

THE DO PROGRAMMING LANGUAGE COMPILER

עדו הירש

ת.ז. 214290249

מנחים: ד"ר נילי נוה ומיכאל צ'רנובילסקי

תאריך: מאי 2022, תשפ"ב

תוכן עניינים

9.....	מבוא
9.....	השראה
9.....	מטרה
9.....	תיאור הפרויקט
10.....	ספר הפרויקט
11.....	ספר השפה Do
12.....	הקדמה
12	קצת על Do
12	מה יהיה בספר השפה
12.....	אבני השפה
12	קבועים – Constants
12	טיפוסי קבועים – Variables
13	משתנים – Variables
13	שמות משתנים
13	טיפוסי משתנים
13	הגדרת משתנים
14	ביטויים – Expressions & Statements
14	Expression
14	Statement
14	אופרטורים – Operators
14	חובונים
14	לוגיים
14	קשרים לוגיים
15.....	תכלות השפה
15	השמה
15	תנאים ולולאות
15	תנאים – Conditions
16	לולאות – Loops
16	True & False
16	הערות – Comments
17.....	דיקון השפה
17	מהי שפה?
17	תחברו לשפה
17	Tokens
18	BNF
19.....	דוגמא לתוכנית בשפת Do
20.....	רקע תאורטי
20.....	קומפイルר - Compiler
20	מהו קומפイルר
20	Compiler vs. Interpreter
20	למה נדרש קומפイルר
21	כיצד עובד קומפイルר
22	שלבי הקומפイルר
22	Lexical analysis
22	Syntax analysis (Parsing)
22	Type checking / Semantic analysis

23	Intermediate code generation
23	Machine independent code optimization
23	Code generation
23	Machine dependent code optimization
23	Register allocation
23	Assembly, linking and loading
24	Symbol table
24	Error handler
24	מבנה לוגי של קומפיילר
24	Front end
24	Middle end
24	Back end
25	מכונת מצבים
25	מהו מנגנון מצבים?
25	מהי שפה פורמלית?
26	סוגים של אוטומטים.
26	Chomsky Hierarchy
26	אוטומט סופי
27	אוטומט מחסנית
28	מכונת טירוגן.
30	תיאור הבעה האלגוריתמית
30	 ניתוח מילוני – Lexical Analysis
30	 ניתוח תחבירי – Syntax Analysis
30	 ניתוח סמנטי – Semantic Analysis
31	סקירת אלגוריתמים בתחום הבעה
31	מנחים
31	Derivation
31	Left-most Derivation
31	Right-most Derivation
32	Left Factoring
33	Parsing Algorithms
33	Top Down Parsing (TDP)
33	Definite Clause Grammar Parsers
34	Recursive Descent Parsing
34	Back tracking
34	דוגמא
35	Predictive parsing
36	LL parser
36	Early parser
37	Bottom Up Parsing (BUP)
37	Shift Reduce
37	Shift step
37	Reduce step
38	דוגמא
39	LR Parser
39	Precedence Parser
39	דוגמא
40	CYK Parser
40	דוגמא
40	Recursive Ascent Parser

41

41	ניתוח מילוני - Lexical Analysis
41	דוגמא לתהילך ניתוח המילוני.
42	מטריצת הסמיוכיות.
42	אלגוריתם ליזיהו אסימון על ידי גרפ זה.
42	פואודו קוד.
43	האוטומט הסופי של המנתה המילוני.
43	אלפבית.
43	אוטומט.

46

46	ניתוח תחבירי – Syntax Analysis
46	Parsing table & Stack
46	Action table
46	Goto table
47	מחסנית.
48	עץ ניתוח.
49	אלגוריתם ניתוח.
49	פואודו קוד.
49	מימוש LR Parser.
49	The Dot notation
50	בניית דיאגרמת המעברים.
51	בנית הטבלאות.
51	מעבר על קשתות - Shift
52	מעבר על מצבים - Reduce
52	First & Follow sets
53	טבלה סופית.
53	דוגמא ניתוח בעזרת ה – Parsing table
54	ניתוח השפה Do.
54	Tokens
54	Grammar
54	BNF
55	Railroad Diagram
58	דיאגרמת המעברים.
59	Action & Goto
59	Action
61	Goto

62

62	ניתוח סמנטי – Semantic Analysis
62	סמנטיקה.
62	מטרוטוי של המנתה הסמנטי.
62	בדיקת נכונות סוג משתנים – Type Checking
62	בדיקת נכונות רצף שימוש במשתנים.
63	בדיקת ייחודיות שמות המשתנים.
63	אסטרטגיה לפתרון.
63	Attribute Grammar
63	SDT & Semantic Actions
63	סיכון.
64	ישום המנתה הסמנטי.
64	בדיקת נכונות סוג משתנים – Type Checking
64	בדיקת נכונות רצף שימוש במשתנים.
64	בדיקת ייחודיות שמות משתנים
65	דוגמא.

66

66	Code Generation
66	get_reg

66	symbol_codegen
67	exp_generator
67	Statements
68	Conditional statements
68	Conditional expressions
69.....	Symbol table
69	שמירה ואחזר מידע על משתנים
69	Scope management
69	מחסנית של Symbol tables
70	טבלה אחת
70	ען של Symbol tables
71.....	Error Handler
71	Syntax errors recovery strategies
71	Exit on Error
71	Panic mode
71	Statement mode
72	Do Error recovery
73.....	Top-Down Level Design
73.....	ابت על
73.....	Compiler
73.....	Lexer
73.....	Parser
74.....	Semantic Analyzer
74.....	Code Generator
74.....	Scope Tree
74.....	Error Handler
75.....	מבנה נתונים
75.....	מנתח מילוני – Lexical Analysis
75	גרף
76	מציאת מצב התחילת של Token
77.....	Syntax Analysis –
77	גרף
77	Parse Table
78	Parse Stack
78	דקדוק השפה – Production Rules
79.....	Parse Tree –
80.....	מנתח סמנטי – Semantic Analysis –
80	Attributes
80	מצביים לפונקציות סמנטיות
81.....	Code Generation
81	מערך רגיסטרים
81	Register Attribute
82.....	Symbol Table
82	Hash table

82	Scope tree
82	תרשים
83.....	Error Handler
83	מציאת מצב התחלתי של Rule
83	מצביים לפונקציות טיפול בשגיאות
84.....	תיאור סביבת העבודה וشفת התוכנות
84.....	شفת התוכנות
84.....	סביבה העבודה
84	Operating System
84	Code Editor
84	C Compiler
85.....	אלגוריתם ראשי
86.....	תרשים מודולים
87.....	מודולים ופונקציות ראשיות
87.....	 מבט על
88.....	מודולים
88	token
89	lexer
91	lexer_fsm
93	parser
95	parse_stack
96	parse_table
98	parse_tree
99	semantic
102	code_generator
105	scope_tree
107	scope
108	symbol_table
110	symbol_table_entry
111	error_handler
115	compiler
116	general
116	io
116	ansi
117.....	מדריך למשתמש
117.....	התקנה
117	דרישות
117	שלבים
117.....	הסירה
117.....	בנייה הקומpileר
117	דרישות
117	שלבים
118.....	שימוש
118	Assembly – Do
119	exe - ↳ Assembly
120.....	xicom אישי

120.....	הסיפור
121.....	אתגרים
121.....	ניהול הפרויקט
121.....	כלים שקיבלת ואכח איתם להמשך
121.....	אילו הייתי מתחילה את הפרויקט היום
122.....	אילו הייתי יכול להמשיך לעבוד על הפרויקט
122.....	השגת מטרות
122.....	סיכום
123.....	ביבליוגרפיה
124.....	קוד הפרויקט
124.....	GitHub
124.....	מבנה קוד הפרויקט
125	bin
125	src
126	scripts
126	tests
127.....	קוד
127	install.bat
127	uninstall.bat
128	scripts
128	build.ps1
128	clean.ps1
128	code_lines.ps1
129	prepend_user_path.ps1
129	remove_user_path.ps1
130	test.ps1
132	src
132	main.c
133	global.h
133	global.c
134	ansi
134	ansi.h
134	ansi.c
135	code_generator
135	code_generator_base.h
137	code_generator.h
140	code_generator.c
160	compiler
160	compiler.h
161	compiler.c
164	error_handler
164	error_handler.h
167	error_handler.c
174	general
174	general.h
175	general.c
176	io

176	io.h
176io.c
178lexer
178lexer.h
179lexer.c
184lexer_fsm
184lexer_fsm.h
186lexer_fsm.c
196parser
196parser_base.h
197parser.h
199parser.c
206parse_stack
206parse_stack.h
206parse_stack.c
209parse_table
209parse_table.h
210parse_table.c
259parse_tree
259parse_tree.h
260parse_tree.c
264scope_tree
264scope_tree.h
265scope_tree.c
270scope
270scope.h
270scope.c
272symbol_table
272symbol_table_base.h
272symbol_table.h
273symbol_table.c
279symbol_table_entry
279symbol_table_entry.h
280symbol_table_entry.c
282semantic
282semantic.h
284semantic.c
291token
291token.h
292token.c
296.....	נספחים
296.....	תרשימים ודיגרמות
296	Github
296	Whimsical
296.....	תרשימים האלגוריתם הראשי

מבוא

השערה

אני אדם שאוהב אתגרים.

לכן, באחד מן השיעורים בכיתה כשהמרצה זרקה לאוויר שפרויקט מתAGER אשר מפתח יכולות הוא בנית קומפיילר, ישר קפצתי על ההזדמנות.

תמיד עניין אותי לדעת כיצד הקוד שאני כותב בשפה שבעיני היא כמו אנגלית, מתרגם ל – 0 | – 1 שהמחשב מסוגל להבין. הדמים אותם我可以 כתוב פשוטה כמו מחשב, שמבנה רק "יש חשמל אין חשמל", לוקחת את המילים שאני כותב לה בשפה שאינו מבין וקלה לי לכתיבת, ותרגמת אותן לפעולות מתמטיות ולוגיות מורכבות.

מטרה

ארצה להוצאה מהפרויקט הזה את המיטב. הן מבחינות לקיחת אתגר ופיתוח עצמי, והן מבחינות רכישת ידע בתחום שלא התעסקתי בו בעבר, קומפיילרים.

ארצה להבין לעומק כיצד קומפיילרעובד ואת התאוריה עליו מתבסס.

ארצה לבחון ולפתח את הידע והקשרים שלו בפרויקט בסדר גודל זהה. פיתוח של אלגוריתמים חכמים ויעילים אשר יפתרו את הבעיות האלגוריתמיות העולות בפרויקט זה, תוך לימוד עצמי של ידע חדש וציבורת ניסיון בנושאים שלא התעמקתי בהם בעבר, כמו מכונות מצבים, עיצוב שפה, מבנה נתונים ועוד.

תיאור הפרויקט

הפרויקט שלי הינו פיתוח מהדר, Compiler, אשר מתרגם מסמך טקסט המכיל קוד בשפה שני עיצורתי, So, לקוד אסטראלי 64 ביט.

שפה התכוונת היא הגדרה של חוקים תחביריים וסמנטיים, שנועדו להגדיר תהליכי חישוב שיבוצעו על ידי המחשב. הגדרת השפה היא חלק בלתי נפרד מבניית המהדר.

כאשר מגדרים שפת תוכנות, מתייחסים כאמור לשולשה מישורים: **מילוני, תחבירי ולשוני**.

المישור المילוני מגדר אילו מילים שיוכות לשפה, ואילו לא.

לדוגמא, המילה if היא מילה המקובלת בשפת C בעוד שהמילה #0@#El איןנה.

المישור התחבירי מגדר אילו רצפי מילים של השפה הם חוקים, ואילו הם לא.

לדוגמא, הרץ; 3 = x אז הוא רצף חוקי בשפת C, בעוד שהרצף 5 then if x אם איןנו.

المישור הלשוני מתייחס למשמעות רצפי המילים, והוא מגדר חוקים כליים שהחיים להתקיים בכל רצף מילים בשפה. לדוגמה, חובת ההצעה – לפני שימוש במשתנה, קיימת חובה להציגו לעילו.

הקומפיילר עושה שימוש בהגדרת השפה על מנת לנתח את קוד שנקלט, וכדי לייצר את תוכנית היעד.

ספר הפרויקט

ספר הפרויקט שאותם קוראים זה עתה מלאה את תהליך ייצור הפרויקט וסביר על כל חלקיו השונים.

הוא מכיל את הגישות המגוונות לפתרון בעיות האלגוריתמיות השונות העולות בפרויקט זה, את ההתלבויות בין האסטרטגיות השונות, את הלך החשיבה שלי בתהליך פיתוח הפרויקט ועוד מידע רב על התאוריות עליון מבוסס הפרויקט, אופן השימוש בו, ומקורות המידע בהם השתמשתי במהלך הפיתוח.

הספר מיועד לכל גורם, מפתחים ולא מפתחים כאחד, המעניין בפרויקט ורוצה להבין אותו ואת תהליך היצירה שלו לעומק.

תודה 

ספר השפה Do

THE DO PROGRAMMING LANGUAGE



Written by Ido Hirsh

FIRST EDITION

הקדמה

קצת על Do

מקור שמה של שפת Do מגיע מק'יזרשמי, OPI, ומהמילה "תעשה!" באנגלית, מילה המעוררת מוטיבציה לעובדה ועשיה. שפת Do דומה בסינטקס שלה לשפות התכנות C ו – Pascal.

מה יהיה בספר השפה

בספר השפה תתואר שפת התכנות Do. יתארו אבני השפה, תוכלת השפה, ודקוק השפה.

אבני השפה

קבועים – Constants

קבוע הוא ערך המופיע ישירות בקוד התוכנית.

קבוע יכול להיות מטיפוסים שונים – מספרשלם,תו.

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

דוגמא: *set x = 81*

בדוגמא שלעיל 81 הוא קבוע, אשר יבן על ידי Compiler קבוע מטיפוס int, מספרשלם.

טיפוסי קבועים

- קבוע מטיפוס מספרשלם - **int**.

על מנת להגיד קבוע מסווג int נכתב את ערכו ישירות בקוד התוכנית:

○ דוגמא:

a / 2

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

קבוע מטיפוסתו – **char**

הגדרת קבוע מסווג char תהיה בתוך שני גרשים בודדים:

○ דוגמאות להגדרה:

char ch = '5'

הערך של התו 5 יכנס 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 מזאה (Identifier).

שמות משתנים

1. שם המשתנה הוא רצף של אותיות בשפה האנגלית, ספרות, והטו '_' (Underscore).
2. רצף זה חייב להתחיל באות בשפה האנגלית או בתו '_'.
3. אין להשתמש במילים שמורות כمزאות.
4. קיימת הבחנה בין אותיות גדולות וקטנות (Case sensitive).

טיפוסי משתנים

כל המשתנה בשפה So יש גם טיפוס (Data-type) אשר מציין את סוג הערבים שהוא יכול להכיל.

ישנם שני סוגי של טיפוסי משתנים:

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

הגדרת משתנים

הגדרה כללית של משתנה:

<Data-type> <Identifier>;

דוגמאות:

int x;

char c;

ביטויים – Expressions & Statements

Expression

יחידה תחבירית בשפת תכנות שנייה להערכה על מנת לקבוע את ערכיה. שילוב של אחד או יותר קבועים, משתנים, פונקציות, אופרטורים (Operators), ו – Expression נוספים, שהשפה מפרשת (לפי הכללים של קידמות ושירות), ומחשבת כדי ליצור ("להציג") ערך. תהליך זה, עבור ביטויים מתמטיים, נקרא הערכה (Evaluation).

בפשטות, הערך המתקיים 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 המבוצעת פעולה כלשהי שיש לבצע. תכנית הנקتابת בשפה צו 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 :

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

אופרטורים – Operators

חסובוניים

- חיבור - $+$
- חיסור - $-$
- כפל - $*$
- חילוק - $/$
- שארית - $\%$

לוגיים

- שווה ל - $==$
- לא שווה ל - $!=$
- גדול מ - $>$
- קטן מ - $<$
- גדול או שווה ל - \geq
- קטן או שווה ל - \leq
- not - !

קשרים לוגיים

- or - ||
- and - &&

תכולת השפה

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

כל פקודה בשפה Do תסתיימ עם נקודה פסיק ; למעט תנאים, לולאות ווסף בלוקים. התוכנית תסתיימ בסטנדרט (Standards) :

השם

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

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

סימול של השמה מבוצע באמצעות המילה השמורה `set` וסימן השווה - =

- הגדירה כללית להשמה:

`set <Identifier> = <Expression>;`

על מנת שהשמה תהיה חוקית, טיפוס המשתנה אליו עושים את ההשמה, כמו למשל ה - -> `<Data-type>` של ה - -> `<Identifier>` ציריך לתאום לטיפוס הערך המושם, כמו למשל המילה השמורה `num`.

- דוגמאות:

`set num = -17;`
`set ch = 'h';`

תנאים ולולאות

תנאים ולולאות הם חלק קוד המתבצעים בהתאם בתנאי מסוים שהוא אמת או שקר.

תנאים – Conditions

لتנאי יכולים להיות שני חלקים:

- if
- else

אם הביטוי נותן תוצאה אמת, הקוד שבחלק של ה – if יבוצע.

ואם נותן תוצאה שקר, הקוד שבחלק של ה – if לא יבוצע.

אם יש חלק של ה – else, הוא יבוצע כאשר הביטוי נותן תוצאה שקרית.

חלקים אלו של ה – if וה – else – יבוצעו 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
done
...
```

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

```
if (<Expression>):
    Do if <Expression> is True
done

else:
    Do if <Expression> is False
done
...
```

לולאות – Loops

לולאה דומה מאוד מבנה שלה לתנאי, if, אך ההבדל היחיד הוא שחלק הקוד שבתוך הלולאה מתבצע **כל עוד** התנאי תקף (כל עוד הביטוי נותן תוצאה אמת), ולאו דווקא 0 או 1 פעמים. יכולות לולאה יכולות להתבצע מספר רב של פעמים.

- while

- דוגמא כללית להגדרת לולאה בעזרת שימוש ב – while:

```
while (<Expression>):
    Do while <Expression> is True
done
...
```

True & False

- false - False .

False = 0

. true – True .

True != 0

הערות – Comments

הערות הן קטעי קוד אשר הקומpileר מתעלם מהם.

לרוב מתכנים משתמשים בהערות על מנת להסביר למה כתבו שורה קוד מסוימת, למה עשו זאת דווקא בצורה הספציפית הזאת ועל מנת להשאיר תזכורות לעצם וلمתכנים אחרים שיקראו את הקוד שלהם בעתיד.

שפת Do גם כן תומכת בהערות. הערות בשפת Do הן הערות של שורה אחת, למשל, כל הקוד מתחילה הערה "# חשב הערה, עד אשר נרד שורה. הערות בשפת Do יתחלו עם #.

- דוגמא להערה:

This is a comment in Do!

דקדוק השפה

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

מהו שפה?

שפה היא אוסף המשפטים שמצויתם לחוקים המוגדרים בתחביר של השפה. משפטים אלו מורכבים מהמילים / האסימונים המוגדרים בשפה. (ראה פרק [רקע תאורטי – מהי שפה פורמלית?](#) עבור ההגדרה של שפה פורמלית.)

תחביר השפה

תחביר השפה ספ, כמו רוב שפות התכנות, הוא תחביר חופשי הקשור (Context free grammar).

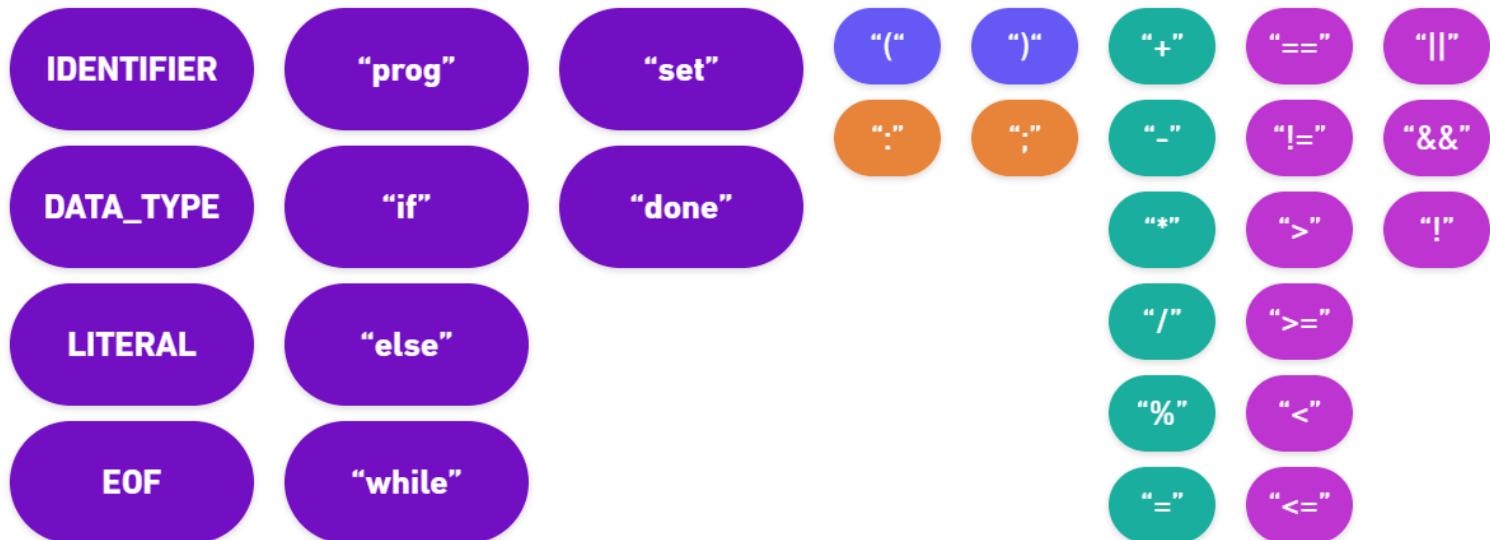
הקדוק מורכב מ – **Terminals** ו – **Non-Terminals**. הסימנים (Terminals) הם המילים (Tokens) שנקלטו קלות מקלט הקוד, בעוד שהמשתנים (Non-Terminals) הם רצפי סימנים ומשתנים.

תחביר השפה מוגדר באמצעות שילוב הסימנים והמשתנים, בכלים שנקבעים כללי יצירה (Production rules). כללי היצירה בעצם מגדירים את המשתנים, באמצעות הסימנים המוגדרים בשפה ומשתנים אחרים.

Tokens

להלן האסימונים, ה – Tokens של השפה Do:

Tokens



BNF

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

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

להלן ה – BNF של השפה Do:

```

# PROGRAM
<program> ::= "prog" <IDENTIFIER> ":" <block> ":)"

# BLOCK
<block> ::= <stmt> <block> | "done"

# STATEMENTS
<stmt> ::= <var_decl_stmt> | <assignment_stmt> | <if_else_stmt> | <while_stmt>
## variable declaration
<var_decl_stmt> ::= <DATA_TYPE> <IDENTIFIER> ";"

## assignment
<assignment_stmt> ::= "set" <IDENTIFIER> "=" <l_log_expr> ";"
## if else
<if_else_stmt> ::= "if" "(" <l_log_expr> ")" ":" <block> <else_stmt>
<else_stmt> ::= "else" ":" <block> | <EMPTY>
## while
<while_stmt> ::= "while" "(" <l_log_expr> ")" ":" <block>

# EXPRESSIONS
<l_log_expr> ::= <h_log_expr> | <l_log_expr> <l_log_op> <h_log_expr>
<h_log_expr> ::= <bool_expr> | <h_log_expr> <h_log_op> <bool_expr>
<bool_expr> ::= <expr> | <bool_expr> <bool_op> <expr>
<expr> ::= <term> | <expr> <expr_op> <term>
<term> ::= <factor> | <term> <term_op> <factor>
<factor> ::= <IDENTIFIER> | <LITERAL> | "(" <l_log_expr> ")" | "!" <factor> | "-" <factor>

<l_log_op> ::= "||"
<h_log_op> ::= "&&"
<bool_op> ::= "==" | "!=" | ">" | ">=" | "<" | "<="
<expr_op> ::= "+" | "-"
<term_op> ::= "*" | "/" | "%"

```

דוגמא לתוכנית בשפת Do

```
prog main:  
    # Variable declaration  
    int result;  
    int i;  
  
    # Assignment  
    set result = 7;  
    set i = 1;  
  
    # Loops  
    while (i <= 10): # Recursive scopes  
  
        # Conditions  
        if (i >= 1 && i <= 10):  
            set result = result * result;  
        done  
        else:  
            # Data types  
            char ch;  
            set ch = 'A';  
  
            # Complicated expressions  
            set result = result + ch * i;  
        done  
        set i = i + 1;  
    done  
done :) # Program with a smile :)
```

רקע תאורטי

קומפיילר - Compiler מהו קומפיילר

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

הקומפיילר גם מייעל וմשפר את קוד המקור כמה שניית. כמו כן, מתריע על השגיאות / אזהרות שמצא, ומצביע הצעות לפתרונות שלדעתו יפתרו שגיאות / אזהרות אלו.

Compiler vs. Interpreter

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

כפי שצווין לעיל, Compiler, עובר על כל קוד המקור לפני הריצה, בודק את תקיןותו, ומתרגם וממיר אותו לחידת הריצה אחת בשפת מכונה. שפות שמתרגםות על ידי Compiler נקראות שפות מקומפלות. דוגמאות לשפות מקומפלות הן C, C++ ו-Java.

בניגוד לו – Interpreter, הוא – Compiler, לא בדיקת תקיןות הקוד לפני הריצה. שפות שמתרגםות על ידי Interpreter נקראות שפות סקריפט. דוגמאות לשפות אלו הן Python, JavaScript ו-JavaScript.

למה נדרש קומפיילר

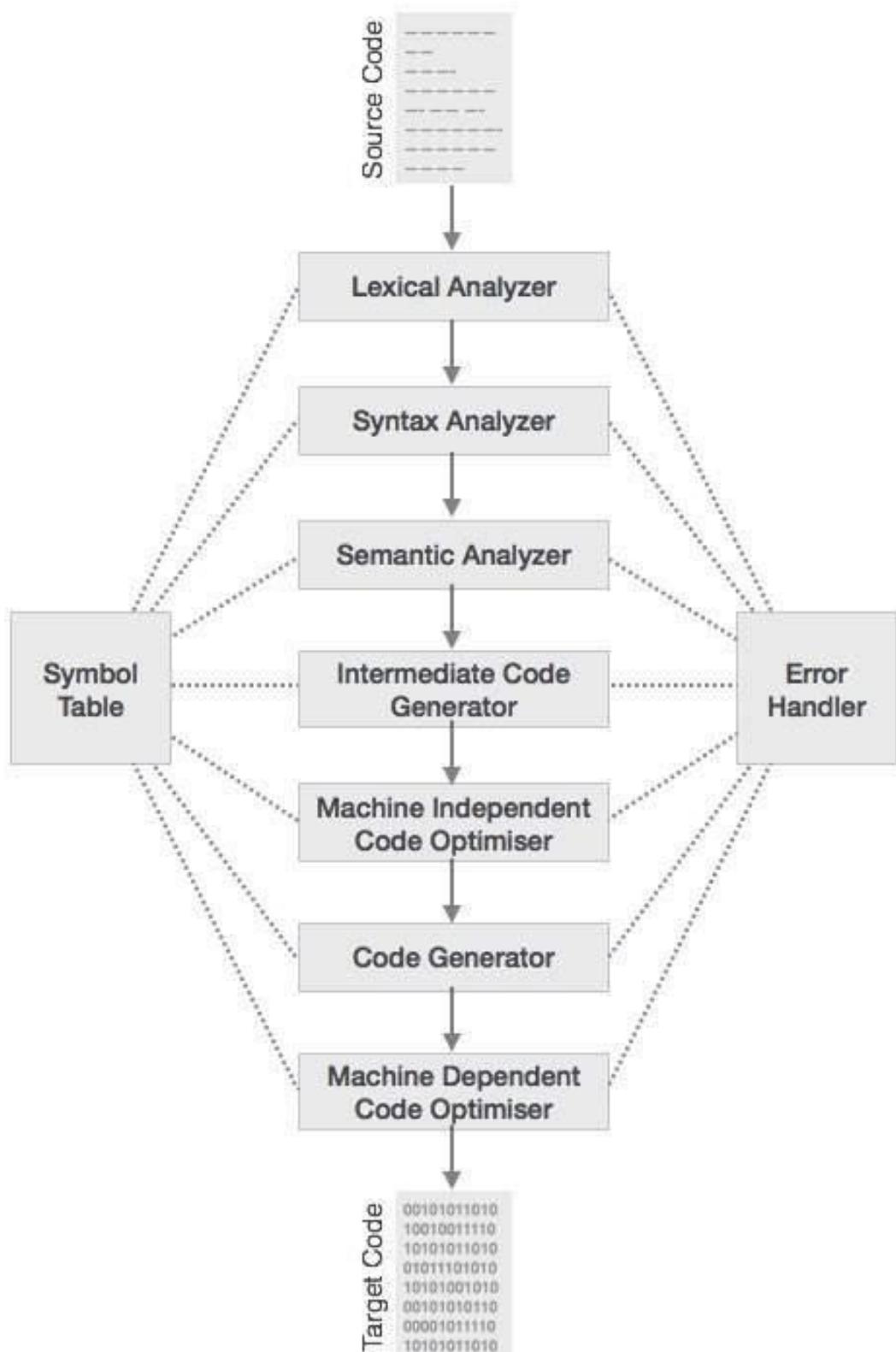
מכיוון שלבני האדם קל יותר לכתוב קוד בשפות תכנות עליות אשר יותר קרובות אליהם (יותר קרובות לאנגלית), מאשר לכתוב קוד בשפת מכונה, אנו משתמשים בשפות עליות אלו לכתיבת קוד. אך המחשב אינו מבין שפות עליות אלו, הוא מבין רק קוד בשפת מכונה.

בשביל לגשר על הפער בין בני האדם לבין המכונה, צריך קומפיילר. שיתרגם את מה שהוא מתכוונים כאשר אכן כתבים קוד בשפה עילית לשפה שהמחשב יבין, לשפת מכונה.

כיצד עובד קומpileר

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

להלן דיאגרמה של השלבים:



שלבי הקומpileר

כפי שציינתי לעיל תהליכי הקומPILEציה הוא תהליך מורכב, ולכן מטרתו לחלק אותו לשלבים. מקובל לחלק כל שלב למודול עצמאי.

להלן חלוקה נפוצה של מודולים:

Lexical analysis

השלב הראשון בתהליכי הקומPILEציה הוא הניתוח המילוני, ה – lexer.

מטרתו של שלב זה הוא לעבור על קוד המקור (source code) שהוא בעצם אוסף תוים נוספים בתוך קובץ, ולהוציא מאוסף תוים זה טוקנים, tokens, הנמצאים בשפה. הטוקנים הם אבני השפה. לדוגמה המילה השמורה `float` בשפת C, או קבוע מסווג מספר שלם, או שם של משתנה כלשהו.

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

הטו q לא יכול להימצא באמצע הגדרת מספר. וכך תוצג שגיאה.

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

Syntax analysis (Parsing)

השלב השני בתהליכי הקומPILEציה הוא הניתוח התחבירי, ה – parser.

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

גם בשלב זה של הקומPILEציה יכולות להיווצר שגיאות. סוג השגיאות שיכולים להיות להיתפס כאן הן שגיאות תחביריות.

דוגמא לטעות תחבירית בשפת C: `int = 5; x`

על פי הגדרת השפה של שפת C על מנת להגדיר משתנה צריך לכתוב את טיפוס המשתנה, אחריו שמו, ואז אם רצים ליצור לשם של ערך. כמוון 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 שמשמעותם מה – lexer לתוך מבנה מאורגן בצורה עץ.

העץ נבנה על פי חוקי ה – grammar. שלב זה משמש חלק מבני השפה, לדוגמה סוגרים, מכיוון שהמבנה של העץ עצמה אומר לנו מה סדר הפעולות לביצוע בשלבים הבאים.

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

Type checking / Semantic analysis

השלב השלישי בתהליכי הקומPILEציה 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 שיכולים להיווצר בשלב זה הן שגיאות של חוסר התאמת טיפוסים, שימוש במשתנה לא מוגדר וכו'.

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

הבדיקה הסמנטית היא בדיקת הקטלט האחורונה בתהליכי הקומPILEציה, ולכן עצ הניתוח שנפלט ממנו מייצג תוכניתת תקינה.

Intermediate code generation

השלב הרביעי בתהליך הקומpileציה הוא שלב ייצור קוד הביניים.

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

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

Machine independent code optimization

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

Code generation

שלב זה לוקה את קוד הביניים מהשלב הקודם ומרתגם אותו לקוד בשפת מכונה.

Machine dependent code optimization

השלב האחרון בתהליך הקומpileציה, מטרתו של שלב זה היא לשפר וליעל את קוד המכונה שנוצר בשלב הקודם. בדומה ל – machine independent code optimizer –

נוסף על כך, ישנו עוד מודולים שכחחים:

Register allocation

لتוכנית יש מספר ערכים שהוא צריך לשמור במהלך הריצה שלו. יתכן שארכיטקטורת מכונת היעד לא מאפשר לכל הערכים להישמר בזיכרון המעבד, או ה – registers. השלב של ה – machine depended code generator, מחליט אילו ערכים לשמר ב – registers, ואילו registers ישמרו ערכים אלו.

Assembly, linking and loading

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

לינקר לחבר כמה קבצי `object` לקובץ `exe` אחד.

כל הקבצים שהוא לחבר יכולים להיות מקובלים על ידי אסמבליים שונים.

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

ה – loader הוא חלק מערכת הפעלה שאחראי על הטעינה של קבצי הריצה (`exe`) לזיכרון, והביצוע שלהם.

הוא מחשב את גודל התוכנית ומוצא لها מספיק מקום בזיכרון. הוא גם מאתחל מספר רגייסטרים שונים שיתחילה את תהליך הביצוע/הריצה 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 לקומpileר למצוא רשומה של מזהה כלשהו בתוכנית ולקבל את הפרטים עליו באופן מהיר יחסית.
ה – **Symbol table** עוזרת גם ב – **Scope management**.

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

Error handler
כפי שתכתבו בתיאור השלבים של הקומpileר, בכל שלב ושלב בתחילת הקומpileציה יכולות להיווצר שגיאות.
בשביל כך יש את ה – **Error handler**.

השגיאות שמתקבלות מדוחות ל – **Error handler** והוא מדווח, ומציג אותן חזרה לתוכנת בתצורה של הודעה. אם לקומpileר יש הצעה מסויימת לפתרון הבעיה, גם היא תוצג בהודעה.

מבנה לוגי של קומpileר
ניתן לחלק את הקומpileר לשלושה חלקים לוגיים: **.back end, middle end, front end**:
Front end
עובד על קוד המקור ומantha אותו.
Middle end
אחראי על ייעול הקוד על מנת לשפר את ביצועי הקו.
Back end
אחראי על ייצור הקוד המובן לשפת מכונה.
.Machine independent code optimization וה – **Intermediate code generation** – בחלק זה משתפים
.Machine dependent code optimization וה – **Machine dependent code generation** – בחלק זה משתפים

מכונת מצבים

מהי מכונת מצבים?

מכונת מצבים (לעתים נקראת גם אוטומט) היא מודל מתמטי חישוב, מכונה אבסטרקטית, אשר נמצאת במצב אחד מתוך אוסף סופי של מצבים בכל רגע נתון, כך שהמעבר בין מצב למשנה בקהלט, והקהלט מוגבל לא"ב המוגדר מראש. (כלומר, תוו' הקטל שאהוטומט מקבל מוגדרים מראש). האוטומט מקבל שפה פורמלית.

מהי שפה פורמלית?

שפה פורמלית, *Formal Language*, בנייתה מיללים שהთווים שלהם נמצאים מעל א"ב מוגדר, והמיללים עצמן בנויות מאוסף מוגדר היטב של חוקים המרכיב תווים מא"ב זה.

הא"ב של שפה פורמלית בניו מאותיות, תווים, סמלים או אסימונים, המוחברים יחד לכדי יצירת מחרוזות בשפה. כל מחרוזת המורכבת מא"ב זה נקראת מילה, ואוסף המיללים השיכוכת לשפה פורמלית נקראות *Well-Formed Words*.

הא"ב של שפה מתואר כך: $\{ \dots, \#, b, 1, 2, 0 \} = \Sigma$, כך שהთווים הכלולים בא"ב נמצאים בתחום הסוגרים המסלולים.

אוסף כל המיללים מעל הא"ב ס מסומן בדרך כלל Σ^* . אורך מילה בשפה נקבע על פי מספר התווים מתוך הא"ב מהם היא מורכبت. לכל שפה יש רק מילה אחת באורך 0 והיא המילה הריקה, לרבות מסומנת, ϵ , e . על ידי שרשור שתי מילים בשפה ניתן ליצור מילה חדשה שהאורך שלה יהיה הסכום של שתי המילים. שרשור מילה עם ϵ ישאיר את המילה כמו שהיא.

שפה פורמלית L מעל הא"ב Σ היא תת-אוסף של Σ^* . ככלומר, אוסף מיללים מעל אותו הא"ב.

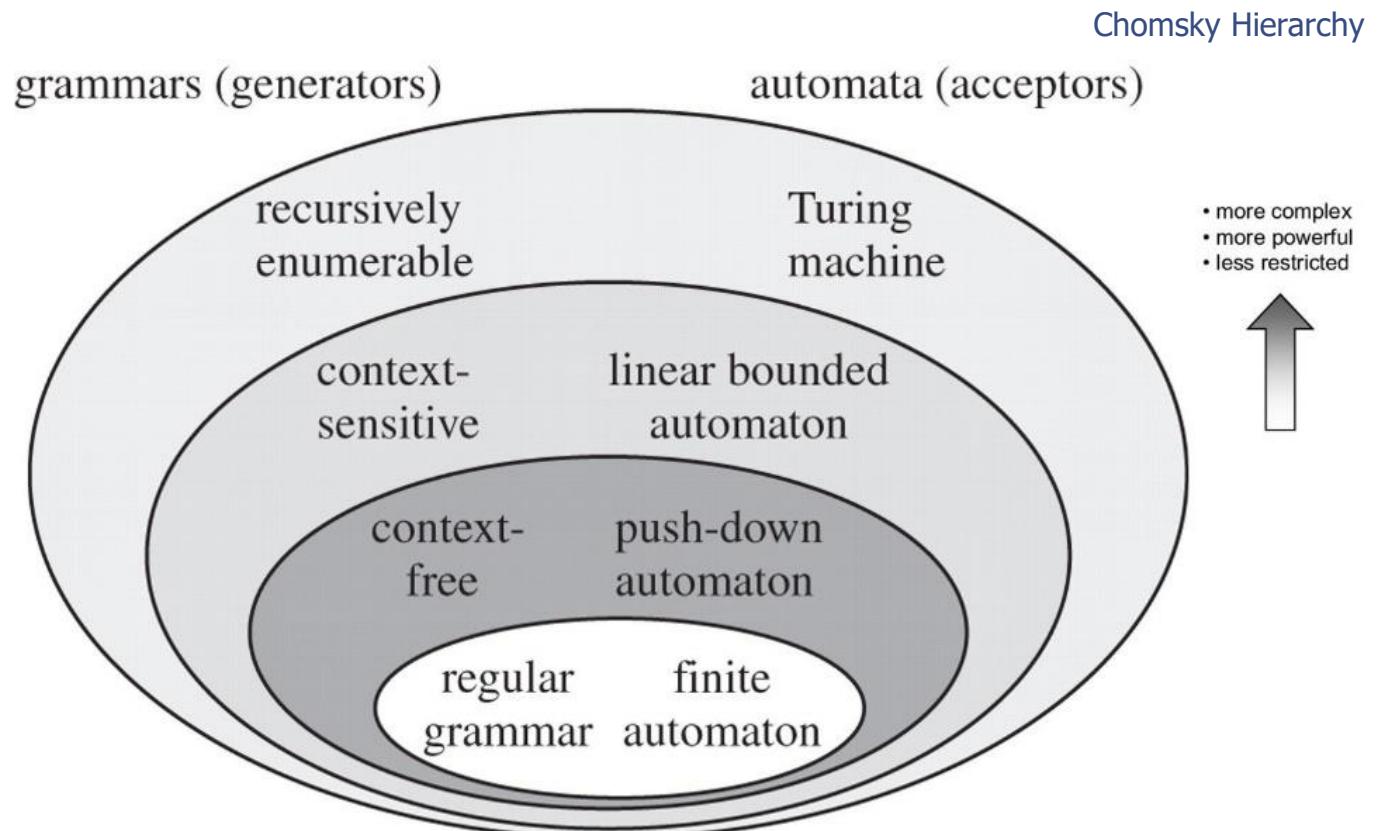
כתיבה פורמלית של שפה: $\{ w \in \Sigma^* \mid \text{Condition} \}$

כלומר השפה מוגדרת ככל המיללים הנמצאות מעל הא"ב של השפה העומדות בתנאי (*Condition*).

ישנם מספר סוגי של שפות פורמליות: שפות רגולריות, שפות חופשיות הקשר, שפות תלויות הקשר, כך שכל אחת מרכיבת יותר מהשניה, ומאפשרת קבלת מחרוזות מורכבות יותר מקודמתה. ההיררכיה בין סוגי השפות מתוארת על פי ה – Chomsky Hierarchy –, כפי שיוצג בהמשך בהסבר על סוגי של אוטומטים.

סוגים של אוטומטים

ישנם סוגים שונים של אוטומטים, כך שיש אוטומטים חזקים יותר, המסוגלים לקבל שפות מורכבות יותר ולפתור בעיות מורכבות יותר, ויש אוטומטים חלשים יותר, ככלומר מסוגלים לקבל רק סוגים מסוימים של שפות, ולפתור רק סוגים מסוימים של בעיות. סוג האוטומטים מסודרים על פי ה – Chomsky Hierarchy, היררכיה של אוטומטים ושפות פורמליות שהגדיר Noam Chomsky, החל מהאוטומט החלש ביותר, אוטומט סופי, לחזק ביותר, מכונת טיורינג, שהוא המודל התאורטי המתאר את המחשב המודרני שלנו כיום.

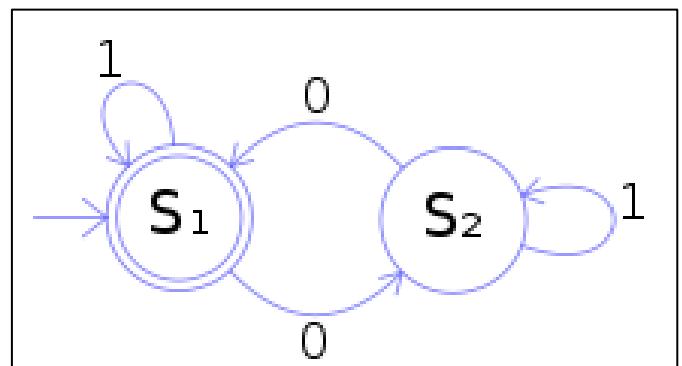


אוטומט סופי

אוטומט סופי (Finite Automaton) / FSM, הוא מכונה מצבים אשר יכולה להימצא במצב אחד מתוך אוסף סופי של מצבים בכל רגע נתון, והמעבר בין מצב למצב מוגנה רק על ידי תווים הקלט.

אוטומט סופי מאופיין בכך שהוא מקבל רק שפות רגולריות, Regular Languages. שפה רגולרית היא שפה פורמלית שיכולה להיות מתווארת על ידי Expression (Regex), והיא נמצאת במקום הנמוך ביותר בהיררכיה של צ'ומסקי (Chomsky Hierarchy).

דיאגרמה המתארת אוטומט סופי מעלה הא"ב {1, 0} = Σ



היעוגלים בדיאגרמה, S_1 ו- S_2 , מတאים מצבים, כך שעיגול כפול, כמו במקורה של S_1 נקרא מצב מקבל. אם האוטומט נעצר על מצב מקבל לאחר שסיים לעבור על כל תווית הקלט, משמע שהקלט הוא חלק מהשפה. לעומת כל מקרה אחר, הקלט לא חלק מהשפה.

החיצים בדיאגרמה מတאים את המעברים בין המצבים השונים של האוטומט כתלות בקלט. לדוגמה, על מנת לעבור מ מצב S_1 למצב S_2 אנו חייבים לקלוט 0. לעבור קלט 1 אנו נשאר במצב S_1 . החץ היחיד בדיאגרמה שרק מסתois במצב ולא מתחילה בשום מצב הוא החץ המצביע על המצב ההתחלתי של האוטומט. לכל אוטומט יש מצב ההתחלתי ממנו מתחילה העבודה.

שוב, מחרוזת תחשב כחלק מהשפה אם ורק אם האוטומט סיים את עובודתו במצב מקבל, ורק לאחר שעבור על כל תווית הקלט.

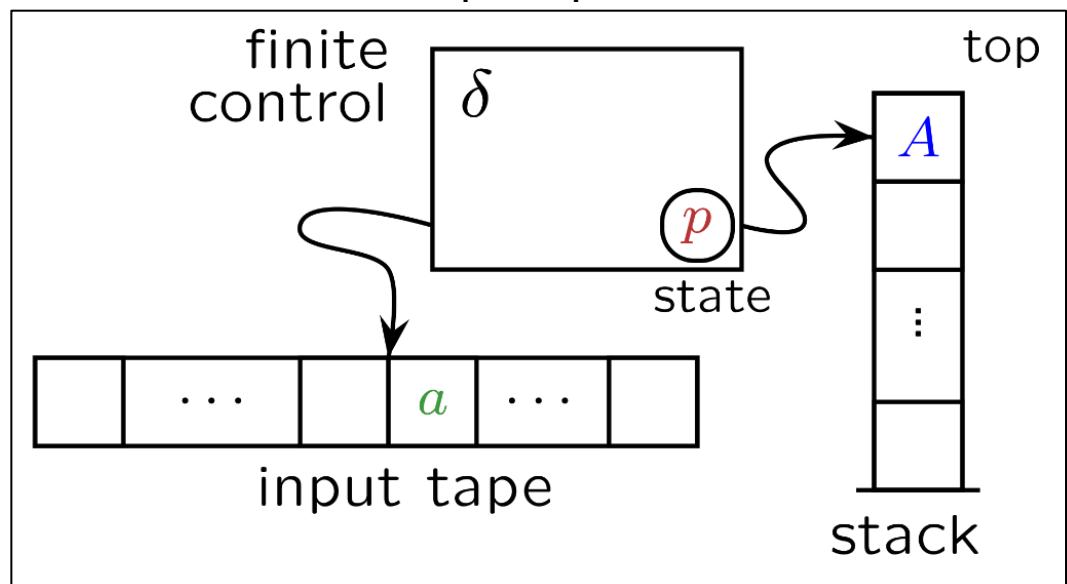
אוטומט מחסנית
אוטומט מחסנית, (Push-Down Automaton) PDA, הוא מכונת מצבים אשר נעזרת במחסנית לביצוע עבודתה, והמעבר בין מצבים מותנה לא רק על ידי תווית הקלט, אלא גם על ידי מה שנמצא בראש המחסנית. הפעולות היחידות האפשריות על המחסנית הם Push ו- Pop של איברים הנמצאים בא"ב של המחסנית.

אוטומט מחסנית מאופיין בכך שהוא מקבל שפות חופשיות הקשורות, Context Free Languages. יsono קשור הדוק בין אוטומט מחסנית ל – CFL. כל מתאר Context Free Grammar יכול להיות מתואר על ידי אוטומט מחסנית.

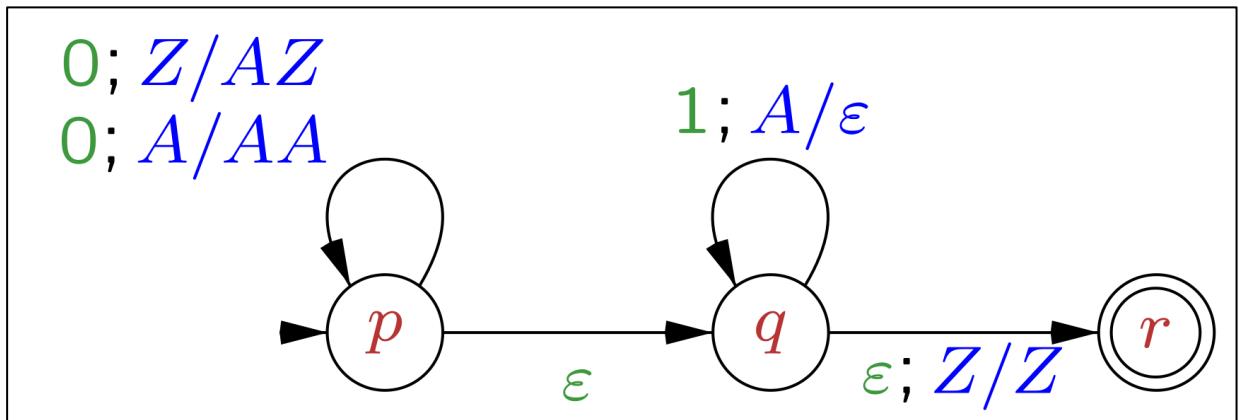
כיוון שכעת ישנה גם מחסנית, PDA מוגדרת כעת על ידי 7 דברים:

- Q – אוסף סופי של מצבים
- Σ – א"ב של הקלט
- Γ – א"ב של המחסנית
- δ – פונקציית מעבר, אוסף סופי של המעברים האפשריים מעלה הא"ב של הקלט והא"ב של המחסנית
- q_0 – המצב בו מתחילה האוטומט
- Z – הסמל ההתחלתי על המחסנית
- F – אוסף המצביעים המתקבלים באוטומט

דיאגרמת המתארת אוטומט מחסנית באופן אבסטרקט



דיאגרמה המתארת אוטומט מחסנית המקביל את השפה $\{ 0^n \mid n \geq 0 \}$ (מחוזות עם מספר שווה של 0 ו-1)



ישן שתי דרכי מקובלות בהן מחוזות תתקבל באוטומט מחסנית. אחת אומרת שם ורק אם האוטומט סיים במצב מקביל (נמצא ב- F), ונקראו כל תווים הקלט המחרוזת תתקבל. השנייה אומרת שם ורק אם האוטומט סיים כאשר המחסנית ריקה, ונקראו כל תווים הקלט המחרוזת תתקבל. הראשונה משתמש בזיכרון הפנימי של המכונה, הלאה הם המצבים, והשנייה משתמש בזיכרון החיצוני של המכונה, הלאה היא המחסנית.

מכונת טיורינג

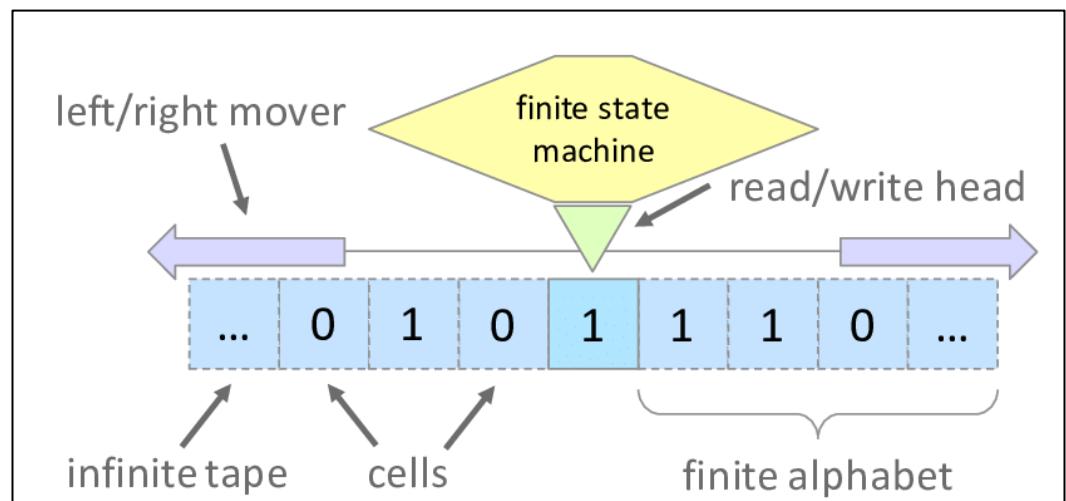
מכונת טיורינג, Turing Machine, היא מודל חישובי מתמטי אשר באמצעותו ניתן לתאר באופן מופשט את פעולתו של מחשב.

מכונת טיורינג מתארת בצורה פורמלית-מתמטית, כיצד ניתן לבצע פעולות חישוביות שונות כגון זהויות מיילים השיקות לשפה פורמלית, ביצוע פעולות חישוב ו邏輯 ובעוד. מכונת טיורינג היא האוטומט חזק ביותר לביצוע חישובים והוא אינו מוגבלת לסוג מסוים של שפה. כל בעיה הניתנת לפתרון יכולה להיפתר באמצעות מכונת טיורינג, ונitinן להוכיח כי קיימות שפות, או במילים אחרות קיימות בעיות, אשר אין ניתנות לחישוב במכונת טיורינג, ולכן לא ניתן לחישוב באמצעות כל מחשב. המושג "חישוב" מוגדר לעתים על סמך הניתנות לביצוע באמצעות מכונת טיורינג.

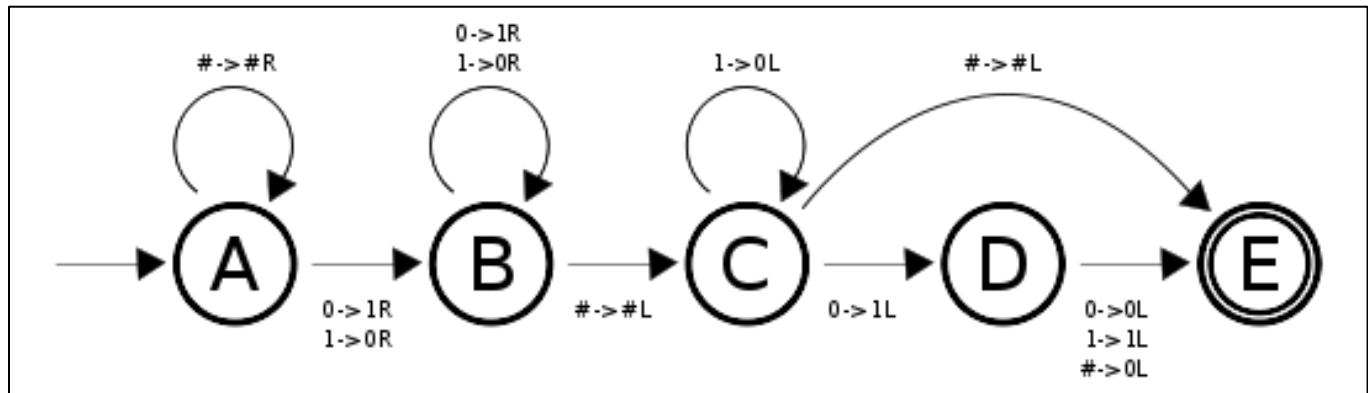
מכונת טיורינג מתוארת באופן דמיוני כסרט אינסופי של תאים, וראש קורא-כותב בעל זיכרון סופי, היכול לקרוא את תוכנו של התא הנוכחי הוא ממוקם, לכתוב באותו התא וכן לנوع ימינה או שמאליה על גבי הסרט. המכונה מקבלת את הקלט שלו באמצעות הסרט, עליו היא יכולה גם לכתוב, ולזוז כרצונה לכל נקודה על גבי.

את המודל הציע אלן טיורינג בשנת 1936, טרם המצאת המחשב המודרני, כדי ליצור הגדרה מתמטית מדוייקת של אלגוריתם, או "תהליך מכני" חישובי.

דיאגרמה אבסטרקטית המתארת את אופן פעולה של מכונת טיורינג



אוטומט המתאר מכונת טיורינג הקוראת מספר בינארי וכותבת את המספר הנגדי לו בשיטת המשלים ל - 2



כפי שציינתי לעיל, מכונת טיורינג היא האוטומט החזק ביותר, המחשב המודרני יכול להיות מתואר בעZRתה, היא אינה מוגבלת לאף שפה או בעיה כל עוד היא ניתנת לפתרון, ואם בעיה אינה ניתנת לפתרון על ידי מכונת טיורינג היא לא יכולה להיפתר על ידי שום מחשב.

מכונת טיורינג מגדרה את הגבול בין הביציאות שאנו יכולים ולא יכולים לפתור על ידי מחשבים.

תיאור הבעה האלגוריתמית

בפרק זה אציג את הבועות האלגוריתמיות העולות בכל שלב מרכזי בתהילך הקומpileציה, אתה אוטן ואtan דוגמאות.

ניתוח מילוני – Lexical Analysis

כיצד נוכל לפרש בכוונה חד משמעותית 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 '='? נדרש להתייחס לשני התווים כ – Token אחד המיצג השווה.

ניתוח תחבירי – Syntax Analysis

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

על מנת לבנות עצ' מדויק, יש להגדיר "נוסחאות" מדויקות שייצגו את השפה, וההיגיון שבה. וכך שציינתי בפרק של [ספר השפה So](#), מקובל לתאר נוסחאות אלו בצורה BNF.

מאחר ומדובר בעץ, ולכל צומת בעץ יש תכונות שונות – "ילדים" שונים זה מזה, צריך למצאו דרך לשמור על התכונות הייחודיות של כל אחד מהם, ועודין לשמר על היכולת להוסטל עלייהם כמקול.

עוד אתגר מעבר להגדרת השפה, הוא "מצבים מתקבלים". למשל ב – #C, מה קורה כאשר המצב הנוכחי הוא Statement, והטוקן הנוכחי הוא Identifier? איך נדע האם לצפות להשמה (Assignment), כמו "3 = X", או לקריאה לפעולה של עצם כמו "()().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 סמנטי – Semantic Analysis

עד כה ראיינו רצף מאוד היגיוני. המנתח המילוני בודק טוקנים ומבודד שוכלים בשפה, המנתח התחבירי מרכיבים מטוקנים אלו משפטים ובודק שמשפטים אלו תקינים בשפה. וcutה המנתח הלשוני צריך לקבל את התוכנית בעץ התחבירי ולבדק האם היא הגיונית. פה יבדק הרצף הלוגית של התוכנית, האם יש שימוש במשתנה שלא הוכרז? האם יש חוסר תאימות בין סוג'י משתנים?

בכדי לפענן את העץ התחבירי, נדרש למצוא שיטה עיליה לעבור עליו, ולפשט אותו. כיצד נזהה את הטיפוס של כל צומת בעץ? כיצד נסורך את העץ בכוונה שתאפשר לנו לקבל את הערכים הנורשים מהחוי, הוריין וילד�?

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 רעוניות. הוא ממוקד ולא צמתים מקשרים, ומתרטתו היחידה היא לייצג את התוכנית במבנה שיאפשר בקלות יחסית לתרגם לייצוג ביניים.

ושוב, כפי שציינתי כבר קודם, המנתח הלשוני (הבדיקה הסמנטית) היא בדיקת הקטל האחורה בתהילך ההידור, ولكن העץ שנפלט ממנו מיצג תוכנית תקינה.

סקירה אלגוריתמים בתחום הבעה

תהליך הניתוח התחבירי, ה – Parsing, הוא התהליך המשמעותי והמורכב ביותר מבחינה אלגוריתמית וריעונית בתהילך הקומpileציה. כתע אציג שיטות שונות ואלגוריתמים שונים הנפוצים בשלב זה.

מונחים

כמו מונחים שאשתמש בהם בתיאור האלגוריתמים.

Derivation

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

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

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

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

Left-most Derivation

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

Right-most Derivation

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

דוגמה

נתון ה – Grammar הבא:

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow id$

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

Left-most derivation

$E \rightarrow E * E$
$E \rightarrow E + E * E$
$E \rightarrow id + E * E$
$E \rightarrow id + id * E$
$E \rightarrow id + id * id$

Right-most derivation

$E \rightarrow E + E$
$E \rightarrow E + E * E$
$E \rightarrow E + E * id$
$E \rightarrow E + id * id$
$E \rightarrow id + id * id$

Left Factoring

אם יותר מ – Production rule אחד מתחילה באותה קיומת, אז ה – Parser לא יכול לבצע הכרעה באיזה מהחוקים הוא צריך לבחור בשבייל לנתח את הקטלט הנוכחי.

דוגמא

אם כללי ייצור מסוימים נראה כך:

$$A \Rightarrow \alpha\beta \mid \gamma \alpha \mid \dots$$

המנתח לא יודע להחליט איזה חוק לעקב, כיוון שני החוקים מתחילהים באותו Terminal (או Non-terminal). על מנת להסידר בעיה זאת משתמש בטכניקה שנקראת Left factoring.

משמעות Left factoring מミירה את ה – Grammar כך שלא יהיו חוסר הוודאות האלו. היא עובדת כך שעבור כל קיומת שימושת יותר מפעם אחת יוצרים כל חדש וההמשך של הכלל הישן משורשר לכל החדש.

דוגמא

הכלל הקודם יוכל כתעת להיראות כך:

$$\begin{aligned} A &= > \alpha A' \\ A' &= > \beta \mid \gamma \mid \dots \end{aligned}$$

עכשו למנתח יש רק הכלל אחד עבור הקיומת המסוימת הזאת, מה שמקל עליו לקבל החלטות.

Parsing Algorithms

על מנת ליצור Parse Tree עליו יتبסס תהליכי הקומpileציה, ישנו כמה אלגוריתמים הנקראים Parsing Algorithms. אלגוריתמים אלה מחלקים לשני סוגים עיקריים.

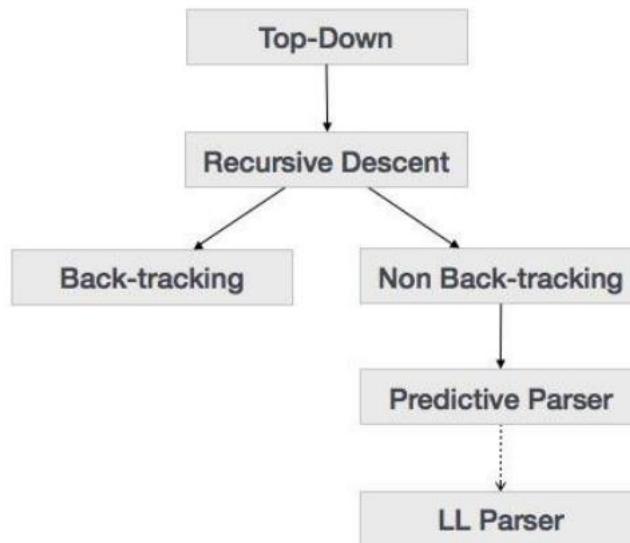
1. Top Down Parsing (TDP)
2. Bottom Up Parsing (BUP)

שפות תכנות הן בדרך כלל CFL במשמעות מוכנות מצבים, ובאופן יותר ספציפי מוכנות מצבים המשמשות במחשנית (Pushdown machines). لكن האלגוריתמים שבעת אציג ישתמשו באופןוט מוחסנית לרוב, על מנת לבצע את פעולה ה – Parsing.

Top Down Parsing (TDP)

טכנית בה עבררים מחלוקתם העליונים לחלקים התחתיונים של העץ התħבורי, על ידי שימוש בכללי השכתב של Grammer השפה. עבררים מה – Grammar קלקלת.

:Top Down Parsing מספור סוגים של



cut אציג ואסביר על כמה אלגוריתמים שימושים בגישה של Top Down.

Definite Clause Grammar Parsers

(DCG) Definite Clause Grammar הוא דרך להביע תħbiri של שפה, בין אם שפה טבעיות או פורמלית. DCGs מזוהים בדרך כלל עם Prolog, שפת תכנות לוגית שבניגוד לשפות תכנות רבות אחרות, מיועדת בעיקר לשפת תכנות הצהרתית, כך שההיגיון של התוכנית מתבטא במנחים של יחסים, המיצגים כעובדות וככלים. ביצוע "חישוב" בה מתבצע על ידי הפעלת שאלתה על היחסים הללו.

```

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

← Prolog – DCG ב –

Recursive Decent Parsing

צורה נפוצה של TDP. בשיטה זו, כיוון שהיא שיטה שמתבססת על הגישה של Top Down, עז הניתוח נוצר מלמעלה למטה, והקלט נקרא משמאלי לימין. שיטה זו משתמשת בפונקציות עבור כל Non-terminal ו-Terminal שנמצא ב- Grammar השפה. Recursive descent parser יוצר את עז הניתוח תוך מעבר רקורסיבי על הקלט, מה שיכל לארום לו לשוב מ- Back tracking. (האם יהיה או לא יהיה תלו依 ב- Grammar השפה, אם הוא Left factored –). (Back tracking, הוא ימנע מ- Left factored).

בגלל ה- Back tracking ייעילותו של האלגוריתם יכולה להיות אף אקספוננציאלית, כלומר (2^n).
גרסה של Recursive decent parsing שלא משתמשת ב- Back tracking נקראת Predictive parsing.

Back tracking

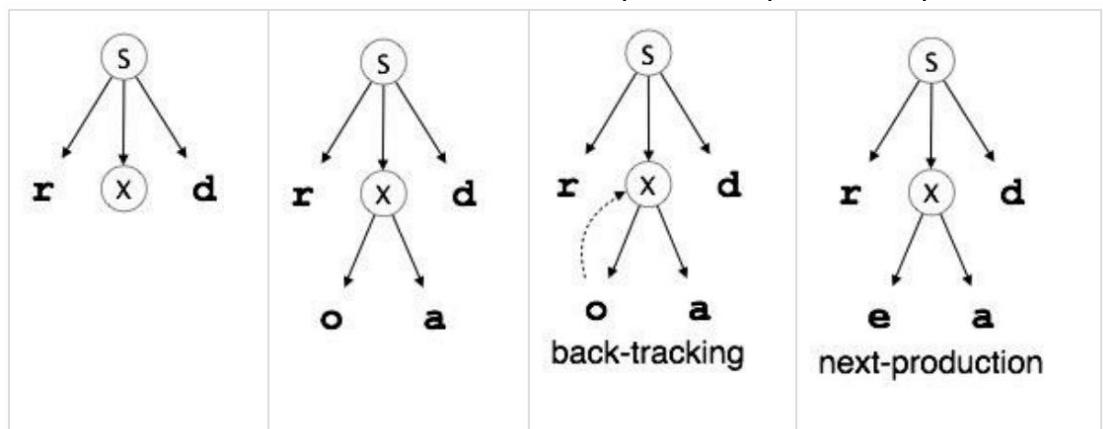
כאשר ה- Parser משתמש בשיטה של Recursive decent parsing (ו- Grammar לא Left factored), ייצרו מצבים במהלך המהלך הניתוח של הקלט בהם המנתח יכול לטעות, וזאת כנראה בגלל שעשו בחירה לא נכונה של כל מסויים בדרך. לכן, המנתח חוזר חוזרת למקומות אחרים בו ביצוע הכרעה, ושם בוחר באופציה אחרת. החזרה החזאת למקום האחרון בו ביצוע הכרעה, על מנת לבצע הכרעה חדשה, נקראת Back-tracking. רק כאשר ניסה את כל האפשרויות ולא הצליח להתאים את הקלט לכליה השפה, ניתן להבין שהקלט הוא לא תקין מבחינת השפה.

דוגמא

נתון ה- Grammar הבא:

$$\begin{aligned} S &\rightarrow rXd \mid rZd \\ X &\rightarrow oa \mid ea \\ Z &\rightarrow ai \end{aligned}$$

עבור מחרוזת הקלט "read" נראה תהליך הניתוח:



Predictive parsing

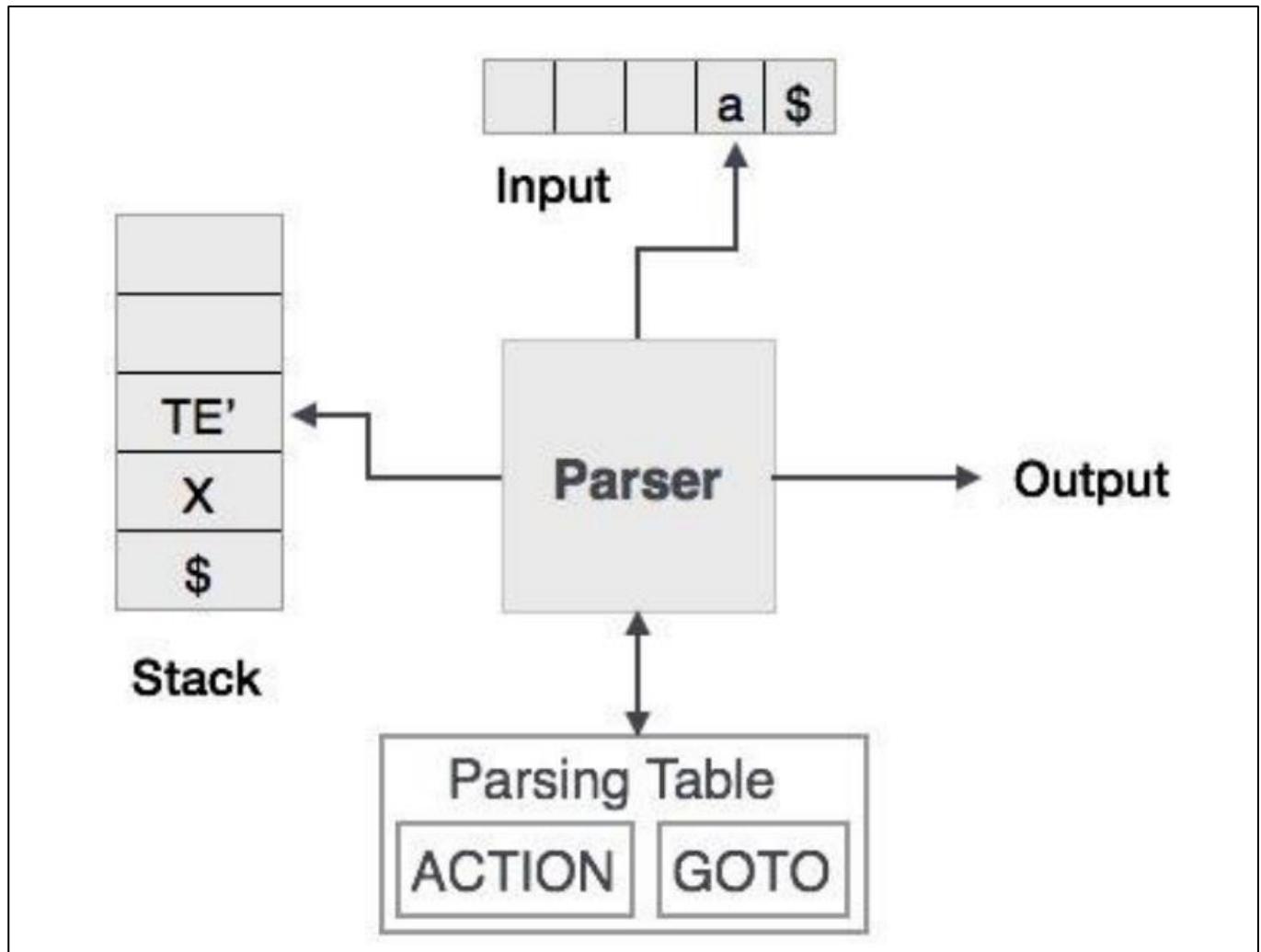
Recursive descent parser הוא יכולת לחזות באיזה Production הוא צריך להשתמש בשביל להחלף את הקלט. עקב כך הוא לא סובל מ – Back tracking.

בשביל להשיג יכולת חיזוי זו, ה – "מציז" לסמלים הבאים בקלט. בשביל שהוא ללא Back tracking מוגביל את ה Grammar – Parser יכול להיות רק מתת-קובוצה של CFGs הנקראת .LL(k) Grammars.

Predictive parser משתמש במחסנית (Stack) ובטבלת ניתוח (Parsing table) בשביל לנתח את הקלט וליצור 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 כלליים, ב – Predictive parsing יש לכל היוטר כל אחד עבור כל קלט מסוים. כך שבמקרים בהם אין אף כלל שתוואם את הקלט, תהליכי הניתוח נכשל.

דיאגרמה הממחישה את העבודה של ה – Predictive parser

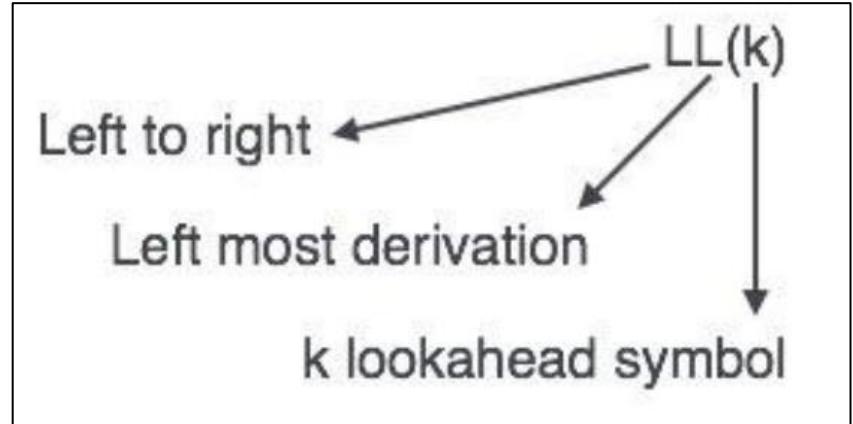


LL parser

.Context free grammars LL parser מקבל LL grammars. כפי שציינו לעיל, סימני האלגוריתמים שהציגו קודם, כולמר, ניתן לישם LL parsing באמצעות שני האלגוריתמים "מציצים" קדימה (Look ahead).

Predictive descent ו Recursive descent ניתנים ליישם LL parsing באמצעות שני האלגוריתמים שהציגו קודם, כולמר, (באמצעות עזרה של טבלה).

parser LL נכתב גם כ – (k)LL, 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 המשמש בטכנית של dynamic programming על מנת לנתח את מחוזת הקלט.

האלגוריתמים הקדומים שתיארתי, לדוגמא Recursive descent, מבוססים על חיפוש רקורסיבי של מבנים תחביריים אפשריים אשר יקבעו את מחוזת הקלט. שיטה זו של חיפוש יכולה לגרום לכך חלקים מהמבנה התחבירי הכללי אשר מקבלים חלק מסוים מחוזת, מייצרים שוב ושוב. החזרה הזאת על פתרונות חלקים בתוך תריליך המבנה התחבירי הכלול, היא התוצאה של ה – Back tracking. dynamic programming נתון חלופה יעילה יותר בה החלקים שכבר יוצרו יישמרו לצורך שימוש חוזר בתהיליך השלים של הניתות. קר לא יהיה צורך לחזור על אותן חישובים שוב ושוב, מה שמייעל את האלגוריתם.

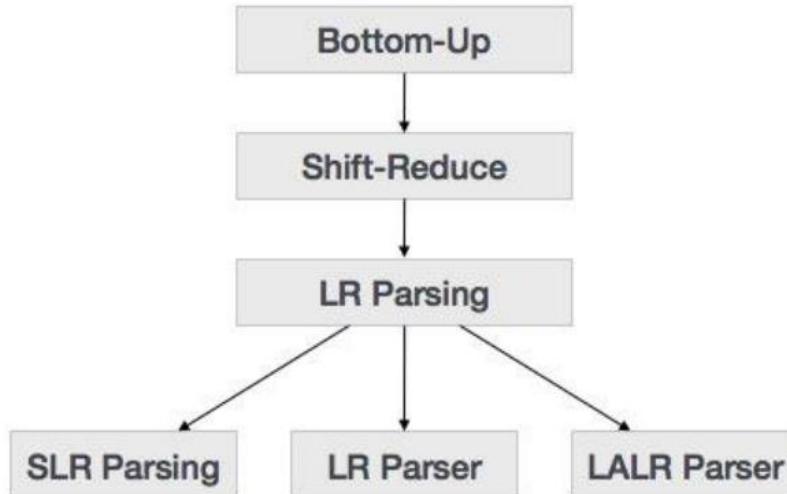
שמירה על פתרונות חלקים אלו מתבצעת באמצעות מבנה נתונים הנקרא טבלה, chart. لكن גרסאות שונות של ה – chart parsing גם Early parser נקראות גם chart.

יעילות האלגוריתם היא $O(n^3)$

Bottom Up Parsing (BUP)

טכנית בה עוברים מהחקקים התחכמוניים לחקלים העליונים של העץ התחבירי, על ידי שימוש בכללי השכתב של Grammer השפה. עוברים מהקלט אל ה - Grammar.

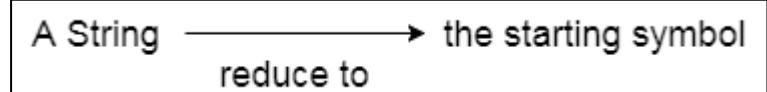
Diagramma המתארת מספר סוגי של Bottom Up Parsing :



cutet אציג וסביר על כמה אלגוריתמים שימושים בגישה של Up Bottom .

Shift Reduce

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



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

Shift step

שלב זה מבצע מעבר לסמל, Symbol , הבא שmagiu מהקלט, והוא נקרא גם symbol . Shifted symbol סמל זה נדחף (PUSH) למחסנית. ה – Non-terminal Shifte מטופל כזאת את של עז הניתוח .

Reduce step

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

יעילות האלגוריתם היא $O(n)$

דוגמה
נתון ה-grammar הבא:

$$\begin{aligned} S &\rightarrow S+S \\ S &\rightarrow S-S \\ S &\rightarrow (S) \\ S &\rightarrow a \end{aligned}$$

כרגע תחילה הניתנו:

a1-(a2+a3)

עבור מחרוזת הקלט

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

LR Parser

Parser LR הוא Parser מסווג בBottom, אשר מנתה שפות חופשיות הקשורות דטרמיניסטיות בזמן לינארי. לרוב משתמש בשיטה של LR Parser. Shift Reduce Parser – LR(k), כר שבדומה ל – LL(k), ה – L מסמנת קריאה של הקלט משמאליימן (Left to Right), ה – R מסמנת derivation most Right, והוא – k מסמן את מספר הסמלים שה – Parser מציין "עליהם קדימה" (Look ahead).

ויראיות שונות של LR Parser הן:

- Look Ahead LR – LALR
- Simple LR – SLR
- Canonical LR – CLR

יעילות האלגוריתם היא $O(n)$

Precedence Parser

בשם המלא, Operator-Precedence Parser, הוא Parser פשוט המשמש בשיטה של Shift Reduce Parser. מסוגל לנתח תתי-קובוצות של Grammars Precedence Parser – LR(1). ה – קידימות (Precedence) של האופרטורים בביטוי מסוים על מנת להחליט איך לנתח אותו.

דוגמה

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

CYK Parser

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

דוגמא

עבור המשפט 'abac' = W, הוא יבדוק אם ניתן ליצר:

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

כל בדיקה תבוצע בתקף על הבדיקה הקודמת.

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

דוגמא לתהיליך 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
$\rightarrow AB \mid BC$ $\rightarrow BA \mid a$ $\rightarrow CC \mid b$ $\rightarrow AB \mid a$					
1. <u>ba</u> , <u>aa</u> , <u>ab</u> , <u>ba</u> 2. <u>baa</u> (<u>b</u> U <u>aa</u> <u>ba</u> U <u>a</u>), <u>aab</u> (<u>a</u> U <u>ab</u> <u>aa</u> U <u>b</u>), <u>aba</u> (<u>a</u> U <u>ba</u> <u>ab</u> U <u>a</u>) 3. <u>baab</u> (<u>baa</u> U <u>b</u> <u>b</u> U <u>aab</u> <u>ba</u> U <u>ab</u>), <u>aaba</u> (<u>aab</u> U <u>a</u> <u>a</u> U <u>aba</u> <u>aa</u> U <u>ba</u>) 4. <u>baaba</u> (<u>baa</u> U <u>ba</u> <u>ba</u> U <u>aba</u> <u>b</u> U <u>aaba</u>)					

יעילות האלגוריתם היא $O(n^3)$

Recursive Ascent Parser

Recursive ascent parser (Look Ahead LR Parser) שמשתמש בפונקציות רקורטיביות, מאשר בטבלאות.

הוא עובד כך שכל פונקציה של ה – Parser מייצגת מצב מסוים אחד במקונית המצבים. בתוך כל פונקציה מתאפשרת הבחירה על איזה פעולה לעשות בהתאם ל – Token הנוכחי. ברגע שה – Token זהה, הפעולה שתילקה מtabססת על המצב הנוכחי. יש שתי פעולות יסודיות שיכלות להילך, Shift ו – Reduce.

אסטרטגייה

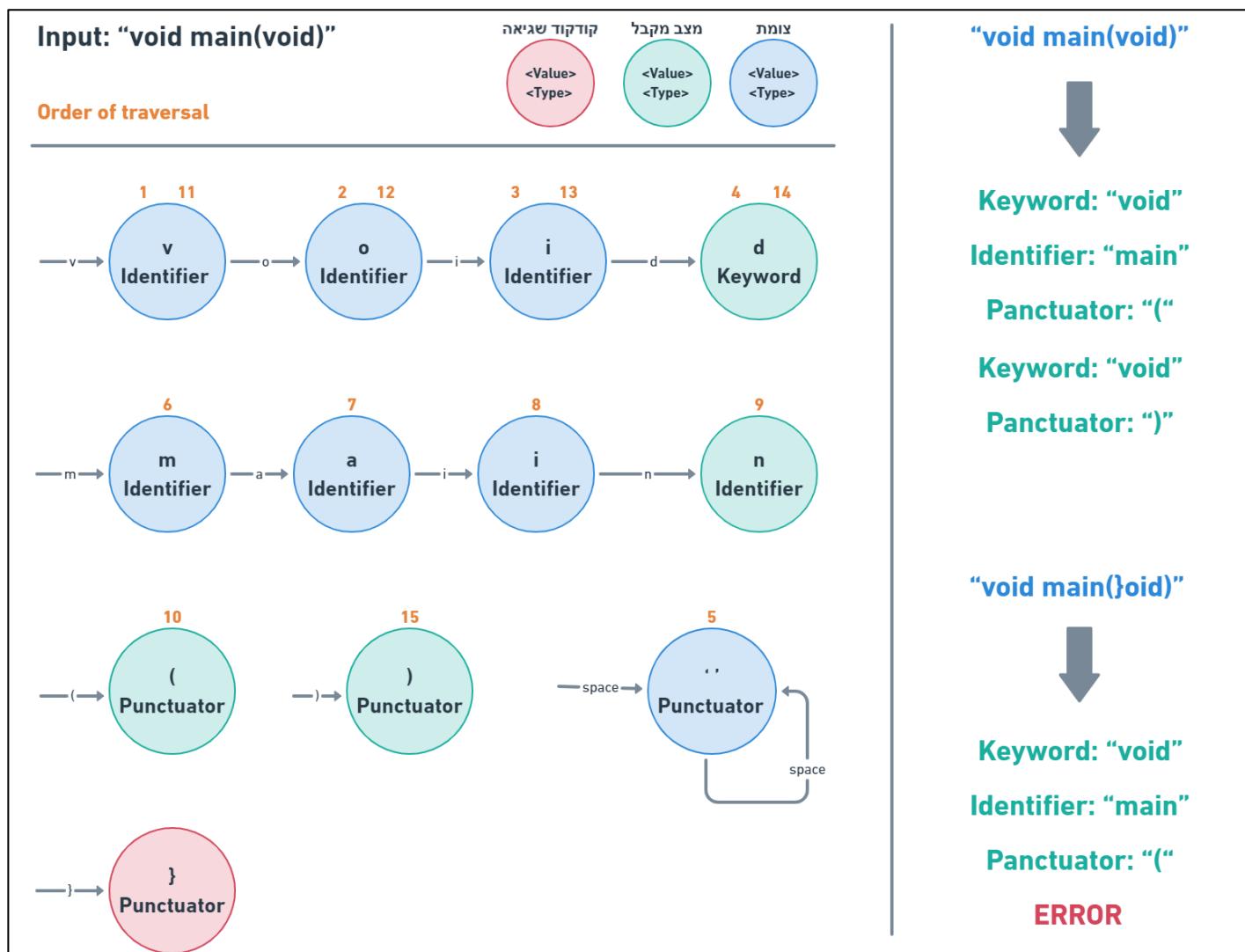
כעת אציג כיצד אסטרטגיה זו בא לידי ביטוי בשלבים השונים של תהליכי הקומpileציה.
ליעילות זמן ריצה לינארית, אشتמש במבנה הנתונים גרפ' שיציג את מكونת המצביעים של Compiler – Compiler.

ניתוח מילוני - ניתוח

שוב, מטרתנו לשאוף לעלות זמן ריצה לנארית. עקבvr אנו לא רוצים להתייחס לקוד המקור אותו אנחנו מתרגמים כטוסט. אך אנו מתרגמים את קוד המקור לאסימונים בעלי משמעות בקוד, כפי שמתואר בחלק של הסקירה התיאורטית. המסלול שיוציא יתבסס על התווים מקוד המקור. המסלול יסתהים כאשר הגיעו לכך שהוא עלה, אשר יחזיר סוג אסימון (Identifier, Literal, Operator, Punctuator, Keyword) ששמור אצלן. במקרה וקוד המקור יחיל רצף תווים לא חוקי, המסלול יוביל לקובדוקוד שגיאה.

דוגמאות לתהיליר הביגטום המילוני

וגם לארוך מחרוזת הקלט "void main(void)" לאסימונים בשפת C על ידי שימוש באוטומט סופי.



מטריצת הסמיכויות

דוגמא עבור מטריצת הסמיכויות של הגרפ' המיצג את מכונת המצבים של המנתה המילוני.

	י	ו	ד	ו	ו	ו	ו	ו	ו	ו	...
1	9	5	5	5	2						
2	9			3	5	5	5				
3	9	5	4	5	5	5	5	5			
4	9	7	5				5	5			
5	9	5		5	5	5	5	5			
6	9						6	6	8		
...											

במטריצה שלහן כל שורה מייצגת קודקוד. כל עמודה מייצגת שכנות של קודקוד. השכנות מסודרת לפי סוג התווים. עבור מצב מסוים ותו הנוכחי מקוד המקור, נדע לאיזה מצב אנחנו צריכים לעבור. תא ריק מצין שאין שכנות בין המצב הנוכחי ובין התו הנוכחי מקוד המקור, כלומר העונה לעלה זהה סוף האסימון.

אלגוריתם ליזיהו אסימון על ידי גרפ' זה

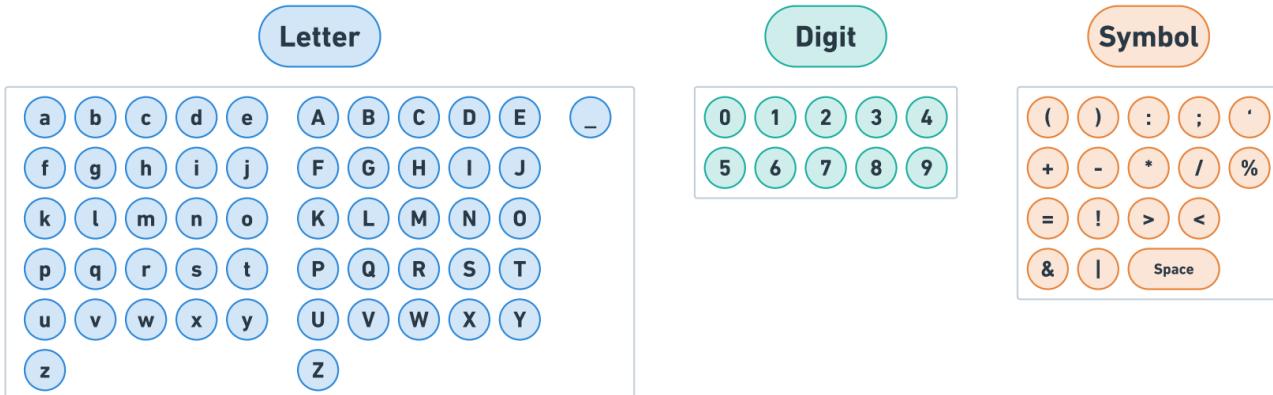
- מעבר על הגרפ' על פי תוו קוד המקור עד הגיעו לעלה (יצירת המסלול תتبoso על פי רצף התווים)
- יצירת אסימון על פי העלה שהגענו אליו

פואדו קוד

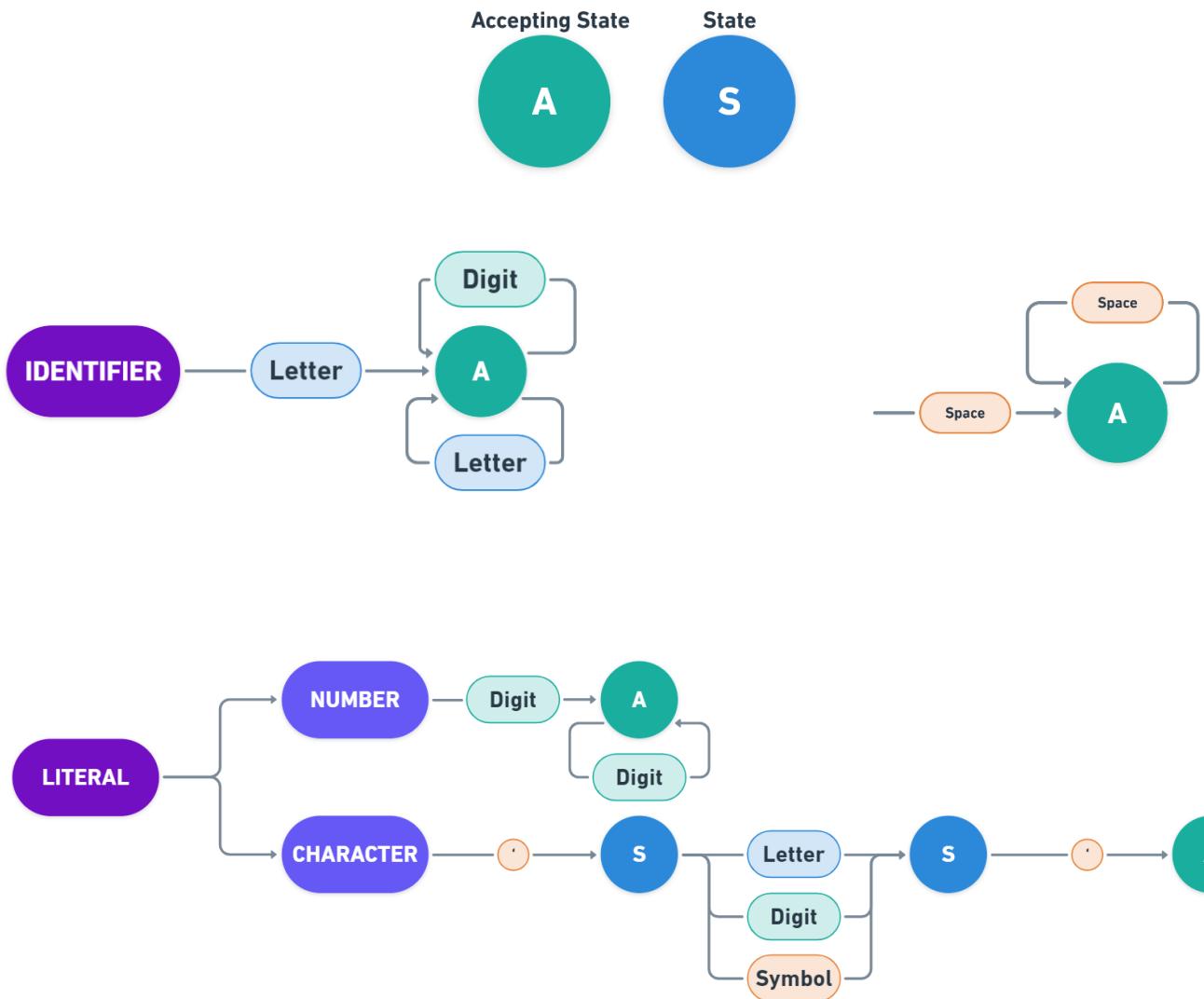
- התחל
- אתגרף האוטומט הסופי של המנתה המילוני
- עבור אל המצב ההתחלתי עبور התו הנוכחי בקוד המקור
- כל עוד לא הגיעו לקודקוד שהוא עלה, בצע:
 - מעבר לקודקוד הבא על פי המצב הנוכחי והטו הנוכחי מקוד המקור
 - התקדםתו אחד בקוד המקור
 - צור אסימון על פי העלה שהגעת אליו
 - החזיר אסימון
- סיום

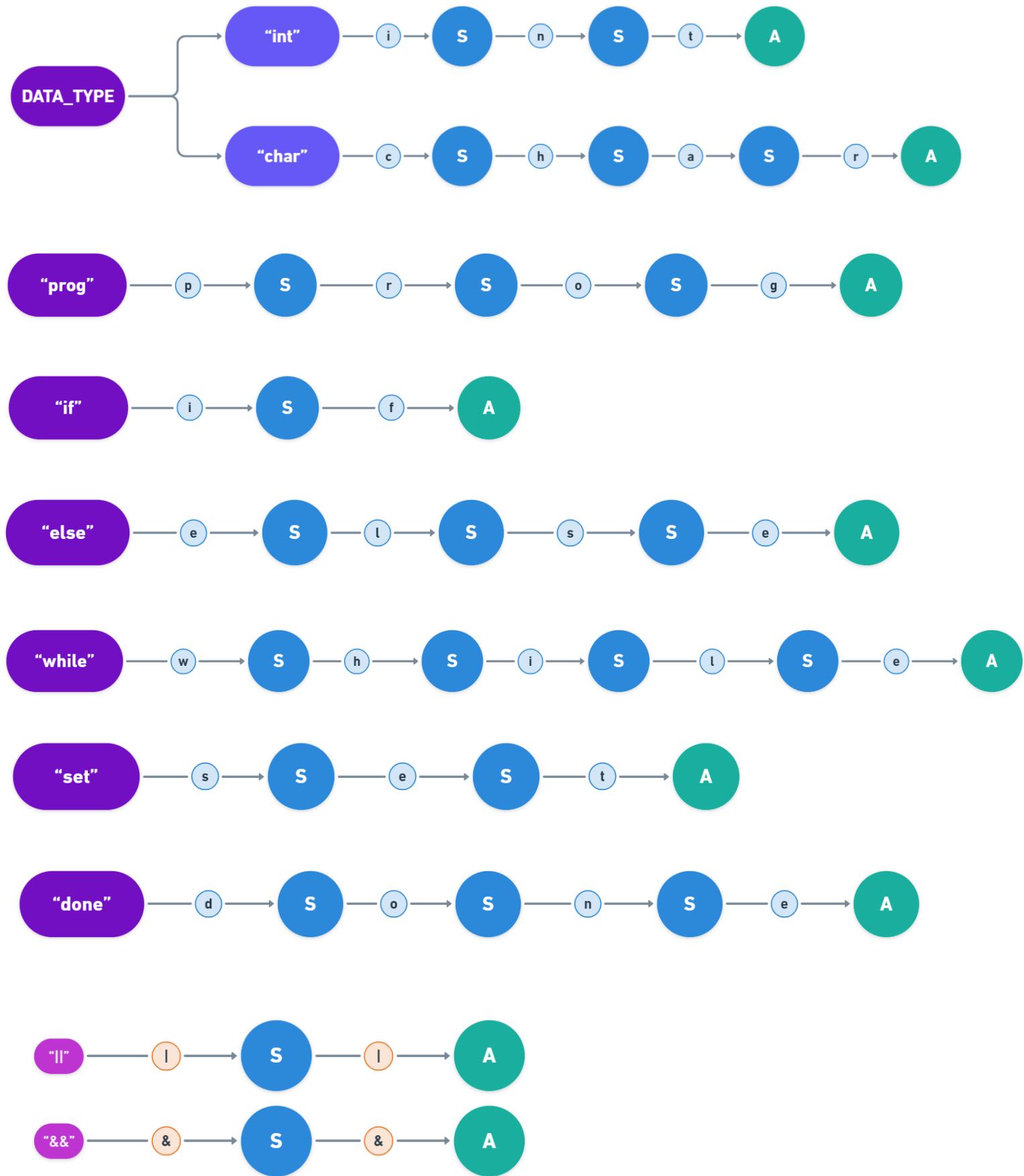
האוטומט הסופי של המנתח המילוני
האוטומט הסופי של המנתח המילוני, ה – Lexer, עברו השפה שלו, Do.

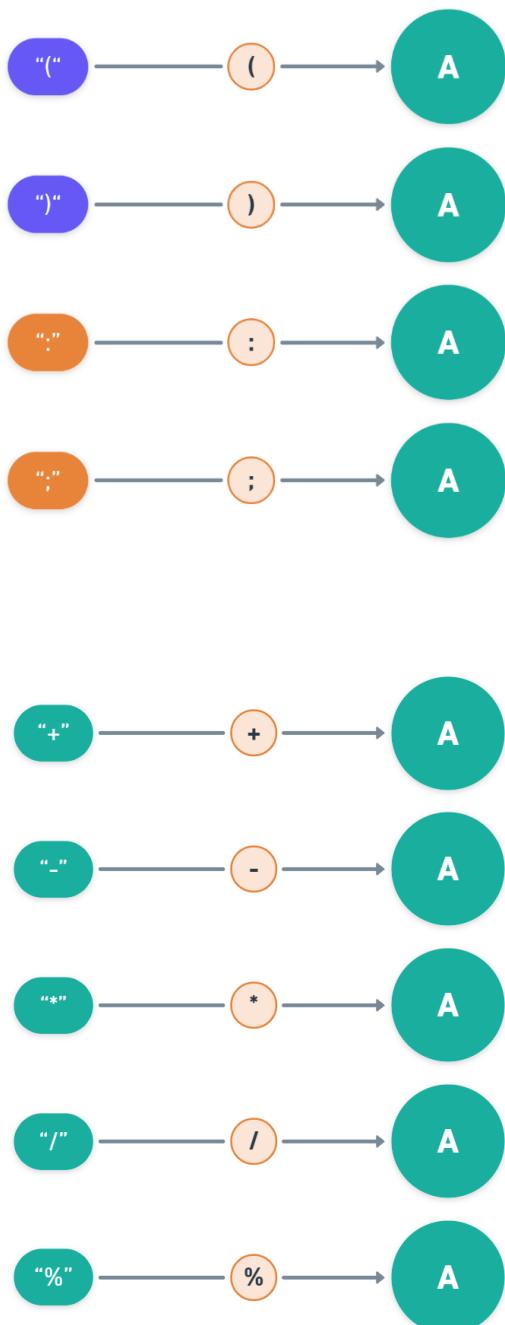
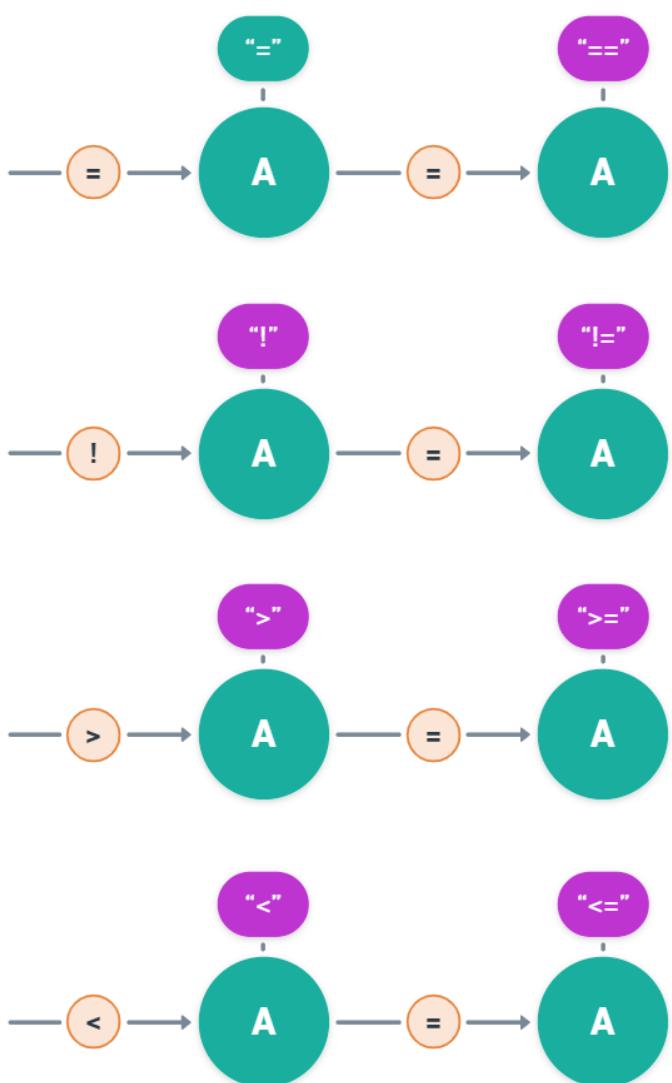
אלפבית



אוטומט







Syntax Analysis – תחבירי

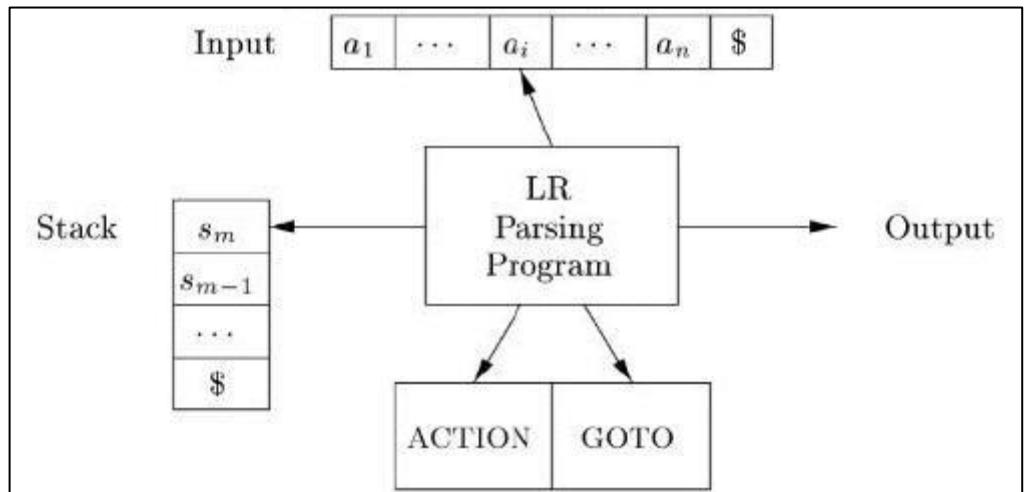
בשלב זה נרצה לבדוק האם רצף האסימונים שקיבלנו משלב הקודם הוא רצף אסימונים תקין על פי דקדוק שפת המקור והוא אכן מתרגם, ובנוסף נרצה ליצור עץ ניתוח שיציג את התוכנית התקינה.

בשלב הקודם, שלב הניתוח המילוני, השתמשנו בטבלה עבור יישום הגרף שמייצג את האוטומט הסופי של המנתה המילוני, בה עבר כל מצב ותו נוכחי ידענו לאיזה מצב עליינו לעבר.

שלב הניתוח התחבירי מורכב יותר. התחבר של שלב זה הוא תחבר חופשי הקשור (CFG) ולכן אינו יכול להתקבל על ידי אוטומט סופי. על מנת למשו השתמש באוטומט מחסנית.

כעת שהבנו זאת, טבלה אחת של מצב ותו נוכחי לא תספיק לנו לביצוע המטלה. אנו עדין שואפים ליעילות זמן ריצה לנארית ולכן השתמש באלגוריתם ניתוח מסוג Up-Bottom, אלגוריתם ה- LR Parser, ובאופן כללי, פירטתי בפרק [סקרת אלגוריתמים בתחום הבעה](#).

דיאגרמה המתארת את פעולתו של LR Parser



Parsing table & Stack

על מנת למש את ה – LR Parser נשתמש במחסנית, ובשתי טבלאות שיחד יקראו Parsing table. שתי הטבלאות הן .Goto table – Action table

Action table

מצינית למתנה איזו פעולה – Shift, Reduce – הוא צריך לבצע, בהתאם למצב הנוכחי ואסימון הבא מהקלט.

כל שורה תציג מצב, וכל עמודה תציג אסימון (Token / Terminal) שנמצא בשפה. נגייע לתא בטבלה על ידי המצב הנוכחי והאסימון הבא מקוד המקור. כל תא בטבלה יגיד לנו איזו פעולה علينا לעשות, Shift או Reduce, ולאייה מצב עליינו לעבר לאחר מכן. תא ריק מצין שהגענו למצב שלא יכול להתקיים על פי תחבר השפה, כולם שגיאה.

בטבלת ה – Action ישנו רק תא אחד שמצוין את פעולה ה – Accept. תא זה הוא התא אליו נגייע לאחר שסימנו את החוק הרראש בדקדוק במלואו, מה שמצוין תוכנית התקינה.

Goto table

מצינית למתנה לאיזה מצב הוא צריך לעבור לאחר ביצוע פעולה ה – Reduce.

בטבלת ה – Goto כל שורה תציג מצב, וכל עמודה תציג משתנה (Non-terminal) שנמצא בת לחבר השפה. נגייע לתא בטבלה על ידי המצב הנוכחי והמשתנה (Non-terminal) אשר נמצא בראש המחסנית. כל תא בטבלה יגיד לנו לאיזה מצב עליינו לעבור בהתאם למצב הנוכחי ולמשתנה שנמצא בראש המחסנית לאחר ביצוע פעולה ה – Shift או ה – Reduce אשר ה – Action table אמרה לנו לעשות.

דוגמה ל – Action table

State	Action Table						Goto Table		
	id	+	*	()	\$	E	T	F
0	S5			S4			1	2	3
1		S6				ACC			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

מחסנית

המחסנית תחזיק בתוכה זוגות של Terminal או Non-terminal ומספר מצב. בראש המחסנית תמיד יימצא מספר המצב הבא אליו צריך לעבור. המחסנית תאותחל עם איבר אחד שהוא מספר המצב הראשון, 0.

עבור כל פעולה Shift נדחוף למחסנית שני איברים: האחד הוא – Terminal הנוכחי מקוד המקור, והשני הוא מספר המצב כפי שצוין ב – Action table.

עבור כל פעולה Reduce נוציה מהמחסנית את כמות האיברים שנמצאת ב – RHS של חוק הדקדוק שצויין ליד פעולה – Reduce בטבלת ה – Action כפול 2 (משום שעבור כל איבר שנדחוף למחסנית אנו גם דוחפים את מספר המצב אליו צריכים לעבור), ונדחוף למחסנית את ה – Non-terminal שנמצא ב – LHS של אותו חוק דקדוק. כעת כל שנוצר הוא לדוחף את מספר המצב הבא לראש המחסנית. ניתן לדעת מספר זה על ידי התא בטבלת ה – Goto שנמצא בעמודה של ה – RHS של חוק הדקדוק שכעת דוחנו, ובשורה של מספר המצב שנמצא תחתו במחסנית.

כך תראה המחסנית בהתאם

0									
---	--	--	--	--	--	--	--	--	--

לאחר שתי פעולות Shift 3 – Terminal שהוא a

0	a	3	a	3					
---	---	---	---	---	--	--	--	--	--

לאחר פעולה Reduce עבר חוק הדקדוק aa → A

0	A								
---	---	--	--	--	--	--	--	--	--

דחיפת מספר המצביע המתאים על פי טבלת ה – Goto במקומות [0, A] במקומות [0, A]

0	A	4							
---	---	---	--	--	--	--	--	--	--

עץ הניתוח

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

בנייה העץ תעשה תוך כדי תהליכי הניתוח, כך שכל פעם שנבצע את פעולה ה – Reduce, נוסיף לעץ את ה – Non-terminal שב – LHS של חוק הדקדוק שעל פיו ביצענו את פעולה ה – Reduce, ונקבע שהבנים שלו יהיו כל ה – Terminal ו – Non-terminal שב – RHS של חוק הדקדוק, אשר נמצאים מהמחסנית.

בונים אלו שהוצאנו מהמחסנית גם הם עצמם עצים, וכך בעצם אנו אט אט בונים את העץ בתוך המחסנית, וכך שבסוף תהליכי הניתוח התקין ישאר לנו רק איבר אחד במחסנית והוא עץ הניתוח המלא של התוכנית שלנו. כל צומת בעץ תכיל את הטיפוס שלו, ורשימה של הבנים שלו.

אלגוריתם הניתוח

cutet אציג את הפסאודו קוד עבור אלגוריתם הניתוח בעזרת המחסנית, ה – Action table ו – Goto table.

פסאודו קוד

1. התחל

2. אתחל את המחסנית ואת הטבלאות Action & Goto

3. הכנס למשתנה a את האסימון הראשון מקוד המוקור

4. כל עוד לא עברת על כל קוד המוקור **בצע**:

4.1. הכנס למשתנה s את המצב שנמצא בראש המחסנית

4.2. אם $a = \text{Shift } t$, **בצע**:

4.2.1. דחוף את a למחסנית

4.2.2. דחוף את t למחסנית

4.2.3. הכנס למשתנה a את האסימון הבא מקוד המוקור

4.3. אחרת אם $a = \text{Reduce}[s, a]$, **בצע**:

4.3.1. קבע את A – Non-terminal שנמצא ב – LHS של s

4.3.2. הוציא $2^{\ell}(u)$ Length(Production rule) איברים ממחסנית

4.3.3. קבע את כל חלקי ה – RHS של s Production rule שהוצאה מהמחסנית כבנים של A

4.3.4. הכנס למשתנה v את המצב הנמצא כתע בראש המחסנית

4.3.5. דחוף את A למחסנית

4.3.6. דחוף את $[v, A]$ Goto למחסנית

4.4. אחרת אם $a = \text{Accept}[s, a]$, **בצע**:

4.4.1. סיום והחזיר את עץ הניתוח שנמצא בראש המחסנית

4.5. אחרת, **בצע**:

4.5.1. דוח על השגיאה שנמצאה

4.5.2. תקן את המחסנית אם יש צורך בכך

4.5.3. הכנס למשתנה a את האסימון הבא מקוד המוקור שיאפשר המשך קומפילציה

5. סיום

מימוש LR Parser

cutet יש לנו את התאוריה והאלגוריתם לניתוח באמצעות המחסנית ושתי הטבלאות. אך על מנת להשתמש בטבלאות אלה נוצרו לבנות אותן קודם. כיצד נבנה שתי טבלאות אלה? נראה כעת.

נתן לחלק את מימוש ה – LR Parser לשני חלקים על מנת לפשט את ההסבר:

1. יירה של דיאגרמת המעברים, בעצם האוטומט, הגרף, של תחבר השפה.

2. יירה של ה – Parsing table, הבניה משתי טבלאות: Goto table ו – Action table בעזרת דיאגרמת המעברים.

The Dot notation

Notation באמצעות נקופה הנמצאת בין חלקים שונים של rule, המציין כמה מאותו rule כבר ראינו. הנקודה, Dot, יכולה להופיע בכל מקום בחולק הימני (RHS) של rule.

$A \rightarrow \cdot XYZ$ $A \rightarrow X \cdot YZ$ $A \rightarrow XY \cdot Z$ $A \rightarrow XYZ \cdot$	$A \rightarrow XYZ$
--	---------------------

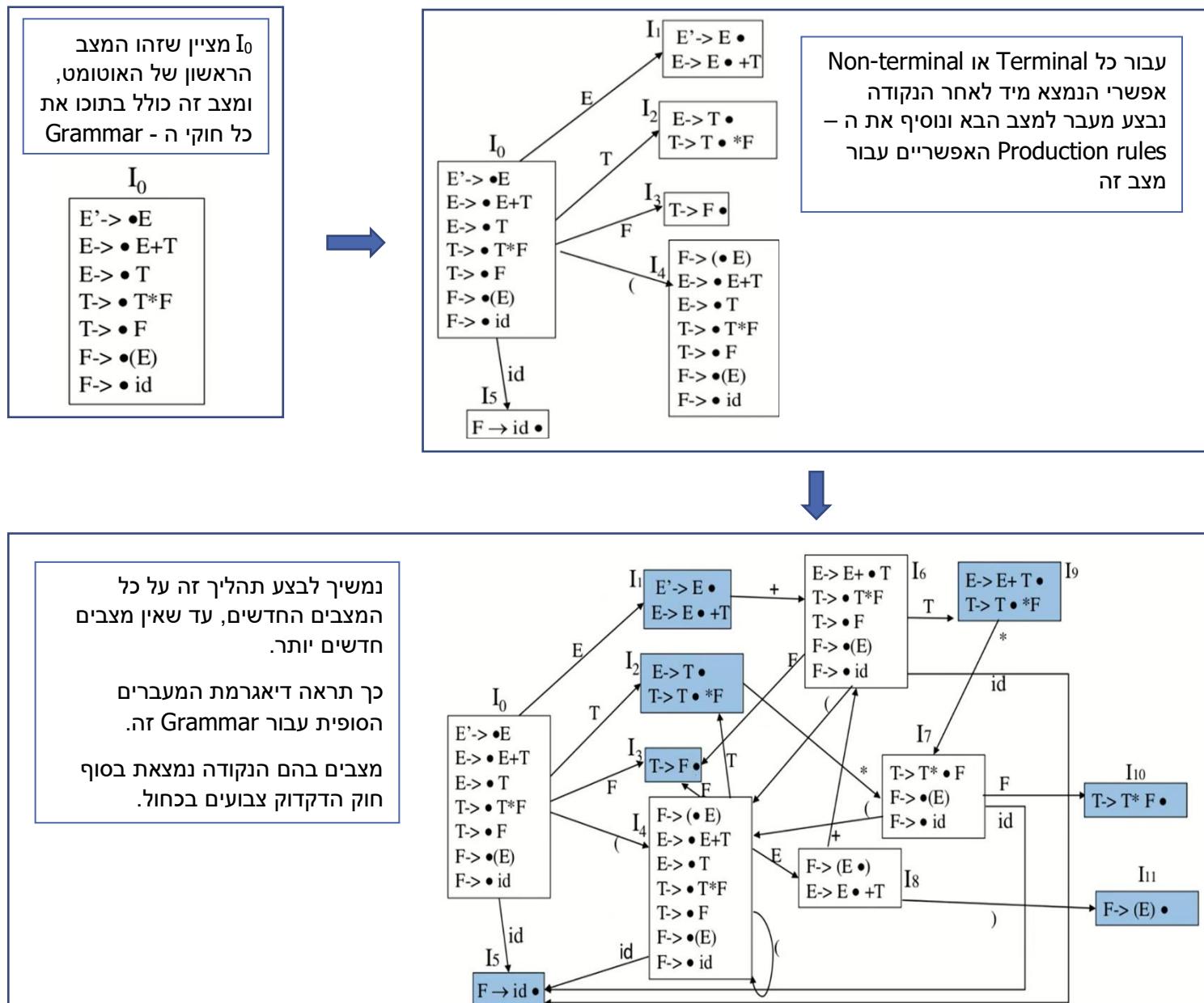
לדוגמא, חוק הדקדוק נתן לנו 4 אופציות:

בנין דיאגרמת המעברים
כפי שציינו לעיל, השלב הראשון בימוש LR Parser הוא ייצור דיאגרמת המעברים עבור תחביר השפה. כתת אספיר
כיצד בשלב זה קורה בעקבות שימוש ב – Dot notation.

ניח את ה - Grammar הבא

$E' \rightarrow \bullet E$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T^* F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$

כך נראה תהליכי הבנייה של דיאגרמת המעברים עבור Grammar זה



כך מתואר תהליכי הבניה של הדיאגרמה

אנו מתחילה מה – Grammar המקורי, ועבור כל סימן שמוופיע מיד אחרי הנקודה אנו עוברים למצב המתאים. חוזרים על פעולה זו עבור כל המצבים החדשים עד שלא מתווספים יותר מצבים חדשים. כך, כאשר יש לנו דיאגרמה זו, אנו יכולים לדעת מהם כל המצבים באוטומט שלו, איזה חוקים אפשריים בכל מצב, ועבור איזה סמל עוברים לאיזה מצב.

בנייה הטבלאות

השלב הקודם נתן לנו את דיאגרמת המעברים, האוטומט, של ה – LR Parser. כעת נרצה ליצור מדיאגרמה זו את שתי הטבלאות שאיתן בפועל נבצע את תהליכי הניתוח התchapiri.

מעבר על קשותות - Shift
 בשיביל לבנות את שתי הטבלאות תחילת נעבור על כל הקשותות, המעברים, בדיאגרמה מהשלב הקודם. עבור קשת עם משקל שהוא Non-terminal נעדכן מעבר זה בטבלת ה – Goto table –, עבור קשת עם משקל שהוא Terminal נעדכן בטבלת ה – Action table וצמוד אליה מספר המצב אליו צריך לעבור לאחר ביצוע פעולה ה – Shift.

כך יראו הטבלאות לאחר שעברנו על כל הקשותות שבדיagramma מהשלב הקודם

State	Action Table						Goto Table		
	id	+	*	()	\$	E	T	F
0	S5			S4			1	2	3
1		S6				ACC			
2			S7						
3									
4	S5			S4			8	2	3
5									
6	S5			S4			9	3	
7	S5			S4					10
8		S6			S11				
9			S7						
10									
11									

מעבר על מצבים בעלי הנקודה בסוף ה – Production rule בדיאגרמה מהשלב הקודם, המצביעים שהגענו לסופו של Production rule ושניתן לבצע Reduce (מסומנים בכחול בדיאגרמה). זאת מטרה לסמן את כל פעולות ה – Reduce set First set – First set. אך על מנת לעשות זאת נוצרת להכיר עוד שני מושגים נוספים: First set ו – Follow set. בטבלת ה – Action.

First & Follow sets

חלק חשוב מהטהlixir של ייצרת ה – Parsing table הוא היצירה של First & Follow sets. אלו יכולים לספק לנו את המיקום המדויק של כל Terminal ב – Derivation. זאת על מנת להקל על קבלת החלטה על פעולה ה – Reduce ב – Parsing table.

בקרצה, First set הם כל ה – Terminals שיכולים להופיע ישירות לאחר שעושים לו – Non-terminal זהה או אחר, ו – Follow set הם כל ה – Terminals שיכולים להופיע מיד לאחר Non-terminal זהה או אחר.

אנו נתמקד ונותנשמש בעיקר ב – Follow set של Non-terminal, נקרא לו A, יהיה כל ה – Terminals שיכולים להופיע מיד לאחר A בכל ה – Production rule בהם A מופיע. גם אם יש Non-terminals בין לבין, אנו מתעלמים מהם וממשיכים לבצע Follow set עד שנגיע ל – Terminals Derivations להיות ישר לאחר A.

לדוגמא, כך נראה ה – Grammar עבור ה – Follow set מהשלבים הקודמים

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T^* F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow id$

$$\begin{aligned} \text{Follow}(E) &= \{ \$, +,) \} \\ \text{Follow}(T) &= \{ *, +,), \$ \} \\ \text{Follow}(F) &= \{ *, +,), \$ \} \end{aligned}$$

אלגוריתם לחישוב Follow set

- אם A הוא המצב ההתחלתי, אז $\text{Follow}(A) \subseteq \$$
- אם A הוא Non-terminal ויש לו Production כדוגמת $A \rightarrow S$, אז $\text{Follow}(S) \subseteq \text{Follow}(A)$
- אם A הוא Non-terminal ויש לו Production כדוגמת $A \rightarrow AB$, אז $\text{Follow}(B) \subseteq \text{Follow}(A)$ למעט ϵ
- אם A הוא Non-terminal ויש לו Production כדוגמת $A \rightarrow ABC$, אז $\text{Follow}(C) \subseteq \text{Follow}(A)$ למעט ϵ

icut שamo יודעים לחשב את ה – Follow, נוכל להמשיך במילוי טבלת ה – Action בפעולות ה – Reduce.

עבור כל מצב בו הנקודה נמצאת בסוף ה – Production rule (מצביעים אלו מסומנים בכחול בדיאגרמה מהשלב הקודם), נסמן בכל העמודות שבשורה של המצב הנוכחי שנמצאות ב – Follow set של המצב הנוכחי, (R) Reduce ועל פי איזה כל עשיינו (בעזרת המספרים שמסומנים בסוגרים ב – Grammar הקבע בכתום שלעיל).

כפי שמוסבר לעיל, אנו מבצעים את פעולה ה – Reduce המתאימה רק בעמודות הנמצאות ב – Follow set של המצב הנוכחי. כלומר, אנו בעצם "מסתכלים קדימה" אסימון אחד, על מנת לקבוע את הפעולה הנוכחיית. הסתכלות קדימה זו הופכת את ה – Parser LR ל – (1) SLR, כלומר אנו מסתכלים קדימה אסימון אחד בכל שלב.

טבלה סופית
כך יראו הtablאות לאחר סיום שלב זה. אלו הtablאות הסופיות אותן נבצע את תהליכי הניתוח התחבירי

State	Action Table						Goto Table		
	id	+	*	()	\$	E	T	F
0	S5			S4			1	2	3
1		S6				ACC			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

דוגמא לניתוח בעזרת ה – Parsing table –
עבור **עboro \$ * id * id \$** (\$ מציין סוף המחרוזת) כך יראה תהליכי הניתוח בעזרת הtablה שלעיל ואלגוריתם הניתוח שהוציא קודם

Stack	Input	Action
0	id * id \$	S5
0 id 5	* Id \$	F->id
0 F	* id \$	
0 F 3	* id \$	T->F
0 T 2	* id \$	S7
0 T 2 * 7	id \$	S5
0 T 2 * 7 id 5	\$	
0 T 2 * 7 id 5	\$	F->id
0 T 2 * 7 F 10	\$	T->T*F
0 T 2	\$	E->T
0 E 1	\$	ACC

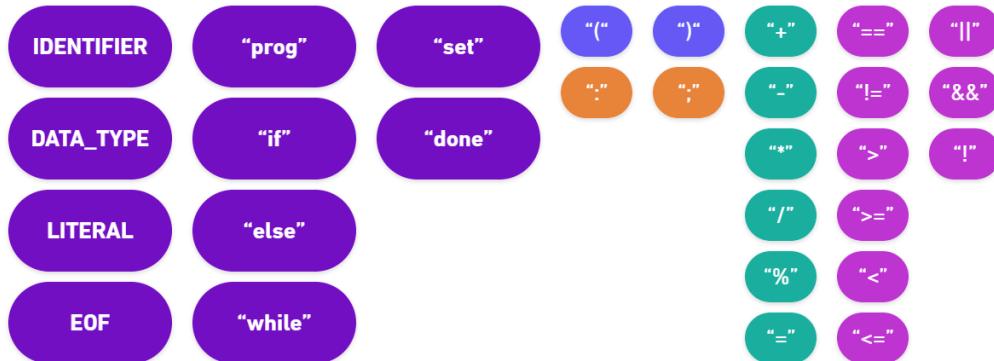
ניתוח השפה Do

כעת אציג כיצד תארויה שהציגו לעיל באה ידי ביטוי בניוthon השפה של Do.

Tokens

אלן הם האטימונים של השפה Do. אלו הם בעצם אבני הבנייה, ה – **Terminals**, של תחביר השפה. כפי שציינתי קודם, אלה מגיעים לשלב זה מהשלב הקודם, **Lexical analysis**.

Tokens



Grammar

כעת אציג את תחביר השפה Do בשתי תצורות, האחת היא BNF Notation, והשנייה היא Railroad Diagram, דיאגרמה ויזואלית לתיאור חוקים של שפה פורמלית. שתי התצורות אמנים נראות שונות, אך זהות לחłówין במשמעותן.

BNF

```

# PROGRAM
<program> ::= "prog" <IDENTIFIER> ":" <block> ":")

# BLOCK
<block> ::= <stmt> <block> | "done"

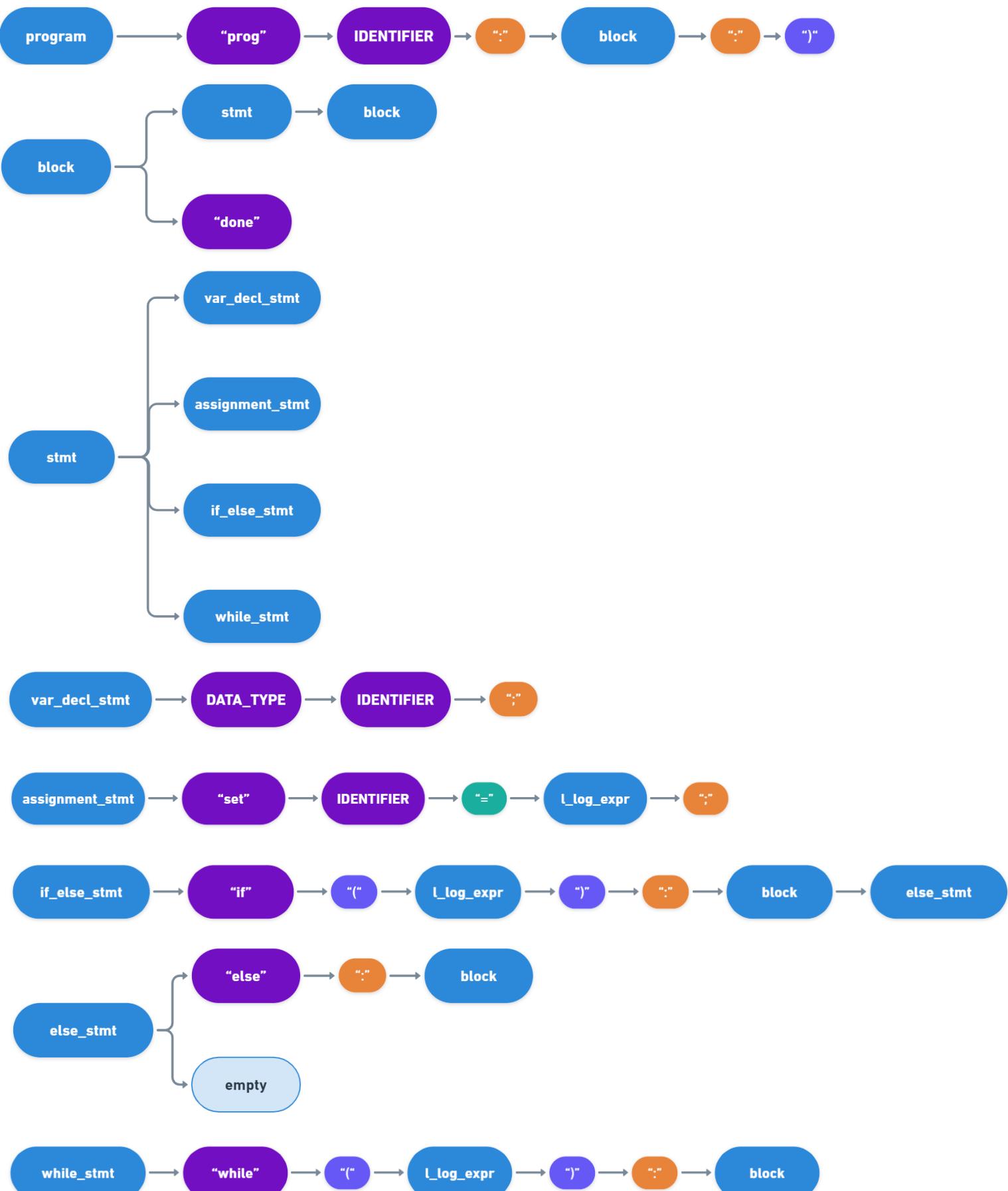
# STATEMENTS
<stmt> ::= <var_decl_stmt> | <assignment_stmt> | <if_else_stmt> | <while_stmt>
## variable declaration
<var_decl_stmt> ::= <DATA_TYPE> <IDENTIFIER> ";"
## assignment
<assignment_stmt> ::= "set" <IDENTIFIER> "=" <l_log_expr> ";"
## if else
<if_else_stmt> ::= "if" "(" <l_log_expr> ")" ":" <block> <else_stmt>
<else_stmt> ::= "else" ":" <block> | <EMPTY>
## while
<while_stmt> ::= "while" "(" <l_log_expr> ")" ":" <block>

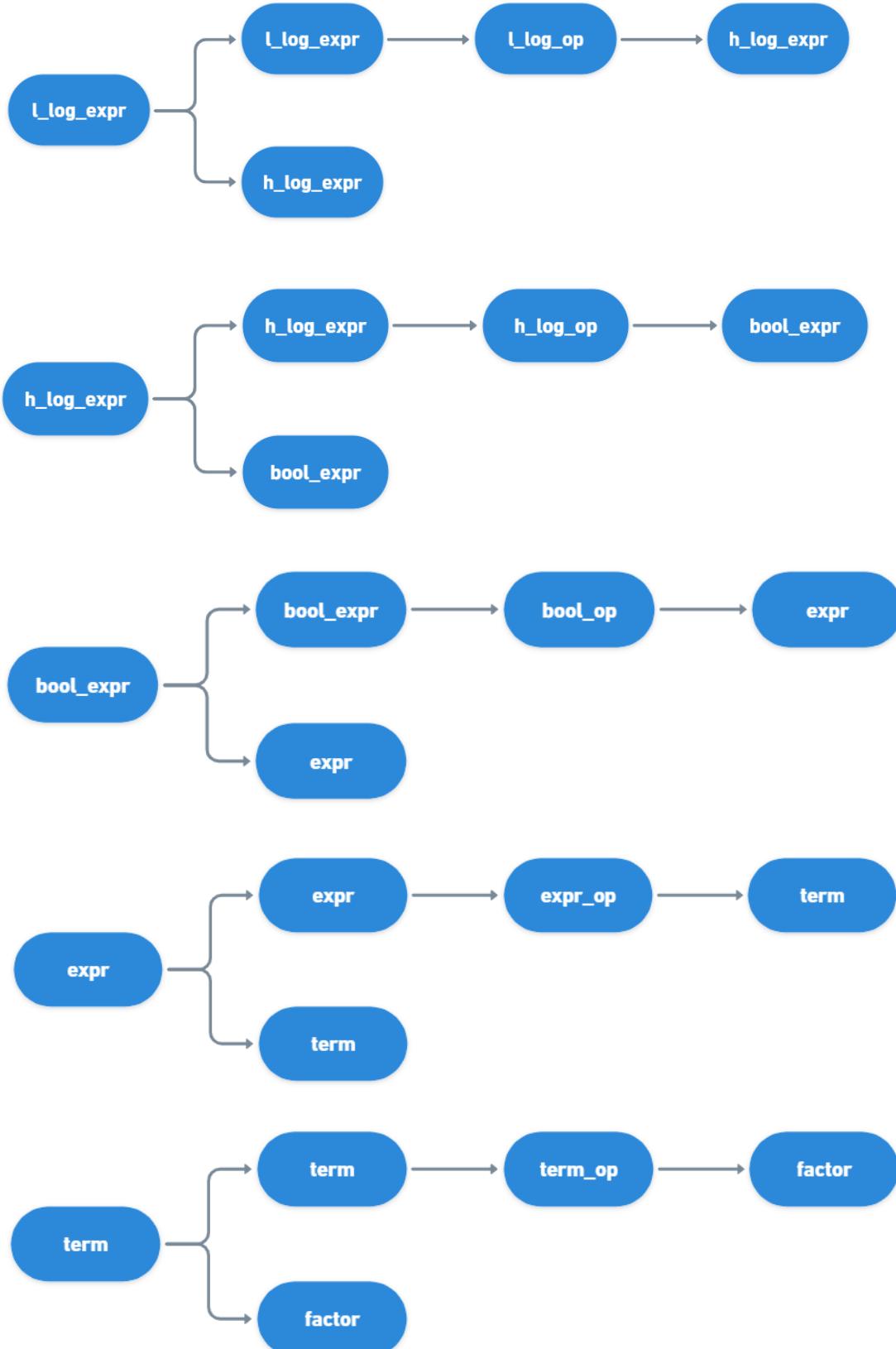
# EXPRESSIONS
<l_log_expr> ::= <h_log_expr> | <l_log_expr> <l_log_op> <h_log_expr>
<h_log_expr> ::= <bool_expr> | <h_log_expr> <h_log_op> <bool_expr>
<bool_expr> ::= <expr> | <bool_expr> <bool_op> <expr>
<expr> ::= <term> | <expr> <expr_op> <term>
<term> ::= <factor> | <term> <term_op> <factor>
<factor> ::= <IDENTIFIER> | <LITERAL> | "(" <l_log_expr> ")" | "!" <factor> | "-" <factor>

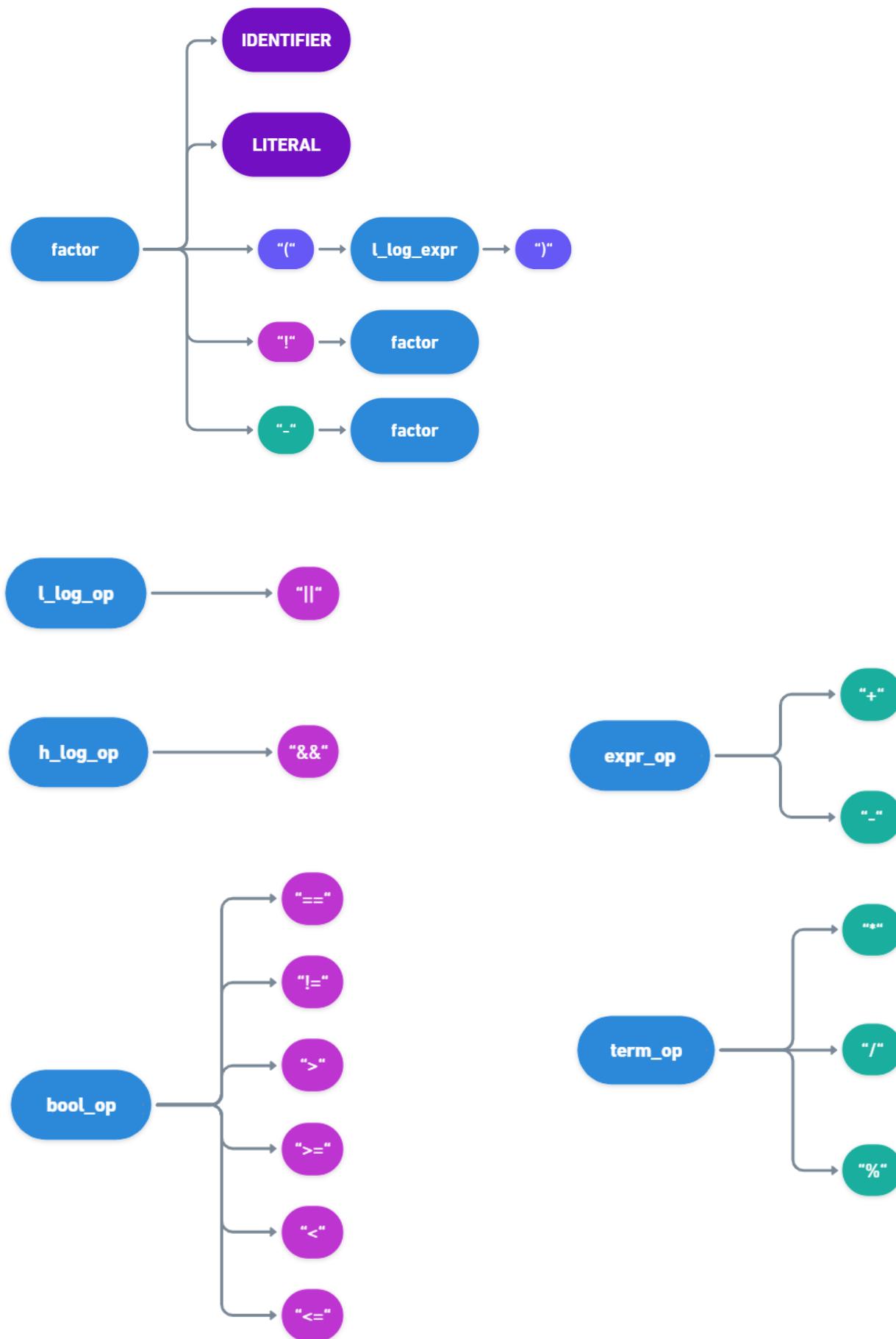
<l_log_op> ::= "||"
<h_log_op> ::= "&&"
<bool_op> ::= "==" | "!=" | ">" | ">=" | "<" | "<="
<expr_op> ::= "+" | "-"
<term_op> ::= "*" | "/" | "%"

```

Railroad Diagram







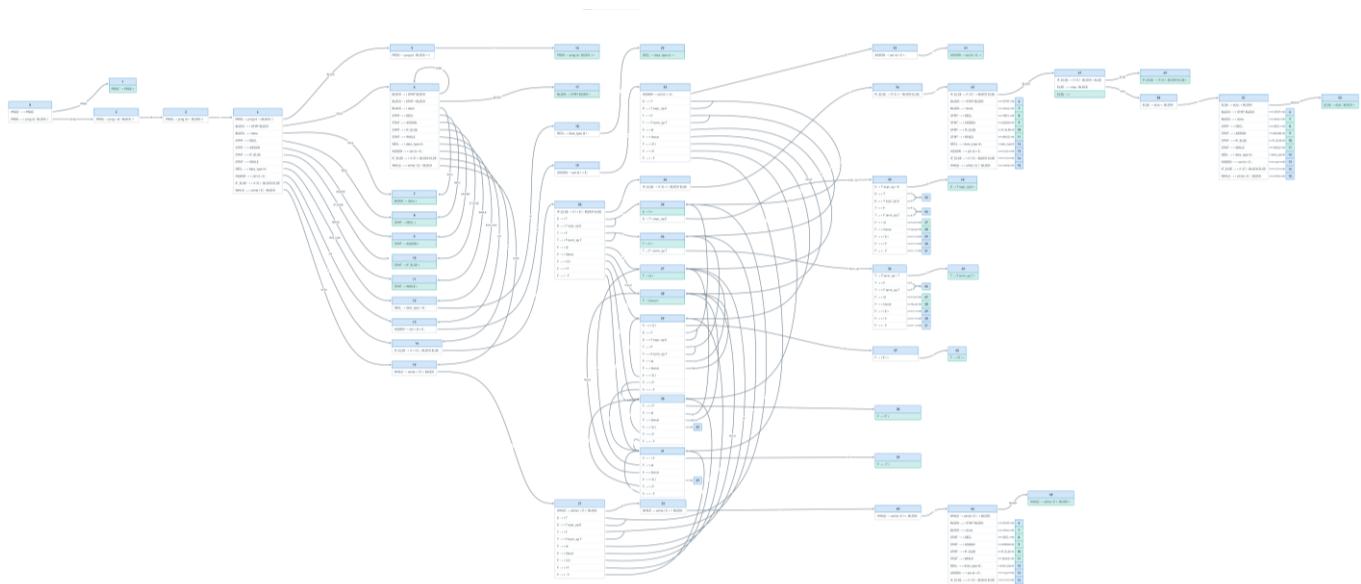
דיאגרמת המעברים

כך נראה דיאגרמת המעברים, האוטומט, הגרפי, עבור Grammar השפה שהציגו לעיל, כשלב הראשון לייצרת טבלת ה – Action וטבלת ה – Goto בהן המנתה המילוני שלנו ישמש.

ה – Action יחד עם מספרי החוקים

GRAMMAR	
1	PROG' → PROG
2	PROG → prog id : BLOCK ;
3	BLOCK → STMT BLOCK
4	BLOCK → done
5	STMT → DECL
6	STMT → ASSIGN
7	STMT → IF_ELSE
8	STMT → WHILE
9	DECL → data_type id :
10	ASSIGN → set id = E ;
11	IF_ELSE → if (E) : BLOCK ELSE
12	ELSE → else : BLOCK
13	ELSE → empty
14	WHILE → while (E) : BLOCK
15	L_LOG_E → L_LOG_E l_log_op H_LOG_E
16	L_LOG_E → H_LOG_E
17	H_LOG_E → H_LOG_E h_log_op BOOL_E
18	H_LOG_E → BOOL_E
19	BOOL_E → BOOL_E bool_op E
20	BOOL_E → E
21	E → T expr_op E
22	E → T
23	T → F term_op T
24	T → F
25	F → id
26	F → literal
27	F → (L_LOG_E)
28	F → ! F
29	F → - F

הダイגרמה



זהו דיאגרמה ענקית ובתמונה קטנה זו הפרטים הרבים בה לא ברורים. צירפתי תמונה זו רק לצורך לקבל מבט על של הדיאגרמה. הדיאגרמה המלאה תצורף כנספה.

Action & Goto

cut כישיש בידינו את דיאגרמת המעברים של תחביר השפה, נתרגמה לטבלאות ה – Action & Goto.

Action

State	Action																	
	prog	id	:)	done	int	:	char	:	set	=	if	()	else	while		&&
0	S2																	
1		S2																
2			S3															
3				S4														
4					S7													
5						S7												
6							S12											
7								S12										
8									S12									
9										S13								
10											S14							
11												S14						
12												S14						
13													S15					
14														R2				
15														R2				
16														R3				
17														R4				
18														R5				
19														R6				
20															S21			
21																S20		
22																	S21	
23																		S22
24																		S23
25																		S32
26																		S32
27																		S32
28																		S32
29																		S32
30																		S32
31																		S32
32																		S32
33																		S32
34																		S32
35																		S32
36																		S32
37																		S32
38																		S32
39																		S32
40																		S32
41																		S32
42																		S32
43																		S32
44																		S32
45																		S32
46																		S32
47																		S32
48																		S32
49																		S32
50																		S32
51																		S32
52																		S32
53																		S32
54																		S32
55																		S32
56																		S32
57																		S32
58																		S32
59																		S32
60																		S32
61																		S32

<code>==</code>	<code>!=</code>	<code>></code>	<code>>=</code>	<code><</code>	<code><=</code>	<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>number</code>	<code>character</code>	<code>!</code>	<code>eof</code>
19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
R0														accept
S40	S40	S40	S40	S40	S40	S41	S41	S42	S42	S42	S42	S31	S31	S33
R18	R18	R18	R18	R18	R18	R20	R20	R22	R22	R22	R22	S31	S31	S33
R20	R20	R20	R20	R20	R20	R22	R22	R23	R23	R23	R23	S31	S31	S33
R22	R22	R22	R22	R22	R22	R23	R23	R24	R24	R24	R24	S31	S31	S33
R23	R23	R23	R23	R23	R23	R24	R24	R24	R24	R24	R24	S31	S31	S33
R24	R24	R24	R24	R24	R24	S34	S34	S34	S34	S34	S34	S31	S31	S33
R26	R26	R26	R26	R26	R26	S34	S34	S34	S34	S34	S34	S31	S31	S33
R27	R27	R27	R27	R27	R27	R27	R27	R27	R27	R27	R27	S31	S31	S33
R26	R26	R26	R26	R26	R26	R26	R26	R26	R26	R26	R26	S31	S31	S33
R27	R27	R27	R27	R27	R27	R27	R27	R27	R27	R27	R27	S31	S31	S33
R17	R17	R17	R17	R17	R17	S41	S41	S42	S42	S42	S42	S31	S31	S33
R19	R19	R19	R19	R19	R19	R21	R21	R21	R21	R21	R21	S31	S31	S33
R21	R21	R21	R21	R21	R21	R25	R25	R25	R25	R25	R25	S31	S31	S33
R25	R25	R25	R25	R25	R25	R25	R25	R25	R25	R25	R25	S31	S31	S33

Goto

Goto table														
PROG	BLOCK	STMT	DECL	ASSIGN	IF_ELSE	WHILE	L_LOG_E	ELSE	H_LOG_E	BOOL_E	E	T	F	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	
1														
	5	6	8	9	10	11								
	17	6	8	9	10	11								
							24		25	26	27	28	29	
							35		25	26	27	28	29	
							36		25	26	27	28	29	
								43		25	26	27	28	
										29				
										44				
										45				
									49	26	27	28	29	
									50	27	28	29		
									51	28	29			
										52	29			
											53			
	56	6	8	9	10	11								
	57	6	8	9	10	11			58					
		61	6	8	9	10	11							

ניתוח סמנטי – Semantic Analysis

מטרתם של השלבים הקודמים הייתה בדיקה של תקינות השפה מבחינה מילונית, Lexer, ובדיקה של תקינות השפה מבחינה תחבירית, Parser. כעת כל שנותר לוודא הוא שהתוכנית אכן תקינה גם מבחינה סמנטית.

לעומת ה – Grammar בשלב הקודם שהוא חופשי הקשר (CFG) ובדק את תקינות תבנית התוכנית, הבדיקה הסמנטית היא תלויות הקשר (Context-Sensitive) ובודקת את שימושות התוכנית. לעומת זאת, תקינותו של משפט בשפה מלאה גם במקרים שלו ולא רק במבנה שלו.

העץ שנוצר על ידי ה – Parser לרוב אינו שימושי לקומpileר בצורתו הפשוטה, זאת משום שהוא אינו מחזיק בערכיהם המציגים איך להעריך את העץ, וכך השפה (כיוון שהיא Context-free) אינה מציינית כיצד לפרש אותו.

דוגמה

$T \rightarrow E + E$

כלל היירה שלעיל אין שום שימושות סמנטית המשוויכת לו, והוא לא יכול לעזור לנו **להבין** את ה – Production.

סמנטיקה

נחזיר בקצרה על מהי סמנטיקה של שפה. הסמנטיקה של השפה היא בעצם הפעולה של לתת שימושות לבניה שלה, כדוגמת ה – Tokens והתחביר שלה. סמנטיקה עוזרת לפרש סמלים (Symbols), את הסוגים שלהם, והקשרים שלהם אחד עם השני.

ניתוח סמנטי קובע האם המבנה התחבירי של התוכנית בעל שימושות או לא.

$CFG + semantic\ rules = Syntax\ Directed\ Definitions$

דוגמה

$; int a = "value";$

שורט הקוד שלעיל לא אמורה ליצור שגיאה בשלב הניתוח המילוני, כמו גם בשלב הניתוח התחבירי, אך מבחינה סמנטית היא לא נכונה (לא ניתן להשים מחוץ למשתנה מסוג int), ואמורה לגרום לשגיאה סמנטית.

מטרתו של המנתח הסמנטי

כעת אתן תזכורת קצרה על הדברים העיקריים אותם נרצה לוודא על מנת לבדוק האם התוכנית שלנו נכונה מבחינה סמנטית.

בדיקות נכונות סוגים משתנים – Type Checking

בדיקת נכונות סוגים משתנים, נקרא גם Type checking, קובעת כי נרצה לוודא משתנה מסווג (Type) מסוים, אכן מקבל את הערך המתאים לו בפועל השמה.

לדוגמא, משתנה מסווג זה יכול לקבל רק ערך מסווג מספרשלם, ולא שום סוג אחר אחר.

כמו כן, Type checking קובע כי ב – Expression ישתתפו רק סוגים מסוימים של משתנים וליטרלים, ורק שילובים אפשריים שלהם.

לדוגמא, בשפת C ה – $"hello" + 5$ אינו תקין, והוא אמור לגרום לשגיאת קומPILEציה בשלב הניתוח הסמנטי.

בדיקות נכונות רצף שימוש במשתנים

עוד בדיקה שנרצה לבצע היא האם יש שימוש במשתנה לפני שהוגדר. זהה בדיקה שלא מתבצעת בשלבים הקודמים של הקומPILEציה, Parser, ונרצה לבצע אותה כעת. בתוכנית תקינה לא ניתן להשתמש במשתנה לפני שהוגדר.

בדיקת ייחודיות שמות המשתנים
שם משתנה לא יכול להיות מוגדר מספר פעמים באותו – Scope. עוד על Scope בהמשך.

אסטרטגיה לפתרון

אז כיצד נבצע את כל זה? כתת אציג את האסטרטגיה לפתרון בעית הניתוח הסמנטי.

Attribute Grammar

Attribute Grammar הוא צורה מיוחדת של Context-free Grammar בו אחד או יותר Non-terminal תכונת נספפת, Attribute, על מנת לספק לנו מידע תליי הקשר, Context-sensitive. לכל תכונה צאת יש תחום מסוים של ערכיים, לדוגמה, מספר, מחורזת, ביטוי ועוד.

כל Attribute היא בעצם צמד של $\langle \text{Attribute name}, \text{Attribute value} \rangle$

Attribute Grammar הוא דרך לספק סמנטיקה ל – CFG, והוא יכול לעזור לנו להגדיר بد בבד את הת לחבר והסמנטיקה של שפה. אם נסתכל על ה – Attribute Grammar עצמו, נראה שהוא מסוגל להעביר מידע בין הצמתים השונים בעץ, מה ש – CFG לבדו לא מסוגל לעשות.

SDT & Semantic Actions

בשביל לתת ולעדכן את ערכי ה – Attributes של הצמתים בעץ, נשתמש בכללים / פעולות סמנטיות, אשר יעדכנו תכונות אלו. הכללים הסמנטיים האלה מסומנים בדרך כלל בצד ימין של rule Production, וקיימים בתוך סוגרים מסולסים.

או Syntax Directed Translation SDT הוא השימוש ב – Attributes ו – Semantic Actionsalo, על מנת לבצע את תהליך Parsing, מה שמייעל את התהילה, ושומר לנו על יעילות זמן ריצה לינארית.

סיכום

בשביל לפתור את בעית הניתוח הסמנטי נסיף לדקדוק השפה שלנו עוד "תכונות", חוקים, אשר צריך לוודא שמתקינים עבור רצף אסימוניים בשבייל לקבוע האם הוא אכן סמנטי.

עבור כל rule Production בת לחבר השפה בו יכולה להיווצר שגיאה סמנטית, נסיף איזושהי פעולה / בדיקה שצריכה להתקיים כאשר משתמשים בו, זאת על מנת לאתר את השגיאה הסמנטית שיכולה להיווצר בעקבות שימוש כלל זה.

דוגמא

$$E \rightarrow E + T \{ E.value = E.value + T.value \}$$

הצד ימני של כלל היצירה (נמצא בתוך סוגרים מסולסים) מצין את הכלל הסמנטי, Semantic Action, הקובע כיצד כל זה צריך להיות מפורש. במקרה זה, סכום הערכיים של E – T, יהיה הערך של E.

בעזרת אסטרטגיה זו נוכל לשמר על יעילות זמן ריצה לינארית, שהוא מעבר ייחיד על העץ שנוצר בהשוואת הפעולות השמורות בדקדוק השפה.

יישום המנתה הסמנטי

cut אציג כיצד נשתמש באסטרטגיה זו על מנת לענות על המטרות השונות, ולממש את המנתה הסמנטי.

בדומה למה שהציגתי לעיל, נוסיף לו – Grammar השפה שלנו את ה – Semantic Actions הדורשים על מנת לוודא שהוא נכון גם מבחינה סמנטית, כמובן, נשימוש ב – SDT. כל פעולה סמנטית זאת בעצם תהיה פונקציה בקומפיאילר שלנו שתבצע את הפעולה הספרטיפית זו, וכל פעם שנבצע אותה פעולת ה – Reduce או ה – Shift ב – Parser, נפעיל פעולה זו. בעצם בתוך ה – Parsing table, נוסיף מצבים לפעולות בהתאם המתאים, וכך יוכל להפעיל כל פעם את הפעולה המתאימה לכל יצירה זהה.

כמו כן, לכל צומת בעץ נוסיף את ה – Attribute הנחוץ לה, השימוש בפעולות הסמנטיות השונות יעדכן אליה מה שיעזר לנו בבדיקה תקינות התוכנית מבחינה סמנטית. כמובן נהפוך את ה – CFG ל – Attributed Grammar. בנוסף, ניעזר בו – Symbol table אשר נבנית במהלך תהליך הקומפיאילציה, עליה ארכחיב עוד בהמשך. cut אציג כיצד תוספות ועדרים אלו עוזרים לנו להשיג את המטרות השונות שהצבנו למנתה הסמנטי.

בדיקות נכונות סוגי משתנים – Type Checking

Type checking אחראי על 2 דברים: השמת ערך מסווג נכון למשתנה, שימוש בערכים מסווגים תקינים בו – Expressions. על מנת לבצע זאת נבין ראשית כי שני דברים אלו תלויים בו – Expressions. לכן, נוסיף להם שיעזרו לנו לוודא זאת.

לכל Non-terminal שלוקח חלק ב – Expression, נוסיף Attribute של Type, כך בכל פעם שנבצע פעולה כלשהי בביטוי מסוים, נבדוק האם הוא – Types מתאים, ואם כן, נעדכן את ה – Type של ה – Non-terminal הבא בהתאם. כמו כן, לאחר שיש בידינו את ה – Type של ה – Expression, נרצה להשם אותו למשתנה. בשביל לעשות זאת, ראשית נצטרך לוודא שסוג המשתנה מתאים לסוג הביטוי שכעת מצאנו. על מנת לבדוק זאת, נשימוש ב – Symbol table. ניגש למשתנה זה ב – Symbol table (בנήאה שהוא קיים ונגיש כموן, עוד על זה בהמשך) ונבדוק מהו סוג שלו. אם סוג זה תואם לסוג של הביטוי שכעת מצאנו, נשים את ערך הביטוי במשתנה, ונמשיך בתהליך הניתוח. אם הסוגים לא תואמים, נדוח על שגיאת חוסר ההתאמה בין סוגים, Type mismatch.

בדיקות נכונות רצף שימוש במשתנים

על מנת לבדוק שלא משתמשים במשנה לפני שהוגדר נשתמש שוב ב – Symbol table. נוסיף כלל סמנטי עבור כל פעם שימושים בשם משתנה הקובע כי צריך לוודא שהוא אכן נמצא ב – Symbol table וכן שהוא בטוחה הגיש לנו. אם בדיקה זו תקינה, נמשיך בתהליך הקומפיאילציה, אם היא לא תקינה נדוח על שגיאת שימוש במשתנה לפני שהוגדר, Undeclared variable.

בדיקות ייחודיות שמות משתנים

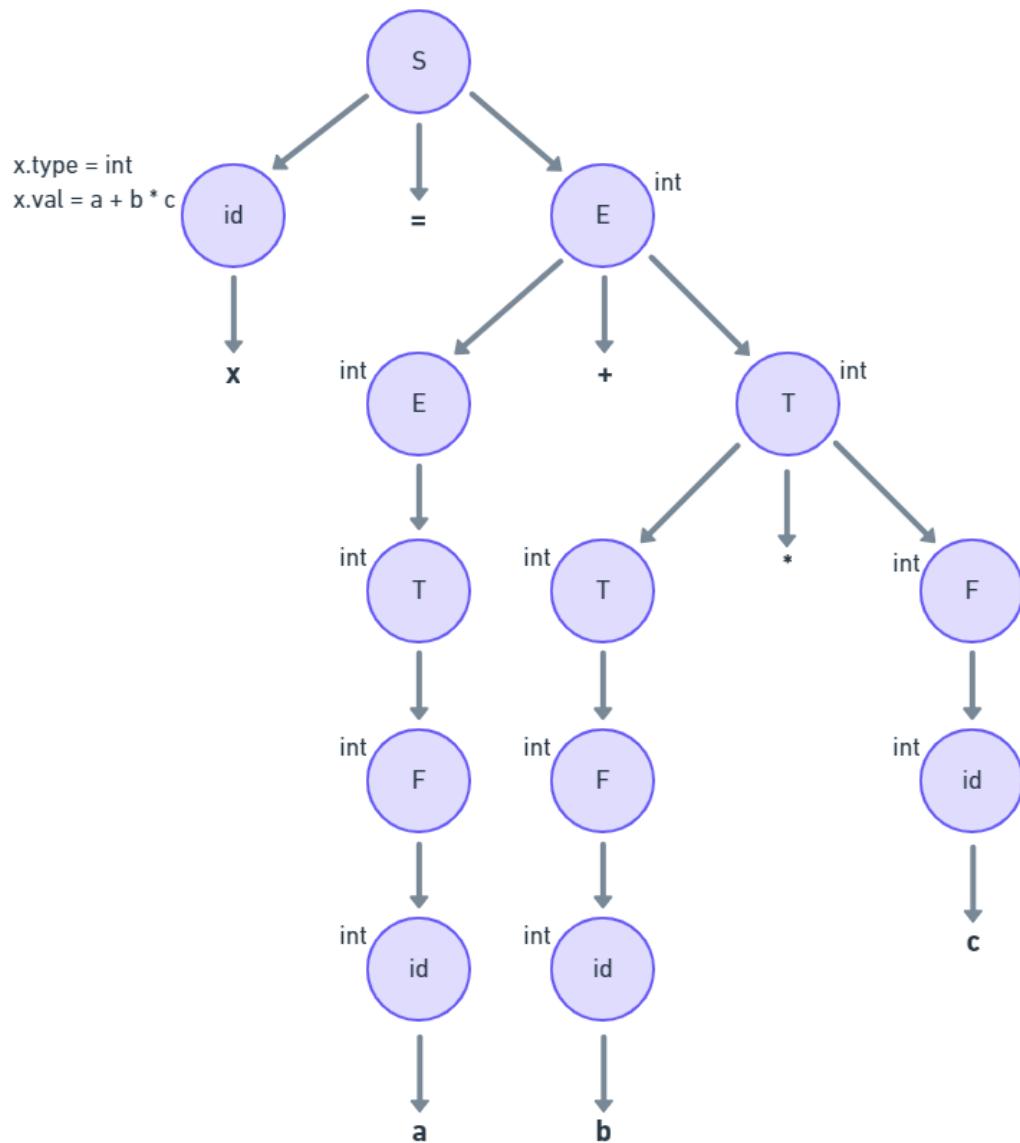
על מנת לבדוק שם משתנה מוגדר אך ורק פעם אחת בכל Scope נשתמש גם כן ב – Symbol table. נוסיף כלל סמנטי עבור כל היצירה של הגדרת משתנה הקובע כי בשבייל שנוכל להגדיר משתנה הוא חייב להיות לא מוגדר באותו – Scope. נפנה לו – Symbol table ונבדוק האם המשתנה הזה מוגדר כבר בו – Scope שלנו. אם לא, נגידר משתנה זה ונמשיך בתהליך הקומפיאילציה, אם כן, נדוח על שגיאת ניסיון הגדרה נוספת של משתנה.

דוגמא
cutet.aczig דוגמא הממחישה שתת כל מה שהציגו לעיל.

Grammar

GRAMMAR	
1	$S \rightarrow id = E \quad \{ \text{if } (\text{look_up}(\text{id}).\text{type} == E.\text{type}) \text{ look_up}(\text{id}).\text{val} = E.\text{val} \}$
2	$E \rightarrow E_1 + T \quad \{ \text{if } (E_1.\text{type} == T.\text{type}) E.\text{type} = T.\text{type} \}$
3	$E \rightarrow T \quad \{ E.\text{type} = T.\text{type} \}$
4	$T \rightarrow T_1 * F \quad \{ \text{if } (T_1.\text{type} == F.\text{type}) T.\text{type} = F.\text{type} \}$
5	$T \rightarrow F \quad \{ T.\text{type} = F.\text{type} \}$
6	$F \rightarrow id \quad \{ F.\text{type} = \text{look_up}(\text{id}).\text{type} \}$

מחרוזת קלט: $x = a + b * c$

Attributes – עץ הניטות יחד עם ה-

Code Generation

הגענו לשלב האחרון בתהליך הקומpileציה. בהגענו לשלב זה יש בידינו את ה- AST ו ה- Symbol table המציגים תוכנית תקינה לאחר שעברו את השלבים הקודמים, Parser, Lexer ו Semantic Analyzer.

מטרתו של שלב זה, ובכך גם נגמר תהליך הקומpileציה, היא תרגום ה- AST להוראות בשפת אסמבלי, שיציגו את התוכנית שלנו.

הסטרטגיה בה ננקוט היא שעבור שלב זה נחזיק פונקציות בסיסיות שיעזרו לנו לתרגם לקוד בשפת אסמבלי. על ידי שימוש באסטרטגיה זו של בניית פונקציות עזר ובבנייה פונקציות מחזיקות אשר מquivול למקבילו לכל ערך בעץ הקוד האבסטרקטי שנוצר כתוצאה מה – Front end של הקומpileציה, נשמר על זמן ריצה לנארו.

להלן חלק מהפונקציות:

`get_reg`

תחזיר אוגר פנוי בשלב שצריך לאחסן בו ערך או משתנה.

Scratch Register Management

- `int scratch_alloc();`
- `void scratch_free(int r);`
- `const char * scratch_name(int r);`

r	0	1	2	3	4	5	6
name	%rbx	%r10	%r11	%r12	%r13	%r14	%r15
inuse	X		X				

`symbol_codegen`

תחזיר את המקום של משתנה כל פעם שאנו חנו ניגשים אליו.

Symbol Generation

- `const char * symbol_codegen(struct symbol *s);`

```
a: string = "hello";

f: function void ( x: integer, y: integer ) = {
    z: integer = 10;

    print a; "a"

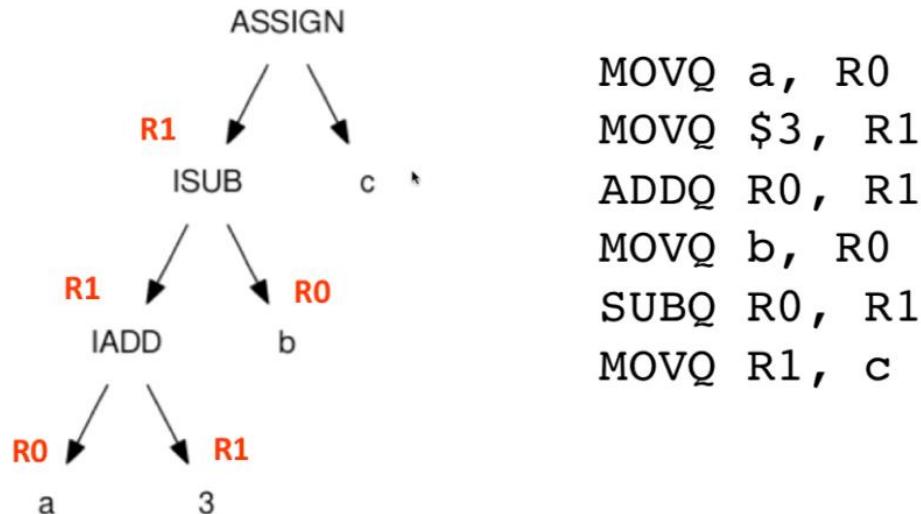
    return x + y + z;
}
```

exp_generator

פונקציה אשר תתרגם קטע קוד המיצג Expression מה – Parse tree לקוד אסמלר.

נושף לכל צומת ב – Expression תכונה המסמלת באיזה אוגר נמצאת התוצאה של תת העץ שלו.

Generating Expressions



Statements

עבור כל Statement נגידר פונקציה אשר מתרגם את כל חלקיו לשפת אסמלר לפי סדר הדקדוק. הפונקציות ישתמשו בפונקציות קודמות.

Generating Statements

```

case STMT_DECL:
    decl_codegen(s->decl);
    break;

case STMT_EXPR:
    expr_codegen(s->expr);
    scratch_free(s->expr->register);
    break;

case STMT_RETURN:
    expr_codegen(s->expr);
    printf("MOV %s, %%rax\n", scratch_name(s->expr->register));
    printf("JMP .%s_epilogue\n", function_name);
    scratch_free(s->expr_register);
    break;
  
```

Conditional statements

עבור קטעי קוד התלויים בתנאי נגדיר מבנה מראש, ונשתמש בפונקציות קודמות בשבייל להשלים חלקים חסרים במבנה.
להלן דוגמא למבנה של תנאי if & else:

Conditional Statements

```
if ( expr ) {
    true-statements
} else {
    false-statements
}
```

expr
 CMP *register*, \$0
 JEQ *false-label*
true-statements
 JMP *done-label*
false-label :
false-statements
done-label :

Conditional expressions

עבור ביטויים לוגיים נגדיר מבנה מראש, נשלים בו את חלקו החסרי בעזרת פונקציות קודמות, ונשווה בהתאם לסימן הנतון.

Conditional Expressions

left-expr < *right-expr*

left-expr
right-expr
 CMP *left-register* *right-register*
 JLT *true-label*
 MOV *false*, *result-register*
 JMP *done-label*
true-label:
 MOV *true*, *result-register*
done-label:

Symbol table

אנו משתמשים ב – **Symbol table** לאורך כל תהליך הקומpileציה. היא עוזרת לנו לשמר מידע ולקבל מידע באופן יעיל על כל משתנה בתוכנית, כמו גם היא עוזרת לנו ב – **Scope management** – עוזרת לנו להבין את התחום (Scope) בו משתנה נגיש או לא נגיש לנו.

כעת אציג כיצד נגרום לו – **Symbol table** לבצע את הדברים שהציגו לעיל.

שמירה ואחזור מידע על משתנים

כפי שציינתי לעיל, טבלת הסמלים עוזרת לנו בשמירה ובאחזור מידע על כל משתנה בתוכנית שלנו, אך איך היא עשוה זאת באופן יעיל?

ניצג את ה – **Symbol table** כ – Dictionary / Hash table. כיוון שככל משתנה בתוכנית חייב להיות בעל שם ייחודי, יוכל למפות כל משתנה למקום ספציפי במילון באמצעות פונקציית Hash.

על ידי שימוש ב – Hash table, יוכל להבטיח ששמירה ואחזור של מידע בסיבוכיות זמן ריצה קבועה, (1)O. כך טבלת הסמלים לא תשפיע על סיבוכיות זמן הריצה שלנו, ובסופה נישאר בזמן ריצה לינארי, (n)O.

Scope management

אז כיצד ניצג את ה – Scope עבור הבלוקים השונים בתוכנית שלנו? כאמור, כיצד נדע איזה משתנה נגיש לנו באיזה שלב בתוכנית?

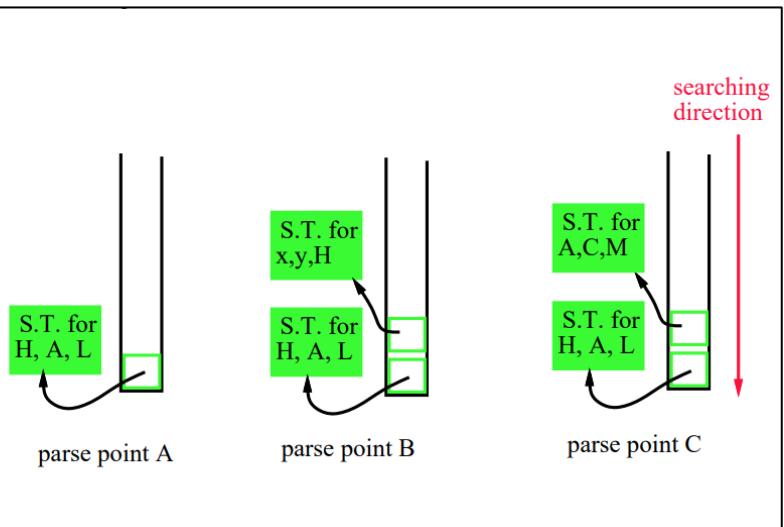
ישן מספר דרכי בהן ניתן למשם Scope:

מחסנית של Symbol tables

נשתמש במחסנית. תהיה לנו מחסנית של Symbol tables כך שכל איבר במחסנית מייצג Scope. האיבר הראשון שנכנס למחסנית הוא ה – Symbol table של ה Scope – הגלובלי של התוכנית, והוא נגיש לכלם. כל פעם שניכנס לבlok חדש נדחוף Scope חדש למחסנית המיצג את הבלוק הנוכחי, וכל פעם שנצא מבלוק נוציא את ה Scope שבראש המחסנית. נדע אם משתנה נמצא ב – Scope הנוכחי שלו אם הוא נמצא בטבלה שנמצאת בראש המחסנית, ונדע אם הוא קיים בכלל אם הוא נמצא בכל המחסנית.

דוגמה

```
main()
{
    /* open a new scope */
    int H,A,L; /* parse point A */
    ...
    { /* open another new scope */
        float x,y,H; /* parse point B */
        ...
        /* x and y can only be used here */
        /* H used here is float */
        ...
    } /* close an old scope */
    ...
    /* H used here is integer */
    ...
    { char A,C,M; /* parse point C */
    }
}
```



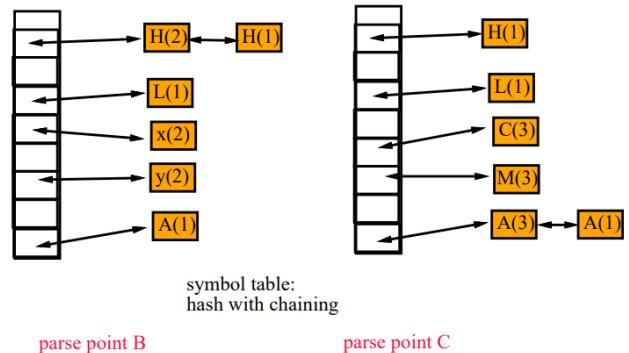
טבלה אחת
נשתמש בטבלה גלובלית אחת, כך שכל Scope יקבל מספר ייחודי, וכל משתנה בטבלה יהיה גם הערך של ה-Scope בו הוא נמצא.

דוגמה

```
{
    /* open a new scope */
    int H,A,L; /* parse point A */

    ...
    { /* open another new scope */
        float x,y,H; /* parse point B */
        ...
        /* x and y can only be used here */
        /* H used here is float */
        ...
    } /* close an old scope */

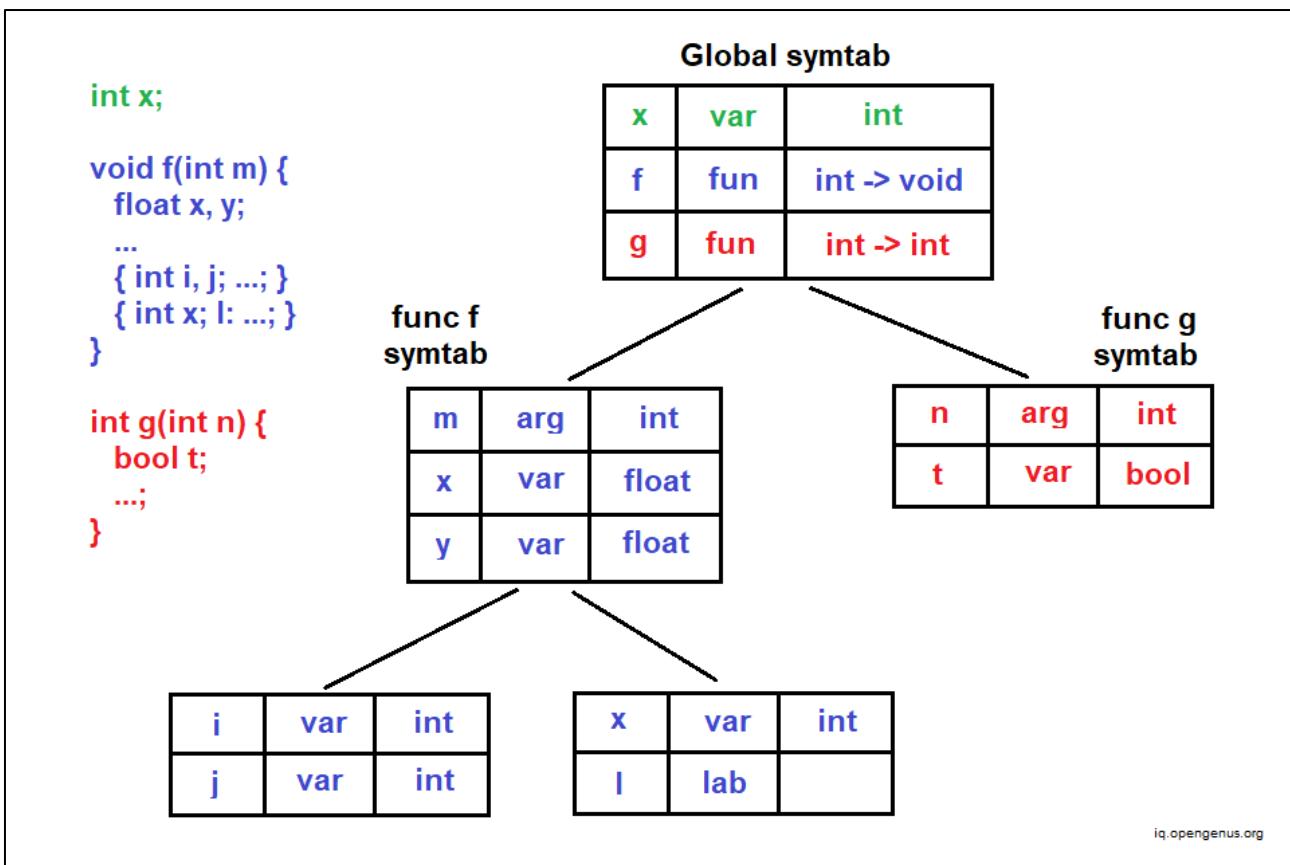
    ...
    /* H used here is integer */
    ...
    { char A,C,M; /* parse point C */
    ...
}
}
```



עץ של Symbol tables

נשתמש בכך. שורש העץ יהיה ה- Global scope של התוכנית, ועבור כל Scope חדש ניצור בן לצומת של ה- Scope הנוכחי. ביציאה מ- Scope נחזיר לאב של ה- Scope הנוכחי. על מנת לבדוק האם משתנה נמצא ב – Scope הנוכחי נבדוק את הטבלה של הצומת הנוכחי בעץ. על מנת לבדוק האם משתנה זמין לנו בתוכנית, נרוץ מהצומת הנוכחי עד השורש ונבדוק בטבלה של כל צומת.

דוגמה



Error Handler

בכל אחד משלבים ב – Front end של תהילך הקומpileציה יכולות להתגלות שגיאות.

- Lexical errors
- Syntax errors
- Semantic errors

אך כיצד נטפל בשגיאה כאשר אנו נתקלים בה? עבור **Semantic errors** ו – **Lexical errors** הטיפול בשגיאה די פשוט, נודיע על שגיאה ונמשיך הלאה. אך עבור **Syntax errors** המשפיעות על מבנה התוכנית שלנו, הטיפול בשגיאה קצר יותר מסובך.

Syntax errors recovery strategies

cutת אציג את הדרכים השונות בהן ניתן לטפל ב – **Syntax errors**.

Exit on Error

גישה זו היא הגישה הפשוטה ביותר לטיפול בשגיאות. כאשר נתקל בשגיאה, נודיע על כך ונסימן את תהליך הקומpileציה. מבחינת כתוב הקוד גישה זו הינה לימיוש ומונעת בעיות היכלות להיווצר אם נמשיך את תהליך הקומpileציה עם שגיאה, אמנם מבחינת כתוב הקוד בשפה גישה זו לא מאד נוחה, הקומpileר כל פעם יתריע למשתמש רק על השגיאה הראשונה שמצא, וכך אם יש הרבה שגיאות נctrar לתקן אותן אחת ולקملל שוב ושוב עד שהקומpileציה תעבור כמו שצורך ללא שגיאות.

Panic mode

כאשר נתקל בשגיאה, נשאר במצב הנוכחי ונתקדם בקוד המקורי עד אשר נמצא התאמה שלא תגרום לשגיאה, ככלומר נתעלם מחלקיק הקוד הבאים עד אשר יגיע חלקיק קוד שמתאים לנו. אמנם בגישה זו תהליך הקומpileציה לא מסתיים בהתקלויות בשגיאה הראשונה, אך גישה זו מדלגת על חלקיק קוד רבים עד אשר תמצא התאמה, או שלא תמצא בכלל, וכך יכולה להת%">
פנספס בדיקה של חלק גדול מהקוד, ובדומה לו – **Exit on Error** נדרש לתקן את השגיאות אחת אחת.

גישה זו אמנם פשוטה רעיונית ומעשית, אך עבור שגיאות קלות בקוד היא עובדת מעולה ובכורה יעה.

Statement mode

כאשר נתקל בשגיאה, נודיע על כך, וננסה לתקן את שאר הקלט כך שתהליך הקומpileציה יוכל להמשיך בהצלחה. לדוגמה, נסיף ; במקומות אחרים, נוריד ; במקומות שיש יותר מדי וכו'. צריך להיות זהירות בגישה זו, משום שתיקון אחד לא יוכל להוביל לlolאה איןסופית.

גישה זו מסובכת יותר למימוש האחריות מכיוון שכותב הקומpileר צריך לחשב על כל מקרי הקצה, ולזוזה שאינו יוצר lolאה איןסופית.

ישנן גישות נוספות למימוש האחריות מכיוון Global correction ו – Error production אך גישות אלו מסובכות יותר, ומורכבות מדי בשבייל שייה שווה לממשן בפרויקט זה.

Do Error recovery במשפט

על מנת לטפל בשגיאות בחורתית לשלב בין שתי גישות: Statement mode ו – Panic mode.

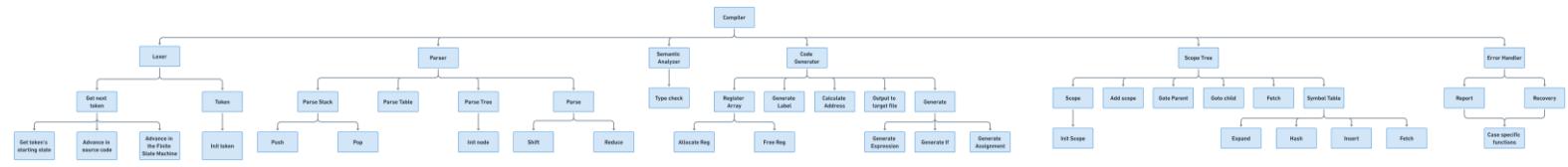
ברגע שנתקל בשגיאה, ה – Parser יודיעו הודעת שגיאה אינפורטטיבית עליה, ויתקדם עד אשר יגיע לאסימון הבא שהוא תחילת של Statement (int, char, set, if, while).

ברגע שהגיע לתחילת Statement מוציא את כל האסימונים שננדפו למחסנית עברו ה – Statement הנוכחי שארם לשגיאה, אם יש צורך דוחף למחסנית את האסימונים שחסרים עברו המצב הנוכחי, עברו למצב אשר אפשר המשך קומpileציה, וממשיך בתהיליך הקומpileציה כאילו הכל היה כרגע.

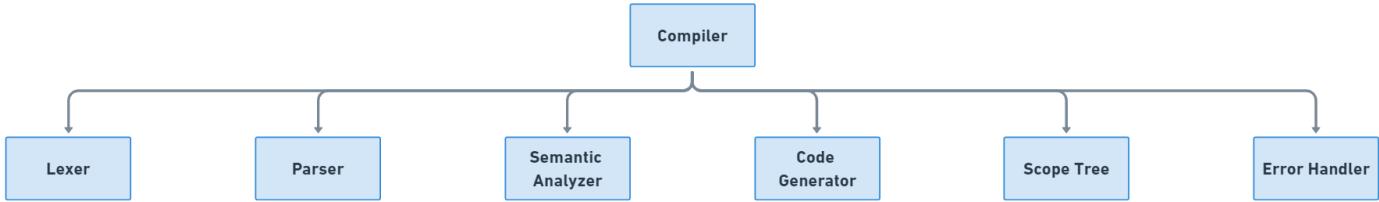
על ידי שימוש זה אני לא מدلג על חלקים גדולים מהקוד, אני מונע לולאות אינסופיות, ובנוסף אני מבטיח כמה שפחות זיהויות שלא קיימות, "שגיאות נגררות", מה שנגרם לרוב על ידי שימוש ב – Panic mode.

Top-Down Level Design

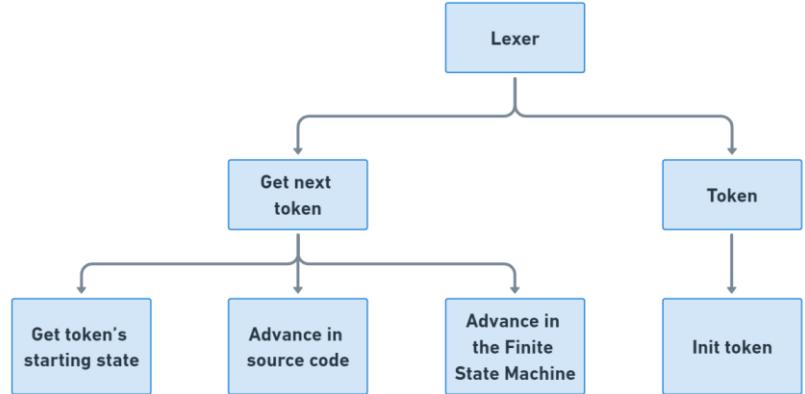
מבט על



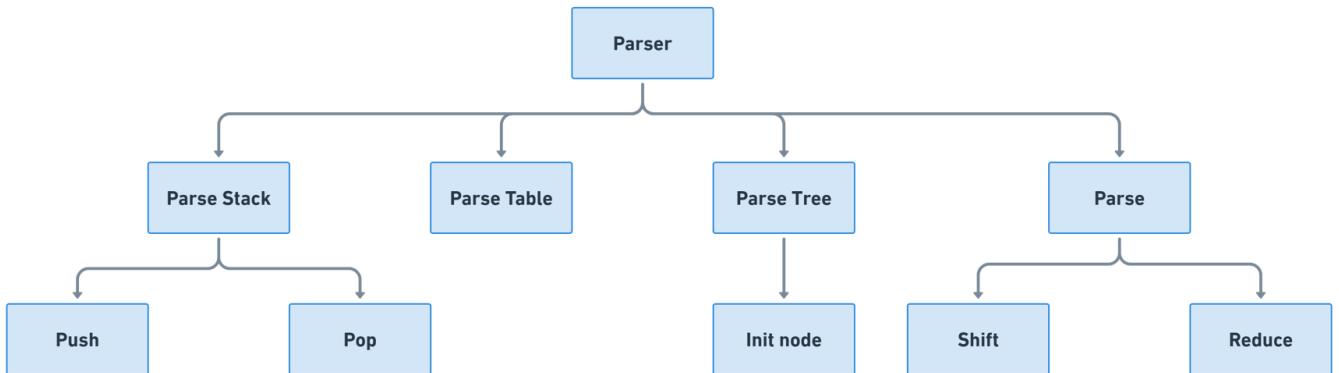
Compiler



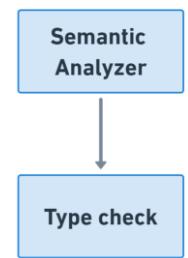
Lexer



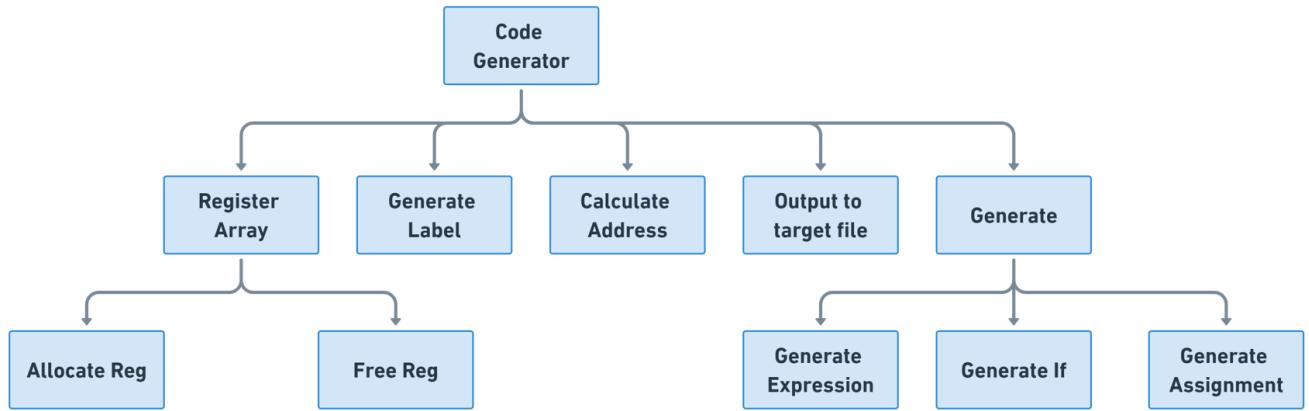
Parser



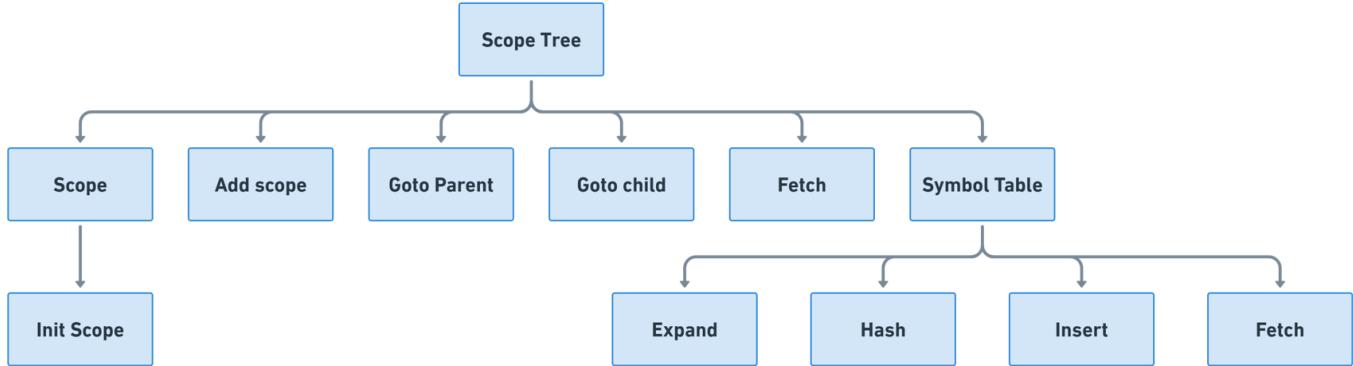
Semantic Analyzer



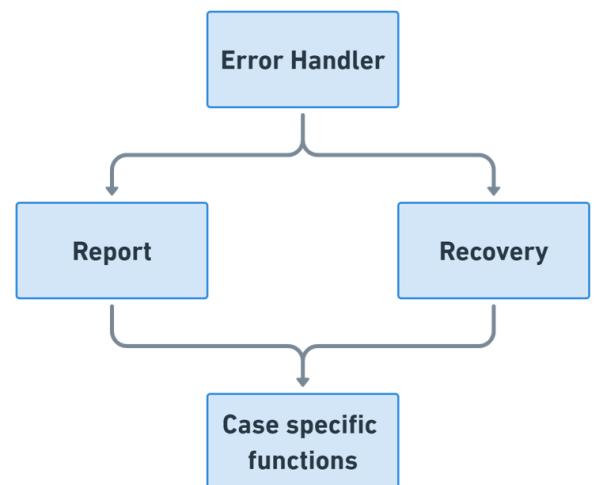
Code Generator



Scope Tree



Error Handler



מבנה נתונים

בפרק זה אציג את מבני הנתונים בהם השתמשתי בכל שלבים השונים של הקומpileר.

מנתח מילוני – Lexical Analysis

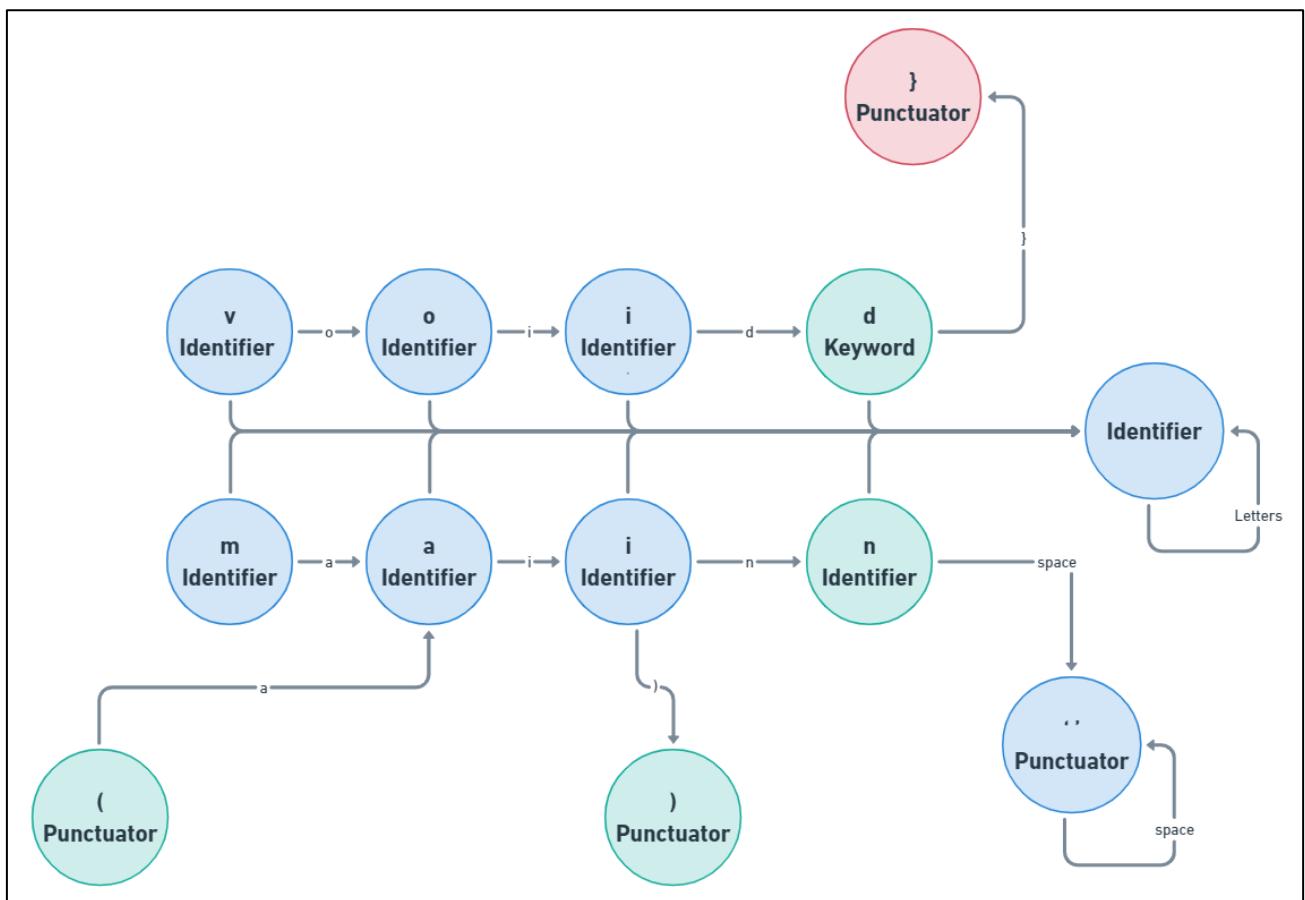
גרף

עבור המנתח המילוני השתמשתי במבנה מצבי סופית המיוצגת על ידי גרף.

משקל הקשרות בgraf הן אוטיות, כך שעבור כל אות ומצב הנוכחי נ עבור למצב ספציפי אחר (אוטומט דטרמיניסטי). כאשר הגיע למצב בו אין לאן להתקדם עבור התו הבא מהקלט, סימן שהגענו למצב סופי, ונחזיר את ה – Token השמור במצב זה. עבורתו לא צפוי להגיע למצב שגיאה.

מבנה נתונים זה בעצם מאפשר לנו לשמר על ייעילות זמן ריצה לינארית, ($O(n)$), משום שאין צורך לשאול שום שאלות, או לבדוק מספר אופציות עבור כל מצב. כל הממצאים האפשריים נמצאים בתוך מבנה הנתונים וכך בכל מצב אנחנו יכולים להזכיר איזה פעולה עליינו לעשות.

תרשים



מציאת מצב התחלתי של Token

נשمر מערך בגודל כל התווים השייכים לשפה ובכל תא את המצב ההתחלתי בו צריך להתחיל לקבל את ה – Token שמתחל בטו זהה.

כרי בסיבוכיות זמן קבועה, (1)O, נוכל לדעת באיזה מצב צריך להתחיל על מנת לקבל את ה – Token המתחילה בכל תו שהוא בשפה.

תרשים

Character	'0'	''	';'	'{'	'}'	...
Starting State	0	2	7	3	...	

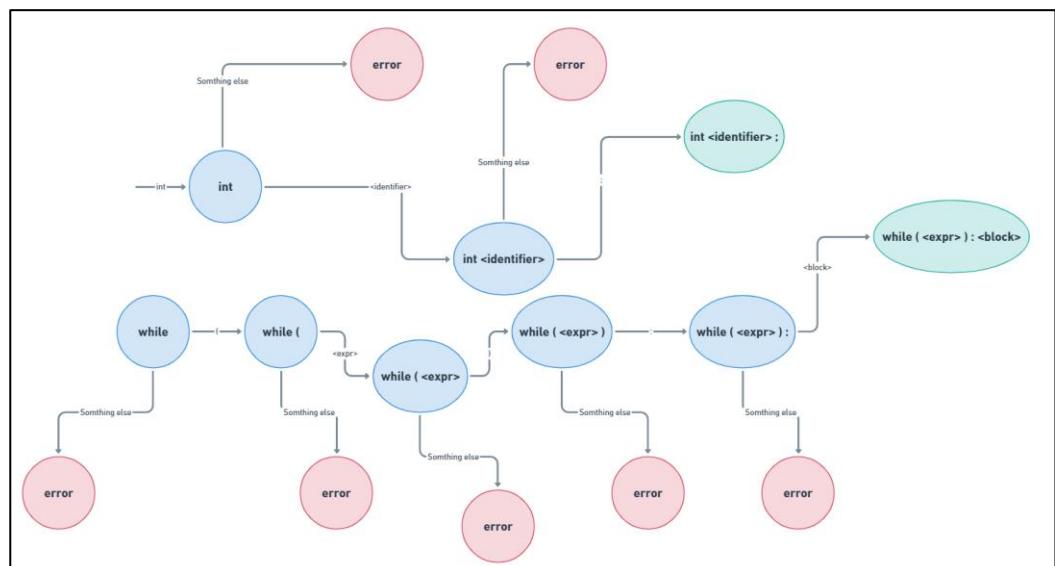
מנתח תחבירי – Syntax Analysis

גרף

עבור המנתח התחבירי השתמשתי באוטומט מוחסנית המיוצג על ידי גרף. כל צומת בgraf מייצג מצב נכון, מיקום, בחוק בתחביר השפה, והקשות בgraf היו Tokens שקיבלו מהמנתך המקורי. עבור מצב נכון, זה – Token הבא מהקלטណ עוזה עליו לעשות בניתוח מטה מעלה (Shift, Reduce, Accept, Error). עבור Token לא צפוי מהקלט נגיע למצב שגיאה.

מבנה נתונים זה מאפשר לנו לקבל החלטה עבור כל מצב אפשרי במהלך הקומpileציה. אנו לא צריכים לשאול שום שאלות, או לבדוק מספר אופציות, מה שעזר לנו לשמור על יעילות זמן ריצה לינארית, (ח)Ο.

תרשים



Parse Table

על מנת למשוך את הגרף שמייצג את אוטומט המוחסנית השתמשתי בשתי טבלאות, Action & Goto tables, אשר יחד נקראות Parse table. אלו בעצם שתי מטריות כך שבטבלת ה – Action כל תא הוא הפעולה שצריך לעשות עבור המצב והאיסימן הנוכחי, ובטבלת ה – Goto כל תא הוא מספר המצב אליו צריך לעבור עבור המצב הנוכחי והוא – Parsing table & Stack נתון למצוא ב – Goto ו – Action ב – Stack.

גישה לתאים השונים במטריות תבצע ביעילות אלגוריתמית קבועה, (1)Ο, ובכך תעזר לנו לשמור על יעילות זמן ריצה לינארית, (ח)Ο, של הקומpileר.

תרשים

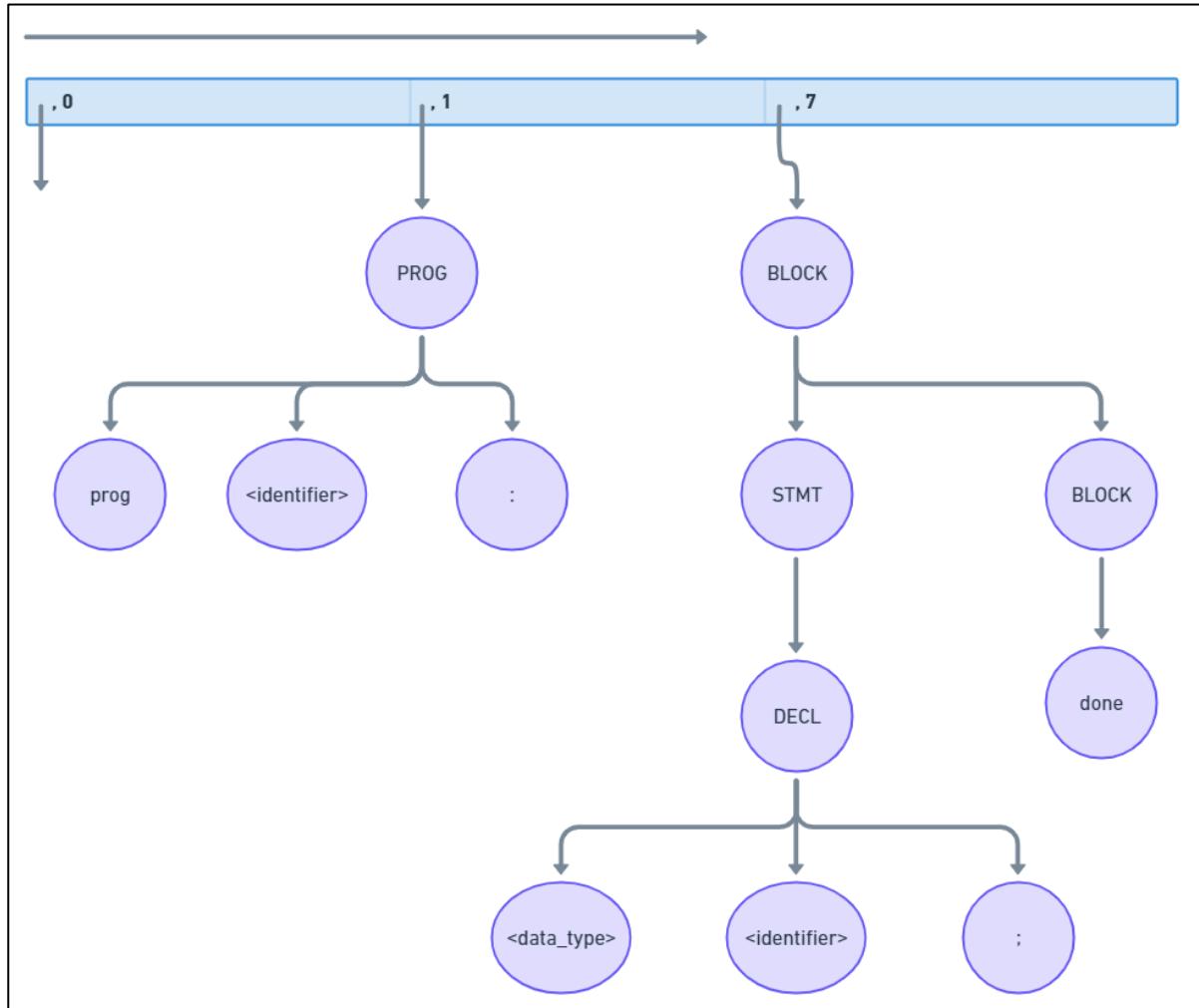
State	Action Table						Goto Table		
	id	+	*	()	\$			
0	S5			S4			1	2	3
1		S6				ACC			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Parse Stack

עבור אוטומט המחסנית אנו כמובן צרכיים ממחסנית. עבור המחסנית של האוטומט השתמשתי במחסנית בה כל איבר מכיל את שורש העץ שנבנה עד כה בתא זה במחסנית, ובנוסף כל איבר במחסנית שומר לאיזה מצב צריך לצלת לאחר מכן. פירוט רב יותר ניתן לקרוא ב - **מחסנית בפרק האסטרטגיה**.

המחסנית מאפשרת לנו "לזכור" את הדברים שראינו כך שבהמשך נוכל להוציא אותם מהמחסנית ולהשתמש בהם במקום לחפשם שוב. בכך היא עוזרת לשמר על ייעילות זמן ריצה לינארית, ($O(n)$, של הקומפילר).

תרשים



דקדוק השפה – Production Rules

על מנת לשמר את כלוי היצירה של השפה המשמש במערך המכיל עבור כל כלל יצירה את ה – LHS Non-terminal שלו, ואת מספר הסמלים שיש ב – RHS של כלל היצירה.

כך נוכל לדעת בסיבוכיות זמן ריצה קבועה, ($O(1)$), כמה איברים צריך להוציא מהמחסנית בכל Reduce, ובנוסף לכך נוכל לדעת איזה Non-terminal צריך להיות בשורש העץ לאחר פעולה הה – Reduce.

תרשים

Rule No.	0	1	...
LHS Non-terminal, No. of symbols on RHS	PROG, 5	BLOCK, 2	...

עץ הניתוח – Parse Tree

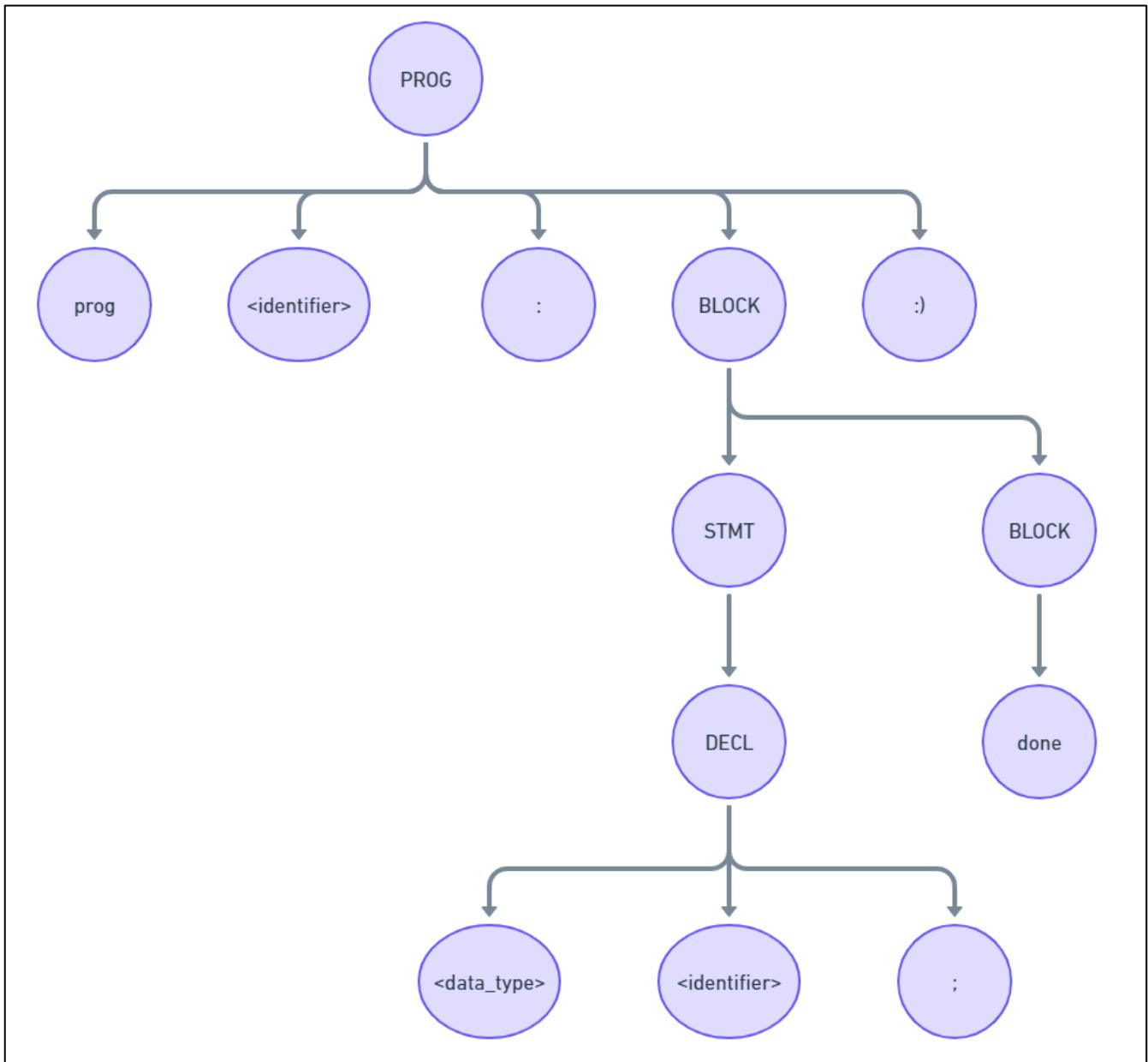
נרצה שהמנתח התחבירי יחזיר לנו עץ מייצג את התוכנית.

כל צומת בעץ יכול Terminal או Non-terminal.

לכל צומת המכיל Non-terminal, יהיה מערך של הבנים שלו, וכל עלה, צומת המכיל Token, יחזיק Token מוקוד המקורי. העץ נבנה אט כתתי עיצים בתוך ה Parse Stack – והוא גבנה כך שאמן עברו עליו בסדרה תוכית, Inorder traversal, קיבל את תוכנית המקור.

העץ מייצג את מבנה התוכנית, ובעצם מתוך המבנה שלו אנו יכולים לקבל החלטות עבור כל מקום בעץ. דבר זה עוזר לנו לשמר על ייעילות אלגוריתמית לינארית, (O(n)), של הקומפיאלה.

תרשים

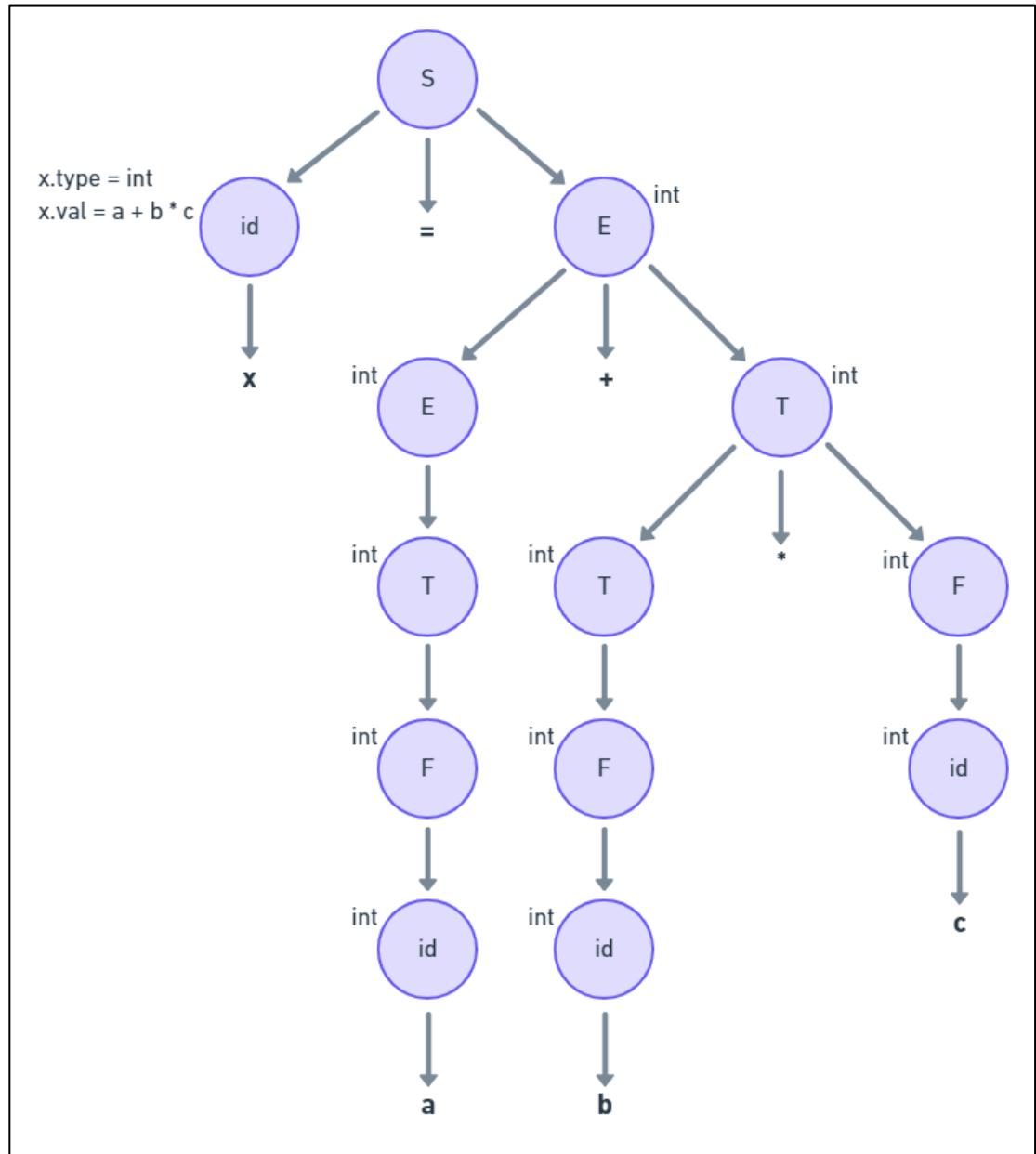


מנתח סמנטי – Attributes

עבור המנתה הסמנטי נוספת לפחות בעץ הניתוח Attribute Shioif לנו סמנטיקה נוספת למבנה של התוכנית. כך יוכל למצוא ולנתה שגיאות סמנטיות אשר לא נתפסות בשלבים הקודמים של הקומpileציה.

התכונות שלעיל נסופות לעזטור כדי תחיליך הניטוח ביעילות זמן ריצה קבועה, (1)O, ובכך לא פוגעת ביעילות זמן הריצה הלינארית, (ח)O, של הקומפלייר.

תרשימים



מצביעים לפונקציות סמנטיות

לכל תא ב Parse table אשר יכול להויצר בו שגיאה סמנטית, נוסיף מצביע לפונקציה המטפלת בשגיאה סמנטית זו.

כך תוך כדי תהליכי הניתוח והתחביבו אנו מבצעים את תהליכי הניתוח הסמנטי, והוא מתבצע ביעילות זמן ריצה קבועה, (1) שיטה זו מייצרת את פועלתו של הקומפיאר.

Code Generation

מערך רגיסטרים

במהלך ה – **Code generation** אנו משתמשים ברגיסטרים על מנת לתרגם את החישובים השונים לשפת אסמבלי.
בשביל לדעת איזה רגיסטרים בשימוש ואייזה זמינים לנו ניצור מערך המכיל את שם הרגיסטר והאם פניו או לא פניו. כך נוכל
בעילות זמן קבועה, (1)O, לקבל רגיסטר פניו עבור החישוב הנוכחי.

תרשים

r	0	1	2	3	4	5	6
name	%rbx	%r10	%r11	%r12	%r13	%r14	%r15
inuse	X		X				

Register Attribute

עבור חישוב של **Expression** אנו משתמשים ברגיסטרים.

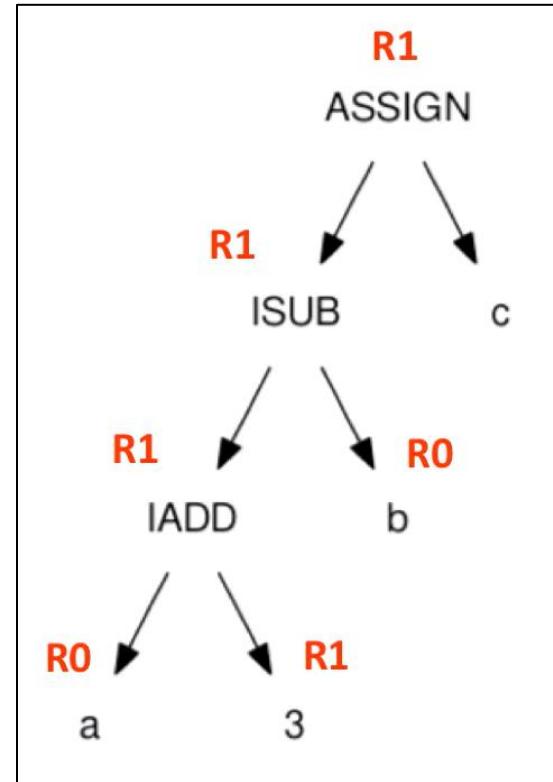
בשביל לדעת איזה רגיסטר מחזיק תוצאה של איזה תת-עץ בביטוי, נוסיף לכל צומת בעץ שהוא חלק מ – **Expression**.
תכונה של מספר הרגיסטר שמחזק את התוצאה שלו.

כך, עבור צומת נכון בעץ נוכל לקרוא רקורסיבית לחישוב תת-עץ שלו, ובחרזה מהרקורסיה נוכל לדעת באיזה רגיסטר
נשמר כל אחד מתתי העצים, ומוכן לבצע את החישוב שהצומת הנוכחי אמרור לבצע.

עבור פעולות "הוראות" נשימוש שוב באחד מן הרגיסטרים של הבנים. זאת על מנת שנוכל לשחרר רגיסטרים, ולהשתמש
בهم שוב.

העברת המידע על הרגיסטרים מתבצעת תוך כדי המעבר על עץ הניתוך ביעילות זמן ריצה קבועה, (1)O, ובכך לא פוגעת
ביעילות האלגוריתמית הלינארית, (n)O, של הקומפיאילר.

תרשים



Symbol Table Hash table

מטרהה של טבלת הסמלים היא שמיירה ואוחזור של מידע אודות המשתנים השונים בתוכנית. על מנת שתבוצע את העבודה בצורה היעילה ביותר, משתמש ב – Hash table.

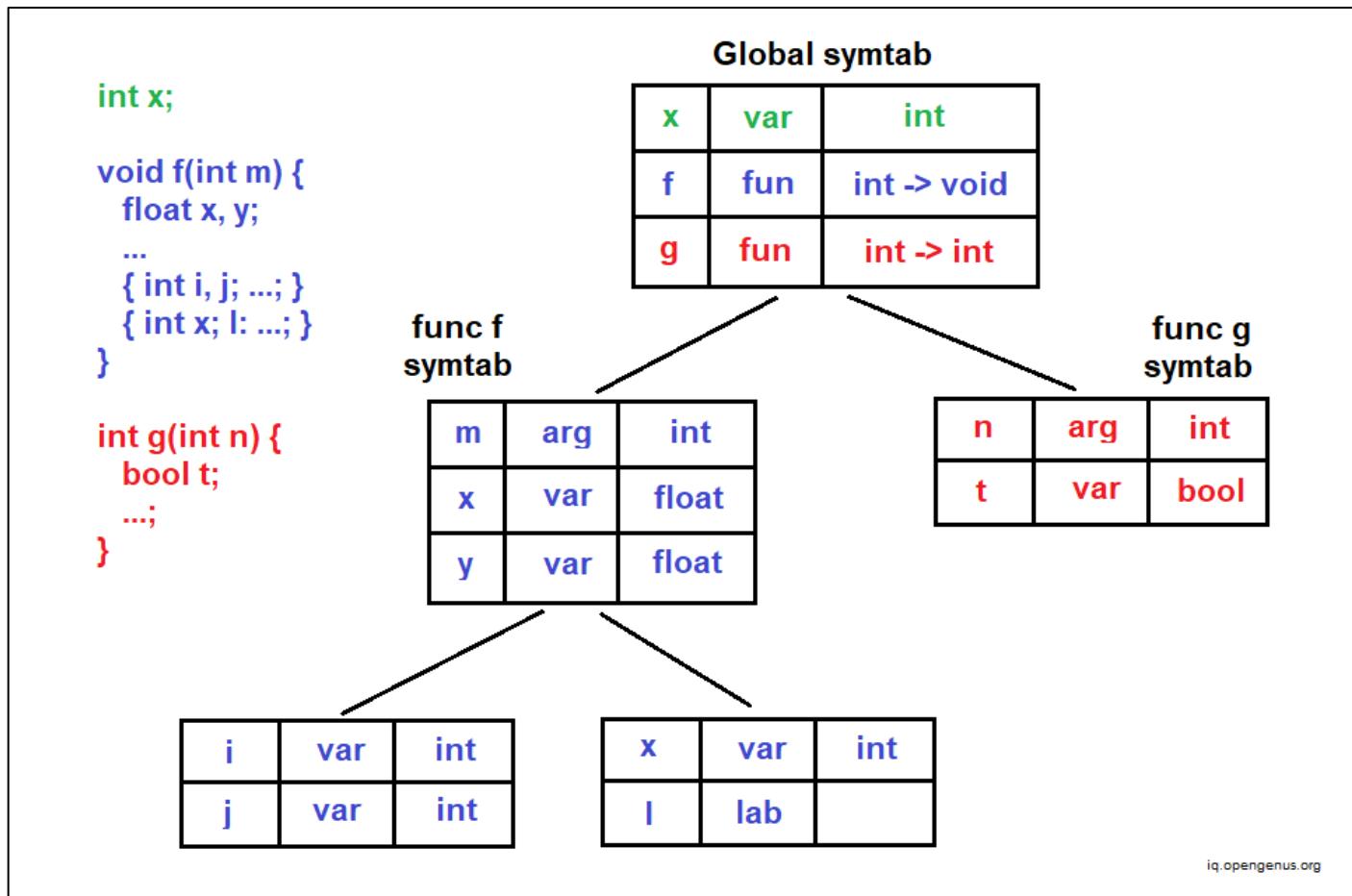
על כל שם משתנה נפעיל פונקציית Hash אשר תיתן לנו את מיקום שם המשתנה בטבלת ה – Hash. כך יוכל לגשת למקום המשתנה הגרפי זהה במערך בעלות זמן ריצה קבועה, (1)O.

על מנת למשתמש את ה – Hash השתמשתי במערך ובפונקציית Hash. פונקציית ה – Hash בה השתמשתי נקראת FNV-1a, ניתן לקרוא על פונקציה זו כאן - https://en.wikipedia.org/wiki/Fowler–Noll–Vo_hash_function.

Scope tree

על מנת למשתמש את ההיררכיה של ה – Scopes בתוכנית השתמש בעץ. כל צומת בעץ תציג Scope ותחזק את טבלת הסמלים של ה – Scope שהוא מייצגת, כמו כן, כל צומת בעץ תחזיק מערך של ה – Scopes שנמצאים בתוכה. שורש העץ יהיה ה – Global scope, ועבור כל Scope חדש יוצר בין עבורו ה – Scope הנוכחי בעץ ונעבור אליו. כאשר נמצא מ – נחזר לאב של ה – Scope הנוכחי.

תרשים



Error Handler

מציאת מצב התחלתי של Production Rule

נרצה להמשיך לבדוק את שאר קוד המקרו גם לאחר התקלות בשגיאה תחבירית. אך כאשר נתקל בשגיאה, לא יוכל לדעת לאיזה מצב לעבור עבורה – Token הבא מהקלט.

על מנת לאפשר זאת, נוסיף מערך הנutan לנו בעבורה – Token הבא מהקלט לאיזה מצב לעבור לאחר התקלות בשגיאה.

כרי ביעילות זמן ריצה קבועה, (1)O, נוכל לדעת מהין להמשיך את הקומpileציה, ובטיח בדיקה של רוב קוד המקרו.

תרשים

Token	int	char	if	...
Starting State	0	2	7	...

מצביעים לפונקציות טיפול בשגיאות

כל תא ב – Parse table המוביל לשגיאה, נוסיף מצביע לפונקציה המתפלת בשגיאה זו – מודעה הוועה אינפורטטיבית אודות השגיאה, מתקנת את המחסנית אם יש צורך, מתקדמת הלאה בקוד המקרו, ועוד.

כרי ביעילות זמן ריצה קבועה, (1)O, ותוך כדי תחילך ניתוח התחבירי, נוכל להודיע למתכנתה הוועה שגיאה אינפורטטיבית אודות השגיאה שביצע, וכן להמשיך לשאר קוד המקרו.

תיאור סביבת העבודה ושפת התוכנות



שפת התוכנות

שפת התוכנות בה השתמשתי לפיתוח הפרויקט היא שפת C.

כמו כן כתבתי מספר סקורייפטים עבור בניה, הרצה, ביצוע בדיקות וניהול של הפרויקט ב – Batch – Powershell .



סביבת העבודה

Operating System

- Microsoft Windows 10



Code Editor

- Visual Studio Code
 - version 1.66.2 (user setup)



C Compiler

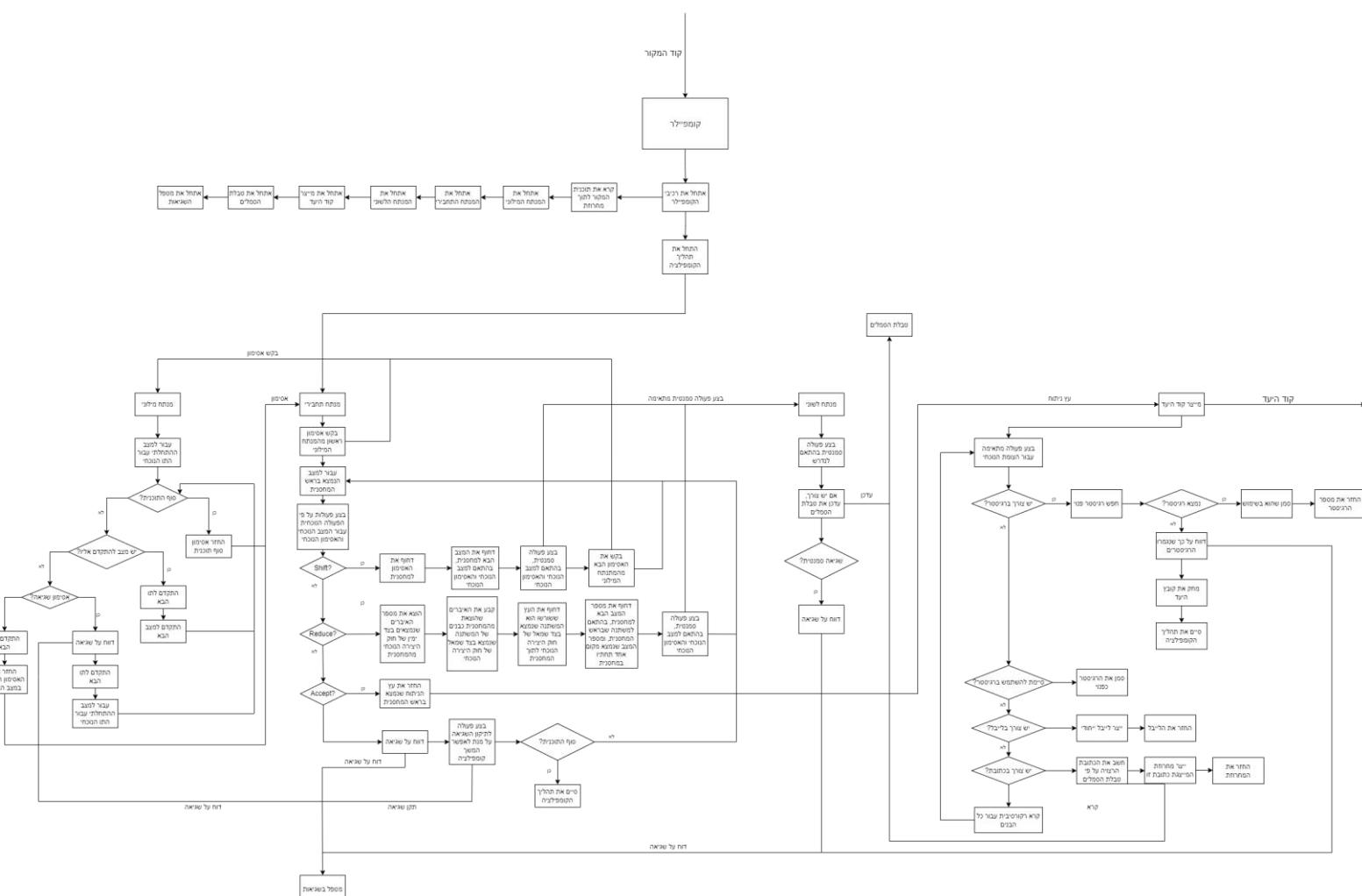
- gcc
 - (x86_64-win32-seh-rev0, Built by MinGW-W64 project) 8.1.0



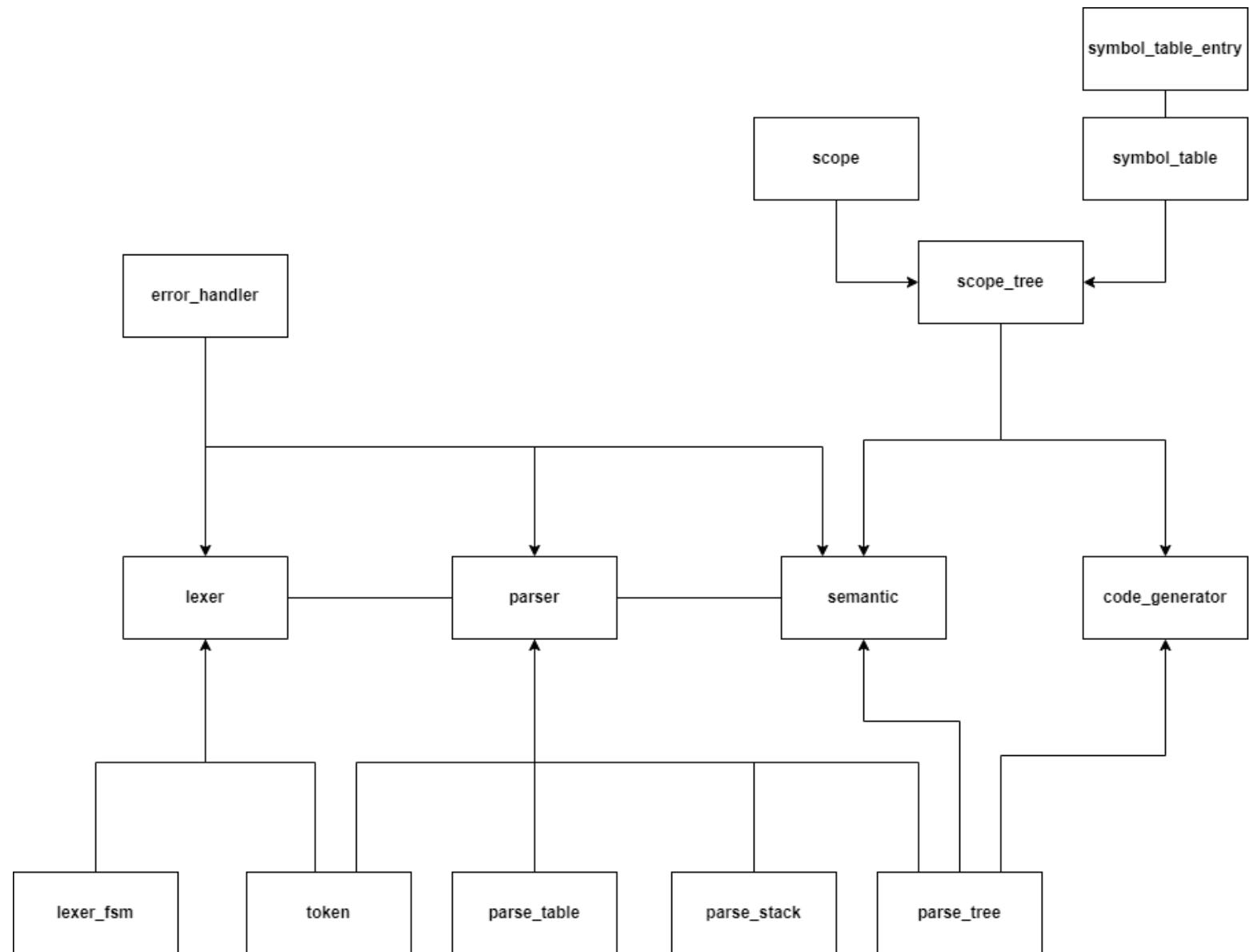
אלגוריתם ראשי

להלן תרשימים זרימה המציג את אלגוריתם העל של הקומpileר. הקומpileר מבצע אלגוריתם זה, ובעצם מתרגם את הקוד בשפת So לקוד אסמבלי, ביעילות אלגוריתמית לינארית, (ח)Ο.

התרשימים גדול ולקן איקוטו ירודה. **התרשימים המלאים** יוצרף כנספה.



תרשים מודולים



מודולים ופונקציות ראשיות

בפרק זה אסביר על המודולים השוניםialiם חילקתי את הפרויקט, ומahan הפונקציות הראשיות בכל מודול. את הקשרים בין המודולים החשובים יציגי בתרשים בפרק הקודם.

מבט על

להלן תרשيم טקסטואלי המתאר את היררכיה המודולים ותתי-המודוליםialiם חילקתי את קוד הפרויקט.

- token
- lexer
 - lexer_fsm
- parser
 - parse_stack
 - parse_table
 - parse_tree
- semantic
- code_generator
- scope_tree
 - scope
 - symbol_table
 - symbol_table_entry
- error_handler
- compiler
- general
- io
- ansi

מודולים

cut-arxiv על כל מודול ואפרט מה תפקידו, מהם ה – **Structs** שמכיל, אילו קבצים שייכים לאותו מודול, מהם המודולים הנמצאים תחתיו, מהן הפונקציות שמכיל ומהי הייעילות האלגוריתמית של כל אחת מהן.

token

מודול המגדיר את המבנה של אסימון, Token, ואופן השימוש בו.

Struct

```
// Token struct
typedef struct Token
{
    char* value;           // The value of the current token from the source code
    int value_len;         // The length of the value
    Token_Type token_type; // The type of the token. From the Token_Type enum
} Token;
```

קבצים

- token.h
- token.c

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	ייצור ומחזירה Token עם התכונות שניתנו.	Token*	מחרוזת ערך, אורך המחרוזת, סוג האסימון	token_init
O(1)	משחררת את הזיכרון של ה – Token שקיבלה.	-	Token*	token_destroy
O(1)	ייצור ומחזירה מחרוזת שמתארת את סוג האסימון שקיבלה.	מחרוזת	Token_Type	token_type_to_str
O(1)	ייצור ומחזירה מחרוזת שמתארת את האסימון שקיבלה.	מחרוזת	Token*	token_to_str

lexer

מודול המגדיר את המנתה המילוני של הקומפיאר.

Struct

```
// Struct of the lexer
typedef struct Lexer
{
    char c;           // Current character in the src code
    int i;            // Current offset from the starting of the source code
    Lexer_FSM* fsm; // The lexer's FSM
} Lexer;
```

קבצים

- lexer.h
- lexer.c

תתי-מודולים

- lexer_fsm

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	יצרת Lexer, וקובעתשה – Lexer של המשתנה глובלי compiler יצבע אליו.	-	-	lexer_create
O(1)	משחררת את הזיכרון של Lexer של המשתנה глובלי compiler.	-	-	lexer_destroy
O(1)	מאתחלת את ה – Lexer של המשתנה глובלי compiler.	-	-	lexer_init
O(1)	מקדמת את ה – Lexerתו אחד בקדם המקור.	-	-	lexer_advance
O(1)	בاهגינו למצוב מקלט במכונת המ מצבים (אין מצבים להתקדם אליו), יוצרת אסימון בהתאם לערך שנאפס עד כה והמצוב מקבל, מקדמת את ה – Lexerתו אחד בקדם המקור, ומוחזירה את האסימון.	Token*	מחרוזת ערך, אורך המחרוזת, מספר המ מצב במכונת ה מצבים של ה Lexer -	lexer_EOT
O(1)	מקבלת סוג אסימון ומוחזירה האם הוא אסימון שצריך לדלג עליו (Whitespace, Comment).	booliani	Token_Type	lexer_is_skip_token
O(n)	מחזירה את האסימון הבא בקדם המקור.	Token*	-	lexer_get_next_token

Lexer_get_next_token

הפונקציה המרכזית של ה – Lexer. בפונקציה זו משתמש ה – Parser על מנת לקבל כל פעם את האסימונ הבא מתוכנית המקור.

פירטתי בהרחבה על האלגוריתם איתה מממשת פונקציה זו בפרק האסטרטגיה, [ניתוח מילוני](#) – . שם גם ניתן למצוא את הפסאודו קוד של האלגוריתם, [פסאודו קוד](#).

lexer_fsm

מודול המגדיר את מكونת המצביעים של ה – .Lexer's Finite State Machine .Lexer

Struct

```
// Struct of the lexer's FSM
typedef struct Lexer_FSM
{
    // An array of starting state indices to help us know at which state to start when we start a new token
    int starting_state_indices[NUM_OF_CHARACTERS];
    // The array of states of the FSM
    Lexer_State states[LEXER_FSM_NUM_OF_STATES];
    // The matrix of edges of the FSM
    Lexer_Edge edges[LEXER_FSM_NUM_OF_STATES][NUM_OF_CHARACTERS];
} Lexer_FSM;
```

קבצים

- lexer_fsm.h
- lexer_fsm.c

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	יצרת את מكونת המצביעים של ה – .Lexer	-	-	lexer_fsm_create
O(1)	משחררת את הזיכרון של מكونת המצביעים של ה – .Lexer	-	-	lexer_fsm_destroy
O(1)	מאתחלת את מكونת המצביעים של ה – .Lexer	-	-	lexer_fsm_init
O(1)	יצרת ומוסיפה את המצביע עם סוג האסימון למكونת המצביעים של ה – .Lexer	-	מספר מצב, Token_Type	lexer_fsm_add_state
O(1)	יצרת ומוסיפה קשת בין המצביע הראשון למצביע השני, עברו התו הנוכחי.	-	מספר מצב, תו, מספר מצב	lexer_fsm_add_edge
O(1)	מוסיפה עברו מיקום התו הנוכחי את מספר המצביע הנוכחי כמספר המצביע ההתחלתי עברו אותותו.	-	מיקום התו, מספר מצב	lexer_fsm_add_starting_state_index
O(1)	יצרת קשת בין מספר המצביע הנוכחי לבין כל תו אלפא-נומרי, חוץ מהתו הנוכחי.	-	מספר מצב, תו	lexer_fsm_set_alnum_identifier
O(1)	מחזירה את מיקום התו הנוכחי במكونת המצביעים של ה – .Lexer	מספר	תו	lexer_fsm_get_char_index

O(1)	מחזירה את מספר המצב הראשוני עבור התו הנוכחי.	מספר	תא	lexer_fsm_get_starting_state_index
O(1)	מדפיסה את הגרפּ (מטריצת הסמיוכיות) של מכונת המצבים של ה –Lexer.	-	-	lexer_fsm_print

parser
מודול המגדיר את המנתה התחבירי של הקומפイルר.

Struct

```
// Struct of the parser
typedef struct Parser
{
    // The current token from the source code produced by the lexer
    Token* token;
    // The parser's parsing table, made from action & goto tables
    // helps up to know which action to perform according to the next token
    Parse_Table* parse_table;
    // The parser's stack
    Parse_Stack_Entry* parse_stack;
    // Array of the production rules
    // This will be used when we reduce by a production rule in the parsing phase
    Production_Rule production_rules[NUM_OF_PRODUCTION_RULES];
    // Array to store the starting state of the tokens at the start of a statement production rule in the language.
    // This array is mainly used for better error reporting and error recovery.
    // Every cell is filled with 0s, except for the tokens that are at the start of a statement production rule.
    // Those token's cells are filled with the number of their produciton rule's starting state in the pushdown automaton
    // according to the parsing table.
    int tokens_starting_state[NUM_OF_TERMINALS];
} Parser;
```

קבצים

- parser_base.h •
- parser.h •
- parser.c •

תתי-מודולים

- parse_stack •
- parse_table •
- parse_tree •

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	יצרת את המנתה התחבירי.	-	-	parser_create
O(1)	משחררת את הזיכרון של המנתה התחבירי.	-	-	parser_destroy
O(1)	מאתחלת את המנתה התחבירי.	-	-	parser_init

O(1)	מאתחلت את המערך של כללי היצירה של תחביר השפה.	-	-	parser_init_production_rules
O(1)	מאתחلت את המערך של המצביעים ההתחלתיים עבור אסימונים שנמצאים בתחילת של Statement.	-	-	parser_init_tokens_starting_state
O(1)	מחזירה את המצביע ההתחלתי עבור סוג האסימון שהתקבל.	מספר מצב	Token_Type	parser_get_token_starting_state
O(1)	מבצעת את פעולה Shift – כחלק מפעולתו של Parser – BU, על פי מספר המצביע הבא הנוכחי.	-	מספר מצב	parser_shift
O(1)	מבצעת את פעולה Reduce – כחלק מפעולתו של Parser – BU, על פי מספר כלל היצירה הנוכחי.	-	מספר כלל ייצור	parser_reduce
O(n)	מנתחת את קוד המקור ומחזירה את עץ הניתוח שמייצג את תוכנית המקור. כמו כן, מעדכנת טבלת הסמלים.	Parse_Tree_Node*	-	parser_parse

parser_parse

פונקציה המרכזית של הקומפיאר.

פונקציה זו מימושת את האלגוריתם הראשי של הפרויקט, הלווא הוא ה – Bottom Up Parser.

פירטתי בהרחבה על אלגוריתם זה בפרק האסטרטגיה, [ניתוח תחבירי – Syntax Analysis](#). שם גם ניתן למצוא את הפסאודו קוד של האלגוריתם, [פסאודו קוד](#).

parse_stack

מודול המגדיר את המחסנית בה משתמש ה – Parser על מנת למש את תהליך הניתוח.

Struct

```
// The struct of a entry of the parser's stack.
// The stack is represents by a linear linked list, where the stack of the list is the top of the stack.
typedef struct Parse_Stack_Entry
{
    Parse_Tree_Node* tree;           // Each entry of the stack holds a tree
    int goto_state;                 // The state to go to in the next iteration of the parser
    struct Parse_Stack_Entry* next_entry; // The next entry of the stack, the one on the bottom of the current one
} Parse_Stack_Entry;
```

קבצים

- parse_stack.h •
- parse_stack.c •

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	יצירת ומחזירה איבר מחסנית חדש עם העץ הנוכחי ועם מספר המצב הבא הנוכחי (goto_state).	Parse_Stack_Entry*	,Parse_Tree_Node* מספר מצב	parse_stack_init_entry
O(1)	שחררת את הזיכרון של מחסנית הניתוח של המנתח התחבירי. הרוסת את כל המחסנית.	-	-	parse_stack_destroy
O(1)	דוחفت את האיבר הנוכחי למחסנית.	-	Parse_Stack_Entry*	parse_stack_push
O(1)	מציאה איבר מראש המחסנית ומחזירה אותו.	Parse_Stack_Entry*	-	parse_stack_pop

parse_table

מודול המגדיר את ה – Parser, מורכבת מהטבלאות Action & Goto, של ה – Parse table בה משתמש על מנת לבצע את תהליכי הניתוח.

Structs

```
// Struct to hold the parsing table. Action & Goto tables all together
typedef struct Parse_Table
{
    // Action table. each cell contains a Action structure
    Action action_table[PARSING_TABLE_NUM_OF_STATES][NUM_OF_TERMINALS];
    // Goto table. Each cell contains only an int that represent a state number
    int goto_table[PARSING_TABLE_NUM_OF_STATES][NUM_OF_NON_TERMINALS];
} Parse_Table;
```

```
// Struct of an entry in the Action table
typedef struct Action
{
    // Error, Shift, Reduce, Accept
    Action_Type action_type;
    // If the current cell is a Shift, then this int represents the state to go next
    // If the current cell is a Reduce, then this int represents the number of the production rule to reduce by
    int state_or_rule;
    // Pointer to a function to be called when we reach that cell. Will also be part of the semantic analysis
    void (*semantic_func) ();
    // Pointer to a function to be called when we reach that cell. Takes part in the error reporting and recovery.
    void (*error_func) ();
} Action;
```

קבצים

- parse_table.h
- parse_table.c

פונקציות

יעילות	תיאור	פלט	קליט	פונקציה
O(1)	יצרת את ה – Parser – Parse table של .Parser	-	-	parse_table_create
O(1)	משחררת את הזיכרון של ה – Parser – Parse table של .Parser	-	-	parse_table_destroy
O(1)	מאתחלת את ה – Parser – Parse table של .Parser	-	-	parse_table_init
O(1)	מחזירה את האינדקס של סוג האסימול הנגון בטבלת הניתוח.	מספר	Terminal_Type	parse_table_get_terminal_index

O(1)	מכניסה את ה – Action הנtentן במקומות הנtentן (מספר מצב, מקום אסימון) לטבלת ה – Action בטבלת הנטיתות.	-	מספר מצב, מקום אסימון, Action	parse_table_insert_action
O(1)	מכניסה לטבלת ה – Goto את מספר המצב הבא הנtentן במקומות הנtentן (מספר מצב, מקום משתנה).	-	מספר מצב, מקום משתנה Non-terminal מספר מצב הבא	parse_table_insert_goto
O(1)	מדפיסה את טבלת הנטיתות.	-	-	parse_table_print

parse_tree
מודול המגדיר את עצם הניתנות.

Struct

```
// Struct of a node in the parse tree
typedef struct Parse_Tree_Node
{
    Symbol_Type symbol_type;           // Terminal / Non-Terminal
    int symbol;                      // The terminal or non-terminal kind of the current node
    Token* token;                    // If a node is a terminal then it will have a token, otherwise it will be NULL
    struct Parse_Tree_Node** children; // Array of a node pointers which represents a node's children
    int num_of_children;             // The length of the childrens array
    // Attribute of a node in the tree.
    // If the node is part of expression it will have a data type.
    Data_Type data_type;
    // The register index in the register array of the code generator that holds this node's result.
    // If the node is part of expression it will have a register holding it's result.
    int register_number;
} Parse_Tree_Node;
```

קבצים

- parse_tree.h
- parse_tree.c

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	ייצור Node חדש עם התכונות הנútנות ומהזירה מצביע אליו.	Parse_Tree_Node*	,Symbol_Type מספר symbol ,Token* ,Parse_Tree_Node** מספר ילדים,	parse_tree_init_node
O(n)	פונקציה רקורסיבית המשחררת את הזיכרון של כל העץ ששורשו נתון.	-	Parse_Tree_Node*	parse_tree_destroy
O(n)	מדפסה את העץ הנútן בפורמט יפה ונוח לקריאה.	-	Parse_Tree_Node*	parse_tree_print

semantic

מודול המגדיר את המנתה הלשוני של הקומpileר.

Enum

```
// Enum of all the possible types of an identifier in the language.
// This enum is only for readability and ease of use.
typedef enum Data_Type
{
    Data_Type_NULL = 0,           // For tree nodes that are not part of an expression
    Data_Type_Int = Token_Int,   // int
    Data_Type_Char = Token_Char, // char
} Data_Type;
```

קבצים

- semantic.h
- semantic.c

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	בודקת האם ניתן לבצע פעולה המשמה בין ה – Data_Type הראשון למשתנה מסווג Data_Type השני.	בוליани	,Data_Type Data_Type	semantic_check_assign_compatibility
O(1)	בודקת האם ניתן לבצע את הפעולה המתמטית שמייצג Token – (אופרטור בינארי) בין שני Token – Data_Types הנתונים.	בוליани	,Data_Type ,Token* Data_Type	semantic_check_binary_op_compatibility
O(1)	בודקת האם ניתן לבצע את הפעולה המתמטית שמייצג Token – (אופרטור אונארי) על סוג המידע הנתון.	בוליани	,Data_Type Token*	semantic_check_unary_op_compatibility
O(1)	נראית בכניסה לכל block חדש. מוסיפה Sub-Scope הנוכחי בעץ ל – Scope וועוברת Scopes – Scope – הזה.	-	-	semantic_enter_block

O(1)	נקראת בכל יציאה מ – block חוזרת ל – Scope האב של ה – Scope הנוכחי בעז ה – Scopes – .Scopes	-	-	semantic_exit_block
O(1)	מבצעת בדיקה סמנטית לאחר הצורה על משתנה. בודקת שלא קיימים כבר ב – Scope – ב – Scope הנוכחי.	-	-	semantic_decl
O(1)	מבצעת בדיקה סמנטית לאחר פעולות השמה. בודקת שהמשתנה אכן קיים, ושסוג המשתנה וסוג הערך המושם תואמים.	-	-	semantic_assign
O(1)	לאחר התקלות בכלל ייצירה של expression מסווג אופרנד אופרטור או פרנד, בודקת שנית לבצע את הפעולה של האופרטור בין שני האופרנדים. כמו כן, קובעת ש – Non-terminal2 יהיה שורש העץ על מנת לפשטו.	-	-	semantic_set_type
O(1)	לאחר התקלות בכלל ייצירה של expression מסווג אופרנד אופרטור או פרנד, בודקת שנית לבצע את הפעולה של האופרטור בין שני האופרנדים.	-	-	semantic_type_check
O(1)	לאחר התקלות בכלל ייצירה של id → F, בודקת האם המשתנה קיים. כמו כן, גורמת לכך שה – id יה שורש העץ במקום F על מנת לפשט את העץ.	-	-	semantic_F_to_id
O(1)	לאחר התקלות בכלל ייצירה של F → literal, גורמת לכך שה – literal יהיה שורש העץ במקום F על מנת לפשט את העץ.	-	-	semantic_F_to_literal

O(1)	לאחר התקלות בכלל יצירה $F \rightarrow (L_LOG_E)$ קובעת את ה expression – כשורש העז על מנת לפשטו.	-	-	semantic_F_to_L_LOG_E
O(1)	עבור כללי היצירה $F \rightarrow -$ או $F \rightarrow !F$, בודקת שנית לבצע פעולה זו על F, ואם כן קובעת את ה Type של ה LHS – להיות זהה ל RHS –	-	-	semantic_F_to_unary_op_F
O(1)	מחדרה מחוזת המיצגת את סוג המידע הנוכחי.	מחוזת	Data_Type	semantic_data_type_to_str

code_generator

מודול המגדיר את מחולל קוד היעד של הקומפיאר.

Structs

```
// Struct of the code generator
typedef struct Code_Generator
{
    Register registers[NUM_OF_REGISTERS]; // Array of registers to be used in the code generation process
    FILE* dest_file; // A pointer to the output file for the generated code
} Code_Generator;
```

```
// Struct of a register in the code generator's register array
typedef struct Register
{
    char name[REGISTER_NAME_LENGTH]; // The name of the register for target code output
    bool inuse; // Whether the register is currently in use or not
} Register;
```

קבצים

- code_generator_base.h
- code_generator.h
- code_genereator.c

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	יצרת את מחולל קוד היעד של הקומפיאר.	-	-	code_generator_create
O(1)	משחררת את זיכרון של מחולל קוד היעד של הקומפיאר.	-	-	code_generator_destroy
O(1)	מאתחלת את מחולל קוד הבניינים של הקומפיאר.	-	-	code_generator_init
O(1)	מחזירה מספר רגיסטר פנוי במערך הרגיסטרים.	מספר רגיסטר	-	code_generator_register_alloc
O(1)	"משחררת", מסמנת רגיסטר כפוני במערך הרגיסטרים.	-	מספר רגיסטר	code_generator_register_free

O(1)	מחזירה מחרוזת המציגת את מספר הרגיסטר הנוכחי.	מחרוזת	מספר רגיסטר	code_generator_register_name
O(1)	יצרת ומחזירה Label ייחודי.	מחרוזת	-	code_generator_label_create
O(1)	מחשבת ומחזירה את כתובות המשתנה הנוכחי.	מחרוזת	Symbol_Table_Entry*	code_generator_symbol_address
O(1)	מדפסה לקובץ היעד.	-	char* format, ...	code_generator_output
O(1)	מדפסה לקובץ היעד את סגמנט הנתונים של התוכנית.	-	-	code_generator_output_data_segment
O(n)	מייצרת את סגמנט הקוד של התוכנית.	-	Parse_Tree_Node*	code_generator_generate
O(n)	.BLOCK מייצרת	-	Parse_Tree_Node*	code_generator_block
O(n)	.STMT מייצרת	-	Parse_Tree_Node*	code_generator_stmt
O(n)	.DECL מייצרת	-	Parse_Tree_Node*	code_generator_decl
O(n)	.ASSIGN מייצרת	-	Parse_Tree_Node*	code_generator_assign
O(n)	.IF_ELSE מייצרת	-	Parse_Tree_Node*	code_generator_if_else
O(n)	.ELSE מייצרת	-	Parse_Tree_Node*	code_generator_else
O(n)	.WHILE מייצרת	-	Parse_Tree_Node*	code_generator_while
O(n)	.expression מייצרת	-	Parse_Tree_Node*	code_generator_expression
O(1)	מייצרת ביטוי ביןארי.	-	מספר רגיסטר, מספר רגיסטר, Token_Type	code_generator_binary_expression
O(1)	מייצרת ביטוי אונארי.	-	מספר רגיסטר, Token_Type	code_generator_unary_expression
O(1)	מייצרת ביטוי לוגי.	-	מספר רגיסטר, מספר רגיסטר, Token_Type	code_generator_bool_e

O(1)	מדפסה לקובץ הקוד המשים Token את ה – Token בהתאם לסוג ה – Token.	-	מספר רגיסטר, Token*	code_generator_mov_token
O(1)	מדפסה לקובץ הקוד המשים Token את ה – Identifier מסוג Token, לרגיסטר הנთoon, בהתאם לסוג המשתנה.	-	מספר רגיסטר, Token*	code_generator_mov_identifier
O(1)	מייצרת OR לוגי ושם את התוצאה (0 / 1) ברגיסטר הראשון.	-	מספר רגיסטר, מספר רגיסטר	code_generator_or
O(1)	מייצרת AND לוגי ושם את התוצאה (0 / 1) ברגיסטר הראשון.	-	מספר רגיסטר, מספר רגיסטר	code_generator_and
O(1)	מייצרת קוד המבצע את פעולה החילוק בין שני הרגיסטרים ושם את התוצאה ברגיסטר הראשון.	-	מספר רגיסטר, מספר רגיסטר	code_generator_divide
O(1)	מייצרת קוד המבצע את פעולה השארית בין שני הרגיסטרים ושם את התוצאה ברגיסטר הראשון.	-	מספר רגיסטר, מספר רגיסטר	code_generator_modulu
O(1)	מייצרת AND לוגי ושם את התוצאה (1 / 0) חזרה ברגיסטר.	-	מספר רגיסטר	code_generator_not

scope_tree
מודול המגדיר את עצה – של Scopes של הקומpileר.

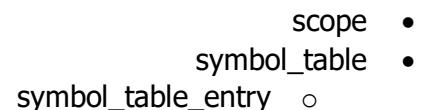
Struct

```
// Struct of the symbol tables scope tree.
// This is the struct we will use as our symbol table.
// The tree represents the hierarchy of scopes in the source program.
typedef struct Scope_Tree
{
    Scope* global_scope; // The root of the tree - the global scope
    Scope* current_scope; // The current scope we are at
} Scope_Tree;
```

קבצים

- scope_tree.h
- scope_tree.c

תתי-מודולים



פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	יוצרת את עצה – Scopes של הקומpileר, מאותחל עם Scope גלובלי ריק.	-	-	scope_tree_create
O(1)	הורסת את עצה – Scopes של הקומpileר.	-	-	scope_tree_destroy
O(n)	פונקציה רקורסיבית להריסת עצה – Scopes הנ庭ון.	-	Scope*	scope_tree_destroy_tree
O(1)	יוצרת Sub-Scope חדש עבור הסקופ הנוכחי.	-	-	scope_tree_add_scope
O(1)	מעבירה את הסקופ הנוכחי לאב שלו.	-	-	scope_tree_goto_parent
O(1)	מעבירה את הסקופ הנוכחי לבן הבא שלו (שעוד לא הינו בו).	-	-	scope_tree_goto_child

O(n)	מחזירה את הרשימה בטבלת הסמלים בה נמצא שם המשתנה זהה.	Symbol_Table_Entry*	שם משתנה	scope_tree_fetch
O(n)	מאפסות עבור כל Scopes בעץ הנ庭ן את מספר הילך האחרון שביקרנו בו.	-	Scope*	scope_tree_reset_child_index
O(n)	מדפיסת עצם Scopes –	-	-	scope_tree_print

scope
מודול המגדיר Scope בז'ה – של הקומpileר.

Struct

```
// Struct of a single node in the tree, scope, in the tree of symbol tables scopes
typedef struct Scope
{
    // Current scope's symbol table
    Symbol_Table* symbol_table;
    // Array of scopes which are the children of the current scope.
    // Each child represent a sub scope of the current scope.
    struct Scope** children;
    int num_of_children;           // Length of the children array
    int current_child_index;       // Index of the current child in children array
    // Pointer to the current scope's parent in the symbol tables scopes tree
    struct Scope* parent;
    // Number of entries seen up to this point.
    // In this scope and in all the scope on the path up to the root scope.
    // Used in the code generation process in the address computation for variables.
    int available_entries;
} Scope;
```

קבצים

- scope.h
- scope.c

פונקציות

פעילות	תיאור	פלט	קלט	פונקציה
O(1)	יוצרת ומוחזירה מצביע ל – Scope חדש עם Symbol table ריק ועם מצביע לאב הנטוון.	Scope*	Scope*	scope_init

מודול המגדיר `Symbol table`, טבלת סמלים, הנמצאת בכל Scope בעז ה – `Scopes` של הקומpileר.

Struct

```
// Struct of a symbol table using chaining
typedef struct Symbol_Table
{
    Symbol_Table_Entry** entries; // Array of the entry pointers of the symbol table
    int capacity; // Max capacity of symbol table
    int num_of_entries; // Current number of entries in the symbol table
    int num_of_indices_occupied; // Number of indices that have entries in them
} Symbol_Table;
```

קבצים

- `symbol_table_base.h`
- `symbol_table.h`
- `symbol_table.c`

תתי-מודולים

- `symbol_table_entry`

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	יוצרת <code>Symbol_Table</code> ומחזירה מצביע אליו.	<code>Symbol_Table*</code>	-	<code>symbol_table_create</code>
O(n)	"הורסת", מחררת את הזיכרון של <code>Symbol_Table</code> ה – <code>הנתון</code> .	-	<code>Symbol_Table*</code>	<code>symbol_table_destroy</code>
O(1)	מבצעת Hash לשם המשתנה על פי האלגוריתם <code>a-FNV-1a</code> .	<code>uint64_t</code>	שם המשתנה	<code>symbol_table_hash</code>
O(1)	מכניסה את ה – <code>Entry</code> לatable הסמלים <code>הנתון</code> .	-	<code>Symbol_Table*</code> <code>Symbol_Table_Entry*</code>	<code>symbol_table_insert</code>
O(1)	מאחזרת את <code>Entry</code> של שם המשתנה <code>הנתון</code> מatable הסמלים <code>הנתון</code> .	<code>Symbol_Table_Entry*</code>	<code>Symbol_Table*</code> שם המשתנה	<code>symbol_table_fetch</code>

O(n)	מרחיבה את טבלת הסמלים.	-	Symbol_Table*	symbol_table_expand
O(n)	מדפיסה את טבלת הסמלים.	-	Symbol_Table*	symbol_table_print

symbol_table_entry
מודול המגדיר איבר, **entry**, בטבלת הסמלים.

Struct

```
// Struct of an entry in the symbol table
typedef struct Symbol_Table_Entry
{
    Entry_Type entry_type;           // The type of the entry - variable, function, ...
    char* identifier;               // The identifier name which is the key for each entry
    Data_Type data_type;            // The type of the identifier - int, char, ...
    struct Scope* scope;           // Pointer to the scope that entry is located at.
    // Used in the code generation process to go in O(1) to the scope of a given entry
    struct Scope* scope;
    // Whether that symbol is a global symbol or not.
    // Used in the code generation process to determine whether a variable
    // should be considered from the data segment or from the stack
    bool is_global;
    // The number of the entry in the current scope.
    // At which order it was entered. If it's the first, second, third, ...
    // Used in the code generation process to resolve variables addresses relative to bp
    int num_in_scope;
    struct Symbol_Table_Entry* next_entry; // Pointer to the next entry in the linked list of entries in that particular index
} Symbol_Table_Entry;
```

קבצים

- symbol_table_entry.h
- symbol_table_entry.c

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	יצירת Entry עם התכונות הנתונות.	Symbol_Table_Entry*	,Entry_Type שם משתנה, Data_Type	symbol_table_entry_init
O(1)	שחררת את הזיכרון של .Entry	-	Symbol_Table_Entry*	symbol_table_entry_destroy
O(1)	.Entry מדפסה	-	Symbol_Table_Entry*	symbol_table_entry_print

error_handler
מודול המגדיר את מטפל השגיאות, Error Handler של הקומpileר.

Enum

```
// Enum for possible error types
typedef enum Error_Type
{
    Error_General = 1,
    Error_Lexical,
    Error_Syntax,
    Error_Semantic,
} Error_Type;
```

קבצים

- error_handler.h
- error_handler.c

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	מדפסה הודעה שגיאה בהתאם לסוג ההדעה וההודעה שהתקבלה. כמו כן, מעדכנת את מספר השגיאות שנראו עד כה בתהיליך הקומpileר. השגיאות יונקציה נרמזו באמצעות סימן נספחים.	-	מספר שורה, Error_Type הודעה, ...	error_handler_report
O(1)	פונקציה גנרטיב בה משתמש הקומpileר עבור חזרה משגיאה, כך יוכל להמשיך בתהיליך הקומpileר גם לאחר שנטקל בשגיאה.	-	-	error_handler_error_recovery

O(1)	מוצג שגਆה מהקומפיילר ציפה לאסימון prog, וודוחת prog למחסנית.	-	-	error_handler_report_expected_prog
O(1)	מוצג שגਆה מהקומפיילר ציפה לאסימון pi שם של תוכנית, וודוח pi למחסנית.	-	-	error_handler_report_expected_prog_name
O(1)	מוצג שגਆה מהקומפיילר ציפה לאסימון : , וודוח : למחסנית יחד עם מספר המצביע המתאים עבור המצביע המתאים.	-	-	error_handler_report_expected_colon_state_3 error_handler_report_expected_colon_state_37 error_handler_report_expected_colon_state_46 error_handler_report_expected_colon_state_59
O(1)	מוצג שגਆה מהקומפיילר ציפה לאסימון ; , וודוח ; למחסנית יחד עם מספר המצביע המתאים עבור המצביע 18.	-	-	error_handler_report_expected_semi_colon_state_18
O(1)	מוצג שגਆה מהקומפיילר ציפה לאסימון .eof	-	-	error_handler_report_expected_eof
O(1)	מוצג שגਆה מהקומפיילר ציפה לאסימון .()	-	-	error_handler_report_expected_smiley

O(1)	מוצג שגਆה שהקומpileר ציפה לאסימון id.	-	-	error_handler_report_expected_identifier
O(1)	מוצג שגਆה שהקומpileר ציפה לאסימון).	-	-	error_handler_report_expected_open_paren
O(1)	מוצג שגਆה שהקומpileר ציפה לאסימון =.	-	-	error_handler_report_expected_assign
O(1)	מוצג שגਆה ספציפית עבור כל המצביעים שדומים למצב .4	-	-	error_handler_report_expected_state_4_group
O(1)	מוצג שגਆה ספציפית עבור כל המצביעים שדומים למצב .7	-	-	error_handler_report_expected_state_7_group
O(1)	מוצג שגਆה ספציפית עבור כל המצביעים שדומים למצב .20	-	-	error_handler_report_expected_state_20_group
O(1)	מוצג שגਆה ספציפית עבור כל המצביעים שדומים למצב .24	-	-	error_handler_report_expected_state_24_group
O(1)	מוצג שגਆה ספציפית עבור כל המצביעים	-	-	error_handler_report_expected_state_25_group

	שדומים למצב .25				
O(1)	מוצג הודעת שגיאה ספציפית עבור כל המצבים שדומים למצב .36	-	-		error_handler_report_expected_state_36_group
O(1)	מוצג הודעת שגיאה ספציפית עבור כל המצבים שדומים למצב .56	-	-		error_handler_report_expected_state_56_group
O(1)	מחזירה מחרוזת המייצגת את סוג השגיאה שהתקבלה.	מחרוזת	Error_Type		error_handler_error_to_str

compiler

מודול המאגד תחתיו את כל חלקי הקומpileר.

אני משתמש במשתנה גלובלי יחיד מסוג `Compiler`, שמוגדר בקבצים `global.h` ו- `global.c`, על מנת לפשט את הקוד (לא צריך להזכיר דברים מיוחדים כפרמטרים) וכדי לנוהל את הזיכרון באופן נוח מכל מקום בתוכנית.

Struct

```
// Struct of the entire compiler
typedef struct Compiler
{
    // The source code to be compiled
    char* src;
    // The name of the destination file
    char* dest_file_name;
    // The lexer of the compiler
    Lexer* lexer;
    // The parser of the compiler
    Parser* parser;
    // The scope tree of the compiler
    Scope_Tree* scope_tree;
    // The code generator of the compiler
    Code_Generator* code_generator;
    // Number of errors found during compilation
    int errors;
    // Current line number in source file for error reporting
    int line;
} Compiler;
```

קבצים

- `compiler.h`
- `compiler.c`

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	מאתחלת את המשתנה הגלובלי <code>compiler</code> ואת כל הרכיבים שלו, על מנת לאפשר התחלת של תהליך הקומPILEר.	-	שם קובץ המקור, שם קובץ היעד	<code>compiler_init</code>
O(1)	משחררת את כל הזיכרון של הקומPILEר.	-	-	<code>compiler_destroy</code>
O(n)	לאחר אתחול, מEMPLת את קוד המקור.	-	-	<code>compiler_compile</code>

general

מודול המכיל הגדרות ופונקציות לשימוש כלל המודולים השונים.

קבצים

- general.h
- general.c

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(n)	מדוחת על שגיאת זיכרון, משחררת את כל הזיכרון שהשתמשנו בו עד כה בתוכנית, ויצאת.	-	שם קובץ, שורה	exit_memory_error
O(n)	מדוחת על שגיאת so file, משחררת את כל הזיכרון שהשתמשנו בו עד כה בתוכנית, ויצאת.	-	שם קובץ	exit_file_io_error

io

מודול האחראי על תמייהה, הפשתה, שימוש וניהול של קבצים, so file.

קבצים

- io.h
- io.c

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(n)	מקבלת שם של קובץ טקסט ומחזירה מהרזהת המכילה את התוכן של קובץ זה.	מחרוזת	שם קובץ	io_read_file

ansi

מודול האחראי על תמייהה – Ascii escape codes, על מנת לאפשר הדפסה של הודעות שגיאיה, הצלחה, מידע עם צבעים למשתמש דרך ה-console – .

קבצים

- ansi.h
- ansi.c

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	.Ascii escape code ל – console תמייהה ל – . כך ניתן להדפיס בצבעים שונים ב – .console נלקח מכאן: https://youtu.be/bQ8qaBjJtYU	-	-	ansi_setup_console

מדריך למשתמש

התקנה

cutת אסביר מהן הדרישות והשלבים על מנת להתקין את הקומפיילר שלו באופן נכון על מחשבכם.

דרישות

על מנת להתקין את הקומפיילר אתם צריכים לוודא שיש לכם `Windows PowerShell` מותקן על המחשב. כשייתחתי את הקומפיילר השתמשתי ב – `Powershell 5.1.19041.1645`.

שלבים

1. העתיקו את הפרויקט אל מחשבכם מהקייםור הבא: <https://github.com/ido-hi/do-compiler.git>
2. העבירו את התקינה של הפרויקט אל המיקום הרצוי במחשבכם.
- ה – Path לתקינות הפרויקט יתווסף למשתנה הסביבה `PATH` של המשתמש שלכם מאוחר יותר.
3. הריצו את הסקריפט `install.bat` שנמצא בתיקיית הפרויקט.
- מוסיף את ה – Path לתקינות `ch0n` אשר כוללת את הקומפיילר שלו ואת הקבצים הנחוצים על מנת לקמפל את האסמלבי, למשתנה הסביבה `PATH` של המשתמש שלכם.

זהו! cutת אתם יכולים להשתמש בקומפיילר שלו מכל מקום שתרצו 😊

הסרה

אם תרצה להסיר את הקומפיילר שלו מהמחשב שלכם (אני לא רואה סיבה שתרצטו, אבל בסדר), תוכל להריץ בפשטות את הסקריפט `uninstall.bat` שנמצא בתיקיית הפרויקט.

הסקריפט דואג לשני דברים:

1. מסיר את ה – Path לתקינות `ch0n` ממשתנה הסביבה `PATH` של המשתמש שלכם.
2. מוחק את תיקית הפרויקט מהמחשב שלכם.

זהו, עכשיו הקומפיילר שלו הוסר מהמחשב שלכם 😊

בנייה הקומפיילר

הפרויקט מגיע בתיקיות `ch0n` עם כל הקבצים הנחוצים על מנת לקמפל קוד בשפת Do לאסמלבי, ולקמפל את האסמלבי הנק"ל לאפליקציה. בין הקבצים האלו נמצא `do.exe`, הלווא הוא – `Do compiler`!

cutת אסביר כיצד תוכלו לבנות וליצור שוב את הקובץ `do.exe` אם במקרה נמחק, או שאתם חושבים שלא עובד כמו שצריך.

דרישות

בשביל לבנות את הקומפיילר תצטרכו להתקין קומפיילר לשפת C, ובאופן יותר ספציפי `GCC`. בהמשך פיתוח הקומפיילר השתמשתי ב – `gcc (x86_64-win32-sse-rev0, Built by MinGW-W64 project) 8.1.10`.

שלבים

1. פתחו `Powershell` בתיקיות `scripts` שבתוכו תיקית הפרויקט.
2. הריצו את הסקריפט `build.ps1`.

זהו! הקובץ `do.exe` אמור להופיע בתיקיה `bin`, ועכשיו תוכלו להשתמש בקומפיילר שלו 😊

שימוש

כעת אסביר כיצד תוכלו להשתמש בקומpileר על מנת לкомפל קוד בשפת Do לקוד אסמלבי 64 ביט, ולקמפל את האסמלבי הנ"ל לאפליקציה שיכולה לרוץ על המחשב שלכם.

Assembly – ל Do

הרכזו את הפקודה `[cmd] do -b [path/to/source.do] [/path/to/destination.asm]` על מנת לкомפל קובץ

לקובץ `.asm`. אם ישן שגיאות בקוד, הן ייצגו כהודעות אינפורטטיביות וברורות למשתמש ותהייר הקומPILEציה ייעצה.

ארגוןメント היעד הוא אופציונלי. אם צוין, יוצר קובץ אסמלבי עם `MASM Assembly` 64ビット. אם לא צוין, יוצר קובץ `.asm.a` במקום בו אתם מרצים את הפקודה.

הרכזו את הפקודה `do` ללא ארגומנטים בשbill להבין איך להשתמש בפקודה.

דוגמא

ניתן לראות דוגמא לתוכנית בשפת Do בפרק של ספר השפה, [דוגמא לתוכנית בשפת Do](#).

```
C:\Windows\System32\cmd.exe
C:\Ido_Hirsh\Projects\Do-Compiler\User Manual>do main.do main.asm

Parsing...
Generating code...

Compilation successful!

C:\Ido_Hirsh\Projects\Do-Compiler\User Manual>
```

PC > OS (C:) > Ido_Hirsh > Projects > Do-Compiler > User Manual		
Name	Type	Size
main.asm	ASM File	2 KB
main.do	DO File	1 KB

exe ל - Assembly

הրיצו את הפקודה `ml64 filename.asm /link /subsystem:windows /entry:main` ב – cmd על מנת לקמפל קובץ exe לקובץ exe.

דוגמא

```
C:\Windows\System32\cmd.exe
C:\Ido_Hirsh\Projects\Do-Compiler\User Manual>ml64 main.asm /link /subsystem:windows /entry:main
Microsoft (R) Macro Assembler (x64) Version 14.31.31105.0
Copyright (C) Microsoft Corporation. All rights reserved.

Assembling: main.asm
Microsoft (R) Incremental Linker Version 14.31.31105.0
Copyright (C) Microsoft Corporation. All rights reserved.

/OUT:main.exe
main.obj
/subsystem:windows
/entry:main

C:\Ido_Hirsh\Projects\Do-Compiler\User Manual>
```

PC > OS (C:) > Ido_Hirsh > Projects > Do-Compiler > User Manual		
Name	Type	Size
main.asm	ASM File	2 KB
main.do	DO File	1 KB
main.exe	Application	3 KB
main.obj	3D Object	1 KB
mllink\$	Shortcut	1 KB

הקבצים `main.obj` ו – `mllink$` הם קבצים הנוצרים בתהליך הקומpileציה על ידי האסמלר `ml64` של מיקרוסופט, בו אני משתמש לקמפל את ה – `MASM Assembly x64` שהקומפイルר שלי יוצר. האפליקציה שלכם היא הקובץ `main.exe`, וניתן למחוק את הקבצים `main.obj` ו – `mllink$`.

האסמלר `ml64` ושאר הקבצים שהוא תלוי בהם על מנת לקמפל את האסמלר מגיעים עם הפרויקט בתיקיית `bin`, ויש לנו גישה אליהם מכל מקום על ידי זה זה ש – `install.bat` הוסיף את ה – `Path` לתיקיית `bin` לשנתנה הסביבה `PATH` של המשתמש שלנו.

זהו! כעת אתם יודעים כיצד להשתמש בקומפイルר שלי על מנת לקמפל את הקוד שלכם בשפת Do לאפליקציה שיכולה לרוץ על המחשב שלכם 😊

סיכום אישי

וואו. אני אפילו לא יודע מאייפה להתחיל...

ה השנה עשית שנת יג כחלק מסלול סייגט ביחידת 8200 באכבה ההגנה לישראל.

למדתי ולקחת קורסים בנושאים ותחומים שונים בתחום המחשב, ביניהם מבני נתונים ואלגוריתמים, שפת C ואסמבלי, מוד< נטונים ושפה SQL, ושפת Java. כפרויקט גמר לשנה זו התבקשתי לבנות פרויקט הכלול פתרון לבעה אלגוריתמית, תוך שימוש במבני נתונים עילים ושילוב הנושאים השונים למדתי השנה.

כפי שציינתי במאוא לספר זה, אני אדם שאוהב אתגרים. ولكن, כשהמרצה דיברה בכיתה על הפרויקט המאתגר, הלא הוא בניית קומפיאיר, לא היססת וקפסטי על ההזדמנות.

בדיעבד, כל מה שאני יכול להגיד על הבחירה הזאת זה וואו!

זה הפרויקט הכלי מאתגר, מרכיב, מלמד, מעוניין, מתקסל, מספק ומפתח שעשייתי בקריירה שלי. הוא פיתח אותי אישיות ולמד אותי המון על עצמי והיכולות שלי, כמו כוח רצון, השקעה ולימוד עצמי הן דבר חזק. פיתח את יכולות שלי בתור מתכנת, לימד אותי רבota על התחום של מדעי המחשב בכלל והתחום של קומפיאירים בפרט, ועזר לי להבין לעומק, לפרטי פרטים, כיצד הדברים שהיו נראים לי כמבנה מאליהם, באמצעות קורדים מאחוריו הקלעים.

הסיפור

از תחילת מההתחלתה...

ברגע שהבנתי שעלוי לבנות פרויקט גמר, ולאחר שנסגרתי על בניית קומפיאיר, ניגשתי לעבודה.

הגעתי בלי שום רקע או ידע קודם על קומפיאירים, ובהתחלה המשימה נראהה לי בלתי אפשרית. איך אני, אמור לבנות תוכנה, שתיקח קווד שמיישהו אחר כתוב ותרגמו אותו ל – 0 + 1 שהמחשב מסוגל להבין?

התחלתי למדוד את התאוריה.

זה מאוד מפתחה להתחיל לכתוב קווד ולרצות שדברים יתחלו לעבוד, אבל בלי תאוריית הבנתי שאני נראה סתם אעשה עבודה כפולה ומיותרת, ואבחן את הזמן שלו.

במשך יותר מחודש רק קראתי ספרים ומארונים, התייעצתי עם אנשים, ראייתי סרטונים, ולקחת קורסים שלמים העוסקים בתאוריה של מדעי המחשב וקומפיאירים. למדתי מהן הגישות המגוונות לפתרון בעויות האלגוריתמיות השונות העולות בפרויקט זה, ומהם האלגוריתמיים ומבנה הנתונים השונים הקיימים בעולם זה. תוך כדי הלמידה למדתי גם רבות על תחומי נספחים לא ידעת מה הקשר בין קומפיאירים. למדתי על התאוריה החישובית, שפות פורמליות ומכונות מצבים, גרפים, עיצוב שפות תכנות, ומבנה נתונים נוספים.

לאחר שחשבתי שיש בידי מספיק ידע על מנת להתחיל, עיצבתי את השפה שלי, So. עיצוב השפה הוא בין החלקים החשובים ביותר בבניית קומפיאיר, כי בסופו של דבר הוא מה שמאגדיר את הדרישות והיכולות שהקומפיאיר צריך למש. ומתוך הדברים, השפה שונתה מספר פעמים במהלך הפרויקט, תוך כדי למידה והבנה של דברים שלא הבנתי בתחלת התחילה.

לאחר שנסגרתי על השפה, התחלתי לכתוב את ספר הפרויקט אותו אתם קוראים עכשו. זאת על מנת להבין מהי האסטרטגיה, האלגוריתמיים, ומבנה הנתונים בהם אני הולך להשתמש בפרויקט.

כעת עברו יותר מחודשים מתחילה העבודה על הפרויקט, ועוד לא כתבתי שורת קווד אחת.

עכשו, חשש בידי את האסטרטגיה, התחלתי לכתוב את קווד. הרבה קווד. זה הפרויקט בסדר הגודל הכלי גדול שכתבתAi פעם, ועוד כתבתAi אותו בשפת C שלמדתי רק השנה. את את התחלתי לפתור את בעיות השונות העולות בפרויקט זה, וכעת, בסופו של דבר, יש בידי מוצר מוגמר שאני גאה בו מאוד.

אתגרים

במהלך הפרויקט נתקلت במספר אתגרים. הן תאורטיים ואלגוריתמיים, והן תכונתיים ומעשיים.

בצד התאורטי האתגר המרכזי היה כמויות החומר העצומות שהייתי צריך למדוד. אני אוהב למדוד דברים באופן עצמאי, ואני גם מאד טוב בהזה, אך אלו היו כמויות עצומות של חומר מורכב יחסית שהייתי צריך למדוד בזמן קצר. אך בסופו של דבר, אני חשב שצלחתית את המשימה על הצד הטוב ביותר.

בצד התכונתי והמעשי גם כן היו אתגרים.

ראשית, חלוקת הקוד למודולים נכונים ויעילים על מנת לאפשר מודולריות של הקוד, ביצוע שינויים, ניהול זיכרון נכון, ויכולות דיבוג קלות שהיא משמשה לא כללה, שחשבתי עליה ריבות גם לפני שהתחלה לכתוב את הקוד, וגם תוך כדי כתיבתו, והיא דבר שהשתנה במהלך הקוד. החלוקה הסופית למודולים בה בחרתי, מאפשר מודולריות, קריאות, ניהול, ודיבוג קל ונוח של הפרויקט, אך שאמ ארצה, יהיה לי קל מאוד להמשיך לעובוד עלייו ולהוסיף לו דברים.

שנייה, הבנת התאוריה היא דבר אחד ומימושה בקוד הוא דבר אחר. חלק מהמושגים והאלגוריתמים שהבנתי באופן תאורטי לא תמיד היו כל כך פשוטים לימייש, ולפעמים היו חסורות לי פיסות של מידע על מנת להבין באופן מלא את הנושא ולמש את התאוריה. מספר פעמים במהלך כתיבת הקוד נתקלת בבעיה שהרגשת שלא אצליח לפטור, אז ישbst, למدت, חקרתי וקרأت עוז בעיה זו, הלמידה חיבור לחלקם רבים לכדי תמונה מלאה אחת, וכך מספר שעות לאחר מכן, הבעיה נפתרה בקלהות. אחת מן ההרגשות הטובות שיש.

ניהול הפרויקט

אני חשב שהנהלותי בתהליכי הבניה של הפרויקט הייתה מעולמת. עבדתי עם המון סבלנות להבנת התאוריה הראשית וdagiti לחשבון ולא לזכור פינוט, הן בהבנת התאוריה והן בכתיבה קוד נקי, מסודר ומתועד. אני חשב שהבחירה להיבין לעומק את התאוריה קודם ורק לאחר מכן להתחיל לכתוב את הקוד, הייתה הבחירה הכי טובה שיכלתי לעשות. אני לא יכול לחשב על כמה זמן שהייתי צריך להשיק מעבר למה שכבר השקעת, אם הייתה מתחילה לכתוב קוד ישר בלי למדוד קודם קודם את התאוריה.

כלים שקיבנתי ואכח איתני להמשך

אחד הכלים שאני לוקח לעצמי מהפרויקט זהה הוא ההבנה שלמידה עצמית היא אחד הכלים הכי חזקים שיש לי. עם השקעה, עקבות וצען, אני מסוגל ללמוד כל דבר שאראה. דבר זה בולט מאוד במהלך הפרויקט בכך שלפעמים הייתי צריך ללמוד נושא מסוים, שתחילתו הוא נראה לי לא מובן, כמו פאצל עם הרבה חלקיים חסרים. אך עם הזמן הבנתי שהלמידה העצמית לוחחת חלקים ממוקורות שונים ומחברת את הפעzel לתוצר שלם, וגורמת להבנה מלאה ועמוקה של הנושא. ההבנה הזאת, היא אחת הרתומות המספקות שיש, כי היא גורמת לך להבין כמה כוח יש לך בידים.

עוד כלי שאני לוקח לעצמי הוא יכולת לנוהל פרויקט בסדר גדול כזה. לפני בניית הקומפיילר לא התנהלתה בפרויקט כזה, הדורש חקוק ולמידה עצמית רבה כל טרם כתיבת הקוד. אני חשב שכעת אוכל לנוהל פרויקטים בסדרי גודל משמעותית גדולים יותר.

אילו הייתה מתחילה את הפרויקט היום

אם הייתה מתחילה היום את הפרויקט, ללא הידע שיש לי, כתעת בנושא, הייתה עשויה כמעט בדיקות אוטומטיות. הייתה מתחילה למדוד את כל התאוריה קודם, עשויה סדר במבני הנתונים, האלגוריתמים והאסטרטגיה בה אנקוט על מנת לבצע את הפרויקט, ורק לאחר מכן הייתה מתחילה לכתוב את הקוד. מה שכנ היהי משנה הוא כמה תיעוד וסידור הקוד, והפרפקציוניזם בכתיבתה שלו. אני חשב שבחזצתי קצת יותר מדי זמן על הኒקון, הסידור והתיעוד של הקוד.

אילו הייתי יכול להמשיך לעבוד על הפרויקט

אם הייתי יכול להמשיך מהנקודה הנוכחית, ללא הגבלת זמן, הייתי מוסיף עוד פיצ'רים לשפה של'. אחד מהפיצ'רים החשובים בעיני הוא הוספת מערכים. כר השפה תהיה Turing complete, ותוכל לפתור כל בעיה חישובית. בנוסף, הייתה רחזה להוסיף לשפה פונקציות, מכיוון שהן אפשרות קוד נקי יותר, מנועות שכפול קוד, ומגבירות את הקוריאות של'. כמו כן, אני חשב שהוספה של **Structs** לשפה תהיה דבר עצמתי הפותח דלת לקראת תכונות מונחה עצמים.

השגת מטרות

במبدأ לספר זה הצבתי לעצמי מספר מטרות, ואני חשב שהשגתן את כולם.לקחתי על עצמי אתגר ופיתחת את עצמי, רכשתי ידע בתחוםים שלא התעסקתי בהם בעבר, הבנתי את התאוריה לעומקה ואיך הדברים עובדים באמת, פיתחת אלגוריתמים יעילים תוך שימוש במבני נתונים שונים על מנת לפתור את הבעיות האלגוריתמיות השונות העולות בפרויקט זה, ופיתחת את יכולתי בניהול ועבודה על פרויקטים.

עוד מטרה שהציבתי לעצמי כחלק מכתיבת ספר הפרויקט, היא לאפשר לקרוא שלו להבין איך קומפיאלים עובדים באמת, מה הידע הנחוץ לו על מנת לכתוב קומפיאילר, ומהן הגישות והאסטרטגיות השונות לפתרון הבעיה. אני באמת ממש שלאחר קריאה עמוקה של ספר הפרויקט, כל אחד יוכל לכתוב קומפיאילר משלו, גם אם אין לו שום רקע בנושא. היה לי חשוב לסקם את כל הידע שרכשתי במהלך העבודה על הפרויקט במקומ אחד, כך שאם בעתיד ארצה להבין פעם נוספת את הדברים שלמדתי ומימשתי בפרויקט זה, אוכל לעשות זאת ב יתר קלות.

סיכום

לסיכום, אני מאוד גאה בפרויקט הזה ובעצמי. שמחתי ותרגשתי לעבוד עליו, והוא לימד אותי המונע על עצמי ופתח אותי אישית.

אני שמח וגאה על כך שבחרתי לחת על עצמי אתגר קשה וצלחתי אותו. אני רואה זאת כהצלחה אישית אשר לימהה אותי המונע וחיזקה אותי לעתיד.

זהו עד צעד קטן במסע שלי... 😊

ביבליוגרפיה

- Atwood, J. & Spolsky, J. (2008, Sep). **Stack Overflow**. Retrieved from <https://stackoverflow.com>
- Bergman, S. D. (2017, May). **Compiler Design**. Retrieved from <https://bit.ly/3vKw55o>
- Dancis, K., & Schoeffel, S. (2017, July). **Whimsical**. Retrieved from <https://whimsical.com>
- Darveshi, R. S. (2011, November). **E.C.E**. Retrieved from Youtube: <https://bit.ly/3vHxR7o>
- Grandinetti, P. (2019, October). **Compilers**. Retrieved from pgrandinetti: <https://bit.ly/3MpMEtm>
- Hoyt, B. (2021, March). **Implement Hash Table in C**. Retrieved from <https://bit.ly/3u0a0xs>
- Hsu, T. s. (1996, Fall). **Symbol Table**. Retrieved from Academia Sinica: <https://bit.ly/3KlsKxA>
- Sanger, L. (2001, January). **Wikipedia**. Retrieved from <https://www.wikipedia.org>
- Simonson, S. (2010, May). **Automata Theory**. Retrieved from Youtube: <https://bit.ly/3HHEiJR>
- Simonson, S. (2010, May). **Theory of Computation**. Retrieved from Youtube: <https://bit.ly/3pGrM7g>
- Simonson, S. (2017, January). **Algorithms**. Retrieved from Youtube: <https://bit.ly/3q0SxDV>
- Soshnikov, D. (2021, April). **Building a Parser**. Retrieved from Youtube: <https://bit.ly/3IRKJM8>
- Thain, D. (2017, Fall). **Code Generation**. Retrieved from Youtube: <https://bit.ly/3HZLb9T>
- Tutorialspoint. (2015). **Learn Compiler Design**. Retrieved from Tutorialspoint: <https://bit.ly/35pWuLi>
- Uncode. (2013, July). **Uncode**. Retrieved from Youtube: <https://bit.ly/3vGe9c4>

קוד הפרויקט

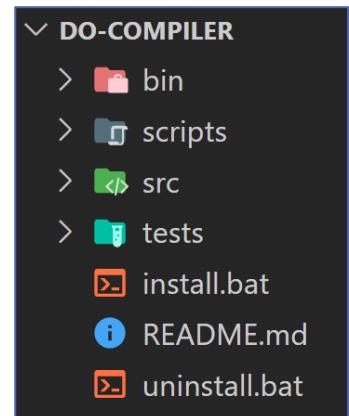
[GitHub](#)

הלאן קישור לקוד הפרויקט ב – GitHub : <https://github.com/ido-hi/do-compiler.git>

מבנה קוד הפרויקט

cutת אסביר את מבנה הקוד של הפרויקט. מה המטרה של כל תיקייה ומה מכילה, וארחיב על קבצים חשובים.

מבנה תיוקית הפרויקט

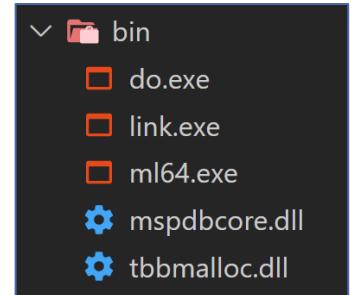


קבצים

- install.bat
 - מתקין את הקומpileר על המחשב.
 - מוסיף את ה – Path לתיקיית bin למשתנה הסביבה PATH של המשתמש, כך תוכל להיות גישה לקומpileר ולאסמבלר מכל מקום!
- uninstall.bat
 - מסיר את הקומpileר מהמחשב.
 - מסיר את Path לתיקיית bin ממשתנה הסביבה PATH של המשתמש.
 - מוחק את התיקייה של הפרויקט מהמחשב.
- README.md
 - קובץ המסביר על הפרויקט. מה מכיל, איך להתקין / להסיר אותו, איך להשתמש בו.

bin

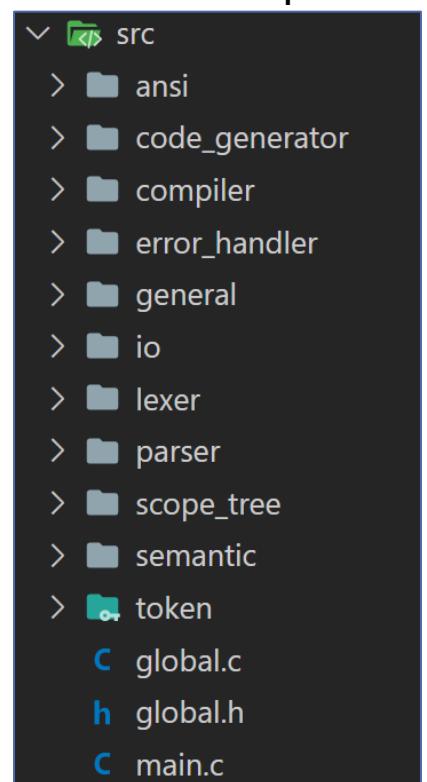
מכליה קבצים בינאריים הנחוצים לשימוש בקומפילר.

מבנה התיקייה**קבצים**

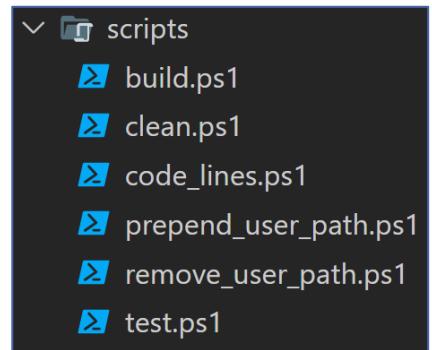
- do.exe •
- הקומפילר! ○
- ml64.exe •
- x64 ○
- link.exe •
- הLINKEr ש – exe ml64.exe משתמש בו. ○
- שני ה – dll האחרים נחוצים עבור exe ml64.exe, על מנת שיוכל לבצע את פעולתו כשרה. •

src

מכליה את קוד הפרויקט. כל קבצי המקור בשפת C (.h, .c) נמצאים כאן.

מבנה התיקייה

scripts
מכליה סקריפטים לעזרה בבנייה, הריצה, בדיקה וניהול של הפרויקט.

מבנה התקינה**קבצים**

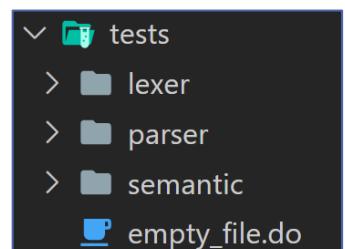
- **build.ps1**
 - מكمפל את כל קבצי הקוד (h, c) של הפרויקט, ויוצר את הקובץ exe.do בתיקייה bin.
- **test.ps1**
 - בונה את הקומפיאר ומריץ את כל הטעיטים שבתיקייה tests כנגד הקומפיאר. בסוף מציג סיכום של תוצאות הבדיקות.
- **prepend_user_path.ps1**
 - מוסיף את ה – Path לתיקיית bin למשתנה הסביבה PATH של המשתמש.
- **remove_user_path.ps1**
 - מסיר את ה – Path לתיקיית bin ממשתנה הסביבה PATH של המשתמש.
- **clean.ps1**
 - מסיר את הקובץ exe.do מהתיקייה bin, ומוחק את כל קבצי האסמבלי בתיקיית הפרויקט (אם קיימים).
- **code_lines.ps1**
 - מציג כמה שורות קוד יש בפרויקט.

tests

מכליה קבצי do עבור בדיקה של הקומפיאר.

הקבצים אמורים להיכשל עם כל הערות השגיאה האפשרות שהקומפיאר אמור לטעות. test נחשב כהצלחה אם הוא נכשל עם השגיאה שהוא אמור לטעות.

הקובץ test.ps1 מרים כנגד הקומפיאר את כל הבדיקות שנמצאות בתיקייה זו, ובסוף מציג אילו בדיקות עברו ואילו נכשלו.

מבנה התקינה

קוד

כלuproject מורכב מ – **6098** שורות קוד. מצורפים אליו הקוד הפעילים, כולל, הסקריפטים והקבצים בשפת C.

install.bat

```
@echo off

:: Add the bin directory path to the user's PATH environment variable
cd scripts
powershell .\prepend_user_path.ps1
cd ..

:: Add the bin directory path to the current session PATH
set PATH=%~dp0bin;%PATH%

:: Clear the screen
cls
```

uninstall.bat

```
@echo off

:: Remove the bin directory path from the user's PATH environment variable
cd scripts
powershell .\remove_user_path.ps1
cd ..

:: Delete the project
cd ..
cls
goto 2>nul & rmdir /s /q %~dp0
```

[scripts](#)[build.ps1](#)

```
# Script to compile all the source files of the project #

Clear-Host

# Get all the source files
$source_files = Get-ChildItem ..\src -Recurse -Include *.c

# Check if source files exist, if so compile
if (Test-Path -Path $source_files) {
    gcc $source_files -o ..\bin\do.exe
}
# If does not exist, print error message
else {
    Write-Host "Couldn't find source files" -ForegroundColor red
}
```

[clean.ps1](#)

```
# Script to delete the executable of the project #

# Delete the executable and .asm files
Remove-Item ..\bin\do.exe
Remove-Item ..\*.asm

Clear-Host
```

[code_lines.ps1](#)

```
# Script to count how many lines of code there are in the project #

Clear-Host

Write-Host "# of code lines in the project: " -NoNewline
Write-Host (Get-ChildItem .. -Include ('*.c', '*.h', '*.ps1', '*.bat', '*.do',
'*.md') -Recurse | Get-Content).Length -ForegroundColor Green
```

```
prepend_user_path.ps1
```

```
# Script to prepend the path to the bin directory to the PATH environment variable #

# Get the path to the bin directory of the project
$bin_path = (Split-Path -Path $(Get-Location) -Parent) + "\bin"

# Get the user's PATH environment variable from the registry
$user_path = (Get-ItemProperty -path HKCU:\Environment\ -Name Path).Path

# Set the user's PATH to be the directory to the bin directory + the previous path
Set-ItemProperty -path HKCU:\Environment\ -Name Path -Value "$bin_path;$user_path"
```

```
remove_user_path.ps1
```

```
# Script to remove the path to the bin directory from the PATH environment variable

# Get the path to the bin directory of the project
$bin_path = (Split-Path -Path $(Get-Location) -Parent) + "\bin"

# Get the user's PATH environment variable from the registry
$user_path = (Get-ItemProperty -path HKCU:\Environment\ -Name Path).Path

# Get the user's PATH without the bin directory's path
$user_path = ($user_path.Split(';') | Where-Object { $_ -ne $bin_path }) -join ','

# Set the user's PATH to be the previous path - the directory to the bin directory
Set-ItemProperty -path HKCU:\Environment\ -Name Path -Value $user_path
```

test.ps1

```
# Script to test the code. All the tests are from the ../tests folder. A test is
passed if the exit code of it is NOT 0 #

# Build

# Build only if doesn't exist already
if (!(Test-Path -Path ..\bin\do.exe)) {
    & .\build.ps1
}

# Test

# Check if cli arguments exist and the path specified exists
if (($args.length -gt 0) -and (Test-Path -Path $args[0])) {
    # Set the test path to the path specified
    $test_path = $args[0]
}
# If either is false, set ..\tests as the test path
else {
    $test_path = "..\tests"
}

$passed_tests = @()
$failed_tests = @()

# Run every test in the tests folder
foreach ($file_path in $(Get-ChildItem $test_path -Include '*.do' -Recurse |
Resolve-Path -Relative)) {
    # Run test
    Write-Host
    $file_path
    ..\bin\do.exe $file_path ..\example.asm

    # If passed test, add to $passed_tests array
    # -ne 0 because the test is checking for error catching, and the error code will
not be 0 in that casetests
    # -or the path starts with V because a test that starts with V should return
exit code 0
    if ((($LASTEXITCODE -ne 0) -or $($Split-Path $file_path -Leaf).StartsWith('V')) {
        $passed_tests += Split-Path $file_path -Leaf
    }
}
```

```
# If passed test, add to $failed_tests array
else {
    $failed_tests += Split-Path $file_path -Leaf
}

# Output tests results
# Prints passed tests
foreach ($file_path in $passed_tests) {
    Write-Host "V " -ForegroundColor green -NoNewline
    Write-Host $file_path -NoNewline
    Write-Host " passed" -ForegroundColor green
}
# Prints failed tests
foreach ($file_path in $failed_tests) {
    Write-Host "X " -ForegroundColor red -NoNewline
    Write-Host $file_path -NoNewline
    Write-Host " failed" -ForegroundColor red
}
# Print summary results
Write-Host
Write-Host $passed_tests.length -ForegroundColor green -NoNewline
Write-Host " / " -NoNewline
Write-Host $($passed_tests.length + $failed_tests.length) -ForegroundColor yellow -
NoNewline
Write-Host " tests passed successfully"
```

src

main.c

```
#include "general/general.h"
#include "compiler/compiler.h"
#include "ansi/ansi.h"

int main(int argc, char* argv[])
{
    // Make the program support the ANSI escape codes
    ansi_setup_console();

    // If no command line arguments were specified
    if (argc < 2)
    {
        printf(BOLD_WHITE "do: " RESET RED "fatal error:" RESET " no input
files\n");
        printf("\tusage: " BOLD_WHITE "do <source.do> [<destination.asm>]\n" RESET);
        return 1;
    }

    // If there are 2 or more arguments, initialize the compiler with the first two
    if (argc >= 3)
        compiler_init(argv[1], argv[2]);

    // If there is 1 argument, initialize the compiler with it as the source and with
    // ./a.asm as the destination
    else
        compiler_init(argv[1], "a.asm");

    // Compile the source code
    compiler_compile();

    // Destroy the compiler
    compiler_destroy();

    return 0;
}
```

global.h

```
#pragma once

#include "compiler/compiler.h"

/* ----- Global variables ----- */

// In that file all the global variables are extern, so every file that includes them
// will be able to use them but not to define them.
// The global variables should only be defined once in the whole project,
// that definition is in the global.c file.

// The global project compiler
extern Compiler compiler;
```

global.c

```
#include "compiler/compiler.h"

/* ----- Global variables ----- */

// Here is the only definition of all of the global variables of the project.

// The global project compiler
Compiler compiler;
```

ansi

ansi.h

```
#pragma once

// The ansi.h and ansi.c files were created to add support for the ANSI escape
// codes.
// This enables printing in different colors in the console.
// Taken from here: https://youtu.be/bQ8qaBjJtYU

/* ----- Functions ----- */

// Set up the console to support the ANSI escape codes,
// so we can print in different colors in the console.
void ansi_setup_console();
```

ansi.c

```
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>

#include "ansi.h"

void ansi_setup_console()
{
    DWORD outMode = 0;
    HANDLE stdoutHandle = GetStdHandle(STD_OUTPUT_HANDLE);

    if (stdoutHandle == INVALID_HANDLE_VALUE)
        exit(GetLastError());

    if (!GetConsoleMode(stdoutHandle, &outMode))
        exit(GetLastError());

    // Enable ANSI escape codes for Windows console
    outMode |= 0x0004;

    if (!SetConsoleMode(stdoutHandle, outMode))
        exit(GetLastError());
}
```

code_generator

code_generator_base.h

```
#pragma once

#include "../lexer/lexer.h"

/* ----- Registers ----- */

// The initial value of the register number for the different tree nodes
#define NO_REGISTER -1

// The number of registers the code generator uses
#define NUM_OF_REGISTERS 6

// The maximum length of register's name
#define REGISTER_NAME_LENGTH 8

// The registers used for code generation
#define R10 "r10"
#define R11 "r11"
#define R12 "r12"
#define R13 "r13"
#define R14 "r14"
#define R15 "r15"

// The registers used for some arithmetic operations (because they're the default
// result registers)
#define RAX "rax"
#define RDX "rdx"

/* ----- Labels ----- */

// The format of a label.
// Expecting label number for the %d.
#define LABEL_FORMAT "$$L%d"

// The maximum length of labels's name
#define LABEL_NAME_LENGTH 8
```

```

/* ----- Symbols Addresses ----- */

// The format of an address in the data segment (global variables).
// Added $ at the start of the address to solve name collisions for global
variables.
// Expecting the identifier name for the %s.
#define GLOBAL_ADDRESS_FORMAT "$%s"

// The format of an address on the stack (relative to rbp).
// Expecting the relative address off of rbp for the %d.
#define STACK_ADDRESS_FORMAT "[rbp - %d]"

// The maximum length of symbol address. It is equal to the maximum length of
// a token because the longest address that will be produced will be the length of
// a long identifier.
#define SYMBOL_ADDRESS_LENGTH LEXER_MAX_TOKEN_SIZE

// The size of an entry in the stack of the program.
// For x64 the size of a stack entry is 8 bytes
#define STACK_ENTRY_BYTES 8

/* ----- Instructions ----- */

#define MOV      "\tmov %s, %s\n"

// Control flow
#define CMP      "\tcmp %s, %s\n"
#define LABEL    "%s:\n"
#define JMP      "\tjmp %s\n"
#define JE       "\tje %s\n"
#define JNE     "\tjne %s\n"
#define JG       "\tjg %s\n"
#define JGE      "\tjge %s\n"
#define JL       "\tjl %s\n"
#define JLE      "\tjle %s\n"
#define JC       "\tjc %s\n"
#define JNC     "\tjnc %s\n"

// Arithmetic
#define ADD      "\tadd %s, %s\n"
#define SUB      "\tsub %s, %s\n"
#define SBB      "\tsbb %s, %s\n"

```

```
#define IMUL    "\timul %s, %s\n"
#define IDIV    "\tidiv %s\n"
#define NEG     "\tneg  %s\n"

/* ----- Data types ----- */

#define DB " db ?\n"
#define DQ " dq ?\n"
```

code_generator.h

```
#pragma once

#include <stdio.h>
#include <stdbool.h>

#include "code_generator_base.h"
#include "../parser/parse_tree/parse_tree.h"

/* ----- Structs ----- */

// Struct of a register in the code generator's register array
typedef struct Register
{
    char name[REGISTER_NAME_LENGTH];           // The name of the register for target
    code output;                                // Whether the register is currently in
    bool inuse;                                 // use or not
} Register;

// Struct of the code generator
typedef struct Code_Generator
{
    Register registers[NUM_OF_REGISTERS];      // Array of registers to be used in the
    code generation process
    FILE* dest_file;                          // A pointer to the output file for the
    generated code
} Code_Generator;

/* ----- Functions ----- */
```

```
// Create a new code generator and points the compiler's code generator to it
void code_generator_create();

// Frees the memory allocated for the code generator of the compiler
void code_generator_destroy();

// Initializes the compiler's code generator
void code_generator_init();

// Searches for a free register in the code generator registers array.
// If found, marks it as inuse and returns the index of that register.
// If not found, terminates the compiler.
int code_generator_register_alloc();

// Marks the register in index r in the code generator registers array as unused
void code_generator_register_free(int r);

// Returns the register name of register r in the code generator registers array
char* code_generator_register_name(int r);

// Allocates a new lable name and returns a pointer to it
char* code_generator_label_create();

// Performs the right address computation for the given symbol according to its
// place in the program (global / local),
// and returns a string that represents that address.
// If the entry is NULL, returns NULL.
char* code_generator_symbol_address(Symbol_Table_Entry* entry);

// Outputs the given formated string to the target file
void code_generator_output(char* format, ...);

// Outputs the data segment of the program to the target file
void code_generator_output_data_segment();

// Generates the assembly code for the given parse tree
void code_generator_generate(Parse_Tree_Node* parse_tree);

// Generates a block (BLOCK)
void code_generator_block(Parse_Tree_Node* block);
```

```
// Generates a statement (STMT)
void code_generator_stmt(Parse_Tree_Node* stmt);

// Generates a variable declaration statement (DECL)
void code_generator_decl(Parse_Tree_Node* decl);

// Generates an assignment statement (ASSIGN)
void code_generator_assign(Parse_Tree_Node* assign);

// Generates an if else statement (IF_ELSE)
void code_generator_if_else(Parse_Tree_Node* if_else);

// Generates an else statement (ELSE)
void code_generator_else(Parse_Tree_Node* _else);

// Generates a while statement (WHILE)
void code_generator_while(Parse_Tree_Node* _while);

// Generates an expression
void code_generator_expression(Parse_Tree_Node* expr);

// Generates a binary expression of the form: Expr -> Expr binary_operator Expr
void code_generator_binary_expression(int left_register, int right_register,
Token_Type operator);

// Generates an unary expression of the form: Expr -> unary_operator Expr
void code_generator_unary_expression(int register_number, Token_Type operator);

// Generates a boolean expression of the form: Expr -> bool_operator Expr
// according to the given operator
void code_generator_bool_e(int left_register, int right_register, Token_Type
operator);

// Generates target code that moves the given token to the given register.
// Moves the token according to its type (identifier / literal)
void code_generator_mov_token(int register_number, Token* token);

// Generates target code that moves the given identifier token to the given
register.
void code_generator_mov_identifier(int register_number, Token* token);

// Generates logical OR and puts the result (0 | 1) in the left register
```

```

void code_generator_or(int left_register, int right_register);

// Generates logical AND and puts the result (0 | 1) in the left register
void code_generator_and(int left_register, int right_register);

// Generates division operation between left and right registers. left_register =
left_register / right_register,
void code_generator_divide(int left_register, int right_register);

// Generates modulu operation between left and right registers. left_register =
left_register % right_register,
void code_generator_modulu(int left_register, int right_register);

// Generate logical NOT and puts the result (0 | 1) back in the register
void code_generator_not(int register_number);

```

code_generator.c

```

#include <stdlib.h>
#include <string.h>
#include <stdarg.h>

#include "../global.h"

#include "code_generator.h"
#include "../general/general.h"
#include "../compiler/compiler.h"
#include "../io/io.h"

void code_generator_create()
{
    compiler.code_generator = (Code_Generator*) calloc(1, sizeof(Code_Generator));
    if (compiler.code_generator == NULL) exit_memory_error(__FILE__, __LINE__);
}

void code_generator_destroy()
{
    if (compiler.code_generator != NULL)
    {
        // Close the destination file
        fclose(compiler.code_generator->dest_file);
    }
}

```

```

        free(compiler.code_generator);
        compiler.code_generator = NULL;
    }
}

void code_generator_init()
{
    // - Open destination file
    compiler.code_generator->dest_file = fopen(compiler.dest_file_name, "w");
    if (compiler.code_generator->dest_file == NULL)
exit_file_io_error(compiler.dest_file_name);

    // - Init registers array
    int r = 0;

    strncpy(compiler.code_generator->registers[r].name, R10, REGISTER_NAME_LENGTH);
    compiler.code_generator->registers[r++].inuse = false;

    strncpy(compiler.code_generator->registers[r].name, R11, REGISTER_NAME_LENGTH);
    compiler.code_generator->registers[r++].inuse = false;

    strncpy(compiler.code_generator->registers[r].name, R12, REGISTER_NAME_LENGTH);
    compiler.code_generator->registers[r++].inuse = false;

    strncpy(compiler.code_generator->registers[r].name, R13, REGISTER_NAME_LENGTH);
    compiler.code_generator->registers[r++].inuse = false;

    strncpy(compiler.code_generator->registers[r].name, R14, REGISTER_NAME_LENGTH);
    compiler.code_generator->registers[r++].inuse = false;

    strncpy(compiler.code_generator->registers[r].name, R15, REGISTER_NAME_LENGTH);
    compiler.code_generator->registers[r++].inuse = false;
}

int code_generator_register_alloc()
{
    for (int r = 0; r < NUM_OF_REGISTERS; r++)
    {
        // If registes is not inuse, return its index
        if (!compiler.code_generator->registers[r].inuse)
        {

```

```
        compiler.code_generator->registers[r].inuse = true;
        return r;
    }

// If couldn't find a free register
printf(RED "Couldn't find a free register" RESET);
compiler_destroy();
remove(compiler.dest_file_name);
exit(1);
}

void code_generator_register_free(int r)
{
    compiler.code_generator->registers[r].inuse = false;
}

char* code_generator_register_name(int r)
{
    return compiler.code_generator->registers[r].name;
}

char* code_generator_label_create()
{
    // A static integer to create unique labels
    static int label_num = 0;

    // Allocate memory for the label
    char* label = (char*) calloc(LABEL_NAME_LENGTH, sizeof(char));
    if (label == NULL) exit_memory_error(__FILE__, __LINE__);

    // Create label
    sprintf(label, LABEL_FORMAT, label_num);

    // Increment static int for next label
    label_num++;

    // Return the new label
    return label;
}

char* code_generator_symbol_address(Symbol_Table_Entry* entry)
```

```
{  
    if (entry == NULL)  
        return NULL;  
  
    // Static string for the address (to prevent allocation each time we need a  
symbol)  
    static char address[SYMBOL_ADDRESS_LENGTH] = { 0 };  
  
    // - If the entry is global, just return the identifier  
    if (entry->is_global)  
    {  
        sprintf(address, GLOBAL_ADDRESS_FORMAT, entry->identifier);  
        return address;  
    }  
  
    // - If the entry is local, compute the appropriate address of that symbol  
  
    // The offset of the symbol is:  
    // ((number of entries seen until the parent scope) + (entry's order in its  
symbol table)) * (the size of an entry in the stack)  
    // For example:  
    //      first symbol will be: 8 * (0 + 1) = 8 bytes -> [rbp - 8]  
    //      second symbol will be: 8 * (0 + 2) = 16 bytes -> [rbp - 16]  
    //      ...  
    int bp_offset = STACK_ENTRY_BYTES * (entry->scope->parent->available_entries +  
entry->num_in_scope);  
  
    // Create the computed stack address for the symbol  
    if (entry->data_type == Data_Type_Char)  
        sprintf(address, "byte ptr " STACK_ADDRESS_FORMAT, bp_offset);  
    else  
        sprintf(address, "qword ptr " STACK_ADDRESS_FORMAT, bp_offset);  
  
    // Return the computed address  
    return address;  
}  
  
void code_generator_output(char* format, ...)  
{  
    va_list args;                                // Declare a va_list  
type variable
```

```

    va_start(args, format);                                // Initialize the
va_list with the ...
    vfprintf(compiler.code_generator->dest_file, format, args); // Forward ... to
vfprintf
    va_end(args);                                         // Clean up the
va_list
}

void code_generator_output_data_segment()
{
    Symbol_Table* global_table = compiler.scope_tree->global_scope->symbol_table;

    code_generator_output(".data\n");

    for (int i = 0; i < global_table->capacity; i++)
    {
        Symbol_Table_Entry* entry = global_table->entries[i];
        while (entry != NULL)
        {
            code_generator_output("\t" GLOBAL_ADDRESS_FORMAT, entry->identifier);
            code_generator_output(entry->data_type == Data_Type_Char ? DB : DQ);
            entry = entry->next_entry;
        }
    }

    code_generator_output("\n");
}

void code_generator_generate(Parse_Tree_Node* parse_tree)
{
    // PROG -> prog id : BLOCK :

    // Reset scope tree's scopes current_child_index to be STARTING_CHILD_INDEX
before starting
    // the code generation process, so that the usage of the scope tree will be the
same as the parsing phase.
    scope_tree_reset_child_index(compiler.scope_tree->global_scope);

    // - Generate the data segment of the program
    code_generator_output_data_segment();

    // - Generate the code segment of the program
}

```

```

code_generator_output(".code\n");
code_generator_output("main proc\n");
code_generator_output("\tpush rbp\n");
code_generator_output("\tmov rbp, rsp\n");
code_generator_output("\tsub rsp, 1000h\n\n");

// Generate code for the main program block
code_generator_block(parse_tree->children[3]);

code_generator_output("\n\tmov rsp, rbp\n");
code_generator_output("\tpop rbp\n");
code_generator_output("\tmov rax, 0\n");
code_generator_output("\tret\n");
code_generator_output("main endp\n");
code_generator_output("end\n");

}

void code_generator_block(Parse_Tree_Node* block)
{
    // BLOCK -> done
    if (block->num_of_children == 1)
        // Go to the parent scope when exiting a block
        scope_tree_goto_parent();

    // BLOCK -> STMT BLOCK
    else
    {
        // Generate the statement
        code_generator_stmt(block->children[0]);

        // Generate the block
        code_generator_block(block->children[1]);
    }
}

void code_generator_stmt(Parse_Tree_Node* stmt)
{
    switch (stmt->children[0]->symbol)
    {
        case Non_Terminal_DECL:
            // STMT -> DECL
            code_generator_decl(stmt->children[0]);
    }
}

```

```
        break;

    case Non_Terminal_ASSIGN:
        // STMT -> ASSIGN
        code_generator_assign(stmt->children[0]);
        break;

    case Non_Terminal_IF_ELSE:
        // STMT -> IF_ELSE
        code_generator_if_else(stmt->children[0]);
        break;

    case Non_Terminal WHILE:
        // STMT -> WHILE
        code_generator_while(stmt->children[0]);
        break;
    }

}

void code_generator_decl(Parse_Tree_Node* decl)
{
    // DECL -> data_type id ;

    // If the current scope is not the global scope, add 1 to the scope's available
    entries
    // to help later calculate the address of the currently declared variable.
    // I don't want to count the global variables because they will be on the data
    segment
    // of the program and not the stack segment.
    if (compiler.scope_tree->current_scope != compiler.scope_tree->global_scope)
        compiler.scope_tree->current_scope->available_entries++;

}

void code_generator_assign(Parse_Tree_Node* assign)
{
    // ASSIGN -> set id = L_LOG_E ;

    // Calculate the expression
    code_generator_expression(assign->children[3]);

    // Assign the expression value to the variable
    Symbol_Table_Entry* entry = scope_tree_fetch(assign->children[1]->token->value);
```

```
if (entry->data_type == Data_Type_Char)
    code_generator_output("\tmov %s, %sb\n",
code_generator_symbol_address(entry), code_generator_register_name(assign-
>children[3]->register_number));
else
    code_generator_output(MOV, code_generator_symbol_address(entry),
code_generator_register_name(assign->children[3]->register_number));

// Free result register because we don't need it anymore
code_generator_register_free(assign->children[3]->register_number);
}

void code_generator_if_else(Parse_Tree_Node* if_else)
{
    // IF_ELSE -> if ( L_LOG_E ) : BLOCK ELSE

    // Go to next children when entering a new scope
    scope_tree_goto_child();

    // Update the child to have the same number of available entries as the current
scope
    compiler.scope_tree->current_scope->available_entries = compiler.scope_tree-
>current_scope->parent->available_entries;

    char* else_label = code_generator_label_create();
    char* done_label = code_generator_label_create();

    // Generate the expression
    code_generator_expression(if_else->children[2]);

    // Check if condition
    code_generator_output(CMP, code_generator_register_name(if_else->children[2]-
>register_number), "0");
    code_generator_output(JE, else_label);

    // Free expression register because we don't need it anymore
    code_generator_register_free(if_else->children[2]->register_number);

    // Generate if block
    code_generator_block(if_else->children[5]);
    code_generator_output(JMP, done_label);
```

```
// Generate else block
code_generator_output(LABEL, else_label);
code_generator_else(if_else->children[6]);

code_generator_output(LABEL, done_label);

// Free labels
free(else_label);
free(done_label);
}

void code_generator_else(Parse_Tree_Node* _else)
{
    // ELSE -> epsilon
    if (_else->num_of_children == 0)
        return;

    // ELSE -> else : BLOCK
    else
    {
        // Go to next children when entering a new scope
        scope_tree_goto_child();

        // Update the child to have the same number of available entries as the
        current_scope
        compiler.scope_tree->current_scope->available_entries = compiler.scope_tree-
>current_scope->parent->available_entries;

        // Generate else block
        code_generator_block(_else->children[2]);
    }
}

void code_generator_while(Parse_Tree_Node* _while)
{
    // WHILE -> while ( L_LOG_E ) : BLOCK

    // Go to next children when entering a new scope
    scope_tree_goto_child();
```

```

    // Update the child to have the same number of available entries as the current
    // scope
    compiler.scope_tree->current_scope->available_entries = compiler.scope_tree-
    >current_scope->parent->available_entries;

    char* while_label = code_generator_label_create();
    char* done_label = code_generator_label_create();

    code_generator_output(LABEL, while_label);

    // Generate the expression
    code_generator_expression(_while->children[2]);

    // Check while condition
    code_generator_output(CMP, code_generator_register_name(_while->children[2]-
    >register_number), "0");
    code_generator_output(JE, done_label);

    // Free expression register because we don't need it anymore
    code_generator_register_free(_while->children[2]->register_number);

    // Generate while block
    code_generator_block(_while->children[5]);
    code_generator_output(JMP, while_label);

    code_generator_output(LABEL, done_label);

    // Free labels
    free(while_label);
    free(done_label);
}

void code_generator_expression(Parse_Tree_Node* expr)
{
    // Terminal
    if (expr->num_of_children == 0)
    {
        // Allocate register for the terminal
        expr->register_number = code_generator_register_alloc();

        // Move the terminal to the allocated register
        code_generator_mov_token(expr->register_number, expr->token);
    }
}

```

```

}

// Unary expression
// In the format: Expr -> operator Expr
else if (expr->num_of_children == 2)
{
    // Expr -> unary_operator Terminal
    if (expr->children[1]->symbol_type == Terminal)
    {
        // Allocate register for the terminal operand
        int register_number = code_generator_register_alloc();

        // Move the terminal to the allocated register
        code_generator_mov_token(register_number, expr->children[1]->token);

        // Generate code according to the unary operator
        code_generator_unary_expression(register_number, expr->children[0]-
>token->token_type);

        // Preserve the register
        expr->register_number = register_number;
    }

    // Expr -> unary_operator Non-Terminal
    else
    {
        // Generate the expression
        code_generator_expression(expr->children[1]);

        // Generate code according to the unary operator
        code_generator_unary_expression(expr->children[1]->register_number,
expr->children[0]->token->token_type);

        // Preserve the register
        expr->register_number = expr->children[1]->register_number;
    }
}

// Binary expression
// In the format: Expr -> Expr operator Expr
else if (expr->num_of_children == 3)
{
}

```

```
// If both children are terminals
// I want to start generating code only when both of the children are
terminals,
    // that way I use the minimum number of registers to generate the target
code, and minimizing
        // the risk of running out of registers.
        if (expr->children[0]->symbol_type == Terminal && expr->children[2]-
>symbol_type == Terminal)
{
    // Allocate registers for the terminals
    int left_register = code_generator_register_alloc();
    int right_register = code_generator_register_alloc();

    // Move terminals to their registers
    code_generator_mov_token(left_register, expr->children[0]->token);
    code_generator_mov_token(right_register, expr->children[2]->token);

    // Generate code using both of the registers and the operator
    code_generator_binary_expression(left_register, right_register, expr-
>children[1]->token->token_type);

    // Preserve the left register for the result
    expr->register_number = left_register;

    // Free the right register because we don't need it anymore
    code_generator_register_free(right_register);
}

// If left is non-terminal and right is terminal
else if (expr->children[0]->symbol_type == Non_Terminal && expr-
>children[2]->symbol_type == Terminal)
{
    // Generate left expression
    code_generator_expression(expr->children[0]);

    // Allocate register for the right terminal
    int right_register = code_generator_register_alloc();

    // Move the right terminal to its register
    code_generator_mov_token(right_register, expr->children[2]->token);
```

```

        // Generate code using the left expression's register, right terminal
and the operator
        code_generator_binary_expression(expr->children[0]->register_number,
right_register, expr->children[1]->token->token_type);

        // Preserve the left register for the result
expr->register_number = expr->children[0]->register_number;

        // Free the right register because we don't need it anymore
code_generator_register_free(right_register);
}

// If left is terminal and right is non-terminal
else if (expr->children[0]->symbol_type == Terminal && expr->children[2]-
>symbol_type == Non_Terminal)
{
    // Generate the right expression
code_generator_expression(expr->children[2]);

    // Allocate register for the left terminal
int left_register = code_generator_register_alloc();

    // Move the left terminal to its register
code_generator_mov_token(left_register, expr->children[0]->token);

        // Generate code using the right expression's register, left terminal
and the operator
        code_generator_binary_expression(left_register, expr->children[2]->register_number, expr->children[1]->token->token_type);

        // Preserve the left register for the result
expr->register_number = left_register;

        // Free the right register because we don't need it anymore
code_generator_register_free(expr->children[2]->register_number);
}

// If both non-terminals
else
{
    // Generate the left and right expressions
code_generator_expression(expr->children[0]);
}

```

```
    code_generator_expression(expr->children[2]);  
  
    // Generate code using the left and the right expression's registers and  
    // the operator  
    code_generator_binary_expression(expr->children[0]->register_number,  
expr->children[2]->register_number, expr->children[1]->token->token_type);  
  
    // Preserve the left register for the result  
    expr->register_number = expr->children[0]->register_number;  
  
    // Free the right register because we don't need it anymore  
    code_generator_register_free(expr->children[2]->register_number);  
}  
}  
}  
  
void code_generator_binary_expression(int left_register, int right_register,  
Token_Type operator)  
{  
    switch (operator)  
    {  
        case Token_Or:  
            code_generator_or(left_register, right_register);  
            break;  
  
        case Token_And:  
            code_generator_and(left_register, right_register);  
            break;  
  
        case Token_Equal:  
        case Token_Not_Equal:  
        case Token_Bigger:  
        case Token_Bigger_Equal:  
        case Token_Smaller:  
        case Token_Smaller_Equal:  
            code_generator_bool_e(left_register, right_register, operator);  
            break;  
  
        case Token_Plus:  
            code_generator_output(ADD, code_generator_register_name(left_register),  
code_generator_register_name(right_register));  
            break;  
    }
```

```
        case Token_Minus:
            code_generator_output(SUB, code_generator_register_name(left_register),
code_generator_register_name(right_register));
            break;

        case Token_Multiply:
            code_generator_output(IMUL, code_generator_register_name(left_register),
code_generator_register_name(right_register));
            break;

        case Token_Divide:
            code_generator_divide(left_register, right_register);
            break;

        case Token_Modulu:
            code_generator_modulu(left_register, right_register);
            break;
    }
}

void code_generator_unary_expression(int register_number, Token_Type operator)
{
    switch (operator)
    {
        case Token_Not:
            code_generator_not(register_number);
            break;

        case Token_Minus:
            code_generator_output(NEG,
code_generator_register_name(register_number));
            break;
    }
}

void code_generator_bool_e(int left_register, int right_register, Token_Type operator)
{
    char* true_label = code_generator_label_create();
    char* done_label = code_generator_label_create();
```

```

// Compare left and right operands
code_generator_output(CMP, code_generator_register_name(left_register),
code_generator_register_name(right_register));

// Perform the right operation between the expr and the term
switch (operator)
{
    case Token_Equal:
        code_generator_output(JE, true_label);
        break;

    case Token_Not_Equal:
        code_generator_output(JNE, true_label);
        break;

    case Token_Bigger:
        code_generator_output(JG, true_label);
        break;

    case Token_Bigger_Equal:
        code_generator_output(JGE, true_label);
        break;

    case Token_Smaller:
        code_generator_output(JL, true_label);
        break;

    case Token_Smaller_Equal:
        code_generator_output(JLE, true_label);
        break;
}

// If didn't jump, condition was not met and the result is 0
code_generator_output(MOV, code_generator_register_name(left_register), "0");
code_generator_output(JMP, done_label);

// If jumped, condition was met and the result is 1
code_generator_output(LABEL, true_label);
code_generator_output(MOV, code_generator_register_name(left_register), "1");

code_generator_output(LABEL, done_label);

```

```
// Free the lables
free(true_label);
free(done_label);
}

void code_generator_mov_token(int register_number, Token* token)
{
    // Identifier
    if (token->token_type == Token_Identifier)
        code_generator_mov_identifier(register_number, token);

    // Literal
    else
        code_generator_output(MOV, code_generator_register_name(register_number),
token->value);
}

void code_generator_mov_identifier(int register_number, Token* token)
{
    // Calculate address for the given id
    Symbol_Table_Entry* entry = scope_tree_fetch(token->value);

    // If the type of the identifier is char
    if (entry->data_type == Data_Type_Char)
    {
        // Zero out the register before moving a value to its lower bytes so the
value will be correct
        code_generator_output(MOV, code_generator_register_name(register_number),
"0");

        // Mov the value to the lower byte of the register
        code_generator_output("\tmov %sb, %s\n",
code_generator_register_name(register_number),
code_generator_symbol_address(entry));
    }

    // If the type of the identifier is not char (int)
    else
        code_generator_output(MOV, code_generator_register_name(register_number),
code_generator_symbol_address(entry));
}
```

```

void code_generator_or(int left_register, int right_register)
{
    char* true_label = code_generator_label_create();
    char* done_label = code_generator_label_create();

    // Check if left operand is not 0
    code_generator_output(NEG, code_generator_register_name(left_register));
    code_generator_output(JC, true_label);

    // Check if right operand is not 0
    code_generator_output(NEG, code_generator_register_name(right_register));
    code_generator_output(JC, true_label);

    // If both 0, result is 0
    code_generator_output(MOV, code_generator_register_name(left_register), "0");
    code_generator_output(JMP, done_label);

    // If either is not 0, result is 1
    code_generator_output(LABEL, true_label);
    code_generator_output(MOV, code_generator_register_name(left_register), "1");

    code_generator_output(LABEL, done_label);

    // Free the labels
    free(true_label);
    free(done_label);
}

void code_generator_and(int left_register, int right_register)
{
    char* false_label = code_generator_label_create();
    char* done_label = code_generator_label_create();

    // Check if left operand is 0
    code_generator_output(NEG, code_generator_register_name(left_register));
    code_generator_output(JNC, false_label);

    // Check if right operand is 0
    code_generator_output(NEG, code_generator_register_name(right_register));
    code_generator_output(JNC, false_label);

    // If both not 0, result is 1

```

```

code_generator_output(MOV, code_generator_register_name(left_register), "1");
code_generator_output(JMP, done_label);

// If either is 0, result is 0
code_generator_output(LABEL, false_label);
code_generator_output(MOV, code_generator_register_name(left_register), "0");

code_generator_output(LABEL, done_label);

// Free the labels
free(false_label);
free(done_label);
}

void code_generator_divide(int left_register, int right_register)
{
    // Clear RDX, because the result of division is in RAX:RDX where RAX is quotient
    // and RDX is remainder
    code_generator_output(MOV, RDX, "0");

    // Mov dividend to RAX for division
    code_generator_output(MOV, RAX, code_generator_register_name(left_register));

    // Divide by divisor
    code_generator_output(IDIV, code_generator_register_name(right_register));

    // Mov quotient to result register
    code_generator_output(MOV, code_generator_register_name(left_register), RAX);
}

void code_generator_modulu(int left_register, int right_register)
{
    // Clear RDX, because the result of division is in RAX:RDX where RAX is quotient
    // and RDX is remainder
    code_generator_output(MOV, RDX, "0");

    // Mov dividend to RAX for division
    code_generator_output(MOV, RAX, code_generator_register_name(left_register));

    // Divide by divisor
    code_generator_output(IDIV, code_generator_register_name(right_register));
}

```

```
// Mov remainder to result register
code_generator_output(MOV, code_generator_register_name(left_register), RDX);
}

void code_generator_not(int register_number)
{
    // Initial result value is 1
    code_generator_output(MOV, RAX, "1");

    // NEG the factor register to know if 0 or not
    code_generator_output(NEG, code_generator_register_name(register_number));

    // Sub carry
    // If the number was not 0 CF = 1 and result will be 0
    // If the number was 0 CF = 0 and result will be 1
    code_generator_output(SBB, RAX, "0");

    // Mov result to result register
    code_generator_output(MOV, code_generator_register_name(register_number), RAX);
}
```

compiler

compiler.h

```
#pragma once

#include "../lexer/lexer.h"
#include "../parser/parser.h"
#include "../scope_tree/scope_tree.h"
#include "../code_generator/code_generator.h"

/* ----- Structs ----- */

// Struct of the entire compiler
typedef struct Compiler
{
    // The source code to be compiled
    char* src;
    // The name of the destination file
    char* dest_file_name;
    // The lexer of the compiler
    Lexer* lexer;
    // The parser of the compiler
    Parser* parser;
    // The scope tree of the compiler
    Scope_Tree* scope_tree;
    // The code generator of the compiler
    Code_Generator* code_generator;
    // Number of errors found during compilation
    int errors;
    // Current line number in source file for error reporting
    int line;
} Compiler;

/* ----- Functions ----- */

// Gets the source code file name and destination file name, and initializes the
// global compiler with all it's necessary components
void compiler_init(char* src_file_name, char* dest_file_name);

// Destroys all the components of the global compiler
void compiler_destroy();
```

```
// Compiles the source code
void compiler_compile();
```

compiler.c

```
#include <stdlib.h>

#include "../global.h"

#include "../general/general.h"
#include "compiler.h"
#include "../io/io.h"
#include "../code_generator/code_generator.h"

void compiler_init(char* src_file_name, char* dest_file_name)
{
    // Source code
    compiler.src = io_read_file(src_file_name);

    // Destination file
    compiler.dest_file_name = dest_file_name;

    // Lexer
    lexer_create();
    lexer_init();

    // Parser
    parser_create();
    parser_init();

    // Symbol table
    scope_tree_create();

    // Code generator
    code_generator_create();
    code_generator_init();

    // Initialize with 0 errors
    compiler.errors = 0;
```

```
// Initialize with line number 1
compiler.line = 1;
}

void compiler_destroy()
{
    // Source code
    free(compiler.src);
    compiler.src = NULL;

    // Lexer
    lexer_destroy();

    // Parser
    parser_destroy();

    // Symbol table
    scope_tree_destroy();

    // Code generator
    code_generator_destroy();
}

void compiler_compile()
{
    // Parse the source code and create a parse tree that represents it.
    // Also updates the symbol table.
    Parse_Tree_Node* parse_tree = parser_parse();

    // If errors were encountered during compilation
    if (compiler.errors > 0)
    {
        // Destroy parse tree
        parse_tree_destroy(parse_tree);

        // Delete destination file
        remove(compiler.dest_file_name);

        // Destroy the compiler
        compiler_destroy();

        printf(RED "\n\nCompilation terminated.\n" RESET);
    }
}
```

```
    exit(1);
}

// Generate the assembly code for the created parse tree
code_generator_generate(parse_tree);

// If reached this point, that means the compilation process was successfull!
printf(BOLD_WHITE "\nParsing...\n");
printf("Generating code...\n");
printf(BOLD_GREEN "\nCompilation successful!\n" RESET);

// Destroy parse tree
parse_tree_destroy(parse_tree);
}
```

error_handler

error_handler.h

```
#pragma once

/* ----- Structs ----- */

// Enum for possible error types
typedef enum Error_Type
{
    Error_General = 1,
    Error_Lexical,
    Error_Syntax,
    Error_Semantic,
} Error_Type;

/* ----- Functions ----- */

// Reports an error message to the user and updated the number of errors in the
compiler
void error_handler_report(int line, Error_Type error_type, char* format, ...);

// A general function that helps the compiler to recover after encountering a syntax
error.
// "eats" tokens from the input until finds a token that is at the start of a
statement production rule.
// When finds one, pops every token that is no longer relevant (shifted in the
current errorful statement) off the stack,
// and shifts said found token with its correct starting state onto the updated
stack.
// At the end, retrieves the next token from the input.
void error_handler_error_recovery();

// The following function are specific functions for each state for better error
reporting & recovery.
// This functions are using the error_handler_error_recovery() function.

// Prints an expected `prog` error message to the user, and pushes a prog entry onto
the parse stack
void error_handler_report_expected_prog();
```

```
// Prints an expected program name error message to the user, and pushes an id entry
onto the parse stack
void error_handler_report_expected_prog_name();

// Prints an expected `:` error message to the user, and pushes a `:` entry onto the
parse stack
// with the right goto state according to the state in the action table
void error_handler_report_expected_colon_state_3();
void error_handler_report_expected_colon_state_37();
void error_handler_report_expected_colon_state_46();
void error_handler_report_expected_colon_state_59();

// Prints an expected `;` error message to the user, and pushes a `;` entry onto the
parse stack
// with the right goto state according to the state in the action table
void error_handler_report_expected_semi_colon_state_18();

// Prints an expected EOF error message to the user
void error_handler_report_expected_eof();

// Prints an expected `:)` error message to the user
void error_handler_report_expected_smiley();

// Prints an expected identifier error message to the user
void error_handler_report_expected_identifier();

// Prints an expected `(` error message to the user
void error_handler_report_expected_open_paren();

// Prints an expected `=` error message to the user
void error_handler_report_expected_assign();

// Prints the expected token group of state 4 states group error message to the
user.
// In the action table this token group contains: `done`, `int`, `char`, `set`,
`if`, `while`.
// States that contain this token group: 4, 6, 8, 9, 10, 11, 22, 47, 48, 55 ,57, 58,
60, 61
void error_handler_report_expected_state_4_group();

// Prints the expected token group of state 7 states group error message to the
user.
```

```

// In the action table this token group contains: `:)`, `done`, `int`, `char`,
`set`, `if`, `else`, `while`.
// States that contain this token group: 7, 17
void error_handler_report_expected_state_7_group();

// Prints the expected token group of state 20 states group error message to the
user.
// In the action table this token group contains: id, `(`, `-`, number, character,
`!`
// States that contain this token group: 20, 21, 23, 32, 33, 34, 38, 39, 40, 41, 42
void error_handler_report_expected_state_20_group();

// Prints the expected token group of state 24 states group error message to the
user.
// In the action table this token group contains: `)``, `||`
// States that contain this token group: 24, 35, 43
void error_handler_report_expected_state_24_group();

// Prints the expected token group of state 25 states group error message to the
user.
// In the action table this token group contains: `;`, `)``, binary_operator
// States that contain this token group: 25, 26, 27, 28, 29, 30, 31, 44, 45, 49, 50,
51, 52, 53, 54
void error_handler_report_expected_state_25_group();

// Prints the expected token group of state 36 states group error message to the
user.
// In the action table this token group contains: `;`, `||`
// States that contain this token group: 36
void error_handler_report_expected_state_36_group();

// Prints the expected token group of state 56 states group error message to the
user.
// In the action table this token group contains: `done`, `int`, `char`, `set`,
`if`, `else`, `while`.
// States that contain this token group: 56
void error_handler_report_expected_state_56_group();

// Converts an error code to its string representation
const char* error_handler_error_to_str(Error_Type error_type);

```

error_handler.c

```
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>

#include "../global.h"

#include "error_handler.h"
#include "../general/general.h"

void error_handler_report(int line, Error_Type error_type, char* format, ...)
{
    printf("\n[ " RED "%s" RESET " on line" CYAN " %d" RESET " ] ", 
error_handler_error_to_str(error_type), line);

    va_list args;           // Declare a va_list type variable
    va_start(args, format); // Initialize the va_list with the ...
    vprintf(format, args); // Forward ... to vprintf
    va_end(args);          // Clean up the va_list

    compiler.errors++;
}

void error_handler_error_recovery()
{
    // Get the starting state of the current token
    int token_starting_state = parser_get_token_starting_state(compiler.parser-
>token->token_type);

    // Loop until found a token which is a start of a statement, or until reached
EOF.
    // Ignore all toknes until findes a token which is a start of a statement.
    while (token_starting_state == 0 && compiler.parser->token->token_type !=
Token_Eof)
    {
        // Destroy previous token
        token_destroy(compiler.parser->token);
        // Get next token
        compiler.parser->token = lexer_get_next_token();
        // Get the starting state of the newly fetched token
        token_starting_state = parser_get_token_starting_state(compiler.parser-
>token->token_type);
    }
}
```

```
}

// Pop not needed, previously shifted tokens of the current part of the block
off the parse stack.
// Stop if only one entry left in the parse stack (the init entry).
Parse_Stack_Entry* entry = parse_stack_pop();
while (compiler.parser->parse_stack->next_entry != NULL)
{
    // If reached the start of the current block that means we've poped enough
    off the stack and can continue parsing.
    if (entry->tree->symbol_type == Terminal && entry->tree->token->token_type
== Token_Colon)
        break;

    // Destroy poped entry
    parse_tree_destroy(entry->tree);
    free(entry);

    // Pop the next entry
    entry = parse_stack_pop();
}

// Push back last poped entry (becasue we're popping one too many entries)
parse_stack_push(entry);

// Shift the found token with it's starting state
parser_shift(token_starting_state);

// Get next token from the source code
compiler.parser->token = lexer_get_next_token();
}

void error_handler_report_expected_prog()
{
    Token_Type type = Token_Prog;
    int goto_state = 2;

    printf(", expected '" BOLD_WHITE "prog" RESET "' at the start of the program");

    parse_stack_push(parse_stack_init_entry(parse_tree_init_node(Terminal, type,
token_init(NULL, 0, type), NULL, 0), goto_state));
}
```

```

void error_handler_report_expected_prog_name()
{
    Token_Type type = Token_Identifier;
    int goto_state = 3;

    printf(", expected " BOLD_WHITE "program name" RESET " after '" BOLD_WHITE
"prog" RESET "'");

    parse_stack_push(parse_stack_init_entry(parse_tree_init_node(Terminal, type,
token_init(NULL, 0, type), NULL, 0), goto_state));
}

void error_handler_report_expected_colon_state_3()
{
    Token_Type type = Token_Colon;
    int goto_state = 4;

    printf(", expected '" BOLD_WHITE ":" RESET "' after " BOLD_WHITE "program name"
RESET);

    parse_stack_push(parse_stack_init_entry(parse_tree_init_node(Terminal, type,
token_init(NULL, 0, type), NULL, 0), goto_state));
}

void error_handler_report_expected_colon_state_37()
{
    Token_Type type = Token_Colon;
    int goto_state = 48;

    printf(", expected '" BOLD_WHITE ":" RESET "' at the start of an if block");

    parse_stack_push(parse_stack_init_entry(parse_tree_init_node(Terminal, type,
token_init(NULL, 0, type), NULL, 0), goto_state));
}

void error_handler_report_expected_colon_state_46()
{
    Token_Type type = Token_Colon;
    int goto_state = 55;

    printf(", expected '" BOLD_WHITE ":" RESET "' at the start of a while block");
}

```

```
    parse_stack_push(parse_stack_init_entry(parse_tree_init_node(Terminal, type,
token_init(NULL, 0, type), NULL, 0), goto_state));
}

void error_handler_report_expected_colon_state_59()
{
    Token_Type type = Token_Colon;
    int goto_state = 60;

    printf(", expected '" BOLD_WHITE ":" RESET "' at the start of an else block");

    parse_stack_push(parse_stack_init_entry(parse_tree_init_node(Terminal, type,
token_init(NULL, 0, type), NULL, 0), goto_state));
}

void error_handler_report_expected_semi_colon_state_18()
{
    Token_Type type = Token_Semi_Colon;
    int goto_state = 22;

    printf(", expected '" BOLD_WHITE ";" RESET "'");

    parse_stack_push(parse_stack_init_entry(parse_tree_init_node(Terminal, type,
token_init(NULL, 0, type), NULL, 0), goto_state));
}

void error_handler_report_expected_eof()
{
    printf(", expected " BOLD_WHITE "EOF" RESET);

    error_handler_error_recovery();
}

void error_handler_report_expected_smiley()
{
    printf(", expected '" BOLD_WHITE ":"") RESET "'");

    error_handler_error_recovery();
}

void error_handler_report_expected_identifier()
```

```
{  
    printf(", expected " BOLD_WHITE "identifier" RESET);  
  
    error_handler_error_recovery();  
}  
  
void error_handler_report_expected_open_paren()  
{  
    printf(", expected '" BOLD_WHITE "(" RESET "'");  
  
    error_handler_error_recovery();  
}  
  
void error_handler_report_expected_assign()  
{  
    printf(", expected '" BOLD_WHITE "=" RESET "'");  
  
    error_handler_error_recovery();  
}  
  
void error_handler_report_expected_state_4_group()  
{  
    printf(", expected '" BOLD_WHITE "done" RESET "'");  
    printf(", '" BOLD_WHITE "int" RESET "'");  
    printf(", '" BOLD_WHITE "char" RESET "'");  
    printf(", '" BOLD_WHITE "set" RESET "'");  
    printf(", '" BOLD_WHITE "if" RESET "'");  
    printf(" or '" BOLD_WHITE "while" RESET "'");  
  
    error_handler_error_recovery();  
}  
  
void error_handler_report_expected_state_7_group()  
{  
    printf(", expected '" BOLD_WHITE ":" RESET "'");  
    printf(", " BOLD_WHITE "done" RESET "'");  
    printf(", " BOLD_WHITE "int" RESET "'");  
    printf(", " BOLD_WHITE "char" RESET "'");  
    printf(", " BOLD_WHITE "set" RESET "'");  
    printf(", " BOLD_WHITE "if" RESET "'");  
    printf(", " BOLD_WHITE "else" RESET "'");  
    printf(" or '" BOLD_WHITE "while" RESET "'");  
}
```

```
    error_handler_error_recovery();
}

void error_handler_report_expected_state_20_group()
{
    printf(", expected '" BOLD_WHITE "(" RESET "'");
    printf("-" RESET "'");
    printf("literal" RESET);
    printf(" or '" BOLD_WHITE "!" RESET "'");

    error_handler_error_recovery();
}

void error_handler_report_expected_state_24_group()
{
    printf(", expected '" BOLD_WHITE ")" RESET "'");
    printf(" or '" BOLD_WHITE "||" RESET "'");

    error_handler_error_recovery();
}

void error_handler_report_expected_state_25_group()
{
    printf(", expected '" BOLD_WHITE ";" RESET "'");
    printf(")" RESET "'");
    printf(" or " BOLD_WHITE "binary operator" RESET);

    error_handler_error_recovery();
}

void error_handler_report_expected_state_36_group()
{
    printf(", expected '" BOLD_WHITE ";" RESET "'");
    printf(" or '" BOLD_WHITE "||" RESET "'");

    error_handler_error_recovery();
}

void error_handler_report_expected_state_56_group()
{
    printf(", expected '" BOLD_WHITE "done" RESET "'");
```

```
printf("", "" BOLD_WHITE "int" RESET "");  
printf("", "" BOLD_WHITE "char" RESET "");  
printf("", "" BOLD_WHITE "set" RESET "");  
printf("", "" BOLD_WHITE "if" RESET "");  
printf("", "" BOLD_WHITE "else" RESET "");  
printf(" or '" BOLD_WHITE "while" RESET "'");  
  
error_handler_error_recovery();  
}  
  
const char* error_handler_error_to_str(Error_Type error_type)  
{  
    switch (error_type)  
    {  
        case Error_Lexical: return "Lexical Error";  
        case Error_Syntax: return "Syntax Error";  
        case Error_Semantic: return "Semantic Error";  
  
        default: return "Error";  
    }  
}
```

general

general.h

```
#pragma once

#include <stdio.h>

/* ----- Macros ----- */

// Used to printf with colors
#define RESET          "\033[0m"
#define BLACK          "\033[30m"      /* Black */
#define RED            "\033[31m"      /* Red */
#define GREEN          "\033[32m"      /* Green */
#define YELLOW         "\033[33m"      /* Yellow */
#define BLUE           "\033[34m"      /* Blue */
#define PURPLE         "\033[35m"      /* Purple */
#define CYAN           "\033[36m"      /* Cyan */
#define WHITE          "\033[37m"      /* White */
#define BOLD_BLACK     "\033[1m\033[30m" /* Bold Black */
#define BOLD_RED       "\033[1m\033[31m" /* Bold Red */
#define BOLD_GREEN     "\033[1m\033[32m" /* Bold Green */
#define BOLD_YELLOW   "\033[1m\033[33m" /* Bold Yellow */
#define BOLD_BLUE      "\033[1m\033[34m" /* Bold Blue */
#define BOLD_MAGENTA   "\033[1m\033[35m" /* Bold Magenta */
#define BOLD_CYAN      "\033[1m\033[36m" /* Bold Cyan */
#define BOLD_WHITE     "\033[1m\033[37m" /* Bold White */

/* ----- Functions ----- */

// Expecting __FILE__ and __LINE__ to report a memory allocation error.
// Reports an memory allocation error, destroys the global compiler and exits with
exit code 1.
void exit_memory_error(char* file, int line);

// Expecting the file name that was not able to perform I/O well to report a file
I/O error.
// Reports an file I/O error, destroys the global compiler and exits with exit code
1.
void exit_file_io_error(char* file_name);
```

general.c

```
#include <stdlib.h>

#include "../global.h"

#include "general.h"
#include "../compiler/compiler.h"

void exit_memory_error(char* file, int line)
{
    printf("[ RED Memory Error  RESET ] Memory allocation failed.\n");
    printf("\tIn file: " BOLD_WHITE "%s\n"  RESET , file);
    printf("\tOn line: " CYAN "%d\n"  RESET , line);

    compiler_destroy();
    remove(compiler.dest_file_name);
    exit(1);
}

void exit_file_io_error(char* file_name)
{
    printf("[ RED File I/O Error  RESET ] Could not open file " BOLD_WHITE "%s\n"  RESET , file_name);

    compiler_destroy();
    remove(compiler.dest_file_name);
    exit(1);
}
```

io

io.h

```
#pragma once

// Gets a text file name and returns a string with the contents of that file
char* io_read_file(char* file_name);
```

io.c

```
#include <stdlib.h>
#include <stdio.h>

#include "io.h"
#include "../general/general.h"

char* io_read_file(char* file_name)
{
    char* buffer = 0;
    unsigned long long length;

    // Opening the file
    FILE* fp = fopen(file_name, "rb");
    if (fp == NULL) exit_file_io_error(file_name);

    // Using fseek & ftell to get length of file
    fseek(fp, 0, SEEK_END);
    length = ftell(fp);

    // Go to start of file again
    fseek(fp, 0, SEEK_SET);

    // Allocate the buffer size according to the length of the file + 1 for \0
    buffer = (char*) calloc(length + 1, sizeof(char));
    if (buffer == NULL)
    {
        fclose(fp);
        exit_memory_error(__FILE__, __LINE__);
    }
}
```

```
// Copying the contents of the file to the buffer
fread(buffer, 1, length, fp);
// Ending the source code buffer with '\0'
buffer[length] = '\0';

// Closing the file
fclose(fp);

return buffer;
}
```

lexer

lexer.h

```
#pragma once

#include <stdbool.h>

#include "lexer_fsm/lexer_fsm.h"
#include "../token/token.h"

// The size to allocate for each new token value
#define LEXER_MAX_TOKEN_SIZE 32 // 31 + 1, the max size of a token is 31 characters
long. +1 for the null terminator

/* ----- Structs ----- */

// Struct of the lexer
typedef struct Lexer
{
    char c;          // Current character in the src code
    int i;           // Current offset from the starting of the source code
    Lexer_FSM* fsm; // The lexer's FSM
} Lexer;

/* ----- Functions ----- */

// Create a new lexer on the heap and updates the compiler's lexer to point to it
void lexer_create();

// Frees everything we've allocated in the lexer_init() function for the compiler's
// lexer
void lexer_destroy();

// Initializes the compiler's lexer
void lexer_init();

// Advances the lexer 1 character forward in the source code
void lexer_advance();

// Creates and returns a new token when reached to the end of a token
```

```

Token* lexer_EOT(char* value, int size, int state);

// Checks whether a token is a skip token according to the given token type.
// The skip tokens are: Whitespace, Comment
bool lexer_is_skip_token(Token_Type type);

// Returns the next token from the source code
Token* lexer_get_next_token();

```

lexer.c

```

#include <stdlib.h>
#include <string.h>

#include "../global.h"

#include "lexer.h"
#include "../general/general.h"
#include "../error_handler/error_handler.h"

void lexer_create()
{
    // Create lexer
    compiler.lexer = (Lexer*) calloc(1, sizeof(Lexer));
    if (compiler.lexer == NULL) exit_memory_error(__FILE__, __LINE__);

    // Create lexer's FSM
    lexer_fsm_create();
}

void lexer_destroy()
{
    // check for NULL pointer
    if (compiler.lexer != NULL)
    {
        // Free the lexer's Finite State Machine
        lexer_fsm_destroy();

        // Free the lexer
        free(compiler.lexer);
        compiler.lexer = NULL;
    }
}

```

```

    }
}

void lexer_init()
{
    // Initialize lexer's components
    compiler.lexer->i = 0;
    compiler.lexer->c = compiler.src[compiler.lexer->i];

    // Initialize lexer's Finite State Machine
    lexer_fsm_init(compiler.lexer->fsm);
}

void lexer_advance()
{
    // Advance to the next character in the source code
    compiler.lexer->i++;
    compiler.lexer->c = compiler.src[compiler.lexer->i];

    // Advance line number if current character is Line Feed / a new line character,
    '\n'
    if (compiler.lexer->c == '\n')
        compiler.line++;
}

Token* lexer_EOT(char* value, int size, int state)
{
    // Reallocating the value to its actual size
    value = (char*) realloc(value, (size + 1) * sizeof(char));
    if (value == NULL) exit_memory_error(__FILE__, __LINE__);

    // Making sure there is a null terminator at the end of the value
    value[size] = '\0';

    // Advancing and returning a new token
    lexer_advance();
    return token_init(value, size, compiler.lexer->fsm->states[state].token_type);
}

bool lexer_is_skip_token(Token_Type type)
{
    if (type == TokenWhitespace || type == TokenComment)

```

```
    return true;

    return false;
}

Token* lexer_get_next_token()
{
    // Get the starting state according to the current character from the source
    // code
    int state = lexer_fsm_get_starting_state_index(compiler.lexer->c);

    // The value of the token that will be returned
    char* value = (char*) calloc(LEXER_MAX_TOKEN_SIZE, sizeof(char));
    if (value == NULL) exit_memory_error(__FILE__, __LINE__);

    // The size of the token's value
    int size = 0;

    // While not EOS
    while (compiler.lexer->c != '\0')
    {
        // Check for Token's max length
        if (size == LEXER_MAX_TOKEN_SIZE - 1)
        {
            // If this is NOT a skip token output error and advance to the next
            // token.
            // If this is a skip token, just ignore the token and continue.
            if (!lexer_is_skip_token(compiler.lexer->fsm->states[state].token_type))
            {
                // Report error
                error_handler_report(compiler.line, Error_Lexical, "Token can't be
longer than " BOLD_WHITE "%d" RESET " characters", LEXER_MAX_TOKEN_SIZE - 1);

                // Continue to the next token
                lexer_advance();
                state = lexer_fsm_get_starting_state_index(compiler.lexer->c);
            }

            // Reset the size to not exceed value's size
            size = 0;
        }
    }
}
```

```

// Update value
value[size++] = compiler.lexer->c;

// If there is no state to advance to according to the current state and
// input character (reached EOT)
if (compiler.lexer->fsm-
>edges[state][lexer_fsm_get_char_index(compiler.src[compiler.lexer->i +
1])].state_number == 0)
{
    // If reached an error token, report the error and continue
    if (compiler.lexer->fsm->states[state].token_type == Token_Error)
    {
        // Making sure there is a null terminator
        value[size] = '\0';

        // Report error
        error_handler_report(compiler.line, Error_Lexical, "Unexpected
character '" BOLD_WHITE "%s" RESET "'", value);

        // Continue to the next token
        lexer_advance();
        state = lexer_fsm_get_starting_state_index(compiler.lexer->c);
        size = 0;
    }

    // If reached a skip token, continue to the next token
    else if (lexer_is_skip_token(compiler.lexer->fsm-
>states[state].token_type))
    {
        lexer_advance();
        state = lexer_fsm_get_starting_state_index(compiler.lexer->c);
        size = 0;
    }

    // If reached to other valid token, return it
    else
        return lexer_EOT(value, size, state);
}
// If not EOT yet, advance to the next state
else
{
    lexer_advance();
}

```

```
        state = compiler.lexer->fsm-
>edges[state][lexer_fsm_get_char_index(compiler.lexer->c)].state_number;
    }
}

// If we've reached the end of the src code but the value is not empty, and the
token is not a skip token, then return a new token with that value
if (size > 0 && !lexer_is_skip_token(compiler.lexer->fsm-
>states[state].token_type))
    return lexer_EOT(value, size, state);

// When we've reached to the end of the src code, return End Of File token
return token_init(NULL, 0, Token_Eof);
}
```

lexer_fsm

lexer_fsm.h

```
#pragma once

#include "../token/token.h"

// Number of character in the language - 128 ascii characters
#define NUM_OF_CHARACTERS 128
// Number of states in the lexer's Finite State Machine
#define LEXER_FSM_NUM_OF_STATES 64

/* ----- Structs ----- */

// Struct of a state in the lexer's FSM
typedef struct Lexer_State
{
    // Every state holds a token type, so if we stoped on that state we would know
    // what token we have
    Token_Type token_type;
} Lexer_State;

// Struct of a connection between states in the lexer's FSM
typedef struct Lexer_Edge
{
    // An edge is representing a transition between states in the FSM
    // Every edge is holding which state to go next
    int state_number;
} Lexer_Edge;

// Struct of the lexer's FSM
typedef struct Lexer_FSM
{
    // An array of starting state indices to help us know at which state to start
    // when we start a new token
    int starting_state_indices[NUM_OF_CHARACTERS];
    // The array of states of the FSM
    Lexer_State states[LEXER_FSM_NUM_OF_STATES];
    // The matrix of edges of the FSM
    Lexer_Edge edges[LEXER_FSM_NUM_OF_STATES][NUM_OF_CHARACTERS];
} Lexer_FSM;
```

```
/* ----- Functions ----- */

// Creates a new lexer fsm on the heap and updates the compiler's lexer finite state
machine to point to it
void lexer_fsm_create();

// Frees everything we've allocated in the lexer_fsm_create() function for the
compiler's lexer finite state machine
void lexer_fsm_destroy();

// Initializes the FSM of the lexer
// Creates all the states and connections between them to make the FSM work
properly
void lexer_fsm_init();

// Sets an edge between state and every alpha-numeric character in the adjacency
matrix except for the except char
void lexer_fsm_set_alnum_identifier(int state, char except);

// Returns the index of a certain character in the FSM adjacency matrix
int lexer_fsm_get_char_index(char value);

// Returns The index of the starting state of a token according to the current
character
int lexer_fsm_get_starting_state_index(char value);

// Adds a starting state index to the lexer's starting state indices array
void lexer_fsm_add_starting_state_index(int char_index, int starting_state_index);

// Adds a state to the lexer's FSM
void lexer_fsm_add_state(int state_number, Token_Type token_type);

// Adds a connection to the lexer's FSM
void lexer_fsm_add_edge(int state, int ch, int to_state);

// Prints the vertices & adjacency matrix of the graph that is representing the FSM
void lexer_fsm_print();
```

lexer_fsm.c

```

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

#include "../global.h"

#include "lexer_fsm.h"
#include "../../general/general.h"
#include "../../error_handler/error_handler.h"

void lexer_fsm_create()
{
    // Create fsm
    compiler.lexer->fsm = (Lexer_FSM*) calloc(1, sizeof(Lexer_FSM));
    if (compiler.lexer->fsm == NULL) exit_memory_error(__FILE__, __LINE__);
}

void lexer_fsm_destroy()
{
    // check for NULL pointer
    if (compiler.lexer != NULL)
    {
        free(compiler.lexer->fsm);
        compiler.lexer->fsm = NULL;
    }
}

void lexer_fsm_setalnum_identifier(int state, char except)
{
    int i;
    for (i = 0; i < NUM_OF_CHARACTERS; i++)
        if ((isalnum(i) || i == '_') && i != except)
            lexer_fsm_add_edge(state, i, lexer_fsm_get_starting_state_index('_'));
}

int lexer_fsm_get_char_index(char value)
{
    // If the character is out of range return the first index in the array, 0
    // Else reutrn it's proper index
    return (value >= 0) ? value : 0;
}

```

```

int lexer_fsm_get_starting_state_index(char value)
{
    return compiler.lexer->fsm-
>starting_state_indices[lexer_fsm_get_char_index(value)];
}

void lexer_fsm_add_starting_state_index(int char_index, int starting_state_index)
{
    compiler.lexer->fsm->starting_state_indices[char_index] = starting_state_index;
}

void lexer_fsm_add_state(int state_number, Token_Type token_type)
{
    compiler.lexer->fsm->states[state_number].token_type = token_type;
}

void lexer_fsm_add_edge(int state, int ch, int to_state)
{
    compiler.lexer->fsm->edges[state][ch].state_number = to_state;
}

void lexer_fsm_print()
{
    int i, j;

    // Edges
    printf("      ");
    for (i = 0; i < NUM_OF_CHARACTERS + 0; i++)
    {
        if (i >= ' ')
            printf("%c", i);
        else
            printf("%d", i % 10);
    }
    printf("\n");

    printf("      ");
    for (i = 0; i < NUM_OF_CHARACTERS; i++)
    {
        printf("%c", '_');
    }
}

```

```

printf("\n");

for (i = 0; i < LEXER_FSM_NUM_OF_STATES; i++)
{
    printf("%2d. | ", i);

    for (j = 0; j < NUM_OF_CHARACTERS; j++)
    {
        if (compiler.lexer->fsm->edges[i][j].state_number != 0)
            printf("%d", compiler.lexer->fsm->edges[i][j].state_number);
        else
            printf("%c", '.');
    }
    printf("\n");
}

printf("\n");
}

void lexer_fsm_init()
{
    // Variable to create current state index for each state in the FSM of the
lexer.
    int s = 1;
    // Iterator
    int i;

    // - Whitespace -> 1 state
    lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index(' '), s);
    lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('\t'), s);
    lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('\r'), s);
    lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('\n'), s);
    lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('\v'), s);
    lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('\f'), s);
    // -- states
    lexer_fsm_add_state(s, TokenWhitespace);
    // -- edges
    lexer_fsm_add_edge(s, lexer_fsm_get_char_index(' '), s);
    lexer_fsm_add_edge(s, lexer_fsm_get_char_index('\t'), s);
    lexer_fsm_add_edge(s, lexer_fsm_get_char_index('\r'), s);
    lexer_fsm_add_edge(s, lexer_fsm_get_char_index('\n'), s);
    lexer_fsm_add_edge(s, lexer_fsm_get_char_index('\v'), s);
    lexer_fsm_add_edge(s, lexer_fsm_get_char_index('\f'), s);
}

```

```

// - Comment -> 1 state
s += 1; // To calculate s add to it the numer of states in the previous part
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('#'), s);
// -- states
lexer_fsm_add_state(s, Token_Comment);
// -- edges
// Starting from 0x20 because this is the space character and the first
character we care about in a comment
for (i = 0x20; i < NUM_OF_CHARACTERS; i++)
    lexer_fsm_add_edge(s, lexer_fsm_get_char_index(i), s);

// - Identifier -> 1 state
s += 1;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('_'), s);
for (i = 0; i < NUM_OF_CHARACTERS; i++)
    if (isalpha(i))
        lexer_fsm_add_starting_state_index(i, s);
// -- states
lexer_fsm_add_state(s, Token_Identifier);
// -- edges
lexer_fsm_set_alnum_identifier(s, 0);

// - Number literal -> 3 state
s += 1;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('0'), s); // s
// -> Starting at a state for first digit 0
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('1'), s + 1); // s
+ 1 -> Starting at a state for first digit different from 0
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('2'), s + 1);
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('3'), s + 1);
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('4'), s + 1);
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('5'), s + 1);
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('6'), s + 1);
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('7'), s + 1);
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('8'), s + 1);
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('9'), s + 1);
// -- states
lexer_fsm_add_state(s, Token_Number); // State for first digit 0
lexer_fsm_add_state(s + 1, Token_Number); // State for first digit other digits
lexer_fsm_add_state(s + 2, Token_Error); // If there are continues 0s at the
start of a number

```

```

// -- edges
// --- Add edge for 0-9 from the first digit 0 state to an error state because
if a number starts with 0 it must not continue
lexer_fsm_add_edge(s, '0', s + 2);
lexer_fsm_add_edge(s, '1', s + 2);
lexer_fsm_add_edge(s, '2', s + 2);
lexer_fsm_add_edge(s, '3', s + 2);
lexer_fsm_add_edge(s, '4', s + 2);
lexer_fsm_add_edge(s, '5', s + 2);
lexer_fsm_add_edge(s, '6', s + 2);
lexer_fsm_add_edge(s, '7', s + 2);
lexer_fsm_add_edge(s, '8', s + 2);
lexer_fsm_add_edge(s, '9', s + 2);
// --- Stay in first digit different from 0 state for every 0-9 digit
lexer_fsm_add_edge(s + 1, '0', s + 1);
lexer_fsm_add_edge(s + 1, '1', s + 1);
lexer_fsm_add_edge(s + 1, '2', s + 1);
lexer_fsm_add_edge(s + 1, '3', s + 1);
lexer_fsm_add_edge(s + 1, '4', s + 1);
lexer_fsm_add_edge(s + 1, '5', s + 1);
lexer_fsm_add_edge(s + 1, '6', s + 1);
lexer_fsm_add_edge(s + 1, '7', s + 1);
lexer_fsm_add_edge(s + 1, '8', s + 1);
lexer_fsm_add_edge(s + 1, '9', s + 1);

// - Character literal -> 3 states
s += 3;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('\'), s);
// -- states
lexer_fsm_add_state(s, Token_Error); // '
lexer_fsm_add_state(s + 1, Token_Error); // 'C
lexer_fsm_add_state(s + 2, Token_Character); // 'C'
// -- edges
for (i = 0; i < NUM_OF_CHARACTERS; i++)
    lexer_fsm_add_edge(s, i, s + 1);
lexer_fsm_add_edge(s + 1, lexer_fsm_get_char_index('\''), s + 2);

// - Keywords
// -- "int" & "if" -> 4 states
s += 3;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('i'), s);
// --- states

```

```

lexer_fsm_add_state(s, Token_Identifier); // i
lexer_fsm_add_state(s + 1, Token_If); // if
lexer_fsm_add_state(s + 2, Token_Identifier); // in
lexer_fsm_add_state(s + 3, Token_Int); // int
// --- edges
// ---- "int"
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('n'), s + 2);
lexer_fsm_set_alnum_identifier(s, 'n');
lexer_fsm_add_edge(s + 2, lexer_fsm_get_char_index('t'), s + 3);
lexer_fsm_set_alnum_identifier(s + 2, 't');
lexer_fsm_set_alnum_identifier(s + 3, 0);
// ---- "if"
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('f'), s + 1);
lexer_fsm_set_alnum_identifier(s + 1, 0);

// -- "char" -> 4 states
s += 4;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('c'), s);
// --- states
lexer_fsm_add_state(s, Token_Identifier); // c
lexer_fsm_add_state(s + 1, Token_Identifier); // ch
lexer_fsm_add_state(s + 2, Token_Identifier); // cha
lexer_fsm_add_state(s + 3, Token_Char); // char
// --- edges
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('h'), s + 1);
lexer_fsm_set_alnum_identifier(s, 'h');
lexer_fsm_add_edge(s + 1, lexer_fsm_get_char_index('a'), s + 2);
lexer_fsm_set_alnum_identifier(s + 1, 'a');
lexer_fsm_add_edge(s + 2, lexer_fsm_get_char_index('r'), s + 3);
lexer_fsm_set_alnum_identifier(s + 2, 'r');
lexer_fsm_set_alnum_identifier(s + 3, 0);

// -- "prog" -> 4 states
s += 4;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('p'), s);
// --- states
lexer_fsm_add_state(s, Token_Identifier); // p
lexer_fsm_add_state(s + 1, Token_Identifier); // pr
lexer_fsm_add_state(s + 2, Token_Identifier); // pro
lexer_fsm_add_state(s + 3, Token_Prog); // prog
// --- edges
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('r'), s + 1);

```

```

lexer_fsm_set_alnum_identifier(s, 'r');
lexer_fsm_add_edge(s + 1, lexer_fsm_get_char_index('o'), s + 2);
lexer_fsm_set_alnum_identifier(s + 1, 'o');
lexer_fsm_add_edge(s + 2, lexer_fsm_get_char_index('g'), s + 3);
lexer_fsm_set_alnum_identifier(s + 2, 'g');
lexer_fsm_set_alnum_identifier(s + 3, 0);

// -- "else" -> 4 states
s += 4;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('e'), s);
// --- states
lexer_fsm_add_state(s, Token_Identifier); // e
lexer_fsm_add_state(s + 1, Token_Identifier); // el
lexer_fsm_add_state(s + 2, Token_Identifier); // els
lexer_fsm_add_state(s + 3, Token_Else); // else
// --- edges
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('l'), s + 1);
lexer_fsm_set_alnum_identifier(s, 'l');
lexer_fsm_add_edge(s + 1, lexer_fsm_get_char_index('s'), s + 2);
lexer_fsm_set_alnum_identifier(s + 1, 's');
lexer_fsm_add_edge(s + 2, lexer_fsm_get_char_index('e'), s + 3);
lexer_fsm_set_alnum_identifier(s + 2, 'e');
lexer_fsm_set_alnum_identifier(s + 3, 0);

// -- "while" -> 5 states
s += 4;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('w'), s);
// --- states
lexer_fsm_add_state(s, Token_Identifier); // w
lexer_fsm_add_state(s + 1, Token_Identifier); // wh
lexer_fsm_add_state(s + 2, Token_Identifier); // whi
lexer_fsm_add_state(s + 3, Token_Identifier); // whil
lexer_fsm_add_state(s + 4, Token_While); // while
// --- edges
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('h'), s + 1);
lexer_fsm_set_alnum_identifier(s, 'h');
lexer_fsm_add_edge(s + 1, lexer_fsm_get_char_index('i'), s + 2);
lexer_fsm_set_alnum_identifier(s + 1, 'i');
lexer_fsm_add_edge(s + 2, lexer_fsm_get_char_index('l'), s + 3);
lexer_fsm_set_alnum_identifier(s + 2, 'l');
lexer_fsm_add_edge(s + 3, lexer_fsm_get_char_index('e'), s + 4);
lexer_fsm_set_alnum_identifier(s + 3, 'e');

```

```

lexer_fsm_set_alnum_identifier(s + 4, 0);

// -- "set" -> 3 states
s += 5;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('s'), s);
// --- states
lexer_fsm_add_state(s, Token_Identifier); // s
lexer_fsm_add_state(s + 1, Token_Identifier); // se
lexer_fsm_add_state(s + 2, Token_Set); // set
// --- edges
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('e'), s + 1);
lexer_fsm_set_alnum_identifier(s, 'e');
lexer_fsm_add_edge(s + 1, lexer_fsm_get_char_index('t'), s + 2);
lexer_fsm_set_alnum_identifier(s + 1, 't');
lexer_fsm_set_alnum_identifier(s + 2, 0);

// -- "done" -> 4 states
s += 3;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('d'), s);
// --- states
lexer_fsm_add_state(s, Token_Identifier); // d
lexer_fsm_add_state(s + 1, Token_Identifier); // do
lexer_fsm_add_state(s + 2, Token_Identifier); // don
lexer_fsm_add_state(s + 3, Token_Done); // done
// --- edges
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('o'), s + 1);
lexer_fsm_set_alnum_identifier(s, 'o');
lexer_fsm_add_edge(s + 1, lexer_fsm_get_char_index('\n'), s + 2);
lexer_fsm_set_alnum_identifier(s + 1, '\n');
lexer_fsm_add_edge(s + 2, lexer_fsm_get_char_index('e'), s + 3);
lexer_fsm_set_alnum_identifier(s + 2, 'e');
lexer_fsm_set_alnum_identifier(s + 3, 0);

// - Double State Symbols
// -- = & == -> 2 states
s += 4;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('='), s);
// --- states
lexer_fsm_add_state(s, Token_Assignment); // =
lexer_fsm_add_state(s + 1, Token_Equal); // ==
// --- edges
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('='), s + 1);

```

```

// -- ! & != -> 2 states
s += 2;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('!'), s);
// --- states
lexer_fsm_add_state(s, Token_Not); // !
lexer_fsm_add_state(s + 1, Token_Not_Equal); // !=

// --- edges
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('='), s + 1);

// -- > & >= -> 2 states
s += 2;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('>'), s);
// --- states
lexer_fsm_add_state(s, Token_Bigger); // >
lexer_fsm_add_state(s + 1, Token_Bigger_Equal); // >=
// --- edges
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('='), s + 1);

// -- < & <= -> 2 states
s += 2;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('<'), s);
// --- states
lexer_fsm_add_state(s, Token_Smaller); // <
lexer_fsm_add_state(s + 1, Token_Smaller_Equal); // <=
// --- edges
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('='), s + 1);

// -- | & || -> 2 states
s += 2;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('|'), s);
// --- states
lexer_fsm_add_state(s, Token_Error); // |
lexer_fsm_add_state(s + 1, Token_Or); // ||
// --- edges
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('|'), s + 1);

// -- & & && -> 2 states
s += 2;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('&'), s);
// --- states
lexer_fsm_add_state(s, Token_Error); // &

```

```

lexer_fsm_add_state(s + 1, Token_And); // &&
// --- edges
lexer_fsm_add_edge(s, lexer_fsm_get_char_index('&'), s + 1);

// -- : & : ) -> 2 states
s += 2;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index(':'), s);
// --- states
lexer_fsm_add_state(s, Token_Colon); // :
lexer_fsm_add_state(s + 1, Token_Smiley); // :)
// --- edges
lexer_fsm_add_edge(s, lexer_fsm_get_char_index(')'), s + 1);

// - Single state symbols
s += 2;
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('+'), s);
lexer_fsm_add_state(s++, Token_Plus); // +
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('-'), s);
lexer_fsm_add_state(s++, Token_Minus); // -
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('*'), s);
lexer_fsm_add_state(s++, Token_Multiply); // *
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('/'), s);
lexer_fsm_add_state(s++, Token_Divide); // /
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('%'), s);
lexer_fsm_add_state(s++, Token_Modulu); // %
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index('('), s);
lexer_fsm_add_state(s++, Token_Open_Paren); // (
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index(')'), s);
lexer_fsm_add_state(s++, Token_Close_Paren); // )
lexer_fsm_add_starting_state_index(lexer_fsm_get_char_index(';'), s);
lexer_fsm_add_state(s++, Token_Semi_Colon); // ;
}

```

parser

parser_base.h

```
#pragma once

#include "../token/token.h"

// Number of terminals in the parasing table
// The terminals are the tokens. So the number of terminals is the same as the
// number of tokens
#define NUM_OF_TERMINALS NUM_OF_TOKENS
// Number of non-terminals in the parasing table
#define NUM_OF_NON_TERMINALS 14
// Number of states in the parser's table
#define PARSING_TABLE_NUM_OF_STATES 62

// Enum of all the possible actions a cell in the parsing table can do
typedef enum Action_Type
{
    Action_Error,
    Action_Shift,
    Action_Reduce,
    Action_Accept
} Action_Type;

// Enum of the possible symbol types
typedef enum Symbol_Type
{
    Terminal,
    Non_Terminal,
} Symbol_Type;

// Enum of all terminals. Exactly the same as the Token_Type enum
// Only for ease of use and readability
typedef Token_Type Terminal_Type;

// Enum of all the non-terminals in the grammar of the language
typedef enum Non_Terminal_Type
{
    Non_Terminal_PROG,
    Non_Terminal_BLOCK,
```

```

Non_Terminal_STMT,
Non_Terminal_DECL,
Non_Terminal_ASSIGN,
Non_Terminal_IF_ELSE,
Non_Terminal_WHILE,
Non_Terminal_L_LOG_E,
Non_Terminal_ELSE,
Non_Terminal_H_LOG_E,
Non_Terminal_BOOL_E,
Non_Terminal_E,
Non_Terminal_T,
Non_Terminal_F,
} Non_Terminal_Type;

```

parser.h

```

#pragma once

#include "parse_tree/parse_tree.h"
#include "parse_table/parse_table.h"
#include "parse_stack/parse_stack.h"
#include "../token/token.h"

// Number of production rules in the grammar of the language
#define NUM_OF_PRODUCTION_RULES 28

/* ----- Structs ----- */

// A struct to hold each production rule's length and non-terminal on its LHS
typedef struct Production_Rule
{
    // The non-terminal that is on the LHS of a production rule
    Non_Terminal_Type non_terminal_type;
    // The number of Terminals and Non-Terminals that are on the RHS of a production
    rule
    int rule_length;
} Production_Rule;

// Struct of the parser
typedef struct Parser
{

```

```
// The current token from the source code produced by the lexer
Token* token;
// The parser's parsing table, made from action & goto tables
// helps up to know which action to perform according to the next token
Parse_Table* parse_table;
// The parser's stack
Parse_Stack_Entry* parse_stack;
// Array of the production rules
// This will be used when we reduce by a production rule in the parsing phase
Production_Rule production_rules[NUM_OF_PRODUCTION_RULES];
// Array to store the starting state of the tokens at the start of a statement
// production rule in the language.
// This array is mainly used for better error reporting and error recovery.
// Every cell is filled with 0s, except for the tokens that are at the start of
// a statement production rule.
// Those token's cells are filled with the number of their produciton rule's
// starting state in the pushdown automaton
// according to the parsing table.
int tokens_starting_state[NUM_OF_TERMINALS];
} Parser;

/* ----- Functions ----- */

// Creates a new parser on the heap and returns a pointer to it
void parser_create();

// Frees everything we've allocated in the parser_create() function
void parser_destroy();

// Initializes the parser. Lexer, parse table, stack, and production rules
void parser_init();

// Initializes the production rules array according to the grammar of the language
void parser_init_production_rules();

// Initializes the array of statement production rule's starting states according to
// the parsing table of the language
void parser_init_tokens_starting_state();

// Get the starting state of a terminal
int parser_get_token_starting_state(Token_Type token_type);
```

```
// Shifts the current parser's token and goto state onto the parse stack
void parser_shift(int goto_state);

// Reduce by the production rule
void parser_reduce(int production_rule_num);

// Parses the source code and returns an Abstract Syntax Tree / Parse Tree that
represents the source code.
// Also updates the scope tree.
Parse_Tree_Node* parser_parse();
```

parser.c

```
#include <stdlib.h>
#include <stdbool.h>

#include "../global.h"

#include "parser.h"
#include "../general/general.h"
#include "../error_handler/error_handler.h"

void parser_create()
{
    // Create parser
    compiler.parser = (Parser*) calloc(1, sizeof(Parser));
    if (compiler.parser == NULL) exit_memory_error(__FILE__, __LINE__);

    // Create parser's parse table
    parse_table_create();
}

void parser_destroy()
{
    // check for NULL pointer
    if (compiler.parser != NULL)
    {
        // Free parser's token
        token_destroy(compiler.parser->token);
        compiler.parser->token = NULL;
```

```

// Free parser's parsing table
parse_table_destroy();

// Free parser's stack
parse_stack_destroy();

// Free the parser
free(compiler.parser);
compiler.parser = NULL;
}

}

void parser_init()
{
    // Initialize token to NULL
    compiler.parser->token = NULL;

    // Initialize parsing table
    parse_table_init();

    // Initialize parser's stack
    // Push the first stack entry onto the stack. For the first entry we only care
    about its goto_state
    // value because it's the first state we start from. In our case it's state
    number 0.
    parse_stack_push(parse_stack_init_entry(NULL, 0));

    // Initialize production_rules array according to grammar rules of the language
    parser_init_production_rules();

    // Initializes the array of statement production rule's starting states
    according to the parsing table of the language
    parser_init_tokens_starting_state();
}

void parser_init_production_rules()
{
    // Holds the number of the current production rule
    int prod_num = 0;
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_PROG, 5 };      // PROG -> prog_id : BLOCK :)
```

```

    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_BLOCK, 2 };      // BLOCK -> STMT_BLOCK
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_BLOCK, 1 };      // BLOCK -> done
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_STMT, 1 };      // STMT -> DECL
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_STMT, 1 };      // STMT -> ASSIGN
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_STMT, 1 };      // STMT -> IF_ELSE
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_STMT, 1 };      // STMT -> WHILE
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_DECL, 3 };      // DECL -> data_type id
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_ASSIGN, 5 };     // ASSIGN -> set id = L_LOG_E
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_IF_ELSE, 7 };   // IF_ELSE -> if ( L_LOG_E ) : BLOCK ELSE
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_ELSE, 3 };      // ELSE -> else : BLOCK
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_ELSE, 0 };      // ELSE -> epsilon
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_WHILE, 6 };     // WHILE -> while ( L_LOG_E ) : BLOCK
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_L_LOG_E, 3 };   // L_LOG_E -> L_LOG_E l_log_op H_LOG_E
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_L_LOG_E, 1 };   // L_LOG_E -> H_LOG_E
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_H_LOG_E, 3 };   // H_LOG_E -> H_LOG_E h_log_op BOOL_E
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_H_LOG_E, 1 };   // H_LOG_E -> BOOL_E
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_BOOL_E, 3 };    // BOOL_E -> BOOL_E bool_op E
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_BOOL_E, 1 };    // BOOL_E -> E
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_E, 3 };         // E -> E expr_op T
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_E, 1 };         // E -> T
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_T, 3 };         // T -> T term_op F

```

```

    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_T, 1 };           // T -> F
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_F, 1 };           // F -> id
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_F, 1 };           // F -> literal
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_F, 3 };           // F -> ( L_LOG_E )
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_F, 2 };           // F -> ! F
    compiler.parser->production_rules[prod_num++] = (Production_Rule) {
Non_Terminal_F, 2 };           // F -> - F
}

void parser_init_tokens_starting_state()
{
    // Those numbers (12, 13, 14, 15) are chosen based on the parsing table. More
precisely, on the shift actions in the action table.
    // The rest of the token's starting state is set to 0 because of the calloc used
to create the parser.
    // Token - 3 because at the enum of the token types, there are 3 additional
tokens at the start (error, whitespace, comment).
    // In my language only `int`, `char`, `set`, `if`, `while` are at the start of a
statement.
    // I've added `done` as well because if we are inside an inner block, then we
don't
    // want to go pass that block (that will make errors later for missmatching
start and end of block). This
    // creates better error reporting & handling.
    compiler.parser->tokens_starting_state[Token_Done - 3] = 7;
    compiler.parser->tokens_starting_state[Token_Int - 3] = 12;
    compiler.parser->tokens_starting_state[Token_Char - 3] = 12;
    compiler.parser->tokens_starting_state[Token_Set - 3] = 13;
    compiler.parser->tokens_starting_state[Token_If - 3] = 14;
    compiler.parser->tokens_starting_state[Token_While - 3] = 15;
}

int parser_get_token_starting_state(Token_Type token_type)
{
    // terminal_type - 3 because at the enum of the token types, there are 3
additional tokens at the start (error, whitespace, comment).
    return compiler.parser->tokens_starting_state[token_type - 3];
}

```

```
}  
  
void parser_shift(int goto_state)  
{  
    // Create a new terminal tree node from the current token  
    Parse_Tree_Node* tree_node = parse_tree_init_node(Terminal, compiler.parser->token->token_type, compiler.parser->token, NULL, 0);  
    // Create a new stack entry for created tree node  
    Parse_Stack_Entry* stack_entry = parse_stack_init_entry(tree_node, goto_state);  
    // Push created stack entry onto the stack  
    parse_stack_push(stack_entry);  
}  
  
void parser_reduce(int production_rule_num)  
{  
    // The produced stack entry  
    Parse_Stack_Entry* stack_entry;  
    // Get the current production rule  
    Production_Rule production_rule = compiler.parser->production_rules[production_rule_num];  
    // Create an array of the size of number of symbols on the RHS of the production  
    // rule that we reduce by  
    Parse_Tree_Node** children = (Parse_Tree_Node**)calloc(production_rule.rule_length, sizeof(Parse_Tree_Node*));  
    if (children == NULL) exit_memory_error(__FILE__, __LINE__);  
  
    // Pop Length(Production rule) entries from the stack, and put the trees of each  
    // entry as a child in the array.  
    // From right of array to the left because the production rule is "reversed" in  
    // the stack  
    for (int i = production_rule.rule_length - 1; i >= 0; i--)  
    {  
        // Pop entry from the stack  
        stack_entry = parse_stack_pop();  
        // Place its tree in the children array  
        children[i] = stack_entry->tree;  
        // Free popped entry  
        free(stack_entry);  
    }  
    // Create a new tree node from the non-terminal on the LHS of the production  
    // rule we reduce by
```

```
Parse_Tree_Node* tree_node = parse_tree_init_node(Non_Terminal,
production_rule.non_terminal_type, NULL, children, production_rule.rule_length);
    // Create a new stack entry using the created tree node and Goto table
    stack_entry = parse_stack_init_entry(tree_node, compiler.parser->parse_table-
>goto_table[compiler.parser->parse_stack-
>goto_state][production_rule.non_terminal_type]);
    // Push created entry onto the stack
    parse_stack_push(stack_entry);
}

Parse_Tree_Node* parser_parse()
{
    // The next state to go to
    int state;
    // Current action table cell
    Action action;

    // Input first token from the source code
    compiler.parser->token = lexer_get_next_token();

    // While not done parsing. We'll be done parsing by an Accept or Error.
    while (true)
    {
        // Get the next state from the top of the stack
        state = compiler.parser->parse_stack->goto_state;

        // Save the current cell in the action table
        action = compiler.parser->parse_table-
>action_table[state][parse_table_get_terminal_index(compiler.parser->token-
>token_type)];

        // If Action[state, token] == Shift
        if (action.action_type == Action_Shift)
        {
            parser_shift(action.state_or_rule);

            // Performs a semantic function if needed
            if (action.semantic_func != NULL)
                action.semantic_func();

            // Get next token from the source code
            compiler.parser->token = lexer_get_next_token();
        }
    }
}
```

```
        }
        // If Action[state, token] == Reduce
        else if (action.action_type == Action_Reduce)
        {
            parser_reduce(action.state_or_rule);

            // Performs a semantic function if needed
            if (action.semantic_func != NULL)
                action.semantic_func();

        }
        // If Action[state, token] == Accept
        else if (action.action_type == Action_Accept)
        {
            // When reached an Accept, the stack only has one entry other than the
            start entry, which contains the parse tree.

            // Disconnect the parse tree from the stack so it won't be destroyed
            later, and return it.

            Parse_Tree_Node* parse_tree = compiler.parser->parse_stack->tree;
            compiler.parser->parse_stack->tree = NULL;
            return parse_tree;

        }
        // If Action[state, token] == Error
        else
        {
            // If reached an Error output error message
            error_handler_report(compiler.line, Error_Syntax, "Unexpected token %s",
token_to_str(compiler.parser->token));

            // Perform the specific error report & recovery function
            action.error_func();

            // If reached EOF, return NULL to prevent further processing and
            infinite loop
            if (compiler.parser->token->token_type == Token_Eof)
                return NULL;
        }
    }
}
```

parse_stack

parse_stack.h

```
#pragma once

#include "../parse_tree/parse_tree.h"

/* ----- Structs ----- */

// The struct of a entry of the parser's stack.
// The stack is represents by a linear linked list, where the stack of the list is
// the top of the stack.
typedef struct Parse_Stack_Entry
{
    Parse_Tree_Node* tree;                      // Each entry of the stack holds a tree
    int goto_state;                            // The state to go to in the next
iteration of the parser
    struct Parse_Stack_Entry* next_entry;      // The next entry of the stack, the one
on the bottom of the current one
} Parse_Stack_Entry;

/* ----- Functions ----- */

// Create a new stack entry, updates its properties, and returns a pointer to it
Parse_Stack_Entry* parse_stack_init_entry(Parse_Tree_Node* tree, int goto_state);

// Frees the whole stack
void parse_stack_destroy();

// Pushes a stack entry onto the the compiler's parse stack
void parse_stack_push(Parse_Stack_Entry* entry);

// Pops the top entry from the stack and returns the pointer to it
Parse_Stack_Entry* parse_stack_pop();
```

parse_stack.c

```
#include <stdlib.h>

#include "../../global.h"
```

```
#include "parse_stack.h"
#include "../../general/general.h"

Parse_Stack_Entry* parse_stack_init_entry(Parse_Tree_Node* tree, int goto_state)
{
    // Create a new stack entry
    Parse_Stack_Entry* entry = (Parse_Stack_Entry*) calloc(1,
sizeof(Parse_Stack_Entry));
    if (entry == NULL)
    {
        parse_tree_destroy(tree);
        exit_memory_error(__FILE__, __LINE__);
    }

    // Update entry properties
    entry->tree = tree;
    entry->goto_state = goto_state;
    entry->next_entry = NULL;

    return entry;
}

void parse_stack_destroy()
{
    Parse_Stack_Entry* entry;

    // While the stack is not empty
    while (compiler.parser->parse_stack != NULL)
    {
        // Pop and destroy the entry
        entry = parse_stack_pop();

        // Free entry if not NULL
        if (entry != NULL)
        {
            parse_tree_destroy(entry->tree);
            free(entry);
        }
    }
}

void parse_stack_push(Parse_Stack_Entry* entry)
```

```
{  
    // Connect the entry as the stack of the stack  
    entry->next_entry = compiler.parser->parse_stack;  
    // Make stack point to the new entry  
    compiler.parser->parse_stack = entry;  
}  
  
Parse_Stack_Entry* parse_stack_pop()  
{  
    // Check for NULL pointer  
    if (compiler.parser->parse_stack == NULL)  
        return NULL;  
  
    // Extract the top node  
    Parse_Stack_Entry* top_entry = compiler.parser->parse_stack;  
    // Advance the stack of the stack list  
    compiler.parser->parse_stack = compiler.parser->parse_stack->next_entry;  
    // Disconnect it from the stack list  
    top_entry->next_entry = NULL;  
  
    return top_entry;  
}
```

parse_table

parse_table.h

```
#pragma once

#include "../parser_base.h"

/* ----- Structs ----- */

// Struct of an entry in the Action table
typedef struct Action
{
    // Error, Shift, Reduce, Accept
    Action_Type action_type;
    // If the current cell is a Shift, then this int represents the state to go next
    // If the current cell is a Reduce, then this int represents the number of the
    production rule to reduce by
    int state_or_rule;
    // Pointer to a function to be called when we reach that cell. Will also be part
    of the semantic analysis
    void (*semantic_func) ();
    // Pointer to a function to be called when we reach that cell. Takes part in the
    error reporting and recovery.
    void (*error_func) ();
} Action;

// Struct to hold the parsing table. Action & Goto tables all together
typedef struct Parse_Table
{
    // Action table. each cell contains a Action structure
    Action action_table[PARSING_TABLE_NUM_OF_STATES][NUM_OF_TERMINALS];
    // Goto table. Each cell contains only an int that represent a state number
    int goto_table[PARSING_TABLE_NUM_OF_STATES][NUM_OF_NON_TERMINALS];
} Parse_Table;

/* ----- Functions ----- */

// Creates a new Parse_Table on the heap and sets the Parser's parse table to point
to it
void parse_table_create();
```

```

// Frees everything we've allocated the parse_table_create() function
void parse_table_destroy();

// Initializes the parsing table according to the grammar rules
void parse_table_init();

// Returns the index of a terminal in the action table
int parse_table_get_terminal_index(Terminal_Type terminal_type);

// Inserts a new cell into the action table
void parse_table_insert_action(int state, int terminal_index, Action action);

// Inserts a new cell into the goto table
void parse_table_insert_goto(int state, int non_terminal_index, int goto_state);

// Prints the parsing table in a nice format
void parse_table_print();

```

parse_table.c

```

#include <stdlib.h>
#include <stdio.h>

#include "../global.h"

#include "parse_table.h"
#include "../general/general.h"
#include "../error_handler/error_handler.h"
#include "../semantic/semantic.h"

void parse_table_create()
{
    // Create a new parsing table
    compiler.parser->parse_table = (Parse_Table*) malloc(1, sizeof(Parse_Table));
    if (compiler.parser->parse_table == NULL) exit_memory_error(__FILE__, __LINE__);
}

void parse_table_destroy()
{
    // check for NULL pointer
    if (compiler.parser != NULL)

```

```
{  
    free(compiler.parser->parse_table);  
    compiler.parser->parse_table = NULL;  
}  
}  
  
int parse_table_get_terminal_index(Terminal_Type terminal_type)  
{  
    // Returning the terminal_type - 3 because the tokens enum has 3 extra tokens at  
    // the start of it,  
    // Token_Error, TokenWhitespace, TokenComment, which we don't care about here.  
    // We want to start the indexing from 0 and not from 3.  
    return terminal_type - 3;  
}  
  
void parse_table_insert_action(int state, int terminal_index, Action action)  
{  
    compiler.parser->parse_table->action_table[state][terminal_index] = action;  
}  
  
void parse_table_insert_goto(int state, int non_terminal_index, int goto_state)  
{  
    compiler.parser->parse_table->goto_table[state][non_terminal_index] =  
    goto_state;  
}  
  
void parse_table_print()  
{  
    printf("\n");  
  
    int state, symbol;  
  
    // Print top row  
    printf("      ");  
    // Terminals  
    for (symbol = 0; symbol < NUM_OF_TERMINALS; symbol++)  
        printf(" %2d", symbol + 2);  
    // Non-Terminals  
    printf(" ");  
    for (symbol = 0; symbol < NUM_OF_NON_TERMINALS; symbol++)  
        printf(" %2d", symbol);  
    printf("\n");  
}
```

```
// Print _
printf("      ");
for (symbol = 0; symbol < NUM_OF_TERMINALS; symbol++)
    printf("____");
for (symbol = 0; symbol < NUM_OF_NON_TERMINALS; symbol++)
    printf("____");
printf("_\n");

// Print table
for (state = 0; state < PARSING_TABLE_NUM_OF_STATES; state++)
{
    printf("%2d. |   ", state);

    // Action
    for (symbol = 0; symbol < NUM_OF_TERMINALS; symbol++)
    {
        if (compiler.parser->parse_table-
>action_table[state][symbol].action_type == Action_Error)
            printf(" . ");
        else if (compiler.parser->parse_table-
>action_table[state][symbol].action_type == Action_Shift)
            printf("%c%-2d ", 'S', compiler.parser->parse_table-
>action_table[state][symbol].state_or_rule);
        else if (compiler.parser->parse_table-
>action_table[state][symbol].action_type == Action_Reduce)
            printf("%c%-2d ", 'R', compiler.parser->parse_table-
>action_table[state][symbol].state_or_rule);
        else
            printf(" A   ");
    }
    // Goto
    for (symbol = 0; symbol < NUM_OF_NON_TERMINALS; symbol++)
        if (compiler.parser->parse_table->goto_table[state][symbol] != 0)
            printf("%2d ", compiler.parser->parse_table-
>goto_table[state][symbol]);
        else
            printf(" . ");

    printf("\n");
}
```

```

    printf("\n");
}

void parse_table_init()
{
    // This function is dependent on the parsing table, Action & Goto, that I've
    // created before hand according to the grammar of the language.
    // I explained how I created it very thoroughly in the strategy chapter of my
project book.

    // The current state in the parsing table
    int s;
    // The current action table entry
    Action action;

    // State 0
    s = 0;
    // - Action
    // -- Shift
    // --- `prog` -> S2
    action = (Action) { Action_Shift, 2, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Prog),
action);
    // - Goto
    // -- PROG -> 1
    parse_table_insert_goto(s, Non_Terminal_PROG, 1);

    // - Error function
    for (int i = 0; i < NUM_OF_TERMINALS; i++)
        compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_prog;

    // State 1
    s = 1;
    // - Action
    // -- Accept
    // --- EOF -> Accept
    action = (Action) { Action_Accept, 0, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Eof), action);
}

```

```

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_eof;

// State 2
s = 2;
// - Action
// -- Shift
// --- id -> S3
action = (Action) { Action_Shift, 3, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_prog_name;

// State 3
s = 3;
// - Action
// -- Shift
// --- `:` -> S4
// ---- Here I'm not using semantic_enter_block() when encountering : because
here is the enter to the global
// ---- scope, which is already created when initializing the scope tree.
action = (Action) { Action_Shift, 4, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Colon),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_colon_state_3;

// State 4
s = 4;
// - Action
// -- Shift

```

```

// --- `done` -> S7
action = (Action) { Action_Shift, 7, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
// --- `int` -> S12
action = (Action) { Action_Shift, 12, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
// --- `char` -> S12
action = (Action) { Action_Shift, 12, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
// --- `set` -> S13
action = (Action) { Action_Shift, 13, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
// --- `if` -> S14
action = (Action) { Action_Shift, 14, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
// --- `while` -> S15
action = (Action) { Action_Shift, 15, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Goto
// -- BLOCK -> 5
parse_table_insert_goto(s, Non_Terminal_BLOCK, 5);
// -- STMT -> 6
parse_table_insert_goto(s, Non_Terminal_STMT, 6);
// -- DECL -> 8
parse_table_insert_goto(s, Non_Terminal_DECL, 8);
// -- ASSIGN -> 9
parse_table_insert_goto(s, Non_Terminal_ASSIGN, 9);
// -- IF_ELSE -> 10
parse_table_insert_goto(s, Non_Terminal_IF_ELSE, 10);
// -- WHILE -> 11
parse_table_insert_goto(s, Non_Terminal_WHILE, 11);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_4_group;

// State 5

```

```

s = 5;
// - Action
// -- Shift
// --- `:`` -> S16
action = (Action) { Action_Shift, 16, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smiley),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_smiley;

// State 6
s = 6;
// - Action
// -- Shift
// --- `done` -> S7
action = (Action) { Action_Shift, 7, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
// --- `int` -> S12
action = (Action) { Action_Shift, 12, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
// --- `char` -> S12
action = (Action) { Action_Shift, 12, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
// --- `set` -> S13
action = (Action) { Action_Shift, 13, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
// --- `if` -> S14
action = (Action) { Action_Shift, 14, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
// --- `while` -> S15
action = (Action) { Action_Shift, 15, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Goto
// -- BLOCK -> 17
parse_table_insert_goto(s, Non_Terminal_BLOCK, 17);

```

```

// -- STMT -> 6
parse_table_insert_goto(s, Non_Terminal_STMT, 6);
// -- DECL -> 8
parse_table_insert_goto(s, Non_Terminal_DECL, 8);
// -- ASSIGN -> 9
parse_table_insert_goto(s, Non_Terminal_ASSIGN, 9);
// -- IF_ELSE -> 10
parse_table_insert_goto(s, Non_Terminal_IF_ELSE, 10);
// -- WHILE -> 11
parse_table_insert_goto(s, Non_Terminal_WHILE, 11);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_4_group;

// State 7
s = 7;
// - Action
// -- Reduce
// --- for (`:`), `done`, `int`, `char`, `set`, `if`, `else`, `while`) -> R2
// ---- Using semantic_exit_block() because when we reduce by rule no. 2 we know
we've exited a scope.
// ---- Rule no. 2: BLOCK -> done
action = (Action) { Action_Reduce, 2, semantic_exit_block, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smiley),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Else),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)

```

```

    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_7_group;

// State 8
s = 8;
// - Action
// -- Reduce
// --- for (`done`, `int`, `char`, `set`, `if`, `while`) -> R3
action = (Action) { Action_Reduce, 3, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_4_group;

// State 9
s = 9;
// - Action
// -- Reduce
// --- for (`done`, `int`, `char`, `set`, `if`, `while`) -> R4
action = (Action) { Action_Reduce, 4, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Error function

```

```
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_4_group;

// State 10
s = 10;
// - Action
// -- Reduce
// --- for (`done`, `int`, `char`, `set`, `if`, `while`) -> R5
action = (Action) { Action_Reduce, 5, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_4_group;

// State 11
s = 11;
// - Action
// -- Reduce
// --- for (`done`, `int`, `char`, `set`, `if`, `while`) -> R6
action = (Action) { Action_Reduce, 6, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);
```

```

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_4_group;

// State 12
s = 12;
// - Action
// -- Shift
// --- id -> S18
action = (Action) { Action_Shift, 18, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_identifier;

// State 13
s = 13;
// - Action
// -- Shift
// --- id -> S19
action = (Action) { Action_Shift, 19, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_identifier;

// State 14
s = 14;
// - Action
// -- Shift
// --- `(` -> S20
action = (Action) { Action_Shift, 20, NULL, NULL };

```

```

    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Open_Paren),
action);

    // - Error function
    for (int i = 0; i < NUM_OF_TERMINALS; i++)
        compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_open_paren;

    // State 15
    s = 15;
    // - Action
    // -- Shift
    // --- `(` -> S21
    action = (Action) { Action_Shift, 21, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Open_Paren),
action);

    // - Error function
    for (int i = 0; i < NUM_OF_TERMINALS; i++)
        compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_open_paren;

    // State 16
    s = 16;
    // - Action
    // -- Reduce
    // --- EOF -> R0
    action = (Action) { Action_Reduce, 0, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Eof), action);

    // - Error function
    for (int i = 0; i < NUM_OF_TERMINALS; i++)
        compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_eof;

    // State 17
    s = 17;
    // - Action
    // -- Reduce
    // --- for (`:`), `done`, `int`, `char`, `set`, `if`, `else`, `while` ) -> R1

```

```

action = (Action) { Action_Reduce, 1, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smiley),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Else),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_7_group;

// State 18
s = 18;
// - Action
// -- Shift
// --- `;` -> S22
action = (Action) { Action_Shift, 22, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_semi_colon_state_18;

// State 19
s = 19;
// - Action
// -- Shift
// --- `=` -> S23
action = (Action) { Action_Shift, 23, NULL, NULL };

```

```

    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Assignment),
action);

    // - Error function
    for (int i = 0; i < NUM_OF_TERMINALS; i++)
        compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_assign;

    // State 20
    s = 20;
    // - Action
    // -- Shift
    // --- id -> S30
    action = (Action) { Action_Shift, 30, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);
    // --- `(` -> S32
    action = (Action) { Action_Shift, 32, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Open_Paren),
action);
    // --- `-` -> S34
    action = (Action) { Action_Shift, 34, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
    // --- number -> S31
    action = (Action) { Action_Shift, 31, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Number),
action);
    // --- character -> S31
    action = (Action) { Action_Shift, 31, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Character),
action);
    // --- `!` -> S33
    action = (Action) { Action_Shift, 33, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not), action);

    // - Goto
    // -- L_LOG_E -> 24
    parse_table_insert_goto(s, Non_Terminal_L_LOG_E, 24);
    // -- H_LOG_E -> 25
    parse_table_insert_goto(s, Non_Terminal_H_LOG_E, 25);
    // -- BOOL_E -> 26

```

```

parse_table_insert_goto(s, Non_Terminal_BOOL_E, 26);
// -- E -> 27
parse_table_insert_goto(s, Non_Terminal_E, 27);
// -- T -> 28
parse_table_insert_goto(s, Non_Terminal_T, 28);
// -- F -> 29
parse_table_insert_goto(s, Non_Terminal_F, 29);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_20_group;

// State 21
s = 21;
// - Action
// -- Shift
// --- id -> S30
action = (Action) { Action_Shift, 30, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);
// --- `(` -> S32
action = (Action) { Action_Shift, 32, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Open_Paren),
action);
// --- `-` -> S34
action = (Action) { Action_Shift, 34, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
// --- number -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Number),
action);
// --- character -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Character),
action);
// --- `!` -> S33
action = (Action) { Action_Shift, 33, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not), action);

// - Goto

```

```

// -- L_LOG_E -> 35
parse_table_insert_goto(s, Non_Terminal_L_LOG_E, 35);
// -- H_LOG_E -> 25
parse_table_insert_goto(s, Non_Terminal_H_LOG_E, 25);
// -- BOOL_E -> 26
parse_table_insert_goto(s, Non_Terminal_BOOL_E, 26);
// -- E -> 27
parse_table_insert_goto(s, Non_Terminal_E, 27);
// -- T -> 28
parse_table_insert_goto(s, Non_Terminal_T, 28);
// -- F -> 29
parse_table_insert_goto(s, Non_Terminal_F, 29);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_20_group;

// State 22
s = 22;
// - Action
// -- Reduce
// --- for (`done`, `int`, `char`, `set`, `if`, `while`) -> R7
// ---- Using semantic_decl() because after reducing by rule no. 7, we must
check for unique identifier.
// ---- Rule no. 7: DECL -> data_type id ;
action = (Action) { Action_Reduce, 7, semantic_decl, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_4_group;

```

```

// State 23
s = 23;
// - Action
// -- Shift
// --- id -> S30
action = (Action) { Action_Shift, 30, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);
// --- `(` -> S32
action = (Action) { Action_Shift, 32, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Open_Paren),
action);
// --- `-' -> S34
action = (Action) { Action_Shift, 34, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
// --- number -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Number),
action);
// --- character -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Character),
action);
// --- `!' -> S33
action = (Action) { Action_Shift, 33, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not), action);

// - Goto
// -- L_LOG_E -> 36
parse_table_insert_goto(s, Non_Terminal_L_LOG_E, 36);
// -- H_LOG_E -> 25
parse_table_insert_goto(s, Non_Terminal_H_LOG_E, 25);
// -- BOOL_E -> 26
parse_table_insert_goto(s, Non_Terminal_BOOL_E, 26);
// -- E -> 27
parse_table_insert_goto(s, Non_Terminal_E, 27);
// -- T -> 28
parse_table_insert_goto(s, Non_Terminal_T, 28);
// -- F -> 29
parse_table_insert_goto(s, Non_Terminal_F, 29);

```

```

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_20_group;

// State 24
s = 24;
// - Action
// -- Shift
// --- `)` -> S37
action = (Action) { Action_Shift, 37, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
// --- `||` -> S38
action = (Action) { Action_Shift, 38, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_24_group;

// State 25
s = 25;
// - Action
// -- Shift
// --- `&&` -> S39
action = (Action) { Action_Shift, 39, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);
// -- Reduce
// --- for (`;`, `)` , `||` ) -> R14
// ---- Using semantic_set_type() because after reducing by rule no. 14, we need
to set the LHS non-terminal's type to the RHS type
// ---- Rule no. 14: L_LOG_E -> H_LOG_E
action = (Action) { Action_Reduce, 14, semantic_set_type, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);

```

```

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

// State 26
s = 26;
// - Action
// -- Shift
// --- for (`==`, `!=`, `>`, `>=`, `<`, `<=`) -> S40
action = (Action) { Action_Shift, 40, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smaller),
action);
parse_table_insert_action(s,
parse_table_get_terminal_index(Token_Smaller_Equal), action);
// -- Reduce
// --- for (`;`, `)` , `||` , `&&` ) -> R16
// ----- Using semantic_set_type() because after reducing by rule no. 16, we need
to set the LHS non-terminal's type to the RHS type
// ----- Rule no. 16: H_LOG_E -> BOOL_E
action = (Action) { Action_Reduce, 16, semantic_set_type, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

```

```

// State 27
s = 27;
// - Action
// -- Shift
// --- for (`+`, `-`) -> S41
action = (Action) { Action_Shift, 41, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Plus),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
// -- Reduce
// --- for (`;`, `)` , `||` , `&&` , `==` , `!=` , `>` , `>=` , `<` , `<=` ) -> R18
// ---- Using semantic_set_type() because after reducing by rule no. 18, we need
to set the LHS non-terminal's type to the RHS type
// ---- Rule no. 18: BOOL_E -> E
action = (Action) { Action_Reduce, 18, semantic_set_type, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smaller),
action);
parse_table_insert_action(s,
parse_table_get_terminal_index(Token_Smaller_Equal), action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

// State 28
s = 28;

```

```

// - Action
// -- Shift
// --- for (`*`, `/`, `%`) -> S42
action = (Action) { Action_Shift, 42, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Multiply),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Divide),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Modulu),
action);
// -- Reduce
// --- for (`;`, `)` , `||` , `&&` , `==` , `!=` , `>` , `>=` , `<` , `<=` , `+` , `-` ) ->
R20
// ---- Using semantic_set_type() because after reducing by rule no. 20, we need
to set the LHS non-terminal's type to the RHS type
// ---- Rule no. 20: E -> T
action = (Action) { Action_Reduce, 20, semantic_set_type, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smaller),
action);
parse_table_insert_action(s,
parse_table_get_terminal_index(Token_Smaller_Equal), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Plus),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)

```

```
compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

// State 29
s = 29;
// - Action
// -- Reduce
// --- for (`;`, `)`), `||`, `&&`, `==`, `!=`, `>`, `>=`, `<`, `<=`, `+`, `-`,
`*`, `/`, `%`) -> R22
// ---- Using semantic_set_type() because after reducing by rule no. 22, we need
to set the LHS non-terminal's type to the RHS type
// ---- Rule no. 22: T -> F
action = (Action) { Action_Reduce, 22, semantic_set_type, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smaller),
action);
parse_table_insert_action(s,
parse_table_get_terminal_index(Token_Smaller_Equal), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Plus),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Multiply),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Divide),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Modulu),
action);
```

```
// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

// State 30
s = 30;
// - Action
// -- Reduce
// --- for (`;`, `)` , `||` , `&&` , `==` , `!=` , `>` , `>=` , `<` , `<=` , `+` , `-` ,
`*` , `/` , `%`) -> R23
// ---- Using semantic_F_to_id() because after reducing by rule no. 23, we must
check that id exists and set type accordingly.
// ---- Rule no. 23: F -> id
action = (Action) { Action_Reduce, 23, semantic_F_to_id, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smaller),
action);
parse_table_insert_action(s,
parse_table_get_terminal_index(Token_Smaller_Equal), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Plus),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Multiply),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Divide),
action);
```

```

    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Modulu),
action);

    // - Error function
    for (int i = 0; i < NUM_OF_TERMINALS; i++)
        compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

    // State 31
    s = 31;
    // - Action
    // -- Reduce
    // --- for (`;`, `)` , `||` , `&&` , `==` , `!=` , `>` , `>=` , `<` , `<=` , `+` , `-` ,
`*` , `/` , `%` ) -> R24
    // ---- Using semantic_F_to_literal() because after reducing by rule no. 24, we
need to set type accordingly.
    // ---- Rule no. 24: F -> literal
    action = (Action) { Action_Reduce, 24, semantic_F_to_literal, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smaller),
action);
    parse_table_insert_action(s,
parse_table_get_terminal_index(Token_Smaller_Equal), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Plus),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Multiply),
action);

```

```

    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Divide),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Modulu),
action);

    // - Error function
    for (int i = 0; i < NUM_OF_TERMINALS; i++)
        compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

    // State 32
    s = 32;
    // - Action
    // -- Shift
    // --- id -> S30
    action = (Action) { Action_Shift, 30, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);
    // --- `(` -> S32
    action = (Action) { Action_Shift, 32, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Open_Paren),
action);
    // --- `-' -> S34
    action = (Action) { Action_Shift, 34, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
    // --- number -> S31
    action = (Action) { Action_Shift, 31, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Number),
action);
    // --- character -> S31
    action = (Action) { Action_Shift, 31, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Character),
action);
    // --- `!` -> S33
    action = (Action) { Action_Shift, 33, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not), action);

    // - Goto
    // -- L_LOG_E -> 43
    parse_table_insert_goto(s, Non_Terminal_L_LOG_E, 43);
    // -- H_LOG_E -> 25

```

```

parse_table_insert_goto(s, Non_Terminal_H_LOG_E, 25);
// -- BOOL_E -> 26
parse_table_insert_goto(s, Non_Terminal_BOOL_E, 26);
// -- E -> 27
parse_table_insert_goto(s, Non_Terminal_E, 27);
// -- T -> 28
parse_table_insert_goto(s, Non_Terminal_T, 28);
// -- F -> 29
parse_table_insert_goto(s, Non_Terminal_F, 29);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_20_group;

// State 33
s = 33;
// - Action
// -- Shift
// --- id -> S30
action = (Action) { Action_Shift, 30, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);
// --- `(` -> S32
action = (Action) { Action_Shift, 32, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Open_Paren),
action);
// --- `-` -> S34
action = (Action) { Action_Shift, 34, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
// --- number -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Number),
action);
// --- character -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Character),
action);
// --- `!` -> S33
action = (Action) { Action_Shift, 33, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not), action);

```

```

// - Goto
// -- F -> 44
parse_table_insert_goto(s, Non_Terminal_F, 44);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_20_group;

// State 34
s = 34;
// - Action
// -- Shift
// --- id -> S30
action = (Action) { Action_Shift, 30, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);
// --- `(` -> S32
action = (Action) { Action_Shift, 32, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Open_Paren),
action);
// --- `-` -> S34
action = (Action) { Action_Shift, 34, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
// --- number -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Number),
action);
// --- character -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Character),
action);
// --- `!` -> S33
action = (Action) { Action_Shift, 33, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not), action);

// - Goto
// -- F -> 44
parse_table_insert_goto(s, Non_Terminal_F, 45);

```

```

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_20_group;

// State 35
s = 35;
// - Action
// -- Shift
// --- `)` -> S46
action = (Action) { Action_Shift, 46, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
// --- `||` -> S38
action = (Action) { Action_Shift, 38, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_24_group;

// State 36
s = 36;
// - Action
// -- Shift
// --- `;` -> S47
action = (Action) { Action_Shift, 47, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
// --- `||` -> S38
action = (Action) { Action_Shift, 38, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_36_group;

// State 37

```

```

s = 37;
// - Action
// -- Shift
// --- `:` -> S48
// ---- Using semantic_enter_block() because every time we encounter : we know
we've entered a new block.
action = (Action) { Action_Shift, 48, semantic_enter_block, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Colon),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_colon_state_37;

// State 38
s = 38;
// - Action
// -- Shift
// --- id -> S30
action = (Action) { Action_Shift, 30, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);
// --- `(` -> S32
action = (Action) { Action_Shift, 32, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Open_Paren),
action);
// --- `-` -> S34
action = (Action) { Action_Shift, 34, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
// --- number -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Number),
action);
// --- character -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Character),
action);
// --- `!` -> S33
action = (Action) { Action_Shift, 33, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not), action);

```

```

// - Goto
// -- H_LOG_E -> 49
parse_table_insert_goto(s, Non_Terminal_H_LOG_E, 49);
// -- BOOL_E -> 26
parse_table_insert_goto(s, Non_Terminal_BOOL_E, 26);
// -- E -> 27
parse_table_insert_goto(s, Non_Terminal_E, 27);
// -- T -> 28
parse_table_insert_goto(s, Non_Terminal_T, 28);
// -- F -> 29
parse_table_insert_goto(s, Non_Terminal_F, 29);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_20_group;

// State 39
s = 39;
// - Action
// -- Shift
// --- id -> S30
action = (Action) { Action_Shift, 30, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);
// --- `(` -> S32
action = (Action) { Action_Shift, 32, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Open_Paren),
action);
// --- `-` -> S34
action = (Action) { Action_Shift, 34, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
// --- number -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Number),
action);
// --- character -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Character),
action);

```

```

// --- `!` -> S33
action = (Action) { Action_Shift, 33, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not), action);

// - Goto
// -- BOOL_E -> 50
parse_table_insert_goto(s, Non_Terminal_BOOL_E, 50);
// -- E -> 27
parse_table_insert_goto(s, Non_Terminal_E, 27);
// -- T -> 28
parse_table_insert_goto(s, Non_Terminal_T, 28);
// -- F -> 29
parse_table_insert_goto(s, Non_Terminal_F, 29);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_20_group;

// State 40
s = 40;
// - Action
// -- Shift
// --- id -> S30
action = (Action) { Action_Shift, 30, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);
// --- `(` -> S32
action = (Action) { Action_Shift, 32, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Open_Paren),
action);
// --- `-' -> S34
action = (Action) { Action_Shift, 34, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
// --- number -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Number),
action);
// --- character -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };

```

```

parse_table_insert_action(s, parse_table_get_terminal_index(Token_Character),
action);
// --- `!` -> S33
action = (Action) { Action_Shift, 33, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not), action);

// - Goto
// -- E -> 51
parse_table_insert_goto(s, Non_Terminal_E, 51);
// -- T -> 28
parse_table_insert_goto(s, Non_Terminal_T, 28);
// -- F -> 29
parse_table_insert_goto(s, Non_Terminal_F, 29);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_20_group;

// State 41
s = 41;
// - Action
// -- Shift
// --- id -> S30
action = (Action) { Action_Shift, 30, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);
// --- `(` -> S32
action = (Action) { Action_Shift, 32, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Open_Paren),
action);
// --- `-' -> S34
action = (Action) { Action_Shift, 34, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
// --- number -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Number),
action);
// --- character -> S31
action = (Action) { Action_Shift, 31, NULL, NULL };

```

```

    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Character),
action);
    // --- `!` -> S33
    action = (Action) { Action_Shift, 33, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not), action);

    // - Goto
    // -- T -> 52
    parse_table_insert_goto(s, Non_Terminal_T, 52);
    // -- F -> 29
    parse_table_insert_goto(s, Non_Terminal_F, 29);

    // - Error function
    for (int i = 0; i < NUM_OF_TERMINALS; i++)
        compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_20_group;

    // State 42
    s = 42;
    // - Action
    // -- Shift
    // --- id -> S30
    action = (Action) { Action_Shift, 30, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Identifier),
action);
    // --- `(` -> S32
    action = (Action) { Action_Shift, 32, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Open_Paren),
action);
    // --- `-' -> S34
    action = (Action) { Action_Shift, 34, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
    // --- number -> S31
    action = (Action) { Action_Shift, 31, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Number),
action);
    // --- character -> S31
    action = (Action) { Action_Shift, 31, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Character),
action);
    // --- `!` -> S33

```

```

action = (Action) { Action_Shift, 33, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not), action);

// - Goto
// -- F -> 53
parse_table_insert_goto(s, Non_Terminal_F, 53);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_20_group;

// State 43
s = 43;
// - Action
// -- Shift
// --- `)` -> S54
action = (Action) { Action_Shift, 54, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
// --- `||` -> S38
action = (Action) { Action_Shift, 38, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_24_group;

// State 44
s = 44;
// - Action
// -- Reduce
// --- for (`;`, `)``, `||`, `&&`, `==`, `!=`, `>`, `>=`, `<`, `<=`, `+`, `-`,
`*`, `/`, `%`) -> R26
// ---- Using semantic_F_to_unary_op_F() because after reducing by rule no. 26,
we need to set type accordingly.
// ---- Rule no. 26: F -> ! F
action = (Action) { Action_Reduce, 26, semantic_F_to_unary_op_F, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);

```

```

    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smaller),
action);
    parse_table_insert_action(s,
parse_table_get_terminal_index(Token_Smaller_Equal), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Plus),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Multiply),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Divide),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Modulu),
action);

    // - Error function
    for (int i = 0; i < NUM_OF_TERMINALS; i++)
        compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

    // State 45
    s = 45;
    // - Action
    // -- Reduce
    // --- for (`;`, `)` , `||` , `&&` , `==` , `!=` , `>` , `>=` , `<` ,
`<=` , `+` , `-` ,
`*` , `/` , `%` ) -> R27
    // ---- Using semantic_F_to_unary_op_F() because after reducing by rule no. 26,
we need to set type accordingly.
    // ---- Rule no. 27: F -> - F
    action = (Action) { Action_Reduce, 27, semantic_F_to_unary_op_F, NULL };

```

```

    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smaller),
action);
    parse_table_insert_action(s,
parse_table_get_terminal_index(Token_Smaller_Equal), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Plus),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Multiply),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Divide),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Modulu),
action);

    // - Error function
    for (int i = 0; i < NUM_OF_TERMINALS; i++)
        compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

    // State 46
    s = 46;
    // - Action
    // -- Shift
    // --- `:` -> S55
    // ---- Using semantic_enter_block() because every time we encounter : we know
we've entered a new block.
    action = (Action) { Action_Shift, 55, semantic_enter_block, NULL };

```

```

    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Colon),
action);

    // - Error function
    for (int i = 0; i < NUM_OF_TERMINALS; i++)
        compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_colon_state_46;

// State 47
s = 47;
// - Action
// -- Reduce
// --- for (`done`, `int`, `char`, `set`, `if`, `while`) -> R8
// ----- Using semantic_assign() because after reducing by rule no. 8, we must
check for:
// ----- 1. Existing identifier
// ----- 2. Matching types
// ----- Rule no. 7: DECL -> data_type id ;
action = (Action) { Action_Reduce, 8, semantic_assign, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_4_group;

// State 48
s = 48;
// - Action
// -- Shift
// --- `done` -> S7
action = (Action) { Action_Shift, 7, NULL, NULL };

```

```
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),  
action);  
    // --- `int` -> S12  
    action = (Action) { Action_Shift, 12, NULL, NULL };  
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);  
    // --- `char` -> S12  
    action = (Action) { Action_Shift, 12, NULL, NULL };  
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),  
action);  
    // --- `set` -> S13  
    action = (Action) { Action_Shift, 13, NULL, NULL };  
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);  
    // --- `if` -> S14  
    action = (Action) { Action_Shift, 14, NULL, NULL };  
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);  
    // --- `while` -> S15  
    action = (Action) { Action_Shift, 15, NULL, NULL };  
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),  
action);  
  
    // - Goto  
    // -- BLOCK -> 56  
    parse_table_insert_goto(s, Non_Terminal_BLOCK, 56);  
    // -- STMT -> 6  
    parse_table_insert_goto(s, Non_Terminal_STMT, 6);  
    // -- DECL -> 8  
    parse_table_insert_goto(s, Non_Terminal_DECL, 8);  
    // -- ASSIGN -> 9  
    parse_table_insert_goto(s, Non_Terminal_ASSIGN, 9);  
    // -- IF_ELSE -> 10  
    parse_table_insert_goto(s, Non_Terminal_IF_ELSE, 10);  
    // -- WHILE -> 11  
    parse_table_insert_goto(s, Non_Terminal_WHILE, 11);  
  
    // - Error function  
    for (int i = 0; i < NUM_OF_TERMINALS; i++)  
        compiler.parser->parse_table->action_table[s][i].error_func =  
error_handler_report_expected_state_4_group;  
  
    // State 49  
    s = 49;  
    // - Action
```

```

// -- Shift
// --- `&&` -> S39
action = (Action) { Action_Shift, 39, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);
// -- Reduce
// --- for (`;`, `)` , `||` ) -> R13
// ---- Using semantic_type_check() because after reducing by rule no. 13, we
must check matching types of the operands.
// ---- Rule no. 13: L_LOG_E -> L_LOG_E l_log_op H_LOG_E
action = (Action) { Action_Reduce, 13, semantic_type_check, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

// State 50
s = 50;
// - Action
// -- Shift
// --- for (`==`, `!=`, `>`, `>=`, `<`, `<=`) -> S40
action = (Action) { Action_Shift, 40, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smaller),
action);
parse_table_insert_action(s,
parse_table_get_terminal_index(Token_Smaller_Equal), action);
// -- Reduce
// --- for (`;`, `)` , `||` , `&&` ) -> R15

```

```

// ---- Using semantic_type_check() because after reducing by rule no. 15, we
must check matching types of the operands.
// ---- Rule no. 15: H_LOG_E -> H_LOG_E h_log_op BOOL_E
action = (Action) { Action_Reduce, 15, semantic_type_check, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

// State 51
s = 51;
// - Action
// -- Shift
// --- for (`+`, `-`) -> S41
action = (Action) { Action_Shift, 41, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Plus),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
// -- Reduce
// --- for (`;`, `)` , `||` , `&&` , `==` , `!=` , `>` , `>=` , `<` , `<=` ) -> R17
// ---- Using semantic_type_check() because after reducing by rule no. 17, we
must check matching types of the operands.
// ---- Rule no. 17: BOOL_E -> BOOL_E bool_op E
action = (Action) { Action_Reduce, 17, semantic_type_check, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not_Equal),
action);

```

```

    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smaller),
action);
    parse_table_insert_action(s,
parse_table_get_terminal_index(Token_Smaller_Equal), action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

// State 52
s = 52;
// - Action
// -- Shift
// --- for (`*`, `/`, `%`) -> S42
action = (Action) { Action_Shift, 42, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Multiply),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Divide),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Modulu),
action);
    // -- Reduce
    // --- for (`;`, `)` , `||` , `&&` , `==` , `!=` , `>` , `>=` , `<` , `<=` , `+` , `-` ) ->
R19
    // ---- Using semantic_type_check() because after reducing by rule no. 19, we
must check matching types of the operands.
    // ---- Rule no. 19: E -> E expr_op T
    action = (Action) { Action_Reduce, 19, semantic_type_check, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Equal),
action);

```

```

    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smaller),
action);
    parse_table_insert_action(s,
parse_table_get_terminal_index(Token_Smaller_Equal), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Plus),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);

    // - Error function
    for (int i = 0; i < NUM_OF_TERMINALS; i++)
        compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

// State 53
s = 53;
// - Action
// -- Reduce
// --- for (`;`, `)` , `||` , `&&` , `==` , `!=` , `>` , `>=` , `<` , `<=` , `+` , `-` ,
`*` , `/` , `%` ) -> R21
// ---- Using semantic_type_check() because after reducing by rule no. 21, we
must check matching types of the operands.
// ---- Rule no. 21: T -> T term_op F
action = (Action) { Action_Reduce, 21, semantic_type_check, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger),
action);

```

```

    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smaller),
action);
    parse_table_insert_action(s,
parse_table_get_terminal_index(Token_Smaller_Equal), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Plus),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Multiply),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Divide),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Modulu),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

// State 54
s = 54;
// - Action
// -- Reduce
// --- for (`;`, `)``, `||`, `&&`, `==`, `!=`, `>`, `>=`, `<`, `<=`, `+`, `-`,
`*`, `/`, `%`) -> R25
// ---- Using semantic_F_to_L_LOG_E() because after reducing by rule no. 25, we
need to set type accordingly.
// ---- Rule no. 25: F -> ( L_LOG_E )
action = (Action) { Action_Reduce, 25, semantic_F_to_L_LOG_E, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Semi_Colon),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Close_Paren),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Or), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_And), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Equal),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Not_Equal),
action);

```

```

    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Bigger_Equal),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Smaller),
action);
    parse_table_insert_action(s,
parse_table_get_terminal_index(Token_Smaller_Equal), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Plus),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Minus),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Multiply),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Divide),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Modulu),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_25_group;

// State 55
s = 55;
// - Action
// -- Shift
// --- `done` -> S7
action = (Action) { Action_Shift, 7, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
// --- `int` -> S12
action = (Action) { Action_Shift, 12, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
// --- `char` -> S12
action = (Action) { Action_Shift, 12, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
// --- `set` -> S13
action = (Action) { Action_Shift, 13, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);

```

```

// --- `if` -> S14
action = (Action) { Action_Shift, 14, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
// --- `while` -> S15
action = (Action) { Action_Shift, 15, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Goto
// -- BLOCK -> 57
parse_table_insert_goto(s, Non_Terminal_BLOCK, 57);
// -- STMT -> 6
parse_table_insert_goto(s, Non_Terminal_STMT, 6);
// -- DECL -> 8
parse_table_insert_goto(s, Non_Terminal_DECL, 8);
// -- ASSIGN -> 9
parse_table_insert_goto(s, Non_Terminal_ASSIGN, 9);
// -- IF_ELSE -> 10
parse_table_insert_goto(s, Non_Terminal_IF_ELSE, 10);
// -- WHILE -> 11
parse_table_insert_goto(s, Non_Terminal WHILE, 11);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_4_group;

// State 56
s = 56;
// - Action
// -- Shift
// --- `else` -> S59
action = (Action) { Action_Shift, 59, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Else),
action);
// -- Reduce
// --- for (`done`, `int`, `char`, `set`, `if`, `while`) -> R11
action = (Action) { Action_Reduce, 11, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);

```

```

    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Goto
// -- ELSE -> 58
parse_table_insert_goto(s, Non_Terminal_ELSE, 58);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_56_group;

// State 57
s = 57;
// - Action
// -- Reduce
// --- for (`done`, `int`, `char`, `set`, `if`, `while`) -> R12
action = (Action) { Action_Reduce, 12, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_4_group;

// State 58
s = 58;
// - Action
// -- Reduce

```

```

// --- for (`done`, `int`, `char`, `set`, `if`, `while`) -> R9
action = (Action) { Action_Reduce, 9, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_4_group;

// State 59
s = 59;
// - Action
// -- Shift
// --- `:` -> S60
// ---- Using semantic_enter_block() because every time we encounter : we know
we've entered a new block.
action = (Action) { Action_Shift, 60, semantic_enter_block, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Colon),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_colon_state_59;

// State 60
s = 60;
// - Action
// -- Shift
// --- `done` -> S7
action = (Action) { Action_Shift, 7, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);

```

```

// --- `int` -> S12
action = (Action) { Action_Shift, 12, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
// --- `char` -> S12
action = (Action) { Action_Shift, 12, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
// --- `set` -> S13
action = (Action) { Action_Shift, 13, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
// --- `if` -> S14
action = (Action) { Action_Shift, 14, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
// --- `while` -> S15
action = (Action) { Action_Shift, 15, NULL, NULL };
parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Goto
// -- BLOCK -> 61
parse_table_insert_goto(s, Non_Terminal_BLOCK, 61);
// -- STMT -> 6
parse_table_insert_goto(s, Non_Terminal_STMT, 6);
// -- DECL -> 8
parse_table_insert_goto(s, Non_Terminal_DECL, 8);
// -- ASSIGN -> 9
parse_table_insert_goto(s, Non_Terminal_ASSIGN, 9);
// -- IF_ELSE -> 10
parse_table_insert_goto(s, Non_Terminal_IF_ELSE, 10);
// -- WHILE -> 11
parse_table_insert_goto(s, Non_Terminal_WHILE, 11);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_4_group;

// State 61
s = 61;
// - Action
// -- Reduce
// --- for (`done`, `int`, `char`, `set`, `if`, `while`) -> R10

```

```
action = (Action) { Action_Reduce, 10, NULL, NULL };
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Done),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Int), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Char),
action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_Set), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_If), action);
    parse_table_insert_action(s, parse_table_get_terminal_index(Token_While),
action);

// - Error function
for (int i = 0; i < NUM_OF_TERMINALS; i++)
    compiler.parser->parse_table->action_table[s][i].error_func =
error_handler_report_expected_state_4_group;
}
```

parse_tree

parse_tree.h

```
#pragma once

#include "../parser_base.h"
#include "../../token/token.h"
#include "../../semantic/semantic.h"

/* ----- Structs ----- */

// Struct of a node in the parse tree
typedef struct Parse_Tree_Node
{
    Symbol_Type symbol_type;           // Terminal / Non-Terminal
    int symbol;                      // The terminal or non-terminal kind of the
current node
    Token* token;                    // If a node is a terminal then it will have
a token, otherwise it will be NULL
    struct Parse_Tree_Node** children; // Array of a node pointers which represents
a node's children
    int num_of_children;             // The length of the childrens array
    // Attribute of a node in the tree.
    // If the node is part of expression it will have a data type.
    Data_Type data_type;
    // The register index in the register array of the code generator that holds
this node's result.
    // If the node is part of expression it will have a register holding it's
result.
    int register_number;
} Parse_Tree_Node;

/* ----- Functions ----- */

// Creates a new Parse_Tree_Node with the given attributes and returns a pointer to
it
Parse_Tree_Node* parse_tree_init_node(Symbol_Type symbol_type, int symbol, Token*
token, Parse_Tree_Node** children, int num_of_children);

// Recursive function to free a node and all of it's children trees
void parse_tree_destroy(Parse_Tree_Node* root);
```

```
// Prints the parse tree in a nice format
void parse_tree_print(Parse_Tree_Node* root);
```

parse_tree.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

#include "parse_tree.h"
#include "../..../general/general.h"
#include "../..../code_generator/code_generator_base.h"

Parse_Tree_Node* parse_tree_init_node(Symbol_Type symbol_type, int symbol, Token* token, Parse_Tree_Node** children, int num_of_children)
{
    // Create a new Parse_Tree_Node
    Parse_Tree_Node* node = (Parse_Tree_Node*) calloc(1, sizeof(Parse_Tree_Node));
    if (node == NULL)
    {
        // Free all of the children trees in the children array
        for (int i = 0; i < num_of_children; i++)
            parse_tree_destroy(children[i]);

        // Free the children pointers array
        free(children);

        // Free the token
        token_destroy(token);

        exit_memory_error(__FILE__, __LINE__);
    }

    // Update node's properties
    node->symbol_type = symbol_type;
    node->symbol = symbol;
    node->token = token;
    node->children = children;
    node->num_of_children = num_of_children;
}
```

```

// Initialize node with no data type
node->data_type = Data_Type_NULL;

// Initialize node with no register
node->register_number = NO_REGISTER;

return node;
}

void parse_tree_destroy(Parse_Tree_Node* root)
{
    // If reached a leaf stop recursion
    if (root == NULL)
        return;

    // Destroy children
    for (int i = 0; i < root->num_of_children; i++)
        parse_tree_destroy(root->children[i]);

    // Free the node's token
    token_destroy(root->token);

    // Destroy current node
    free(root->children);
    free(root);
}

// ----- Print tree helper functions -----

// Converts a non-terminal type to a string representation
const char* parser_tree_non_terminal_to_str(Non_Terminal_Type non_terminal_type);

// The recursive function that actually prints the parse tree
void parse_tree_print_tree_rec(Parse_Tree_Node* root, char* indent, bool is_last);

// Prints a single node in the parse tree
void parse_tree_print_node(Parse_Tree_Node* node);

const char* parser_tree_non_terminal_to_str(Non_Terminal_Type non_terminal_type)
{
    switch (non_terminal_type)

```

```

{
    case Non_Terminal_PROG: return "PROG";
    case Non_Terminal_BLOCK: return "BLOCK";
    case Non_Terminal_STMT: return "STMT";
    case Non_Terminal_DECL: return "DECL";
    case Non_Terminal_ASSIGN: return "ASSIGN";
    case Non_Terminal_IF_ELSE: return "IF_ELSE";
    case Non_Terminal WHILE: return "WHILE";
    case Non_Terminal_L_LOG_E: return "L_LOG_E";
    case Non_Terminal_ELSE: return "ELSE";
    case Non_Terminal_H_LOG_E: return "H_LOG_E";
    case Non_Terminal_BOOL_E: return "BOOL_E";
    case Non_Terminal_E: return "E";
    case Non_Terminal_T: return "T";
    case Non_Terminal_F: return "F";

    default: return "Don't know that non-terminal type... ;|";
}
}

void parse_tree_print(Parse_Tree_Node* root)
{
    char indent[256] = { 0 };
    parse_tree_print_tree_rec(root, indent, true);
}

void parse_tree_print_node(Parse_Tree_Node* node)
{
    if (node->symbol_type == Terminal)
        printf("%s", token_to_str(node->token));
    else
        printf("%s", parser_tree_non_terminal_to_str(node->symbol));
}

void parse_tree_print_tree_rec(Parse_Tree_Node* root, char* indent, bool is_last)
{
    if (root == NULL)
        return;

    char marker[256] = " |---";
    char cur_indent[256] = " |   ";

```

```
if (is_last)
{
    sprintf(marker, " `---");
    sprintf(cur_indent, "      ");
}

printf("%s", indent);
printf("%s", marker);

printf(" ");
parse_tree_print_node(root);
printf("\n");

strcat(indent, cur_indent);

// Save current indentation before recursing
char tmp[256];
strcpy(tmp, indent);

for (int i = 0; i < root->num_of_children; i++)
{
    parse_tree_print_tree_rec(root->children[i], indent, i == root-
>num_of_children - 1);
    strcpy(indent, tmp);
}
}
```

scope_tree

scope_tree.h

```
#pragma once

#include "scope/scoped.h"
#include "symbol_table/symbol_table_entry/symbol_table_entry.h"

/* ----- Structs ----- */

// Struct of the symbol tables scope tree.
// This is the struct we will use as our symbol table.
// The tree represents the hierarchy of scopes in the source program.
typedef struct Scope_Tree
{
    Scope* global_scope; // The root of the tree - the global scope
    Scope* current_scope; // The current scope we are at
} Scope_Tree;

/* ----- Functions ----- */

// Create a new symbol tables scopes tree, initialized with an empty global scope
void scope_tree_create();

// Destory the scope tree
void scope_tree_destroy();

// Frees all allocated memory in the tree of scopes
void scope_tree_destroy_tree(Scope* global_scope);

// Creates a new scope child for the current scope in the scope tree.
void scope_tree_add_scope();

// Moves the current scope in the scope tree to point to its parent scope
void scope_tree_goto_parent();

// Moves the current scope in the scope tree to point to its next child scope.
// Also updates the current scope's current child to the next child.
void scope_tree_goto_child();
```

```
// Travers the tree from the current scope up to the global scope and tries to find
the given
// identifier. If managed to find it, returns a pointer to it. Otherwise return
NULL.
Symbol_Table_Entry* scope_tree_fetch(char* identifier);

// Resets the current_child_index field to be STARTING_CHILD_INDEX for every scope
in the scope tree.
// Used before code generation so I could traverse the scope tree the same as in the
parsing stage.
void scope_tree_reset_child_index(Scope* global_scope);

// Prints the scope tree in a nice format
void scope_tree_print();
```

scope_tree.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

#include "../global.h"

#include "scope_tree.h"
#include "../general/general.h"

void scope_tree_create()
{
    // Create a new scope tree
    compiler.scope_tree = (Scope_Tree*) calloc(1, sizeof(Scope_Tree));
    if (compiler.scope_tree == NULL) exit_memory_error(__FILE__, __LINE__);

    // Create global scope with it's parent set to NULL (The only scope with NULL
    // father)
    compiler.scope_tree->global_scope = scope_init(NULL);
    compiler.scope_tree->current_scope = compiler.scope_tree->global_scope;
}

void scope_tree_destroy()
{
```

```

// Check for NULL pointer
if (compiler.scope_tree != NULL)
{
    scope_tree_destroy_tree(compiler.scope_tree->global_scope);
    free(compiler.scope_tree);
    compiler.scope_tree = NULL;
}

void scope_tree_destroy_tree(Scope* global_scope)
{
    if (global_scope == NULL)
        return;

    // Destroy all of the current node's children
    for (int i = 0; i < global_scope->num_of_children; i++)
        scope_tree_destroy_tree(global_scope->children[i]);

    // Free current scope
    free(global_scope->children);
    symbol_table_destroy(global_scope->symbol_table);
    free(global_scope);
}

void scope_tree_add_scope()
{
    // Create a new scope and set its parent to the current scope in the scope tree
    Scope* new_scope = scope_init(compiler.scope_tree->current_scope);

    // Resizes the children array of the current scope in the scope tree, and add 1
    // to the number of children
    compiler.scope_tree->current_scope->children = (Scope**)
realloc(compiler.scope_tree->current_scope->children, ++compiler.scope_tree-
>current_scope->num_of_children * sizeof(Scope*));
    if (compiler.scope_tree->current_scope->children == NULL)
exit_memory_error(__FILE__, __LINE__);

    // Add the new scope to the last index of the children array of the current
    // scope in the scope tree
    compiler.scope_tree->current_scope->children[compiler.scope_tree->current_scope-
>num_of_children - 1] = new_scope;
}

```

```
void scope_tree_goto_parent()
{
    // Check if the current scope is the global scope.
    // If not, advance the current scope to the parent scope of the current scope.
    // If yes, remain at the global scope.
    if (compiler.scope_tree->current_scope->parent != NULL)
    {
        compiler.scope_tree->current_scope = compiler.scope_tree->current_scope-
>parent;
    }
}

void scope_tree_goto_child()
{
    // Check if there is a child to go to next.
    // If there is, advances the current scope's current_child_index and advances
    // the current scope to the next child scope.
    // If there is not, the current scope remains as it is.
    if (compiler.scope_tree->current_scope->current_child_index + 1 <=
compiler.scope_tree->current_scope->num_of_children - 1)
    {
        // Advance the current child to the next child
        compiler.scope_tree->current_scope->current_child_index++;

        // Advance the current scope to the next child scope
        compiler.scope_tree->current_scope = compiler.scope_tree->current_scope-
>children[compiler.scope_tree->current_scope->current_child_index];
    }
}

Symbol_Table_Entry* scope_tree_fetch(char* identifier)
{
    Scope* current_scope = compiler.scope_tree->current_scope;
    Symbol_Table_Entry* entry = NULL;

    // Loop up to the global scope, or until identifier found
    while (current_scope != NULL && entry == NULL)
    {
        entry = symbol_table_fetch(current_scope->symbol_table, identifier);
        current_scope = current_scope->parent;
    }
}
```

```
    return entry;
}

void scope_tree_reset_child_index(Scope* global_scope)
{
    if (global_scope == NULL)
        return;

    // Reset current scope's child index
    global_scope->current_child_index = STARTING_CHILD_INDEX;

    // Reset all of the current node's children
    for (int i = 0; i < global_scope->num_of_children; i++)
        scope_tree_reset_child_index(global_scope->children[i]);
}

void scope_tree_print_tree(Scope* global_scope, char* indent, bool is_last);

void scope_tree_print()
{
    char indent[256] = { 0 };
    scope_tree_print_tree(compiler.scope_tree->global_scope, indent, true);
}

void scope_tree_print_tree(Scope* global_scope, char* indent, bool is_last)
{
    if (global_scope == NULL)
        return;

    char marker[256] = " |---";
    char cur_indent[256] = " |     ";

    if (is_last)
    {
        sprintf(marker, " `---");
        sprintf(cur_indent, "       ");
    }

    printf("%s", indent);
    printf("%s", marker);
```

```
printf(" ");
printf("Entries: %d", global_scope->symbol_table->num_of_entries);
printf("\n");

strcat(indent, cur_indent);

// Save current indentation before recursing
char tmp[256];
strcpy(tmp, indent);

for (int i = 0; i < global_scope->num_of_children; i++)
{
    scope_tree_print_tree(global_scope->children[i], indent, i == global_scope-
>num_of_children - 1);
    strcpy(indent, tmp);
}
}
```

scope

scope.h

```
#pragma once

#include "../symbol_table/symbol_table.h"

// The starting value of the current_child_index field
#define STARTING_CHILD_INDEX -1

/* ----- Structs ----- */

// Struct of a single node in the tree, scope, in the tree of symbol tables scopes
typedef struct Scope
{
    // Current scope's symbol table
    Symbol_Table* symbol_table;
    // Array of scopes which are the children of the current scope.
    // Each child represent a sub scope of the current scope.
    struct Scope** children;
    int num_of_children;           // Length of the children array
    int current_child_index;       // Index of the current child in children array
    // Pointer to the current scope's parent in the symbol tables scopes tree
    struct Scope* parent;
    // Number of entries seen up to this point.
    // In this scope and in all the scope on the path up to the root scope.
    // Used in the code generation process in the address computation for variables.
    int available_entries;
} Scope;

/* ----- Functions ----- */

// Creates a scope with an empty symbol table and the given parent pointer
Scope* scope_init(Scope* parent);
```

scope.c

```
#include <stdlib.h>

#include "scope.h"
#include "../../general/general.h"
```

```
Scope* scope_init(Scope* parent)
{
    // Create a new scope
    Scope* scope = (Scope*) calloc(1, sizeof(Scope));
    if (scope == NULL) exit_memory_error(__FILE__, __LINE__);

    // Initialize without children
    scope->children = NULL;
    scope->num_of_children = 0;
    scope->current_child_index = STARTING_CHILD_INDEX;

    // Set initial value of available entries to 0
    scope->available_entries = 0;

    // Create empty symbol table
    scope->symbol_table = symbol_table_create();

    // Set parent
    scope->parent = parent;

    return scope;
}
```

symbol_table

symbol_table_base.h

```
#pragma once

#include "../token/token.h"

// The initial capacity of a symbol table when creating it.
// Number of indices in the array.
#define SYMBOL_TABLE_INITIAL_CAPACITY 32

// The maximum value the load factor of the hash table can have before expanding the
// table
#define SYMBOL_TABLE_MAX_LAMBDA 2.5

// Enum of all the possible types of entries in the symbol table.
// In my language this is really not that necessary because all the identifiers
// are variables. But I added it so I will have the option in the future to support
// other types of identifiers, functions for example.
typedef enum Entry_Type
{
    Entry_Type_Variable
} Entry_Type;
```

symbol_table.h

```
#pragma once

#include <stdint.h>

#include "symbol_table_base.h"
#include "symbol_table_entry/symbol_table_entry.h"

/* ----- Structs ----- */

// Struct of a symbol table using chaining
typedef struct Symbol_Table
{
    Symbol_Table_Entry** entries; // Array of the entry pointers of the symbol
table
```

```

int capacity;           // Max capacity of symbol table
int num_of_entries;    // Current number of entries in the symbol table
int num_of_indices_occupied; // Number of indices that have entries in them
} Symbol_Table;

/* ----- Functions ----- */

// Creates a new symbol table with initial capacity and return a pointer to it
Symbol_Table* symbol_table_create();

// Frees all memory allocated for a symbol table
void symbol_table_destroy(Symbol_Table* symbol_table);

// Constants used by the hash function
#define FNV_OFFSET 14695981039346656037UL
#define FNV_PRIME 1099511628211UL

// Calculates and returns a 64-bit FNV-1a hash for the given identifier.
// See description: https://en.wikipedia.org/wiki/Fowler-Noll-Vo\_hash\_function
uint64_t symbol_table_hash(char* identifier);

// Inserts an entry into the symbol table
void symbol_table_insert(Symbol_Table* symbol_table, Symbol_Table_Entry* entry);

// Returns a pointer to the entry in the symbol table that corresponds to the given
// identifier.
// If the identifier was not found, returns NULL
Symbol_Table_Entry* symbol_table_fetch(Symbol_Table* symbol_table, char*
identifier);

// Expands the symbol table to be twice its current size.
// To assure constant time operations on the symbol table
void symbol_table_expand(Symbol_Table* symbol_table);

// Prints the symbol table in a nice format
void symbol_table_print(Symbol_Table* symbol_table);

```

symbol_table.c

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>

#include "symbol_table.h"
#include "../general/general.h"

Symbol_Table* symbol_table_create()
{
    // Create a new symbol table
    Symbol_Table* symbol_table = (Symbol_Table*) calloc(1, sizeof(Symbol_Table));
    if (symbol_table == NULL) exit_memory_error(__FILE__, __LINE__);

    // Initial symbol table properties
    symbol_table->capacity = SYMBOL_TABLE_INITIAL_CAPACITY;
    symbol_table->num_of_entries = 0;
    symbol_table->num_of_indices_occupied = 0;

    // Create symbol table entries array
    symbol_table->entries = (Symbol_Table_Entry**) calloc(symbol_table->capacity,
    sizeof(Symbol_Table_Entry));
    // Check for allocation error
    if (symbol_table->entries == NULL)
    {
        free(symbol_table);
        exit_memory_error(__FILE__, __LINE__);
    }

    return symbol_table;
}

void symbol_table_destroy(Symbol_Table* symbol_table)
{
    Symbol_Table_Entry* cur_entry;
    Symbol_Table_Entry* prev_entry;

    // Free symbol table entries
    for (int i = 0; i < symbol_table->capacity; i++)
    {
        // For each index, free all the entries in it
        cur_entry = symbol_table->entries[i];
        while (cur_entry != NULL)
        {
            prev_entry = cur_entry;
```

```

        cur_entry = cur_entry->next_entry;
        symbol_table_entry_destroy(prev_entry);
    }

}

// Free entries array
free(symbol_table->entries);

// Free symbol table
free(symbol_table);
}

uint64_t symbol_table_hash(char* identifier)
{
    uint64_t hash = FNV_OFFSET;
    while (*identifier != '\0')
    {
        hash ^= (uint64_t) (unsigned char) (*identifier);
        hash *= FNV_PRIME;
        identifier++;
    }

    return hash;
}

void symbol_table_insert(Symbol_Table* symbol_table, Symbol_Table_Entry* entry)
{
    // Get index of the entry in the table
    uint64_t hash = symbol_table_hash(entry->identifier);
    int index = (int) (hash % (uint64_t) (symbol_table->capacity - 1));

    // Check if added to new index, if so adds 1 to the number of indices occupied
    // in the table
    if (symbol_table->entries[index] == NULL)
        symbol_table->num_of_indices_occupied++;

    // Insert the entry into the table
    entry->next_entry = symbol_table->entries[index];
    symbol_table->entries[index] = entry;

    // Increment the number of entries in the table
    symbol_table->num_of_entries++;
}

```

```
// Calculate lambda - The load factor of the hash table
// λ = num_of_entries / number_of_indices_occupied
float lambda = (float) symbol_table->num_of_entries / (float) symbol_table-
>num_of_indices_occupied;

// If λ is greater then the SYMBOL_TABLE_MAX_LAMBDA, expand table to ensure
constant time operations on the hash table
if (lambda > SYMBOL_TABLE_MAX_LAMBDA)
    symbol_table_expand(symbol_table);
}

Symbol_Table_Entry* symbol_table_fetch(Symbol_Table* symbol_table, char* identifier)
{
    // Get index of the entry in the table
    uint64_t hash = symbol_table_hash(identifier);
    int index = (int) (hash % (uint64_t) (symbol_table->capacity - 1));

    Symbol_Table_Entry* entry = symbol_table->entries[index];

    // Go over all the entries in that index and find the wanted entry
    while (entry != NULL)
    {
        // If found wanted entry then return it
        if (strcmp(entry->identifier, identifier) == 0)
            return entry;

        // If not found, advance to the next entry
        entry = entry->next_entry;
    }

    // If entry was not found, return NULL
    return NULL;
}

void symbol_table_expand(Symbol_Table* symbol_table)
{
    // Create new entries array
    int new_capacity = symbol_table->capacity * 2;
    Symbol_Table_Entry** new_entries = (Symbol_Table_Entry**) calloc(new_capacity,
sizeof(Symbol_Table_Entry));
    // Check for allocation error
```

```
if (new_entries == NULL)
{
    symbol_table_destroy(symbol_table);
    exit_memory_error(__FILE__, __LINE__);
}

// Zero the symbol table's number of entries and number of occupied indices
symbol_table->num_of_entries = 0;
symbol_table->num_of_indices_occupied = 0;

Symbol_Table_Entry* old_entry;
Symbol_Table_Entry* new_entry;
uint64_t hash;
int new_index;

// Move already existing entries into the new entries table
for (int i = 0; i < symbol_table->capacity; i++)
{
    old_entry = symbol_table->entries[i];

    // Go over all the entries in that index
    while (old_entry != NULL)
    {
        // Save the current entry in the old array to the new array, and advance
        new_entry = old_entry;
        old_entry = old_entry->next_entry;

        // Get old_entry's new index in the new table
        hash = symbol_table_hash(new_entry->identifier);
        new_index = (int) (hash % (uint64_t) (new_capacity - 1));

        // Check if added to new index, if so adds 1 to the number of indices
        // occupied in the table
        if (new_entries[new_index] == NULL)
            symbol_table->num_of_indices_occupied++;

        // Insert the old entry into the new table
        new_entry->next_entry = new_entries[new_index];
        new_entries[new_index] = new_entry;

        // Increment the number of entries in the table
        symbol_table->num_of_entries++;
    }
}
```

```
    }

}

// Free old entries array, and update the new entries array in the table
free(symbol_table->entries);
symbol_table->capacity = new_capacity;
symbol_table->entries = new_entries;
}

void symbol_table_print(Symbol_Table* symbol_table)
{
    printf("%d entries in total\n", symbol_table->num_of_entries);
    printf("%d / %d indices are occupied\n\n", symbol_table-
>num_of_indices_occupied, symbol_table->capacity);

    Symbol_Table_Entry* entry;
    for (int i = 0; i < symbol_table->capacity; i++)
    {
        printf("%2d| ", i);

        entry = symbol_table->entries[i];

        while (entry != NULL)
        {
            printf("[");
            symbol_table_entry_print(entry);
            printf("] -> ");
            entry = entry->next_entry;
        }

        printf("\n");
    }
}
```

symbol_table_entry

symbol_table_entry.h

```
#pragma once

#include <stdbool.h>

#include "../symbol_table_base.h"
#include "../../semantic/semantic.h"

/* ----- Structs ----- */

// Declare struct Scope (Defined in ../../scope/scope.h file)
struct Scope;

// Struct of an entry in the symbol table
typedef struct Symbol_Table_Entry
{
    Entry_Type entry_type;                      // The type of the entry - variable,
function, ...
    char* identifier;                          // The identifier name which is the key
for each entry
    Data_Type data_type;                      // The type of the identifier - int,
char, ...
    // Pointer to the scope that entry is located at.
    // Used in the code generation process to go in O(1) to the scope of a given
entry
    struct Scope* scope;
    // Whether that symbol is a global symbol or not.
    // Used in the code generation process to determine whether a variable
    // should be considred from the data segment of from the stack
    bool is_global;
    // The number of the entry in the current scope.
    // At which order it was entred. If it's the first, second, third, ...
    // Used in the code generation process to resolve variables addresses relative
to bp
    int num_in_scope;
    struct Symbol_Table_Entry* next_entry; // Pointer to the next entry in the
linked list of entries in that particular index
} Symbol_Table_Entry;
```

```
/* ----- Functions ----- */

// Creates a new symbol table entry with the specified properties
Symbol_Table_Entry* symbol_table_entry_init(Entry_Type entry_type, char* identifier,
Data_Type data_type);

// Frees memory allocated for an entry
void symbol_table_entry_destroy(Symbol_Table_Entry* entry);

// Prints an entry in a nice format
void symbol_table_entry_print(Symbol_Table_Entry* entry);
```

symbol_table_entry.c

```
#include <stdlib.h>
#include <stdio.h>

#include "../.../global.h"

#include "symbol_table_entry.h"
#include "../.../general/general.h"
#include "../.../semantic/semantic.h"

Symbol_Table_Entry* symbol_table_entry_init(Entry_Type entry_type, char* identifier,
Data_Type data_type)
{
    // Create entry
    Symbol_Table_Entry* entry = (Symbol_Table_Entry*) calloc(1,
sizeof(Symbol_Table_Entry));
    if (entry == NULL)
    {
        free(identifier);
        exit_memory_error(__FILE__, __LINE__);
    }

    // Update entry's properties
    entry->scope = compiler.scope_tree->current_scope;
    entry->entry_type = entry_type;
    entry->identifier = identifier;
    entry->data_type = data_type;
    entry->next_entry = NULL;
```

```
// The number in the scope is the number of entries in the current scope's
symbol_table + 1
// because the number of entries in the current scope start from 0
entry->num_in_scope = compiler.scope_tree->current_scope->symbol_table-
>num_of_entries + 1;

// If the current scope is the global scope then is_global will be true.
// Else, is_global will be false.
entry->is_global = compiler.scope_tree->current_scope == compiler.scope_tree-
>global_scope;

return entry;
}

void symbol_table_entry_destroy(Symbol_Table_Entry* entry)
{
    // check for NULL pointer
    if (entry != NULL)
    {
        free(entry->identifier);
        free(entry);
    }
}

void symbol_table_entry_print(Symbol_Table_Entry* entry)
{
    // Check for NULL pointer
    if (entry == NULL)
        return;

    if (entry->data_type == Data_Type_Int)
        printf("%d. int %s", entry->num_in_scope, entry->identifier);

    else if (entry->data_type == Data_Type_Char)
        printf("%d. char %s", entry->num_in_scope, entry->identifier);

    else
        printf("Don't know that data type... ;|");
}
```

semantic

semantic.h

```
#pragma once

#include <stdbool.h>

/* ----- Structs ----- */

// Enum of all the possible types of an identifier in the language.
// This enum is only for readability and ease of use.
typedef enum Data_Type
{
    Data_Type_NULL = 0,           // For tree nodes that are not part of an
expression
    Data_Type_Int = Token_Int,    // int
    Data_Type_Char = Token_Char, // char
} Data_Type;

/* ----- Functions ----- */

// Checks if it's possible to assing expression's type to the identifier's type
bool semantic_check_assign_compatibility(Data_Type id_type, Data_Type expr_type);

// Checks if the data type of the left and right operands is compatible with the
operator.
// If yes returns the data type of the combined expression.
// If not return Data_Type_NULL
Data_Type semantic_check_binary_op_compatibility(Data_Type left_op, Token* operator,
Data_Type right_op);

// Checks if the data type of the operand is compatible with the operator.
// If yes returns the data type of the expression.
// If not return Data_Type_NULL
Data_Type semantic_check_unary_op_compatibility(Data_Type operand, Token* operator);

// Called after encountering :
// After : we know we are now in a new block, which means a new scope.
// Adds a scope child to the current scope, and moves the current scope to that
child.
void semantic_enter_block();
```

```
// Called after reduction by the production rule: BLOCK -> done (R2)
// When done is encountered, we know we've came out of a scope,
// so move back to the parent scope in the scope tree.
void semantic_exit_block();

// Checks after reduction by the production rule: DECL -> data_type id ; (R7)
// if the identifier is declared in the current scope or not.
// If not inserts a new entry into the current scope's symbol with the known data.
// If yes, reports a semantic error of identifier already exists.
void semantic_decl();

// Checks after reduction by the production rule: ASSIGN -> set id = L_LOG_E ; (R8)
// if id exists in the symbol table, and if the type of that id equals the type of
L_LOG_E.
// If either of the above is false, reports a semantic error for the matching error.
void semantic_assign();

// After reduction by an expression production rule of the form: Non_Terminal_1 ->
Non_terminal_2
// sets Non_Terminal_1.type = Non_Terminal_2.type
// Also makes Non_Terminal_1 to be Non_Terminal_2 to make the tree simpler
void semantic_set_type();

// After reduction by any prodution rule with the form: Operand Operator Operand
// the 3 children of the tree at the top of the parse stack are the Operand Operator
Operand accordingly.
// Checking if the type of the Operand and the Operand matches according to the
limitation of the language.
// If the types match, sets the LHS_Non_Terminal.type = the matching type.
// If don't match, reports a type mismatch error.
void semantic_type_check();

// Checks after reduction by the production rule: F -> id (R23)
// Need to validate that the identifier exists, and if so need to set F.type =
id.type
// If the identifier exists, also sets it to be the tree at the top of the parse
stack
// instead of F, to make the tree simpler.
void semantic_F_to_id();

// After reduction by the production rule: F -> literal (R24)
```

```

// Need to set F.type = literal.type
// Also sets the literal to be the tree at the top of the parse stack
// instead of F, to make the tree simpler.
void semantic_F_to_literal();

// After reduction by the production rule: F -> ( L_LOG_E ) (R25)
// Need to set F.type = L_LOG_E.type
// Also sets the expression to be the tree at the top of the parse stack
// instead of F, to make the tree simpler.
void semantic_F_to_L_LOG_E();

// After reduction by the production rule: F -> ! F (R26)
// And reduction by the production rule: F -> - F (R27)
// Need to set F.type = F.type
void semantic_F_to_unary_op_F();

// Converts a Data_Type to it's string representation
const char* semantic_data_type_to_str(Data_Type data_type);

```

semantic.c

```

#include <string.h>

#include "../global.h"

#include "semantic.h"
#include "../general/general.h"
#include "../scope_tree/scope_tree.h"
#include "../error_handler/error_handler.h"
#include "../token/token.h"

bool semantic_check_assign_compatibility(Data_Type id_type, Data_Type expr_type)
{
    // For now this will always return true, because I only have int and char in my
    // language.
    // If I'll add more types this will have more logic.
    return true;
}

Data_Type semantic_check_binary_op_compatibility(Data_Type left_op, Token* operator_,
Data_Type right_op)

```

```

{

    // For now this will always return Data_Type_Int, because I only have int and
    char in my language,
    // And all the operations between int and char should return int type.
    // If I'll add more types this will have more logic.
    return Data_Type_Int;
}

Data_Type semantic_check_unary_op_compatibility(Data_Type operand, Token* operator)
{
    // For now this will always return Data_Type_Int, because I only have int and
    char in my language,
    // And all the operations on int or char should return int type.
    // If I'll add more types this will have more logic.
    return Data_Type_Int;
}

void semantic_enter_block()
{
    // Adds a new child scope for the current scope
    scope_tree_add_scope();

    // Advance to that new sub scope
    scope_tree_goto_child();
}

void semantic_exit_block()
{
    // Just go the the parent scope in the scope tree
    scope_tree_goto_parent();
}

void semantic_decl()
{
    // After reduction by the production rule: DECL -> data_type id ;

    // The identifier is the 2nd child in the tree that is currently on top of the
    stack (Because it's after reduce).
    // The identifier is the value of the token of the 2nd child.
    char* identifier = compiler.parser->parse_stack->tree->children[1]->token-
>value;
}

```

```

// Fetch the identifier entry from current scope.
Symbol_Table_Entry* entry = symbol_table_fetch(compiler.scope_tree-
>current_scope->symbol_table, identifier);

// If the entry is NULL, insert identifier to the current scope.
if (entry == NULL)
{
    // I'm using strdup(identifier) because the symbol table and a token in the
tree should not point to the same memory location.
    // In order to prevent segfaults when destroying the compiler.
    identifier = strdup(compiler.parser->parse_stack->tree->children[1]->token-
>value);
    if (identifier == NULL) exit_memory_error(__FILE__, __LINE__);

    // Insert new symbol table entry with the known data in to the current
scope's symbol table.
    entry = symbol_table_entry_init(Entry_Type_Variable, identifier,
compiler.parser->parse_stack->tree->children[0]->symbol);
    symbol_table_insert(compiler.scope_tree->current_scope->symbol_table,
entry);
}

// If the entry is NOT NULL, that means this identifier is already declared, so
report semantic error.
else
{
    // compiler.line - 1 because after reduction the lexer's line was probably
already advanced.
    // So for better error reporting I subtracted 1.
    error_handler_report(compiler.line - 1, Error_Semantic, "" BOLD_WHITE "%s"
RESET "' already declared", identifier);
}

void semantic_assign()
{
    // After reduction by the production rule: ASSIGN -> set id = L_LOG_E ;

    // Get the L_LOG_E from the tree at the top of the stack
Parse_Tree_Node* L_LOG_E = compiler.parser->parse_stack->tree->children[3];

    // Get the identifier from the tree at the top of the stack
    char* identifier = compiler.parser->parse_stack->tree->children[1]->token-
>value;
}

```

```

// Try to fetch the identifier from the scope tree
Symbol_Table_Entry* entry = scope_tree_fetch(identifier);

// If the entry is NULL that means the identifier was not found in the scope
tree,
// which means the programmer tries to assign value to an undeclared variable.
if (entry == NULL)
    error_handler_report(compiler.line - 1, Error_Semantic, "" BOLD_WHITE "%s"
RESET "' undeclared", identifier);

// Check for matching types
else if (semantic_check_assign_compatibility(entry->data_type, L_LOG_E-
>data_type) == false)
    error_handler_report(compiler.line - 1, Error_Semantic, "Assignment of type "
" BOLD_WHITE "%s" RESET "' to identifier '" BOLD_WHITE "%s" RESET "' of type '" 
BOLD_WHITE "%s" RESET "'", semantic_data_type_to_str(L_LOG_E->data_type),
identifier, semantic_data_type_to_str(entry->data_type));
}

void semantic_set_type()
{
    // Make the child of the current tree that is on top of the stack
    // to be the tree at the top of the stack.
    // To make the tree simpler.
    compiler.parser->parse_stack->tree = compiler.parser->parse_stack->tree-
>children[0];
}

void semantic_type_check()
{
    // Get the left operand from the tree at the top of the stack
    Parse_Tree_Node* left_op = compiler.parser->parse_stack->tree->children[0];

    // Get the operator from the tree at the top of the stack
    Parse_Tree_Node* operator = compiler.parser->parse_stack->tree->children[1];

    // Get the right operand from the tree at the top of the stack
    Parse_Tree_Node* right_op = compiler.parser->parse_stack->tree->children[2];

    // If the types match with that operator
    Data_Type result_type = semantic_check_binary_op_compatibility(left_op-
>data_type, operator->token, right_op->data_type);
}

```

```

if (result_type != Data_Type_NULL)
    // Set the parent type to that type
    compiler.parser->parse_stack->tree->data_type = result_type;

else
    error_handler_report(compiler.line, Error_Semantic, "Invalid operands to
operator " BOLD_WHITE "%s" RESET ", have '' BOLD_WHITE "%s" RESET '' and ''
BOLD_WHITE "%s" RESET "", operator->token->value,
semantic_data_type_to_str(left_op->data_type), semantic_data_type_to_str(right_op-
>data_type));
}

void semantic_F_to_id()
{
    // Get the identifier from the tree at the top of the stack
    char* identifier = compiler.parser->parse_stack->tree->children[0]->token-
>value;
    // Try to fetch the identifier from the scope tree
    Symbol_Table_Entry* entry = scope_tree_fetch(identifier);

    // If Identifier exists
    if (entry != NULL)
    {
        // Make the identifier to be the tree at the top of the stack instead of F.
        // To make the tree simpler.
        compiler.parser->parse_stack->tree = compiler.parser->parse_stack->tree-
>children[0];

        // F.type = id.type
        compiler.parser->parse_stack->tree->data_type = entry->data_type;
    }

    else
        error_handler_report(compiler.line, Error_Semantic, """ BOLD_WHITE "%s"
RESET '' undeclared", identifier);
}

void semantic_F_to_literal()
{
    // Make the literal to be the tree at the top of the stack instead of F.
    // To make the tree simpler.
}

```

```

compiler.parser->parse_stack->tree = compiler.parser->parse_stack->tree-
>children[0];

// Right now I only have int an char literals in my programming language.
// My compiler interprets both of them as int literals.
// So for now I just set every literal to int literal.
// If I'll add more literals to my language then I'll add a translation function
from literal to data type.
compiler.parser->parse_stack->tree->data_type = Data_Type_Int;
}

void semantic_F_to_L_LOG_E()
{
    //      0   1   2
    // F -> ( L_LOG_E )

    // Make the expression to be the tree at the top of the stack instead of F.
    // To make the tree simpler.
    compiler.parser->parse_stack->tree = compiler.parser->parse_stack->tree-
>children[1];
}

void semantic_F_to_unary_op_F()
{
    // Get the operand from the tree at the top of the stack
    Parse_Tree_Node* operand = compiler.parser->parse_stack->tree->children[1];

    // Get the operator from the tree at the top of the stack
    Parse_Tree_Node* operator = compiler.parser->parse_stack->tree->children[0];

    compiler.parser->parse_stack->tree->data_type =
semantic_check_unary_op_compatibility(operand->data_type, operator->token);
}

const char* semantic_data_type_to_str(Data_Type data_type)
{
    switch (data_type)
    {
        case Data_Type_Int: return "int";
        case Data_Type_Char: return "char";

        default: return "data_type";
    }
}

```

```
    }  
}
```

token

token.h

```
#pragma once

/* ----- Structs ----- */

#define NUM_OF_TOKENS (Token_Eof - 1)

// Enum of all the token types
typedef enum Token_Type
{
    Token_Error,           // Error token, must be 0
    Token_Whitespace,      // Whitespace characters: ' ', '\n', '\t', '\r', '\v',
    '\f',
    Token_Comment,         // Comment. # This is a comment
    Token_Prog,            // `prog`
    Token_Identifier,      // Identifier
    Token_Colon,           // `:`
    Token_Smiley,          // `:)`
    Token_Done,             // `done`
    Token_Int,              // `int`
    Token_Char,             // `char`
    Token_Semi_Colon,       // `;`
    Token_Set,              // `set`
    Token_Assignment,       // `=`
    Token_If,                // `if`
    Token_Open_Paren,        // `(`
    Token_Close_Paren,       // `)`
    Token_Else,              // `else`
    Token_While,             // `while`
    Token_Or,                // `||`
    Token_And,               // `&&`
    Token_Equal,             // `==`
    Token_Not_Equal,         // `!=`
    Token_Bigger,             // `>`
    Token_Bigger_Equal,       // `>=`
    Token_Smaller,            // `<`
    Token_Smaller_Equal,       // `<=`
    Token_Plus,              // `+`
    Token_Minus,             // `-`
```

```

Token_Multiply,           // `*'
Token_Divide,            // `/'
Token_Modulu,            // `%'
Token_Number,             // Number literal
Token_Character,          // Character literal
Token_Not,                // `!'
Token_Eof,                 // End Of File token
} Token_Type;

// Token struct
typedef struct Token
{
    char* value;           // The value of the current token from the source code
    int value_len;          // The length of the value
    Token_Type token_type; // The type of the token. From the Token_Type enum
} Token;

/* ----- Functions ----- */

// Gets a value and a token type, and returns a Token with those inputs
Token* token_init(char* value, int value_len, Token_Type token_type);

// Frees a token
void token_destroy(Token* token);

// Converts the Token_Type to its matching token_type name in the enum of the types
// of a Token
const char* token_type_to_str(Token_Type token_type);

// Creates a string that describes the inputed token
char* token_to_str(Token* token);

```

token.c

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "token.h"
#include "../general/general.h"

```

```

Token* token_init(char* value, int value_len, Token_Type token_type)
{
    // Create a new token
    Token* token = (Token*) calloc(1, sizeof(Token));
    if (token == NULL)
    {
        free(value);
        exit_memory_error(__FILE__, __LINE__);
    }

    // Update token's properties
    token->value = value;
    token->value_len = value_len;
    token->token_type = token_type;

    return token;
}

void token_destroy(Token* token)
{
    // check for NULL pointer
    if (token != NULL)
    {
        // Free the memory allocated for the token's value
        free(token->value);

        // Free the token
        free(token);
    }
}

const char* token_type_to_str(Token_Type token_type)
{
    switch (token_type)
    {
        case Token_Whitespace: return "Whitespace";
        case Token_Identifier: return "Identifier";
        case Token_Number: return "Number";
        case Token_Character: return "Character";
        case Token_Int: return "Int";
        case Token_Char: return "Char";
        case Token_Prog: return "Prog";
    }
}

```

```

        case Token_If: return "If";
        case Token_Else: return "Else";
        case Token_While: return "While";
        case Token_Set: return "Set";
        case Token_Done: return "Done";
        case Token_Assignment: return "Assignment";
        case Token_Equal: return "Equal";
        case Token_Not: return "Not";
        case Token_Not_Equal: return "Not_Equal";
        case Token_Bigger: return "Bigger";
        case Token_Bigger_Equal: return "Bigger_Equal";
        case Token_Smaller: return "Smaller";
        case Token_Smaller_Equal: return "Smaller_Equal";
        case Token_Or: return "Or";
        case Token_And: return "And";
        case Token_Plus: return "Plus";
        case Token_Minus: return "Minus";
        case Token_Multiply: return "Multiply";
        case Token_Divide: return "Divide";
        case Token_Modulu: return "Modulu";
        case Token_Open_Paren: return "Open_Paren";
        case Token_Close_Paren: return "Close_Paren";
        case Token_Colon: return "Colon";
        case Token_Smiley: return "Smiley";
        case Token_Semi_Colon: return "Semi_Colon";
        case Token_Eof: return "EOF";

    default: return "Don't know that token type... ;|";
}
}

char* token_to_str(Token* token)
{
    const char* type_str = token_type_to_str(token->token_type);

    char* str = (char*) calloc(strlen(type_str) + 16, sizeof(char));
    if (str == NULL)
    {
        token_destroy(token);
        exit_memory_error(__FILE__, __LINE__);
    }
}

```

```
if (token->token_type == Token_Eof)
    sprintf(str, BOLD_WHITE "%s" RESET, type_str);
else
    sprintf(str, "" BOLD_WHITE "%s" RESET "", token->value);

return str;
}
```

נספחים

תרשימים ודיagramות

GitHub

להלן קישור לפרויקט ב – GitHub המכיל את כל הקבצים, תМОונת, נספחים אשר השתמשתי בהם במהלך יצירת ספר הפרויקט. שם ניתן למצוא ולהוריד את כל הדיאגרמות והתרשימים בהם השתמשתי בספר הפרויקט באיכות גבוהה.

קישור

<https://github.com/ido-hi/do-compiler-project.git>

Whimsical

חלק מהסתוטוטים יצרתי באתר Whimsical. להלן קישור לתיקייה שלי באתר Whimsical בה יש את כל התרשימים והדיאגרמות שיצרתי עבור ספר הפרויקט. כך ניתן לראות את התרשימים בצורה הטובה ביותר.

קישור

<https://whimsical.com/compiler-E3qhLg38Wn42PsB9kMY11k>

תרשימים האלגוריתם הראשי

להלן קישור לתרשימים המלא של האלגוריתם הראשי, כפי שציינתי שיצורף בפרק **האלגוריתם הראשי**. את התרשימים יצרתי .GitHub באתר draw.io וניתן למצוא את הקובץ של התרשימים (Super-Algorithm.drawio) גם בפרויקט ב –

קישור

https://viewer.diagrams.net/?tags=%7B%7D&highlight=0000ff&edit=_blank&layers=1&nav=1&title=Super-Algorithm.drawio#Uhttps%3A%2F%2Fraw.githubusercontent.com%2Fido-hi%2Fdo-compiler-project%2Fmaster%2Fsuper-algorithm%2FSuper-Algorithm.drawio