על קוד מקור, בינארי ומה שביניהם

מאת שליו שגן

הקדמה

אי פעם תהיתם כיצד המחשב מבין את הקוד שאנו כותבים? או איך הוא יודע שהקוד שכתבנו מקיים את כללי התחביר (syntax) של השפה? איך הוא יודע כשיש unreachable code? אני תהיתי, ולכן התחלתי לחקור, והיום אשתף אתכם בידע שלי ואענה על השאלות האלה. התוכנה האחראית על כך היא המהדר (קומפיילר).

המהדר, יודע את תחביר השפה שלו ואחראי על בדיקות התקינות ועל הסמנטיקה. במאמר אציג כיצד המהדר עובד בצורה המוחשית והברורה ביותר, על ידי כתיבת אחד כזה, בשילוב עם LLVM כדי לייצר את קוד האסמבלי. אז ללא יותר מידי הקדמות, בואו נתחיל.

אני ממליץ לקרוא את המאמר עם ידע ב־C++ משום שנכתוב איתה את המהדר.

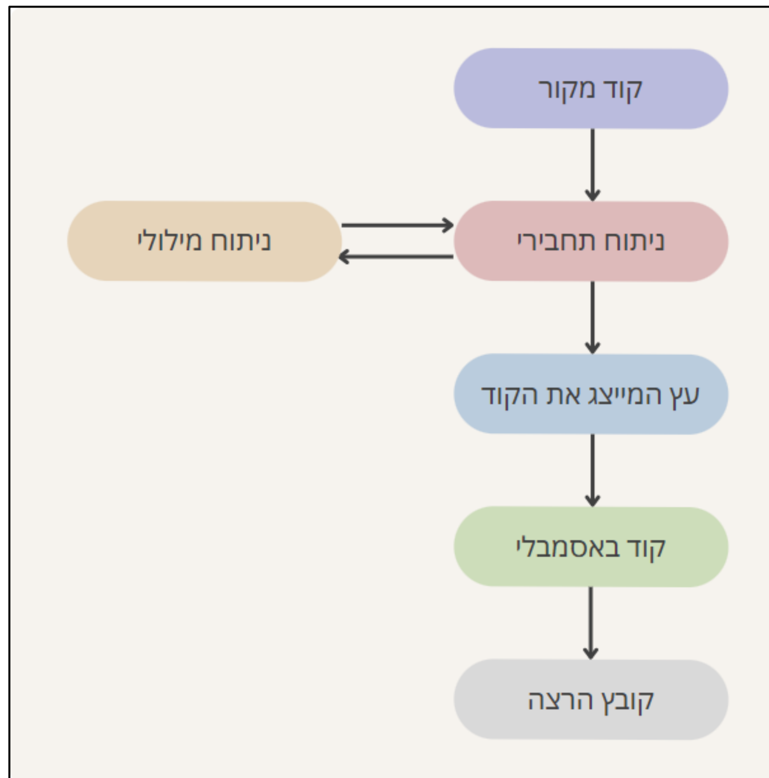
חשוב לי להדגיש כי משום שאני רוצה שהמאמר יפנה לקהל יעד כמה שיותר רחב, לא אכתוב את הקוד ברמה גבוהה או אקפיד על good practices ואתעדף קוד פשוט וקצר על קוד בעל יכולת. לכן המהדר לא יבדוק שגיאות בקוד בכלל!

הערה: ממליץ לא להעתיק את הקוד שנכתוב במאמר, רק להבין אותו ולהוריד את הקוד שמופיע בצורה מסודרת בסיום המאמר.

קצת תיאוריה

תהליך ההידור הינו תהליך מורכב, הכולל שלבים רבים אותם עובר קוד המקור עד שמגיע לפורמט אותו המחשב יכול להבין, ובשלבים האלה נדון במאמר זה. המטרה שלנו: לנתח את קוד המקור, ולהעביר אותו מרצף מילים באנגלית אשר בתקווה עונות על כללי התחביר של השפה, לקוד בשפת אסמבלי. לאחר מכן נוכל להשתמש באסמבלר קיים, ולפלוט קובץ הרצה.

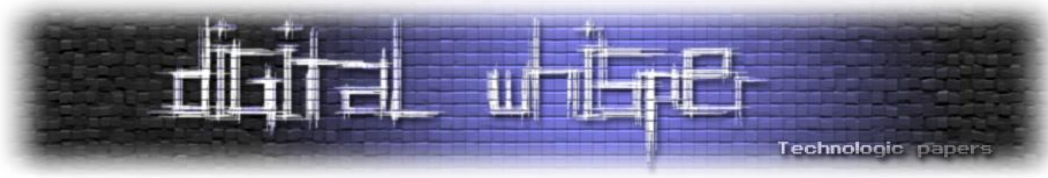
בתרשים הבא תוכלו לראות את שלבי ההידור:



נתחיל להסביר בקצרה על כל אחד מהם על מנת שתדעו לקראת מה אנחנו עומדים. השלב הראשון הוא הניתוח התחבירי אשר רץ במקביל עם הניתוח המילולי. בקצרה, הניתוח המילולי שואף להתעלם מהתווים הלא הרלוונטיים, ולשמור את אלה שכן בחלוקה משלנו, כך הוא יכול להבין מתי יש תווים לא מוכרים ולהתעלם מהם.

הניתוח התחבירי קורא בכל פעם לפונקציה המבצעת את הניתוח המילולי, ומחזירה את החלק הבא בקוד, ולאחר שיש לו רצף מוכר של חלקים, לדוגמה הגדרה של פונקציה, הוא יכול לשים כל אחד מהם במבנה נתונים משלנו. בשלב הזה סיימנו את שלב הניתוח התחבירי ויש לנו מבנה נתונים משלנו המייצג את הקוד בתוכנית. כעת, עלינו ליצור קוד אסמבלי המייצג את העץ שלנו. לכל מבנה נתונים תהיה פונקציה מיוחדת, שיודעת כיצד לפלוט קוד אסמבלי עבור מבנה הנתונים הזה, אך אם נחשוב איך פרקטית העיצוב עובד, נבין שלמבני הנתונים תהיה היררכיה מסוימת. כך, נוכל לחלק את ייצור הקוד לפעולות קטנות, כאשר כל אחת יודעת לעשות את שלה, והן קוראות אחת לשנייה לפי הצורך.

ככלל, אני מאמין שיהיה קל יותר להעביר את שלבי התהליך על ידי אם נעבור אותו יחד. לכן, נכתוב היום מיני מהדר, התומך (כנראה) רק בקוד הדוגמה שאספק. למרות זאת, הקוד ייכתב בצורה שתתן תשתית להרחבה, מלבד ניהול שגיאות שהוא מעבר לתוכן המאמר. לכן מי שירצה להרחיב את המהדר יוכל לעשות זאת, ולתמוך בעוד פיצ'רים מגניבים. רקע נוסף שאני מעוניין לתת, הוא שבכל המאמר אני עומד להשתמש ב"מצביעים חכמים" (smart pointers) מסוג unique_ptr, וזאת בכדי ליהנות ממספר יתרונות אותם מציעים אותם מצביעים. מוזמנים לקרוא עוד כאן.



שלב הניתוח המילולי

ראשית, עלינו לחלק את הקוד שלנו לחלקים הנקראים טוקנים.

טוקן הינו רצף מוגדר מראש של אותיות המייצג סוג הקיים בשפה. הטוקנים מייצגים את המשמעות של החלקים הקטנים ביותר בשפה, דוגמאות לטוקנים כאלו יהיו: מספר, המילים השמורות בשפה, אופרטורים, שמות של משתנים או של פונקציות. בשלב זה אנו אחראיים על הסינון של הרווחים, ירידות שורה, טאבים וכו', וההתייחסות רק לטוקנים.

אצרף ביטוי התחלתי, לפיו נגדיר את כללי התחביר של השפה שנהדר בעזרת המהדר שלנו. הביטוי כתוב בשפה פשוטה הדומה תחבירית לשפת פייתון. ככל שנתקדם בשלבים נראה את השפעתם על הביטוי:

```
def main()
    print(123)
```

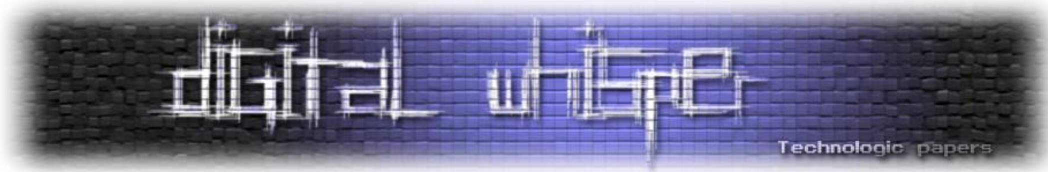
בואו נגדיר את המחלקה עבור הניתוח המילולי ואת המשתנים והפעולות שלה ואסביר על הצורך בכל אחד מהם:

```
class Lexer
{
public:
    static char lastchar;
    static int numvalue;
    static std::string identifierstr;

    enum Token
    {
        tok_EOF = -1,
        tok_function = -2,
        tok_number = -3,
        tok_return = -4,
        tok_identifier = -5
    };
```

המשתנה lastchar הינו משתנה עזר של המחלקה ומשמש אותה לשמירת התו הנוכחי. המשתנה numvalue הינו משתנה האחראי על שמירת הערך המספרי כשנמצא tok_number. דמיינו את השימוש של השלב הבא, הוא יקרא לפעולה get_token, יראה שהטוקן הוא של מספר, אבל איפה ימצא את המספר? במשתנה המחלקה numvalue.

המשתנה identifierstr הוא בדיוק על אותו עקרון, אם יש tok_identifier שם ישמר השם.

ה־Enum הינו מבנה נתונים הנותן שם למספרים בצורה נוחה למקרה שלנו, המספרים שבחרתי שליליים משום שאם לא ימצא טוקן עבור תו מסוים, נחזיר את ערך ה־ASCII שלו. המספרים שליליים בכדי למנוע התנגשות בין ערך ASCII של תו לבין מספר מזהה של טוקן.



נכתוב פעולה אשר בהינתן הקוד מחזירה את הטוקן הבא בו:

```
int get_token(std::string& content) {
```

נתחיל מלדלג על הרווחים שבהתחלה, באמצעות הפעולה isspace אשר תדלג על כל תווי ה־white space, ותוביל אותנו ישר אל תו ASCII. כעת נבדוק מהו התו הראשון, ובהתאם נקרא לפונקציה מתאימה נוספת. אם התו הוא אות, הטוקן יכול להיות שם או keyword, ועלינו לוודא שבאות אחריו אותיות או ספרות. אם התו הוא ספרה הטוקן יהיה tok_number, אך עלינו לוודא שההמשך שלו הוא רק ספרות:

```
  while (isspace(Lexer::lastchar))
  {
    Lexer::lastchar = getnext(content);
  }
 if (isalpha(Lexer::lastchar))
     return Lexer::is_identifier(content);

 if (isdigit(Lexer::lastchar))
     return Lexer::is_numeric(content);
```

אם לא נכנסו לאף פעולה, נבדוק אם הגענו ל־EOF, ואם לא נחזיר את ערך ה־ASCII של התו:

```
    if (Lexer::lastchar == EOF)
        return tok_EOF;

    return Lexer::lastchar;
}
```

כתבנו את הפעולה הראשית, אליה יקראו בשלב הבא. נממש כעת את תת הפעולות להן הפעולה הראשית קראה. נתחיל מהפעולה getnext:

```
    static int getnext(std::string &s)
    {
        if (s.length() > 0)
        {
            char c = s.at(0);
            s = s.substr(1, s.length() - 1);
            return c;
        }
        return EOF;
    }
```

הפעולה תחזיר את התו הראשון של המחרוזת שהיא תקבל, ותמחק את התו מהמחרוזת, כך נוכל לעבור על התווים בצורה נוחה, היא כמובן תוודא שהאורך של המחרוזת גדול מ־0 ואם לא תחזיר 0 ונדע שנגמר הקוד.

כעת נעבור למימוש של פעולות הניתוח עבור מזהה ועבור מספר:

```
    static int is_identifier(std::string& content)
    {
        Lexer::identifierstr = lastchar;
        while (isalnum(lastchar))
        {
            if (isalnum(content.at(0)))
            {
                lastchar = getnext(content);
                identifierstr += lastchar;
            }
            else break;
        }
        if (identifierstr == "def")
        {
            return tok_function;
        }
        return tok_identifier;
    }

    static int is_numeric(std::string &content)
    {
        std::string num(1, lastchar);
        while (isdigit(lastchar))
        {
            if (isalnum(content.at(0)))
            {
                lastchar = getnext(content);
                num += lastchar;
            }
            else
                break;
        }
        Lexer::numvalue = std::stoi(num);
        return tok_number;
    }
};
```

המימוש הוא בדיוק מה שהיינו מצפים, הפעולות עוברות תו אחר תו ובודקות תנאי מסוים, עבור משתנים, שהתו הראשון הוא אות והשאר הוא אותיות או ספרות, ועבור מספרים, שהכל ספרות. שימו לב, הפעולות משתמשות במשתנה המחלקה המתאים להן, ונותנות לו את הערך של התווים שעליהם היא עברה, בהתאם להסבר על משתני המחלקה מוקדם יותר. פה נסגור את המחלקה Lexer, כעת עלינו להקצות את הזיכרון עבור המשתנים הסטטיים שהצהרנו על קיומם בתוך המחלקה:

```
std::string Lexer::identifierstr;
int Lexer::numvalue;
char Lexer::lastchar;
```

ובזאת סיימנו את חלק הניתוח המילולי. ובכן, כעת תוכלו להסתכל על קוד הדוגמה ולומר אילו טוקנים יצאו? התשובה משמאל לימין:

keyword, identifier, (,), identifier, (, number,)

הערה: מעוניין לציין שבפייתון ההזחה רלוונטית כטוקן, אך לצורך הפשטות של הדוגמה נתעלם ממנה.

שלב הניתוח התחבירי

בשלב זה, אנו נבנה עץ המייצג את הטוקנים שאנו קוראים.

הערה על תקינות התחביר, תחביר תקין, משמע סדר הטוקנים תקין, ואין טוקן באמצע שלא ציפינו לו. במהדרים לשפות רציניות, אם התחביר אינו תקין, יש לזהות את השגיאה, "לתקן אותה" (בכדי לאפשר המשך ניתוח מוצלח) ולהודיע על קיומה, ולאחר מכן להמשיך לנתח את שאר הקוד. כאמור אנחנו לא נתעסק עם שגיאות בתחביר.

לכל סוג ביטוי בשפה יהיה טיפוס עץ משל עצמו. לדוגמה: עץ אב־טיפוס של פונקציה יהיה אחראי על ההצהרות על הפונקציה, עץ תנאי יהיה אחראי על ביטוי if-else, עץ לולאה מסוג while וכל סוג אחר של ביטוי שנרצה.

נתחיל מלהגדיר משתנה עזר גלובלי בשם current_token, המשתנה מייצג את הטוקן הנוכחי, כך ערכו העדכני של הטוקן תמיד יהיה זמין לנו, ונוכל לבצע בדיקות שונות עליו. נגדיר גם פעולת מעטפת (wrapper) שתשים את ערך החזרה של הפעולה Lexer::get_token במשתנה שלנו, ותחזיר את ערכו של הטוקן:

```
static int current_token;
int getNextToken(std::string &content)
{
    current_token = Lexer::get_token(content);
    return current_token;
}
```

בניית מבני הנתונים הדרושים

ראשית, עלינו לחשוב כיצד נוכל לייצג את גוף הפונקציה? לרוב הגוף מכיל רשימה של ביטויים מסוגים שונים, אנחנו נתמוך רק בביטוי בודד אבל אסביר גם כיצד תוכלו להרחיב את התמיכה.

כעת אנו נתקלים בבעיה: כיצד נוכל לשמור ביטוי (או רצף של ביטויים) המהווה בלוק בקוד שלנו, אם כל ביטוי ישתייך למחלקה אחרת? לדוגמה: הקריאה לפונקציה תהיה מהטיפוס עץ קריאה לפונקציה, בעוד שביטוי if בתוך הפונקציה ישתייך למחלקה אחרת, ולא נוכל לדעת מראש אילו ביטויים נצטרך לשמור.

המסקנה: עלינו להגדיר מחלקת אם משותפת המייצגת ביטוי כללי. כך, באמצעות פולימורפיזם, נוכל להגדיר מצביע ממחלקת האם, והוא יוכל להצביע על כל טיפוס היורש ממנה.

נגדיר את המחלקה עבור ביטוי כללי, חשוב להדגיש כי זו לא מחלקה שאמורים להיווצר ממנה מופעים (instances), ולכן נגדיר אותה כמחלקה מופשטת (abstract) על ידי כך שנגדיר בה פעולה וירטואלית טהורה, אשר תכפה עבור כל מחלקת בת היורשת ממנה לממש את הפעולה, ותמנע יצירת מופעים ממחלקת האם.

נדון בסוג הפעולה codegen וערך החזרה *Value בהמשך:

```
class ExprAST
{
public:
    virtual Value* codegen() = 0;
};
```

כעת, נגדיר את הטיפוס ExpressionList, בתור מערך דינאמי של מצביעים למחלקת האם שלנו:

```
typedef std::vector<std::unique_ptr<ExprAST>> ExpressionList;
```

כך תוכלו להשתמש בטיפוס הזה מתי שתצטרכו בלוק של ביטויים, בין אם בתוך פונקציה, עבור פרמטרים של פונקציה, לאחר לולאה ובכל מקום אחר.

לצערי לא נתמוך כאן בבלוקים, אך מי שמעוניין יצטרך להמשיך ולכתוב פונקציית Parse ייעודית לבלוק, הנקראת בכל פעם שמזוהה בלוק על פי כללי השפה, הפעולה תהיה אחראית לעבור על כל הביטויים שבבלוק, לקרוא לפעולת הניתוח הכללי, וליצור ולהחזיר את הטיפוס ExpressionList הכולל את רשימת הביטויים שניתחנו.

נמשיך ונגדיר את המחלקות העיקריות עבור כל סוגי הביטויים בהם נתמוך. ובכן, אם נסתכל על הקוד שנתתי כדוגמה, נראה כי הוא מגדיר פונקציה, משם נתחיל:

```
class PrototypeAST
{
    std::string name;
    std::vector<std::string> args;

public:
    PrototypeAST(const std::string &name, std::vector<std::string> Args)
    {
        this->name = name;
        this->args = std::move(Args);
    }
      Function* codegen();
};
```

בקוד אנו מגדירים את העץ רק עבור אב־הטיפוס של הפונקציה, ללא הגוף שלה. אתם שואלים בשביל מה זה שימושי ומדוע יש צורך בהפרדה?

ובכן, ישנן שפות בהן ניתן להצהיר על פונקציה (declare) בלי להגדיר אותה. כך הפונקציה מתווספת אל טבלת הפונקציות של המהדר, והוא מעביר ללינקר את האחריות על פתירת כתובת הפונקציה (Symbol Resolution). אם נוסיף את המילה “extern” לפני החתימה של הפונקציה נקבל הצהרה ללא גוף.

בהמשך, נגדיר מחלקה גם עבור הגדרת פונקציה, במה היא שונה אמרנו? היא מכילה את גוף הפונקציה:

```
class FunctionAST
{
    std::unique_ptr<PrototypeAST> proto;
    std::unique_ptr<ExprAST> body;  // ExpressionList body

public:
    FunctionAST(std::unique_ptr<PrototypeAST> Proto, ExpressionList
Body)
    {
        this->proto = std::move(Proto);
        for (auto &&expr : Body)
        {
            this->body.emplace_back(std::move(expr));
        }
    }
      Function *codegen();
};
```

הקוד בסך הכל עוטף את אב־הטיפוס ומוסיף את גוף הפונקציה ממחלקת האם ExprAST. לאחר מכן אנחנו כותבים את הפעולה הבונה (constructor) של המחלקה, המקבלת אב־טיפוס ומצביע לביטוי כללי ומאתחלת את המשתנים שלה.

כעת נממש את שאר המחלקות. אם נסתכל על השורה השנייה בקוד הדוגמה:

```
    print(123)
```

הקוד כולל קריאה לפונקציה print, עם פרמטר מספרי בעל הערך 123. נגדיר מחלקה עבור קריאה לפונקציה, הבנו שהפרמטרים הנדרשים הם שם הפונקציה, והפרמטרים איתם קוראים לה:

```
class CallExprAST : public ExprAST
{
    std::string callee;
     ExpressionList args;

public:
    CallExprAST(const std::string& callee, ExpressionList args)
    {
        this->callee = callee;
        this->args = std::move(args);
    }
    Value* codegen() override;
};
```

המחלקה יורשת ממחלקת האם עבור ביטוי, מגדירה את שני המשתנים הדרושים ופעולה בונה המשימה את ערכיהם של הפרמטרים במשתני המחלקה.

נמשיך הלאה, טיפוס הפרמטר היה מספר שלם. עלינו להגדיר מחלקה עבור מספרים, ולה תכונה אחת והיא: ניחשתם נכון – הערך המספרי שלו:

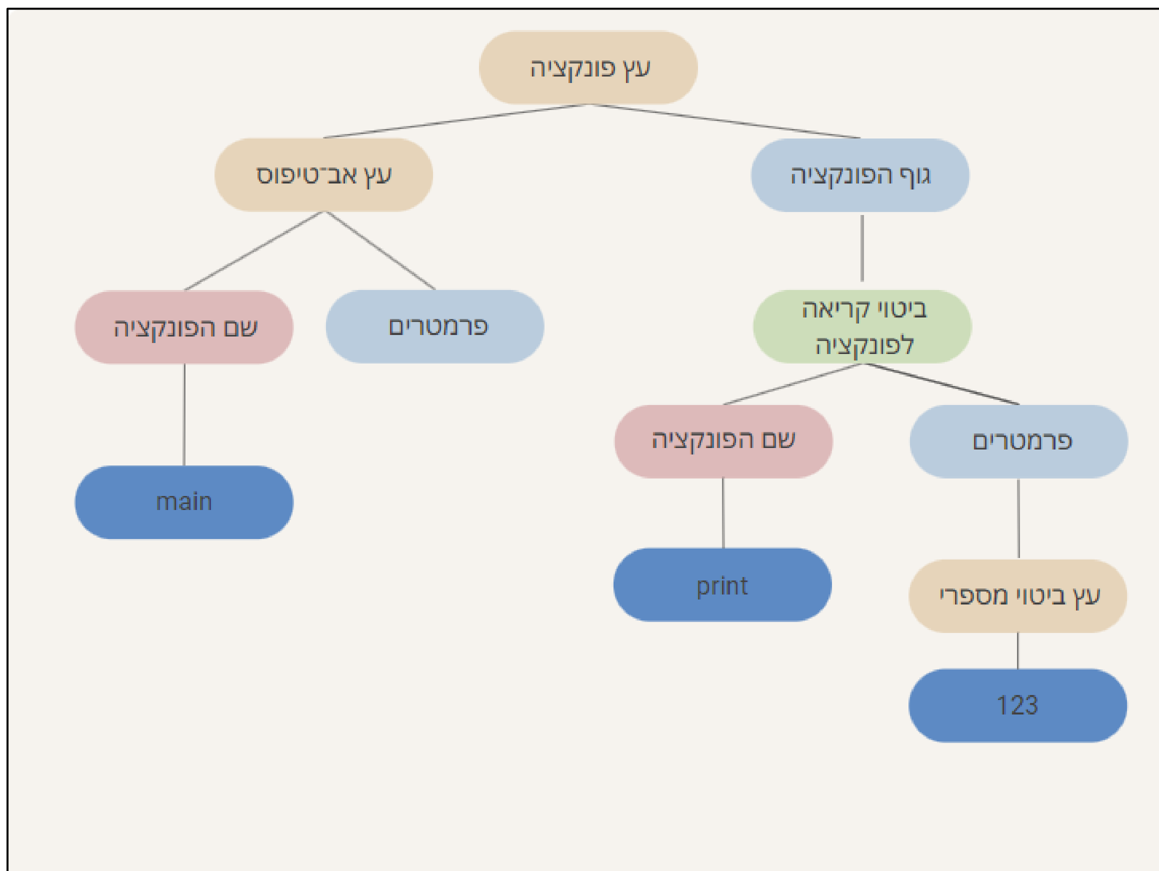
```
class NumberExprAST : public ExprAST
{
    int value;

public:
    NumberExprAST(int value)
    {
        this->value = value;
    }
    Value* codegen() override;
};
```

הפעולה הבונה מקבלת מספר שלם ומשימה אותו אל המשתנה בתוך המחלקה.

סיימנו לכתוב את כל המחלקות הדרושות בכדי לנתח את קוד הדוגמה. כמובן שלמהדר אמיתי יהיו עוד המון מבני נתונים בכדי לייצג את כל שאר הביטויים הקיימים בשפה.

משימה: חישבו כיצד יראה העץ שייוצר מקוד הדוגמה?

תשובה:



כתיבת פעולות הניתוח

כעת נכתוב את הפעולות האחראיות על בניית העצים שלנו. תהיה לולאה, שבודקת מהו הטוקן הנוכחי, ובהתאם תקרא לפעולת הניתוח הרלוונטית, האחראית על אימות תחביר השפה, על ידי קריאות נוספות ל־get_token, כך תקבל את הטוקנים שבאים אחרי, ותוודא כי הם בסדר הנכון ויוצרים ביטוי תקין, לבסוף תקרא לבונה הרלוונטית. התחביר בו אנו נשתמש דומה לתחביר של שפת פייתון אך ללא ההזחה, זאת אומרת שלא נתמוך בבלוקים, וגוף הפונקציה יהיה השורה שאחריה.

נתחיל מפעולת הניתוח על פונקציה, היוצרת עץ המייצג פונקציה. עליה לנתח את אב־הטיפוס ולאחר מכן את גופה של הפונקציה. נחלק את שתי המשימות האלה לפעולות, נתחיל בניתוח של אב־טיפוס.

נזכר בתכנון, ישנה לולאה הקוראת לפעולת הניתוח הרלוונטית. נחשוב כיצד הלולאה יודעת לאיזו פונקציה לקרוא? היא בודקת מיהו הטוקן שהיא מקבלת, ולפיו קוראת לפונקציה המתאימה. חשוב להדגיש, הפעולה המנתחת אב־טיפוס לעולם לא תקרא מהלולאה. משום שתמיד תבוא לפניה אחת מהמילים def או extern, אב־טיפוס לא יכול להתקיים ללא אחת המילים האלה, לפיהן נקרא לפעולה המתאימה.

לכן, נחליט שהפעולה נקראת כאשר הטוקן הנוכחי הוא כבר שם הפונקציה ולא אחת המילים שלפניה. הנחת היסוד של כל אחת מן הפונקציות היא שהיא נקראת אך ורק כאשר הטוקן הנוכחי הוא שייך אליה. במקרה שלנו, הטוקן צריך להיות טוקן ההגדרה של פונקציה: המילה השמורה "def". והטוקן הראשון בו הפונקציה נתקלת הוא מזהה הפונקציה (identifier):

```
static std::unique_ptr<PrototypeAST> ParsePrototype(std::string&
content)
{
    std::string FnName = Lexer::identifierstr;
    getNextToken(content);
```

שמרנו את שם הפונקציה ועברנו לטוקן הבא. כעת, הטוקן יהיה פתיחת סוגריים עבור הארגומנטים המועברים לפונקציה, אותם נשמור במערך דינאמי של מחרוזות:

```
    std::vector<std::string> ArgNames;
    while (getNextToken(content) == Lexer::tok_identifier)
        ArgNames.push_back(Lexer::identifierstr);
```

הלולאה עוברת על כל הטוקנים של מחרוזות:

```
    getNextToken(content);
    return std::make_unique<PrototypeAST>(FnName, std::move(ArgNames));
}
```

לבסוף, נעבור על התו) וניצור את העץ לאחר שניתחנו את הפרמטרים. נמשיך אל השלב השני ביצירת עץ הפונקציה והוא ניתוח הגוף שלה. הניתוח יתבצע על ידי הפעולה ParsePrimary, האחראי על ניתוח ביטוי כללי, ונכתוב אותה בסוף משום שהיא תלויה בפונקציות אחרות. חשוב מאוד שנוסיף את החתימה שלה בשלב הזה, בכדי להצהיר על קיומה ולתת לפונקציה המנתחת לקרוא לה:

```
static std::unique_ptr<ExprAST> ParsePrimary(std::string &content);
```

כעת נוכל לכתוב את הפונקציה הבונה עץ פונקציה, וקוראת לשתי הפונקציה שהוגדרו קודם:

```
static std::unique_ptr<FunctionAST> ParseDefinition(std::string
&content)
{
    getNextToken(content);  // eat def
    auto Proto = ParsePrototype(content);
    auto body_block = ParsePrimary(content);
    return std::make_unique<FunctionAST>(std::move(Proto),
            std::move(body_block));
}
```

מצוין. משום שכל הקוד נמצא בתוך פונקציות, פונקציית הניתוח שנותרה היא ParsePrimary, ותתי הפעולות אליהן היא קוראת. חשוב להדגיש כי על הפונקציה הזו לנתח ביטוי כללי, אך בפועל היא תקרא לתת פעולות אשר מנתחות כל אחד מהביטויים המופיעים בקוד שלנו. חישבו אילו ביטויים אנו צריכים לנתח? התשובה היא ביטוי מספר ליטרלי, וביטוי קריאה לפונקציה.

נכתוב את פעולת ניתוח עבור מספר שלם, אשר נקראת כשיש tok_number, לכן המספר כבר יחכה לנו במשתנה המחלקה Lexer::numvalue, ואז ניצור מופע של העץ המייצג מספר:

```
static std::unique_ptr<ExprAST> ParseNumberExpr(std::string &content)
{
    auto result = std::make_unique<NumberExprAST>(Lexer::numvalue);
    getNextToken(content);
    return std::move(result);
}
```

פעולת הניתוח הבאה תהיה עבור מזהה, אנחנו נתמוך רק בקריאות לפונקציה, לכן נניח שהטוקן שאחרי המזהה הוא '(' עבור קריאה לפונקציה, עבור מהדר אמיתי הפעולה תבדוק מה הטוקן הבא, אם הוא לא סוגריים זה אזכור משתנה ולא קריאה לפונקציה:

```
static std::unique_ptr<ExprAST> ParseIdentifierExpr(std::string
&content)
{
    std::string func_name = Lexer::identifierstr;
    getNextToken(content); // skip function identifier
    getNextToken(content); // skip '('
    ExpressionList args; // function args

    while (true)
    {
        if (auto arg = ParsePrimary(content))
        {
            args.push_back(std::move(arg));
        }
        else
        {
            getNextToken(content);
            return nullptr;
        }
        if (current_token == ')')
            break; // Done parsing args
        getNextToken(content);
    }
    getNextToken(content);
```

```
    return std::make_unique<CallExprAST>(id_name, std::move(args));
}
```

עכשיו נותר לנו לכתוב את הפונקציה הכללי לניתוח ביטוי:

```
static std::unique_ptr<ExprAST> ParsePrimary(std::string &content)
{
    switch (current_token)
    {
        case Lexer::tok_identifier:
            return ParseIdentifierExpr(content);
        case Lexer::tok_number:
            return ParseNumberExpr(content);
    }
}
```

יצירת קוד אסמבלי

מה נותר

מגניב, הצלחנו לנתח את קוד המקור ולאחסן אותו במבנה נתונים משלנו. בואו נחשוב יחד, מה נותר? לאחר שלב הניתוח, כשהקוד נותח ומאוחסן במבנה נתונים משלנו, ניתן להמשיך בכמה דרכים, אפשר להשתמש בספרייה חיצונית שתעשה את עבודת ה־backend עבורנו, כמו LLVM הפופולארית, האפשרות השנייה היא לכתוב את החלק הזה בעצמנו. במאמר אציג את האפשרות הראשונה.

הכרות עם LLVM

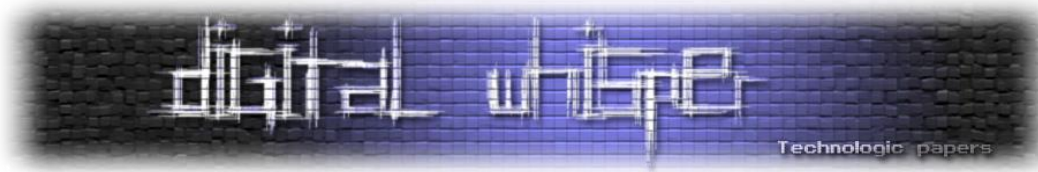
מהם תפקידיה של LLVM? היא אחראית על מתן ממשק לבניית קוד ביניים בתחביר משלה, בשם LLVM Intermediate Representation או IR, ולאחר מכן תפעיל אופטימיזציות אם נרצה, ותהדר את הקובץ לאסמבלי המתאים למעבד המבוקש ואז לקובץ object המתאים למערכת ההפעלה המבוקשת.

נתחיל מלהתקין את גרסה 15.0.7 של הספרייה, ישנן מספר סביבות אפשרויות, כמו להשתמש ב־ Visual Studio ו־MSVC, אך הסביבה הזאת דורשת שנהדר את LLVM בעצמנו לכן נשתמש היום ב־MSYS2 המכילה את הקבצים מהודרים מראש. כמובן שניתן להוריד את LLVM גם עבור מערכות הפעלה אחרות, יש מדריך רשמי של LLVM על כל אחת מהן.

MSYS2 היא סביבת פיתוח והרצה עבור Windows, וכוללת את ++g המהדר בו נשתמש היום ועוד מהדרים רבים ואת הספרייה LLVM מהודרת מראש ומוכנה לשימוש.

נתקין את הסביבה מכאן, נריץ את הפקודה מסעיף 6 עבור התקנת המהדר שבו נשתמש:

```
pacman -S mingw-w64-ucrt-x86_64-gcc
```



חשוב לזכור שמשום ש־LLVM זו ספרייה שמשתנה כל הזמן וכך גם ה־API שלה, נוריד את הספרייה בגרסה שלנו מהארכיון שלהם, ונריץ את הפקודה (בטרמינל שלהם):

```
Pacman -U mingw-w64-x86_64-llvm-15.0.7-3-any.pkg.tar.zst
```

אם הקישור כבר לא עובד, יהיה עליכם להיכנס ידנית לארכיון ולמצוא גרסה של LLVM איתה אתם יודעים לעבוד, לא רע מידי פעם לשדרג לגרסה העדכנית ביותר.

השימוש שלנו בספרייה יהיה מצומצם, ולשפת הביניים של LLVM יש תיעוד מלא והיא כוללת המון אפשרויות מגניבות. המטרה שלנו במאמר היא לעבור את תהליך ההידור אז לא נעבור על כל אלה.

לאחר התקנת הספרייה, נכתוב פעולת Code Generation עבור כל מבנה נתונים, ואז נוכל להשתמש בה כדי ליצור את קוד הביניים. הפונקציה הראשית תהיה הפונקציה המטפלת בעץ פונקציה, והיא תהיה אחראית על הקריאה לשאר הפעולות, בהתאם לקוד שנותח.

ראשית נתחיל מלהגדיר שלושה משתנים גלובליים שיעזרו לנו ביצירת קוד הביניים:

```
std::unique_ptr<LLVMContext> TheContext;
std::unique_ptr<Module> TheModule;
std::unique_ptr<IRBuilder<>> Builder;
```

- **TheContext** הוא אובייקט בסיסי של LLVM המכיל את מבני הנתונים, וצריך להיות קיים בכל תוכנית המייצרת קוד, אין צורך להבין לעומק.

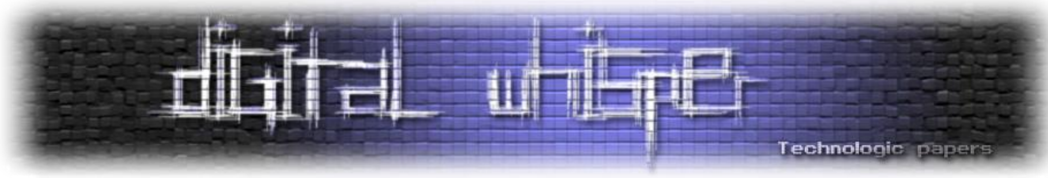
- **TheModule** מכיל את קוד הביניים שאנו יוצרים.

- **Builder** הוא אובייקט הבנייה של ההוראות בשפת הביניים של LLVM, נקרא לפונקציות עליו כשניצור הוראות.

פונקציה נוספת שאנו צריכים, היא פונקציה המאתחלת כל אחד מהמשתנים האלה:

```
void InitializeModule()
{
    TheContext = std::make_unique<LLVMContext>();
    TheModule = std::make_unique<Module>("MyCompiler", *TheContext);
    Builder = std::make_unique<IRBuilder<>>(*TheContext);
}
```

רקע נוסף שאני מעוניין לתת הוא לגבי ערכי החזרה של כל פונקציות ה־codegen, שהבטחתי הסברים על הטיפוסים האלה כבר מהשלב הקודם. הטיפוס llvm::Value אחראי על ייצוג כל ערך משפת הביניים שלהם, כלומר הוא יכול לייצג תוצאה של חישוב, כמו מספר שיצרנו, וגם לייצג ערך של הוראה בשפת הביניים כמו קריאה לפונקציה. הטיפוס llvm::Function אחראי על פונקציות בשפת הביניים, ומכיל רשימה מהטיפוס llvm::BasicBlock שיוצרים את ה־control flow של התוכנית. אנחנו נקרא לטיפוסים האלה בשמותיהם המקוצרים ללא ::llvm, בשלב מאוחר יותר נכתוב יחד את שורת הקוד שמאפשרת זאת.



נתחיל לכתוב את הקוד, וכמובן שחלקה הראשון של הפונקציה יהיה האב־טיפוס שלה, נמצא במבנה נתונים משלו, לכן נכתוב קודם את פונקציית יצירת קוד הביניים עבורה:

```
Function* PrototypeAST::codegen()
{
FunctionType *FT = FunctionType::get(Type::getInt32Ty(*TheContext),
true);
    Function *F = Function::Create(FT, Function::ExternalLinkage, this-
>name, TheModule.get());
    return F;
}
```

הפונקציה אחראית על יצירת החתימה של הפונקציה. כלומר, איזה טיפוס היא מחזירה ואילו טיפוסים היא מקבלת. אנחנו נתמוך כרגע ביצירת פונקציה עם חתימה כמו של הפונקציה הראשית של התוכנית:

```
int main()
```

וזה מה שקורה בשורה הראשונה שלה, בשורה שאחריה אנחנו מוסיפים את הפונקציה לטבלת הפונקציות הקיימת שנמצאת ב־Module שלנו.

נמשיך ונכתוב את codegen עבור עץ פונקציה:

```
Function* FunctionAST::codegen()
{
    Function* TheFunction = this->proto->codegen();
```

נתחיל מלקרוא לפעולה הקודמת שכתבנו ולשמור את ערך החזרה. עכשיו ניצור עצם מהטיפוס llvm::BasicBlock הכולל בתוכו רשימה של instructions, נשים את נקודת הכניסה של ההוראות לבלוק שיצרנו. מה שיגרום לכל קוד שניצור באמצעות ה־Builder להיווצר בבלוק שיצרנו עכשיו. ונקרא לפעולת ה־codegen עבור הביטוי הכללי. נשאיר למהדר להבין איזה ביטוי זה אבל במקרה שלנו יש אפשרות אחת והיא קריאה לפונקציה אחרת:

```
    BasicBlock* BB = BasicBlock::Create(*TheContext, "entry",
TheFunction);
    Builder->SetInsertPoint(BB);
    expr->codegen();
    verifyFunction(*TheFunction);

    return TheFunction;
}
```

נמשיך לכתוב את שאר פעולות codegen עבור שאר מבני הנתונים, אליהן אנו קוראים בלולאה. עבור מספרים ליטרלים:

```
Value *NumberExprAST::codegen()
{
    return ConstantInt::get(Type::getInt32Ty(*TheContext), this->value,
true);
}
```

ועבור ביטוי קריאה לפונקציה:

```
Value* CallExprAST::codegen()
{
    // Look up the name in the global module table.
    Function *CalleeF = TheModule->getFunction(this->callee);

    std::vector<Value*> ArgsV;
    for (size_t i = 0; i != args.size(); i++)
    {
        ArgsV.push_back(args[i]->codegen());
        if (!ArgsV.back())
            return nullptr;
    }
    return Builder->CreateCall(CalleeF, ArgsV, "calltmp");
}
```

הפונקציה בודקת האם קיימת פונקציה בשם שאליה אנו קוראים, מוצאת את כתובתה בעזרת ה־Module, יוצרת וקטור של הפרמטרים שלה וקוראת לה איתם.

פליטת קובץ Object

לאחר שכתבנו את כל פונקציה ה־codegen, ייצוג הביניים של ערכי החזרה מכל הפונקציות נשמר באובייקט מיוחד של LLVM שכבר הכרנו בשם Module.

כעת, עלינו להדר את קוד הביניים שנוצר עם התחביר של LLVM, באמצעות ה־API של המהדר שלהם. התהליך יכול לפלוט להיות קובץ אסמבלי או לתת לו להריץ את האסמבלר עבורנו ולפלוט ישירות קובץ object. אנו נעדיף את האפשרות השנייה, משום ש־LLVM מכילה הרבה אסמבלרים ותתמוך ביותר ארכיטקטורות, וגם בשביל לפשט את התהליך.

הפונקציה תכיל הרבה קריאות ל־API של LLVM, אני מתכוון להסביר את העיקריות אם כי לא את כולן, מי שמעוניין השארתי קישור לתיעוד שלהם בסיום המאמר:

```
void GenerateObject(std::string target)
{
        InitializeAllTargetInfos();
        InitializeAllTargets();
        InitializeAllTargetMCs();
        InitializeAllAsmParsers();
        InitializeAllAsmPrinters();

        TheModule->setTargetTriple(target);
        auto Target = TargetRegistry::lookupTarget(target, Error);
        auto CPU = "generic";
        auto Features = "";

        TargetOptions opt;
        auto RM = Optional<Reloc::Model>();
        auto TargetMachine = Target->createTargetMachine(target, CPU,
                                                          Features, opt, RM);

        TheModule->setDataLayout(TargetMachine->createDataLayout());
```

```
        std::error_code EC;
        raw_fd_ostream dest("output.o", EC, sys::fs::OF_None);
        legacy::PassManager pass;
        auto FileType = CGFT_ObjectFile;

        TargetMachine->addPassesToEmitFile(pass, dest, nullptr,
                                            FileType);
    }
    pass.run(*TheModule);
    dest.flush();
}
```

נעבור על הקוד. הוא מתחיל בלקרוא לפונקציות האתחול החלקים בספרייה בהם נשתמש. מגדיר את ה"Target Triple", שהמבנה שלה הוא השלשה הבאה ARCHITECTURE-VENDOR-OPERATING_SYSTEM הערך שלה בו אשתמש יהיה "x86_64-PC-Windows", כמובן שהערך מגיע כפרמטר לפונקציה ואתם מוזמנים לנסות ערכים נוספים, ממליץ להסתכל בתיעוד איך בדיוק לכתוב את השלשה שלכם. לאחר מכן יש קריאה ליצירת המכונה עליה תהדרו את הקוד לפי שלשת היעד שלכם. לבסוף, פותחים את הקובץ, שומרים את הסוג כ־object file של LLVM, וקוראים לפונקציה שכותבת את המידע לקובץ.

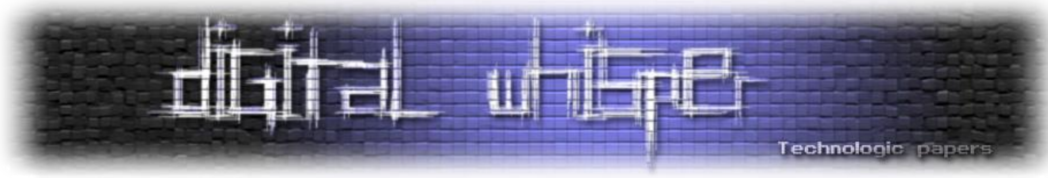
אם השתמשתם באותה שלשה שלי, כעת יש לכם קובץ object עם הפורמט של Windows בשם COFF. סוג הקובץ הזה אינו ניתן להרצה, אז הנה מגיע השלב האחרון.

קישור

קישור או Linking הוא השלב שקורה כאשר מחברים יחד כמה קבצים מסוג object או static library לקובץ הרצה אחד המכיל את כל הקוד. לדוגמה, אנו מקשרים את הקוד של המהדר עצמו, עם הקוד של ספריית LLVM המגיע בתור ספרייה מהודרת, ועלינו לקשר איתה בכדי להשתמש בפונקציות שלה. ונקשר גם עם הספרייה הסטנדרטית של C, אם תהיתם למה אנחנו צריכים אותה, חישבו על הפונקציה print לה אנו קוראים, היא מהווה פונקציה עוטפת עבור הפונקציה printf שלא אנחנו הגדרנו, לכן אנחנו צריכים לקשר איתה.

לאלה שחסר להם ידע תיאורטי או שסתם רוצים ללמוד עוד על התהליך ממליץ לקרוא את פרק 8 של ספר מערכות ההפעלה של ברק גונן, הספר כולו בעברית ומכיל הסברים על נושאים מתקדמים במערכות הפעלה.

נוכל לקרוא למקשר על קובץ ה־object שיצרנו משורת הפקודה, או שנוכל לכתוב קוד ב־C++ שיעשה זאת עבורנו. ממליץ על האפשרות השנייה, אבל אדגים פה את הראשונה.



איך הכל מתחבר

סיימנו לכתוב את כל השלבים ומימשנו את הפעולות הנדרשות, אבל מישהו צריך לקרוא לפעולות האלה. עכשיו נכתוב את הלולאה הראשית של התוכנית.

הלולאה תעבור על הטוקנים, ובהתאם תקרא לפעולת הניתוח הרלוונטית. משם, תקרא לפעולת codegen על הטיפוס שחזר מפעולת הניתוח. וחוזר חלילה:

```
void MainLoop(std::string& content)
{
    getNextToken(content);
    while (1)
    {
        switch (current_token)
        {
        case Lexer::tok_EOF:
            return;
        case Lexer::tok_function:
            auto F = ParseDefinition(content);
           F->codgen();
           break;
        }
    }
}
```

פונקציה פשוטה, יוצאת כשיש End of File, וקוראת לפונקציה האחראית על ניתוח פונקציה. הסיבה לפיצול הלולאה הפשוטה לפונקציה משלה היא בשביל לתת תשתית, כשקוד מורכב יותר יגיע יהיו יותר מקרים ב־switch והפעולה תגדל די מהר.

למי שתהה, הסיבה שאין קריאה ישירה לפעולה של הניתוח המילולי gettoken היא שאין צורך, משום שהפעולה האחראית על ניתוח פונקציה כבר אחראית לעבור על כל הטוקנים שלה, ומשאירה לנו את הטוקן הבא.

יצירת הפונקציה print

ובכן, הפונקציה לא תוסיף את עצמה. כרגע ב־symbol table של LLVM אין פעולה בשם print לכן נקבל שגיאה אם ננסה להריץ קוד שמשתמש בה, אבל לא נצפה מהמשתמש להגדיר אותה בעצמו. לכן אנו צריכים להוסיף בעצמנו את הפונקציה לטבלה, ולהגדיר את הגוף שלה בעצמנו דרך הממשק של LLVM. נגדיר פעולה שמוסיפה פונקציה לטבלה של המהדר, משום שנשתמש בקוד הזה יותר מפעם אחת:

```
Function* AddExternalFunc(std::string name) {
      FunctionType* FuncType = FunctionType::get(Type::
getInt32Ty(*TheContext), {Type::getInt32Ty(*TheContext)}, false); //
signature is: int f(int)
    return Function::Create(FuncType, Function::ExternalLinkage, name,
TheModule.get());
}
```

עכשיו הפונקציה מוכרת למהדר, הצהרנו עליה, אבל אין לה גוף. נחשוב מה הגוף שלה יכיל? משום שלא נרצה שהקוד יהיה תלוי מערכת הפעלה, נשתמש בפונקציה מהספרייה הסטנדרטית של C בשם printf, כמובן שהיא תקבל כפורמט %d כי אנחנו לא תומכים במשהו אחר, וגם עליה עלינו להצהיר מראש:

```
void CreatePrintFunction() {
    Function* PrintfFunc = AddExternalFunc("printf");
    Function* PrintFunc = AddExternalFunc("print");

    BasicBlock *entryBlock = BasicBlock::Create(*TheContext, "entry",
PrintFunc);
    Builder->SetInsertPoint(entryBlock);
    Constant* formatStr = Builder->CreateGlobalStringPtr("%d\n");
    Builder->CreateCall(PrintfFunc, {formatStr, &*PrintFunc-
>arg_begin()});
    Builder->CreateRet(ConstantInt::get(Type::getInt32Ty(*TheContext),
0, false)); // return 0
}
```

לצורכי המחשה, הפונקציה print שיצרנו בקוד, תהיה זהה לקוד המקור המצורף למטה, שהודר לקובץ object וקושר יחד עם main:

```
int print(int x)
{
    printf("%d", x);
    return 0;
}
```

הידור הפונקציה הראשית

מי שעקב אחרי כל הקוד שכתבנו, ולפי הסדר, רק צריך להוסיף את הפונקציה main בצורה הזו:

```
int main() {
    InitializeModule();
    CreatePrintFunction();
    std::string content = "def main()\nprint(123)";
    MainLoop(content);
    GenerateObject("x86_64-pc-windows");
}
```

כמובן שתוכלו להחליף את ההשמה של המחרוזת ב־content לפונקציה שקוראת קובץ ששמו מתקבל מ־argv, ומחזירה את תוכנו.

עכשיו יש להוסיף את הוראות ה־include הבאות בראש התוכנית:

```
#include <string>
#include <vector>
#include <memory>
#include <iostream>
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/Verifier.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/MC/TargetRegistry.h"
#include "llvm/Support/FileSystem.h"
```

```
#include "llvm/Support/Host.h"
#include "llvm/Support/TargetSelect.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Target/TargetOptions.h"
using namespace llvm;
```

ההוראה האחרונה נועדה בשביל הנוחות שלנו, שלא נצטרך לכתוב ::llvm לפני כל טיפוס מהספרייה.

הידור הקוד שלנו

כעת, כל מה שנותר לנו הוא להדר את קוד המקור שלנו, ולקבל מהדר משלנו. אסביר אילו שלבים יש על מנת להדר על Windows באמצעות המהדר ++g שהתקנו עם MSYS2. עבור מערכות הפעלה אחרות הקוד אמור לעבוד אך ידרשו שלבים אחרים בכדי להתקין את הספרייה LLVM ולהדר את הקוד. כמובן שכל הקוד יהיה זמין להורדה בצורה מסודרת בסיום המאמר.

נוסיף את כתובתן המלאה של התיקיות mingw64/bin ו־ucrt64/bin שנמצאות תחת /MSYS2, למשתנה הסביבה Path, מי שלא יודע איך, חיפוש אחד בגוגל כבר יסביר כל מה שתצטרכו לדעת.

כעת עלינו להריץ את הפקודה:

```
llvm-config --cxxflags --ldflags --system-libs --libs all
```

הפקודה קוראת לכלי עזר של LLVM האחראי לספק לנו את הארגומנטים אותם עלינו לספק למהדר של ++C על מנת להדר בהצלחה קובץ המשתמש בספריית LLVM, דברים כמו include path עבור קבצי header, libpath עבור קבצי lib שמכילים את הפונקציות של LLVM בהן אנו משתמשים, שמותיהם של הקבצים איתם אנו צריכים לקשר ועוד כמה דגלים.

לאחר מכן נעתיק את הפלט ונריץ את הפקודה הבאה:

```
g++ compiler.cpp <פלט מהפקודה הקודמת>  -o compiler.exe
```

כעת יפלט לנו קובץ הרצה בשם compiler.exe, זה המהדר שלנו! במקרה שלנו קוד המקור מוטמע בתוכו, אבל כמובן שניתן לשנות זאת כמו שהסברתי קודם.

עכשיו נריץ את המהדר, ונוצר לנו קובץ output.o, זהו קובץ ה־object שיצרנו!

נקשר את הקוד שוב באמצעות המקשר הכלול ב־++g:

```
g++ output.o -o ourmain.exe
```

נוצר קובץ ההרצה שלנו. נוכל כעת להריץ אותו, ונראה שאכן הודפס למסך 123.

סיכום

במהלך המאמר עברנו על השלבים העיקריים אותם עובר הקוד, ואף כתבנו PoC של מהדר עבור פייתון, עם backend של LLVM.

מאוד נהניתי לכתוב את המאמר, כיוון שזה נושא שתמיד עניין אותי, ואפילו לקחתי אותו כפרויקט סיום בכיתה י"ב.

אני מזמין לפנות אליי למייל shalevshagan1@gmail.com עם תגובות, שאלות והערות על המאמר, ובמיוחד אם כתבתם מהדר מתקדם משלכם!

לקריאה נוספת

במאמר זה הסברתי כל מה שיכולתי בגבולות המסמך, כמו שהשמטתי המון דברים חשובים ומעניינים. המטרה היא לתת לכם היכרות עם התהליך, ולא לעבור את כולו איתכם. אצרף רשימה של מקורות בהם תוכלו להמשיך את המחקר ולבנות מהדר מתפקד משלכם.

- קישור לקוד המלא ב־GitHub
- עוד על LLVM

יש המון ספרים טובים על מהדרים, אחד שאהבתי הוא Compilers: Principles, Techniques, and Tools.