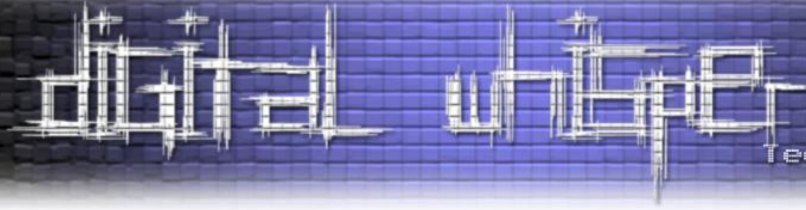


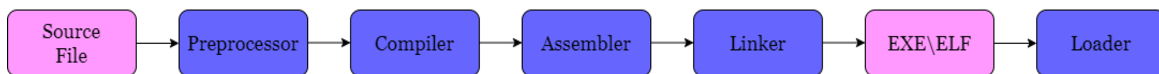
מקובץ טקסט לשפת מכונה

מאת אלון נסים

הקדמה

קובץ קוד בכל שפת תכנות שהיא הוא לא יותר קובץ טקסט. אז מה נותן לקובץ טקסט שלנו את היכולת לבצע חישובים, לממש אלגוריתמיקה מורכבת? זהו תהליך מורכב שבמרכזו נמצא הקומפיילר! למחשב יש יכולת להבין רק 1 ו-0 (שפת מכונה). כשאנחנו רוצים להריץ קוד כלשהו, למחשב אין יכולת לקרוא אנגלית, ולכן אנחנו צריכים לבצע תרגום מהשפה שלנו לשפת מכונה.

אז איך תרגום זה מתבצע? תהליך זה הוא ייחודי לשפת התכנות, במאמר זה נתעסק בתהליך עבור שפת C. נתבונן באיור הבא:



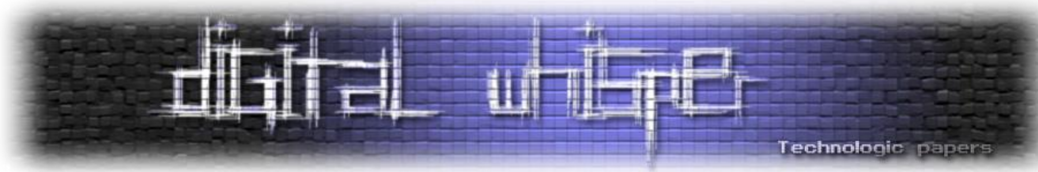
באיור מתואר התהליך שעובר קובץ C להפוך לתוכנית רצה. נפרט על כל שלב.

Preprocessor

לפני שאנו מתחילים את החישובים, אנו קודם צריכים שיהיה לנו את כל המידע הנחוץ. נתבונן בקטע קוד הבא:

```
//File name: proc.c
#include <stdio.h>
int main(void){
        printf("Digital whisper");
      return 0;
}
```

כמתכנתים זה נראה לנו טריוויאלי שניתן להשתמש בפונקציה "printf", אך בפועל איפה היא מוגדרת? הפונקציה מוגדרת בקובץ "stdio.h". איך הקומפיילר יודע להגיד האם הקריאה לפונקציה תקינה אם אין לו את חתימת הפונקציה?



זהו תפקידו של ה-Preprocessor, הוא דואג שלתהליך הקומפילציה יהיה את כל המידע הנדרש. שיטת המימוש של ה-Preprocessor פשוטה מאוד: הוא עובר על הקוד ועושה העתק הדבק לכל המידע שאין לו. כך יראה הקובץ שלנו לאחר שלב ה-Preprocessing:

```
// File name: proc.i
printf (const char *format, ...);
// copy of the stdio.h file
int main(void){
        printf("Digital whisper");
      return 0;
}
```

Compiler

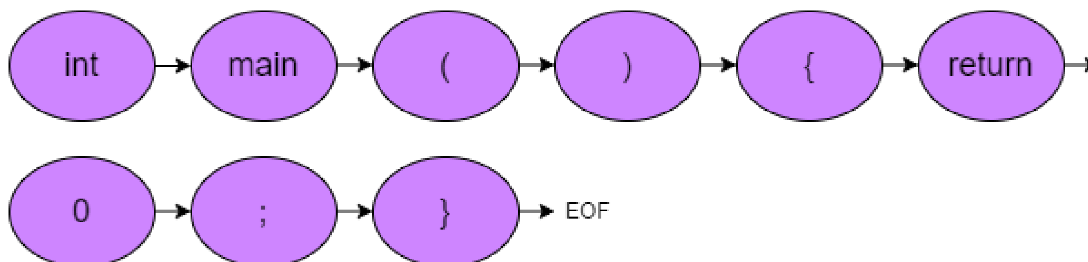
תפקידו המרכזי של הקומפיילר של C הוא לתרגם מ-C לאסמבלי. תהליך הקומפילציה מתבצע ב-3 חלקים:

lexical analysis: בתהליך זה אנו מפרקים את הקובץ ומייצגים אותו כרשימה של tokens. כל token מדמה את הרכיב הקטן ביותר לייצוג ה-parser. אם התוכנית היא כמו פסקה אז token היא מילה בפסקה. tokens רבים הם מילים מופרדים ברווח, ותווים המייצגים סיום שורה, סוגרים וכו'

נתבונן בקטע קוד הבא, נפרק אותו לרשימה של tokens:

```
int main() {
    return 0;
}
```

רשימת ה-tokens עבור קוד זה נראה כך:

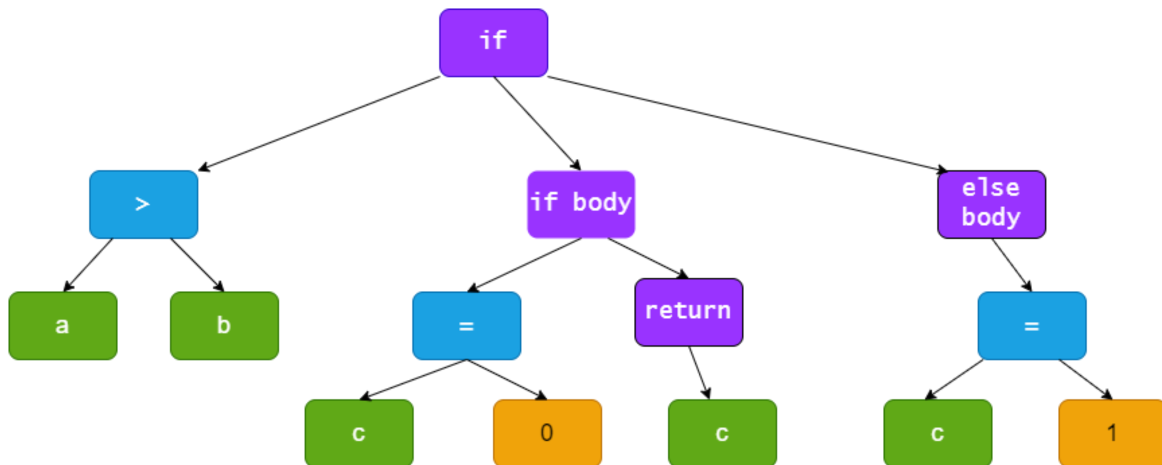


parser: כאת ניצור מרשימת ה-tokens שלנו abstract syntax tree (AST). ה-AST הוא שיטה לייצוג מבנה התוכנית (intermediate representation). עץ זה מדמה את ההיררכיה של התוכנית, ראש העץ ידמה את התוכנית כולה וכל רכיב אחר ייצג יהיה ילדים אשר ייצגו את שימוש הרכיב. באותה הדרך ש- "int a" מייצג את השפה שלנו בטקסט, ה-AST מייצג את השפה בזכרון.

נתבונן בקטע קוד הבא:

```
if (a > b) {
    c = 0;
    return c;
} else {
    c = 1;
}
```

ניצור לו AST מתאים:



לבנות עץ זה היא לא משימה כזו פשוטה: בשביל לפרסר את הקוד אנחנו קודם צריכים להגדיר מה נחשב חוקי, וצריך להגדיר syntax ברור שה-parser יוכל לבנות את העץ לפיו. נעבור על שלבי פירסור באמצעות דוגמא- נפרסר מחרוזת המייצגת ביטוי מתמטי פשוט.

נתעסק בביטויים רק עם אופרטור '+', ונייצג בעזרת AST ביטוי כמו: "2+4+8". נגדיר את חוקי הביטוי כך שלדוגמא הביטוי "2+" אינו תקין. לכן נצטרך להחזיר syntax error. נגדיר grammar ב-Form Backus-Naur:

```
<expression> ::= <int> |
                 <expression> '+' <expression>
```

ישנם כל מני סוגים של parsers. נבנה את ה-AST בעזרת recursive descent parser: הגדרת השפה היא רקורסיבית ולכן גם בניית העץ יתבצע באופן רקורסיבי.

האלגוריתם: נקרא את האיבר הראשון ברשימה של ה-tokens, אם הוא מספר נייצר אובייקט מסוג מספר, נקרא את ה-token הבא ונצפה לאופרטור. אם אינו אופרטור נסיים את התוכנית עם syntax error. אם הוא כן אופרטור, ניצור אובייקט מסוג אופרטור ונקרא את ה-token הבא ונצפה למספר וכך ברקורסיה עד לסוף המחרוזת.

כך יראה פסאודו קוד למימוש recursive descent parser:

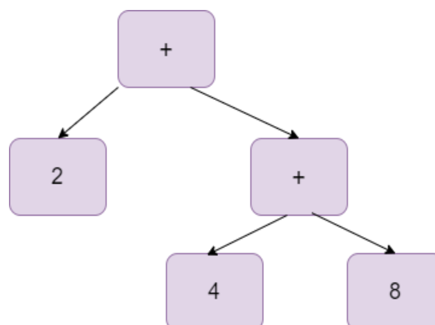
```
// This function parses an <expression>
function parseExpression():
    if currentToken is integer:
        return currentToken // Return the integer token
    else:
        leftExpression = parseExpression()
        match('+')
        rightExpression = parseExpression()
        // Return a list representing the addition
        return new expression('+', leftExpression, rightExpression)

// This function parses an integer
function parseInt():
```

```
    if currentToken is an integer:
        value = currentToken // Return the integer token
        // Consume the token and advance to the next one
        currentToken = getNextToken()
        return new number(value)

// This function matches the current token with an expected token
function match(expectedToken):
    if currentToken is equal to expectedToken:
        // Consume the token and advance to the next one
        currentToken = getNextToken()
    else:
        // Syntax error: Mismatched token
        reportError("Expected " + expectedToken + ", but found " +
currentToken)
```

עבור הביטוי "2+4+8" ה-AST יראה כך:



Assembly generator: יצרנו את ה-AST, הגיע הזמן לאסמבלי! (חלק זה דורש הבנה בסיסית של 32-bit x86 Assembly. למי שלא בקיא, ממליץ לקרוא כאן: A Tiny Guide to Programming in 32-bit x86 Assembly Language). בשביל להתקרב לשפת המכונה נתרגם את העץ לשפת אסמבלי. "נלך" על ה-AST ועבור כל פקודה נייצר קוד אסמבלי מתאים: נתבונן בקוד הבא שנית:

```
// File name: return0.c
int main() {
    return 0;
}
```

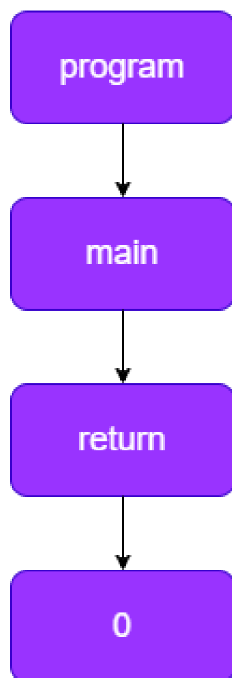
נקמפל עם GCC, ונראה את האסמבלי שהוא מייצר:

```
$ gcc -S -O3 -fno-asynchronous-unwind-tables -masm=intel return0.c
```

נקבל קובץ return0.s, ולאחר סידור קטן הוא יראה כך:

```
;File name: proc.s
.globl main
main:
    xorl eax,eax
    ret
```

בשביל מעבר זה, ראשית ניצור AST מתאים:



נעבור על העץ.

תוכנית זאת מאוד פשוטה ולכן program לא יצור קוד. עבור main נכתוב לקובץ אסמבלי:

```
.globl main
main:
```

פקודה זו מגדירה פונקציה. עבור return נכתוב לקובץ אסמבלי:

```
xor eax,eax
ret
```

פקודה זו מאפסת את הרגיסטר eax, ולאחר מכן באמצעות ret מחזירה ערך 0. בכך יצרנו קובץ אסמבלי מתאים בעזרת ה-AST.

Compiler Optimization: יצרנו קובץ אסמבלי תקין, אבל אולי ניתן לשנות את הקוד ככה שיהיה יעיל יותר?

ישנן המון סוגים של compiler optimizations. נרחיב על peephole optimization ו-loop unrolling peephole optimizations: אופטימיזציה זו מנסה לגנרט קוד אחר יעיל יותר אשר עושה את אותה הפעילות בדיוק.

נתבונן בקוד הבא:

```
a = b + c;
d = a + e;
```

ה-code generator יגנרט את האסמבלי הבא:

```
mov ebx, b
add ebx, c
mov a, ebx
mov ebx, a
mov ebx, e
mov d, ebx
```

אבל, ניתן גם לגנרט את האסמבלי הבא:

```
mov ebx, b
add ebx, c
mov a, ebx
mov ebx, e
mov d, ebx
```

כלומר, הייתה לנו שורה מיותרת.

loop unrolling:

לולאות הם מרכז הכובד של התוכניות שלנו, לכן לאפטם אותם עשוי לתת שיפור משמעותי. שיטה זו מנסה להקטין את כמות הפקודות שהלולאה תבצע. נתבונן בקוד הבא:

```
#include<stdio.h>
int main(void)
{
    for (int i=0; i<5; ++i)
        printf("Digital whisper\n");
    return 0;
}
```

נוכל להקטין את כמות הפקודות בצורה כזו:

```
#include<stdio.h>
int main(void)
{
    printf("Digital whisper\n");
    printf("Digital whisper\n");
    printf("Digital whisper\n");
    printf("Digital whisper\n");
    printf("Digital whisper\n");
    return 0;
}
```

בתוכנית זו חסכנו אתחול משתנה, 5 בדיקות גבול עבור המשנה(i<5) ו-5 עדכונים של המשתנה (++i).

Assembler

כמעט סיימנו, יצרנו קובץ אסמבלי. עכשיו נרצה לתרגם לשפת מכונה.Assembler הוא הקומפיילר של שפת אסמבלי. כיוון אסמבלי הרבה יותר קרוב לשפת המכונה מ-C, שיטת התרגום הרבה יותר פשוטה. לכל פקודת אסמבלי מספר המתאים לה, מספר זה נקרא opcode (רשימה מלאה: opcode table). נתרגם את הפקודה הבאה:

```
mov eax, 2
opcode(mov) = 0xb8, Destination(eax) = 0x0, Source = 0x2.
```

והפקודה תתרגם כך:

```
//File name: move.o
b8 02 00 00 00
```

נשים לב כי שיטת התרגום עבור פקודות שונות היא אחרת, לדוגמא עבור פקודות JMP, נצטרך לתרגם גם את כתובת הקפיצה.

Linker

פעמים רבות נרצה לבנות תוכנית המורכבת ממספר תכניות שונות. כרגע תרגמנו כל תוכנה כזו בנפרד לשפת מכונה, אבל צריך משהו שיחבר תוכנות אלו ביחד לקובץ ריצה יחיד. זהו תפקידו של ה-Linker.

נתבונן שנית בקוד הבא:

```
// File name: proc.i
printf (const char *format, ...);
// copy of the stdio.h file
int main(void){
    printf("Digital whisper");
      return 0;
}
```

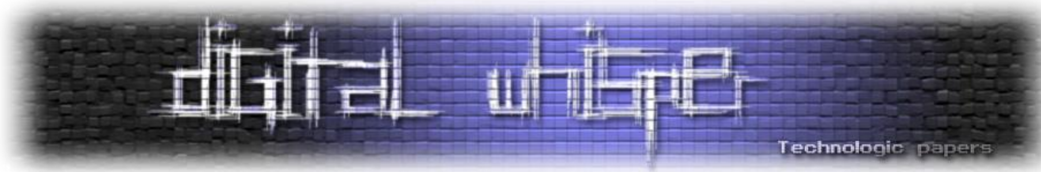
ה-Preprocessor העתיק את חתימת הפונקציה של printf בשביל שהקומפלייר. אבל כרגע לתוכנית יש רק את חתימת הפונקציה ולא את המימוש שלה, אשר הכרחי לביצוע פעולת ההדפסה. לכן "נקשר" את שני הקבצים כך התכניות תצליח לרוץ.

ישנם שני דרכים לקישור זה:

Static linking:

בשיטה זו הקישור מתבצע לפני ריצת התוכנית. ה-linker לוקח את כל הקבצים הדרושים (קבצי .o) ומעתיק אותם לקובץ ריצה יחיד.

- **יתרונות**: ריצת התוכנית מהירה כיוון שאין צורך לגשת לשום מקום אחר בזיכרון בזמן ריצת התוכנית.
- **חסרונות**: קובץ ההרצה הינו גדול ובנוסף מתבצעת פה שכפול קוד עבור ספריות שנקראות מספר פעמים.



:Dynamic linking

בשיטה זו הקישור מתבצע בזמן ריצת התוכנית. ה-linker יוצר קובץ ספריה משותף, ובכל פעם שנדרש הוא ניגש לספריה זו ומריץ את הקוד הנדרש.

- **יתרונות**: גודל קובץ ההרצה קטן, יש גמישות בכך שניתן לשנות פעולות בספריות בלי לקמפל את כל התכניות.
- **חסרונות**: ריצת התוכנית איטית יותר שכן צריך לגשת לספריות בזמן ריצה.

Loader

מזל טוב! יצרנו קובץ ריצה שהמחשב מבין. אבל מבין זה לא הסוף! צריך לקשר קובץ זה עם רכיבי זיכרון הנדרשים ולאתחל לתוכנית שלנו מרחב זיכרון שתוכל לעבוד בו. זהו תפקידו של ה-Loader. הוא מעתיק את קובץ ההרצה למקום בזיכרון שיוכל לרוץ בו, ובנוסף הוא מגדיר סימבולים עבור פונקציות ומשתנים שיוודא שהוא ניגש למקום הנכון בזיכרון.

מימוש קומפיילר של C מ-0

הבנו איך קומפיילר עובד, המבחן האמיתי יהיה לממש קומפיילר משלנו!

הערה: כל קבצי הקוד נמצאים בגיטהאב, בנתיב הבא: https://github.com/alloknight/C-compiler

כעת נממש קומפיילר לסט פקודות קטן של שפת C. התוכנית תהיה מורכבת מפונקציית main שמחזירה ביטוי מתמטי. דוגמה לתוכנית:

```
int main(){
    return 2+2*3;
}
```

בקומפיילר שאציג נשתמש בשלוש טכנולוגיות. את ה- lexer נממש ב- FLEX. את ה-parser נממש ב- GNU Bison. את ה- assembly generator נממש ב-C++. אם אתם לא מכירים את טכנולוגיות אלו, אל דאגה.

Lexical Analysis with FLEX

זהו השלב הפשוט ביותר בקומפיילר- בהינתן grammer מוגדרת, אנו צריכים לפרק את קובץ הקוד לרשימה של tokens. כפי שהוזכר קודם, grammar הוא ה-Syntax של שפת התכנות שלנו, עבור הקומפיילר שלנו, ה- Syntax הוא מאוד פשוט, אופרטורים מתמטים (+,-,*,/), פונקציית main, מספרים שלמים חיוביים קבועים ומילות מפתח כמו int ו-return.

קובץ ה-FLEX מאוד פשוט ויראה כך:

```
// File name: lex.l
%{
#include "node.h"
#include "parse.tab.h"
#define SAVE_TOKEN yylval.string = new std::string(yytext, yyleng)
#define TOKEN(type) (yylval.token = type)
extern void yyerror(const char* s);
%}

%%

[ \t\n]              ; // Ignore whitespace
[0-9]+                 { SAVE_TOKEN; return INTEGER; }
"return"               { SAVE_TOKEN; return RETURN_KEYWORD;}
"int"                  { SAVE_TOKEN; return INT_KEYWORD;}
[a-zA-Z]+              { SAVE_TOKEN; return IDENTIFIER; }
"("                    { return TOKEN(LEFT_BRACE);}
")"                    { return TOKEN(RIGHT_BRACE);}
"{"                    { return TOKEN(LEFT_PAREN);}
"}"                    { return TOKEN(RIGHT_PAREN);}
"+"                    { return TOKEN(PLUS);}
"-"                    { return TOKEN(MINUS);}
"*"                    { return TOKEN(MULT);}
"/"                    { return TOKEN(DIV);}
.                  { yyerror("Invalid character"); }

%%

int yywrap() {
    return 1;
}
```

תוכנית ב-FLEX, מחולקת לשלושה חלקים. החלק הראשון זה הגדרות C. אנו מגדירים מאקרו SAVE_TOKEN לשמור את הערך של ה-token הנוכחי, נבין את המשמעות של YYLVAL כאשר נתעסק בפירסור וב-Bison.

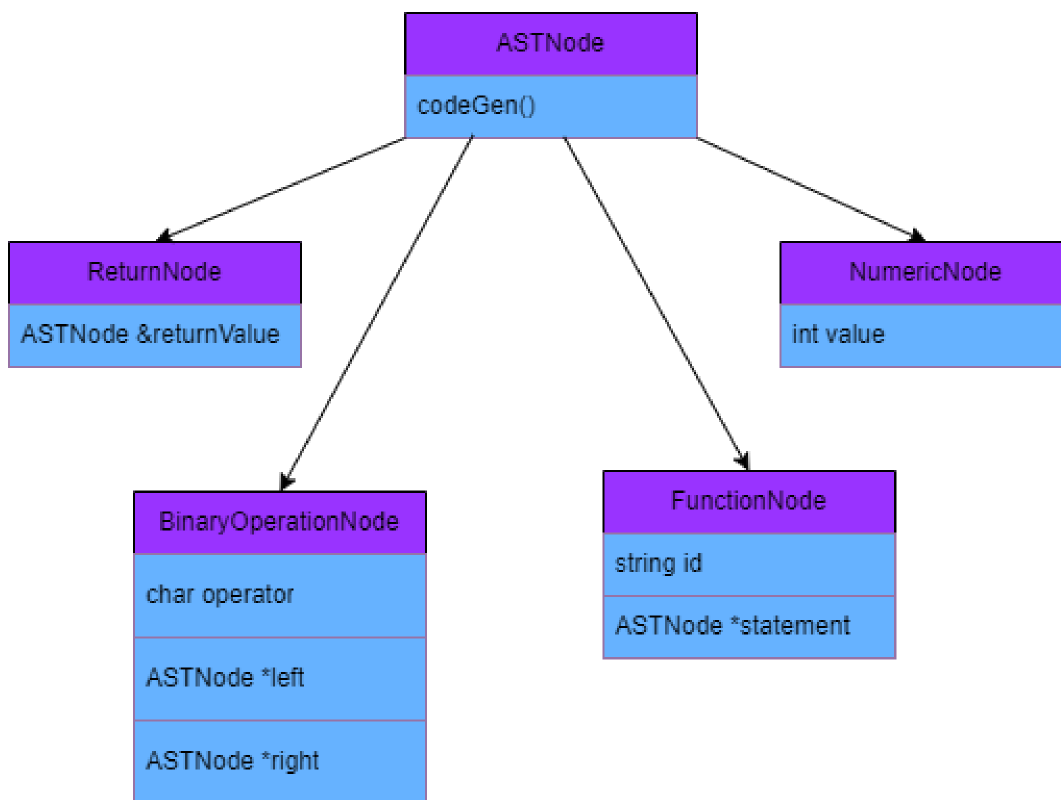
החלק השני זה הגדרת Tokens. ה-token הראשון אומר לדלג על כל הרווחים בתוכנית שכן אין להם חשיבות. נראה ששארית ה-tokens מוגדרים באמצעות קובעים אשר לא מוגדרים, קובעים אלה נמצאים בקובץ parse.tab.h שגם הוא מוגדר ב-Bison.

החלק השלישי היא הגדרת פונקציות. yywrap נקרא כאשר סיימנו לעבור על כל הקובץ וכך יודעים מתי לסיים.

לבנות Grammer מדויק לשפה זה לא משהו פשוט, כרגע השפה שלנו מאוד מצומצמת ולכן יהיה קל יותר. לפני המימוש נצטרך קודם להסכים על Design ברור: התוצאה הסופית של הפרסר יהיה AST. נראה תכף ש-Bison יודע לבנות עץ זה בצורה מאוד אלגנטית.

עיקר העבודה תהיה להגדיר את ה-grammer ולעבוד יחד עם FLEX. לפני היפרסור עצמו, ראשית אנחנו צריכים להגדיר מבנה נתונים המייצג את ה-AST.

נייצג מבנה נתונים בעזרת מחלקות ב-C++ בצורה הבאה:



קוד עבור מבנה נתונים זה יראה כך:

```
// File name: node.h
#ifndef NODE_H_
#define NODE_H_
#include <iostream>
#include <string>
#include <fstream>

class ASTNode {
public:
    virtual ~ASTNode() = default;
    virtual void codeGen(std::ofstream& output) const = 0;
};

class NumericNode : public ASTNode {
```

```
public:
    NumericNode(std::string value);
    void codeGen(std::ofstream& output) const override;
    int value() const;
private:
    int _value;
};

class BinaryOperatorNode : public ASTNode {
public:
    BinaryOperatorNode(char op, ASTNode &left, ASTNode &right);
    void codeGen(std::ofstream& output) const override;
private:
    char _op;
    ASTNode& _left;
    ASTNode& _right;
};

class ReturnNode : public ASTNode{
public:
   ReturnNode(ASTNode &returnValue);
   void codeGen(std::ofstream& output) const override;
private:
   ASTNode& _returnValue;
};

class FunctionNode : public ASTNode{
public:
    FunctionNode(std::string id,ASTNode &statement);
    void codeGen(std::ofstream& output) const override;
    const ASTNode& statement() const;
private:
    std::string _id;
    ASTNode& _statement;
};

#endif
```

ניתן כרגע להתעלם מהפונקציה codeGen שכן תהיה רלוונטית מאוחר יותר.

בחזרה ל-**Bison**:

Bison בונה את ה-AST באמצעות Recursive Descent Parsing שהוזכר בתחילת המאמר. נגדיר מה ה-Syntax של השפה ומה אנחנו רוצים לעשות עם הפעולה:

כך זה יראה עבור ביטוי המייצג ביטוי חיבור:

```
expression:
expression PLUS expression {$$ = new BinaryOperatorNode('+', *$1, *$3); }
    ;
```

ראשית אנחנו מגדירים את שם האובייקט, כאן הוא ביטוי (expression), שנית אנחנו מגדירים את ה-Syntax של האובייקט (expression PLUS expression).

בסוף נגדיר מה אנחנו רוצים לעשות עם אובייקט זה, במקרה זה ניצור רכיב ל-AST. $$ מייצג את הרכיב הנוכחי ב-AST. $k מייצג את האיבר ה-k בשורה של ה-Syntax. כך יראה קוד עבור ה-parser:

```
// File name: parse.y
%{
#include "node.h"
ASTNode *root = nullptr;

extern int yylex();
void yyerror(const char*);

%}

%union {
    int token;
    std::string *string;
    ASTNode *node;
}

%token <string> IDENTIFIER INTEGER RETURN_KEYWORD INT_KEYWORD

%token <token> LEFT_PAREN RIGHT_PAREN LEFT_BRACE RIGHT_BRACE
%token <token> PLUS MINUS MULT DIV

%type <node> expression program statement function factor term

%left PLUS MINUS
%left MUL DIV

%start program

%%

program: function {root = $1;}
        ;
function: INT_KEYWORD IDENTIFIER LEFT_BRACE RIGHT_BRACE LEFT_PAREN statement
RIGHT_PAREN {$$ = new FunctionNode(*$2,*$6);} //$2 = identifier, $6 = statement.
         ;
expression: factor
      | expression PLUS expression {$$ = new BinaryOperatorNode('+', *$1, *$3); }
      | expression MINUS expression {$$ = new BinaryOperatorNode('-', *$1, *$3); }
      | term
    ;
```

```
term: factor MULT factor {$$ = new BinaryOperatorNode('*', *$1, *$3); }
    | factor DIV factor {$$ = new BinaryOperatorNode('/', *$1, *$3); }
    ;
factor: INTEGER {$$ = new NumericNode(*$1); }
    ;
statement: RETURN_KEYWORD expression {$$ = new ReturnNode(*$2);}
    ;
%%

void yyerror(const char *s) {
    std::cerr << "Error: " << s << std::endl;
}
```

נדגיש כמה דברים:

- **union** - זה YYLVAL, אשר ככה FLEX שומר את הערך של ה-Tokens וכך מתקשר עם Bison.
- **%token** - ככה אנחנו מגדירים Tokens טריוויאלים (מילות מפתח, אופרטורים וכו'), בנוסף ככה אנחנו מגדירים את ה-Tokens עבור FLEX, כאשר אנחנו מקמפלים את Bison, הוא מוציא קובץ parse.tab.h אשר ראינו ב-lex.l, קובץ זה מכיל את הגדרות אלו.
- **%type** - ככה אנחנו מגדירים Tokens שאינם טריוויאלים (ביטויים מתמטיים, פונקציות וכו').
- **%start** - אומר לתוכנית איזה אובייקט ליצור ראשון, שכן פה אנחנו רוצים להגדיר תוכנית ולכן ראש ה-AST יהיה מסוג תוכנית כפי שהוצג באיור בתחילת המאמר.
- **factor, term** - ככה אנחנו ממשים את סדר פעולות חשבון, נרצה ליצור אובייקטים של כפל וחילוק לפני חיבור וחיסור.

Assembly generator

עבור כל רכיב בעץ נייצר אסמבלי מתאים, NumericNode: נרצה לשמור את הערך זיכרון, נשתמש ברגיסטר eax. לכן האסמבלי יראה כך:

```
mov eax, value
```

ReturnNode: נרצה להחזיר את הערך של eax בנוסף לקפוץ לכתובת הקודמת. לכן האסמבלי יראה כך:

```
ret
```

FuncitonNode: כרגע יש לנו רק פונקציה אחת בתוכנית ולכן האסמבלי יראה כך:

```
.globl main
main:
```

BinaryOperatorNode: כאן נצטרך להתאים את עצמנו לפי כל אופרטור.

- חיבור נשתמש ב-ADD
- חיסור נשתמש ב-SUB
- כפל נשתמש ב-IMUL
- חילוק נשתמש ב-IDIV

בנוסף אנחנו נדחוף ונוציא מהמחסנית שכן נצטרך שני רגיסטרים. הקוד עבור כל רכיב יראה כך:

```
// name: node.cpp
#include "node.h"

// NumericNode implementation
NumericNode::NumericNode(std::string value) : _value(std::stoi(value)) {}

void NumericNode::codeGen(std::ofstream& output) const {
    if(output.is_open()){
        output << "mov\teax," <<_value << std::endl;
    }
}

int NumericNode::value() const{
    return _value;
}
// BinaryOperatorNode implementation
BinaryOperatorNode::BinaryOperatorNode(char op, ASTNode& left, ASTNode& right)
    : _op(op), _left(left), _right(right) {}

void BinaryOperatorNode::codeGen(std::ofstream& output) const {
    // First, evaluate the left operand and push the result onto the stack
    _left.codeGen(output);
    output << "\tpush eax" << std::endl;

    // Then, evaluate the right operand
    _right.codeGen(output);

    // Now, pop the left operand back into another register (e.g., EBX)
    output << "\tpop ebx" << std::endl;

    // Perform the operation based on the operator
    switch (_op) {
        case '+':
            output << "\tadd eax, ebx" << std::endl;
            break;
        case '-':
            output << "\tsub ebx, eax" << std::endl;
            output << "\tmov eax, ebx" << std::endl;
            break;
        case '*':
            output << "\timul eax, ebx" << std::endl;
            break;
        case '/':
            output << "\txchg eax, ebx" << std::endl; // Swap EAX and EBX
            output << "\tcdq" << std::endl; // Sign-extend EAX into EDX
            output << "\tidiv ebx" << std::endl;
            break;
        default:
            // Handle error: unknown operator
            break;
    }
}

// ReturnNode implementation
ReturnNode::ReturnNode(ASTNode& returnValue) : _returnValue(returnValue) {}

void ReturnNode::codeGen(std::ofstream& output) const {
    NumericNode* numericNode = dynamic_cast<NumericNode*>(&_returnValue);

   _returnValue.codeGen(output);
    if(output.is_open()){
        output << "\tret" << std::endl;
```

```
    }

}

// FunctionNode implementation
FunctionNode::FunctionNode(std::string id,ASTNode& statement) :
_id(id),_statement(statement) {}

void FunctionNode::codeGen(std::ofstream& output) const {
    std::string code = ".globl "+ _id+ "\n"+ _id+ ":";
    if(output.is_open()){
        output << code << std::endl;
    }
}
const ASTNode& FunctionNode::statement() const{
    return _statement;
}
```

כעת כל מה שנשאר זה לעבור על ה-AST זה יהיה מאוד פשוט שכן ה-AST מאוד בסיסי. קוד זה יראה כך:

```
#include "node.h"
extern ASTNode *root;
extern int yyparse();

int main(int argc, char* argv[]) {
    yyparse();
    FunctionNode* functionRoot = dynamic_cast<FunctionNode*>(root);// root in type
functionNode

    if (root != nullptr)  {
        std::ofstream output("code.s");
        if (output.is_open()) {
            functionRoot->codeGen(output);
            functionRoot->statement().codeGen(output);
            output.close();
        }
        else{
            std::cerr << "cant open file" << std::endl;
        }
    }
    return 0;
}
```

וכך יש לנו קומפיילר! כל קבצי הקוד זמינים להורדה בקישור הבא:

https://github.com/alloknight/C-compiler

לקמפל:

```
$ bison -d parse.y // -d flag that generates parse.tab.h
$ flex lex.l
$ g++ -o compile parse.tab.c lex.yy.c node.cpp main.cpp
```

להריץ:

```
$ ./compile < proc.c
```

וכך יוצר קובץ אסמבלי code.s.

לקמפל אסמבלי:

```
$ gcc -m32 -o program code.s
```

להריץ:

```
./program
```

הערה - בשביל להשתמש ב-intel syntax נצטרך להוסיף לפני השורה הראשונה של קובץ אסמבלי את הדגל הבא:

```
.intel_syntax noprefix
```

בשביל לראות את ערך החזרה:

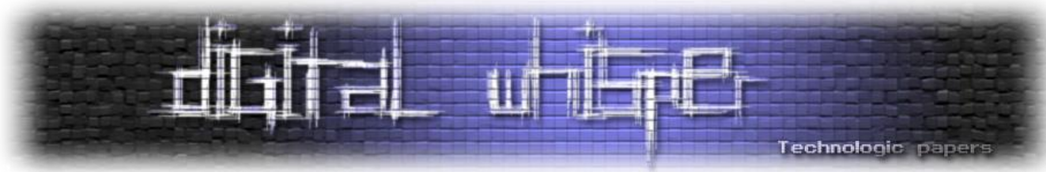
```
$ echo $?
```

סיכום

למדנו על תהליך הקומפילציה ואפילו מימשנו בעצמנו קומפיילר, אבל זו רק ההתחלה! מכאן אפשר להוסיף עוד פונקציות, לממש אופטימזציות למיניהן ואפילו ליצור שפת תכנות חדשה לגמרי.

איך הקומפיילר ממש פונקציות? איך הוא ממש משתנים? איך הוא מקצה זיכרון דינמי? במאמר זה בחרתי לממש קומפיילר מאוד בסיסי. אם אהבתם את הפרויקט הזה, אני ממליץ לכם להמשיך אותו ולענות על השאלות אלו. שכן לעקוב אחרי הסבר זה מגניב, אבל הדרך הכי מעניינת לדעתי היא בעזרת מחקר מעמיק והמון טעויות בדרך.

הבנה וכתיבה של הקומפיילר לימדה אותי המון על איך הדברים שאנחנו פוגשים כל יום כמתכנתים שנראים מובן מאליו עובדים מאחורי הקלעים וחידדה את החושים שלי גם בצד הפיתוחי וגם וצד המחקרי.



על המחבר

שמי אלון נסים, סטודנט למדעי המחשב בסמסטר האחרון לתואר באוניברסיטה הפתוחה. מאוד מתחבר לעולם המחקר וה-low level.

לכל שאלה מוזמנים ליצור קשר במייל - allon.nissim@gmail.com.

ביבליוגרפיה

An Incremental Approach to Compiler Construction:

http://scheme2006.cs.uchicago.edu/11-ghuloum.pdf

Introduction of Compiler Design:

https://www.geeksforgeeks.org/introduction-of-compiler-design

GNU bison:

https://www.gnu.org/software/bison

FLEX:

http://alumni.cs.ucr.edu/~lgao/teaching/flex.html

תמונת GNU:

https://upload.wikimedia.org/wikipedia/commons/8/83/The_GNU_logo.png