Сутність компіляторів

Відомо, що для керування роботою процесору потрібні інструкції на зрозумілій йому (машинній) мові. Але у процесі розробки програми програмісту більш зручно використовувати мови високого рівня. То ж необхідні певні програми, які виконують переклад програм з ***вхідної мови*** високого рівня на ***вихідну***, яка є машинною мовою або максимально до неї наближена, а саме ***компілятори***.

За класифікацією компілятори відносяться до системного програмного забезпечення та призначені для перекладу програм з однієї мови програмування на іншу (Рис. 1.). Важною особливістю перекладу є те, що після перекладу програма повинна зберегти свою функціональність, яку вона мала первісно. Третьою мовою, яка фігурує при розробці компіляторів є ***мова реалізації***, тобто мова, на якої написана програма самого компілятору.

Рис. 1. Загальна схема використання мов у процесі компіляції

Звичайно, компілятор є досить потужною та складною програмою, але покроковий підхід дозволяє розділити процес розробки компілятора на декілька етапів у яких відпрацьовуються основні кроки розробки компілятору:

- Розробка та опрацювання базової структури компілятора
- Унарні та бінарні операції
- Обробка змінних. Локальні змінні.
- Вкладені конструкції. Оператори розгалуження. Тернарний оператор
- Функції
- Цикли

Таким чином, кожний наступний крок базується на знаннях, що отримані під час виконання попереднього, та у свою чергу слугує певним етапом побудови компілятору з більш розширеним функціоналом.

Покроковий підхід дозволяє почати розробку компілятора з самих простих виразів та шляхом поступового їх ускладнення отримати досить функціонально складний програмний продукт. Також цей підхід допоможе не загубитись у великих об’ємах написаного коду, просто його підтримувати та продовжувати розробку. І якщо необхідно щось кардинально змінити, це робиться одразу і без зайвих переписувань.

Цей принцип, який має назву «An Incremental Approach to Compiler Construction», викладений у [6]. Його також можна використовувати у процесі написання коду:

- писати та модифікувати код поступово;
- не робити величезні заготовки для того, чого може взагалі не бути;
- вирішувати проблеми за мірою їх надходження.

Щодо питання, а навіщо взагалі писати компілятор, то для цього є декілька причин [7]:

- з дізнатися про те, як обробляти та аналізувати текст програми, про абстрактні синтаксичні дерева (AST), лексери, парсери тощо, про те, для чого потрібні і як працюють ті елементи мов програмування, з якими ви зустрічаєтесь кожен день;
- зрозуміти низькорівневі деталі того, як працюють комп'ютери з програмами та як програми переходять з одного рівня на інший;
- застосовувати принцип Річарда Фейнмана: «What I cannot create, I do not understand».

Деякі техніки використовуються не лише в компіляторах, а й у інших областях: виділення синтаксису у текстових редакторах, регулярні вирази, парсинг URL, форматування, парсинг SQL, різноманітні програми-перекладачі тощо.

Хоча мова реалізації компілятору може бути будь-якою, для прикладів в якості вхідної мови обрано класична мова ***C*** та псевдокод для пояснень та формулювань.

В свою чергу, і якості вхідної мови у компіляторі буде використовуватися залежно від варіанту або мова ***C***, або мова ***Python***. Вибір саме цих мов ґрунтується на тому, що вони добре знайомі студентам та вивчалися ними раніше.

У мові ***C*** немає об'єктно орієнтованого програмування та інших складних парадигм і це найкращий варіант для простенького компілятора, тому усі приклади у методичці надаються саме для мови ***C***. Що стосується ***Python***, то це доволі складна і комплексна мова за своєю структурою та можливостями, але увагу насамперед буде звернено на відміну синтаксису. Якимось унікальним особливостям мови (у ***Python*** будь-що є об'єктом, особливості трансляції тощо) увага приділятись не буде. Інакше кажучи, це такий собі ***C*** з іншим синтаксисом і ця мова обрана лише у навчальних цілях.

Описи етапів містять численні приклади, які використовуються для пояснення основних ідей. Вони також дають змогу перевіряти правильність виконання програми на кожному кроці, але дозволяється використовувати свої методи та підходи.

Усі етапи мають схожу структуру та складаються з

- необхідних теоретичних відомостей;
- сутності загального завдання на даний етап;
- індивідуальних завдань по варіантам для кожного студента;
- контрольних питань для перевірки засвоєння матеріалу.

Етап 1
Розробка та опрацювання базової структури компілятора

Мета етапу: знайомство і опанування основних етапів компілювання, а також створення найпростішого компілятору.

Необхідні теоретичні відомості

Зазвичай, коли пояснюються основи будь-якої мови програмування, то перший приклад програми містить речення на кшталт «Hello, world!», яке за думкою авторів повинно продемонструвати її працездатність. Не будемо порушувати цю традицію, але перша програма, для якої треба буде розробити компілятор, не буде містить нічого, навіть славнозвісного речення. Це буде програми типу

```
int main(){           // C language
     return 2;
}
```

або

```
def main():           // Python language
     return 2
```

які повинен обробити компілятор.

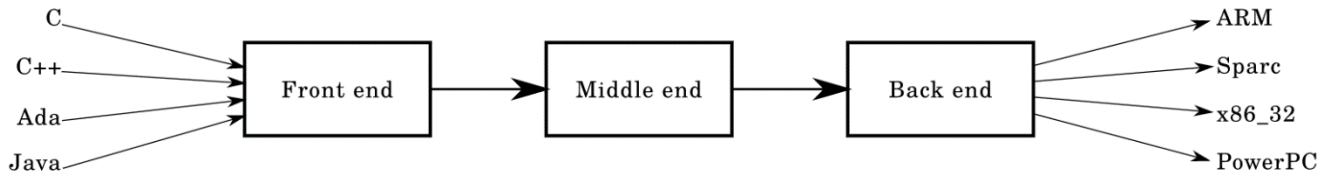
У програмі присутня тільки одна функція **`main`**, що складається лише з одного виразу **`return`**. Єдине що буде відрізнятися – це **тип**, **значення та система числення** числа, що повертається.

Написати лексер та парсер для такої програми виглядає дуже простим завданням, якщо використовувати регулярний вираз. На мові ***Python*** це можна зробити за 20 рядків. Проте цей спосіб має свої недоліки. Наприклад, він не може бути використаний у великих програмах, тобто не є універсальним [8].

Замість комплексних регулярних виразів розділимо компіляцію на 3 етапи]:

- Лексичний аналіз / лексер / ***lexer*** (***scanner, tokenizer***). - **Front end**
- Граматичний аналіз / парсер / ***parser***. - **Middle end**
- Генерація коду / ***code generation***. - **Back end**

Це типова архітектура компілятора, якщо не брати до уваги декілька етапів оптимізації:



Розділення компілятора на таку кількість окремих частин було необхідним десь до 1980 року, бо пам'яті комп'ютерів не вистачало для зберігання всього компілятора. Вміщались тільки його частини і завантажувались в пам’ять по черзі. Ці частини мали назву «проходи» (passes). Вихідні дані одного етапу є вхідними даними для наступного.

Сучасні технічні можливості дозволяють позбавитись такого ділення і зайвого доступу до пам'яті, та проміжних перетворень даних. Компілятори з одним проходом працюють набагато швидше. Але в цьому курсі розглядатиметься варіант з явним розділенням на етапи заради кращого розуміння всього процесу.

LEXING

Лексичний аналіз – етап компіляції, на якому вихідний код розділяється на список токенів (**лексем**). Лексеми будуть вхідними даними для наступного етапу компіляції.

Програма зчитується з файлу у рядок і лексер працює з рядком, в якому розпізнає шаблони і розрізає на лексеми. Через дослідження вхідного рядка тексту посимвольно, лексичний аналіз є найбільш ресурсоємним етапом компіляції, тому треба намагатись оптимізувати його як можна краще.

Токен – найменша одиниця, яку може зрозуміти парсер.

До токенів відносяться: назви змінних, ключові слова, константи, дужки, оператори тощо.

Деякі токени мають значення (***value***), деякі – ні. Також в деяких мовах пробіли можуть не бути токенами, а у інших – бути (***Python***).

Для наведеного вище прикладу програми на мові С список токенів може бути таким:

`int` – int_keyword
`main` – identifier
`(` – open parentheses
`)` – close parentheses
`{` – open brace
`return` – return_keyword
«`2`» – int_constant
‘ – open_quote
’ – close_quote
`f` – character (можуть бути лише у форматі ASCII)
«`3.14`» – float_constant
`;` – semicolon
`}` – close brace

Розділяти на токени можна по-різному. Наприклад, сприймати «**0xdeadbeef**, **1231234234**, **3.1412**, **55.5555**, **0b0001**» просто як числа, або відносити їх до окремих категорій:

0xdeadbeef – HexNumber
1231234234 – WholeNumber
3.1412 – FloatingNumber
55.5555 – FloatingNumber
0b0001 – BinaryNumber

Більш детальне розділення дозволить простіше працювати з обробкою різних типів.

В мові ***Python*** кількість пробілів виступає у ролі ідентифікатора блоків на кшталт фігурних дужок {} в ***C***. Стандартна кількість пробілів для виділення блоку (тіла функції в даному випадку) – чотири, але фактично їх може бути скільки завгодно.

Програма на мові ***C*** може бути написана в один рядок. У випадку ***Python*** для написання декількох інструкцій в одному рядку необхідно розділити їх крапкою з комою «;».

Ці особливості треба обов'язково враховувати при розділенні на лексеми.

Різниця представлення різних систем числення в мові ***C*** та ***Python***

C	Python	Система числення
int a = 3;	a = 3	десяткова
int a = 0b11;	a = 0b11	двійкова
int a = 011;	a = 0o11	вісімкова
int a = 0x11;	a = 0x11	шістнадцяткова

PARSING

Парсер – перевіряє правильність синтаксису мови, тобто правильність поєднання токенів у виразі. Процес має назву **синтаксичний аналіз**. Береться набір лексем і перетворюється у структуру (*AST/parse tree*), яка описує кожний елемент синтаксису та взаємодію цих елементів.

На прикладі псевдокоду це виглядає наступним чином:

```
TokenList lex (char* input){...}
AST parse (TokenList tokens){...}

char* input = "int main() { return 0; }";
TokenList tokens = lex(input);
AST parse_tree = parse(tokens);
```

Як вже зазначалося, компілятори складаються з декількох складових. Окремі, пов'язані між собою компоненти, приймають вхідні дані та перетворюють їх у якісь вихідні дані. Саме через таку особливість функціональні мови гарно підходять до написання компілятора.

Оскільки мова описується за допомогою **граматики**, то для перевірки правильності будь якого виразу використовується **дерево розбору** або **синтаксичне дерево *(parse tree***), яке будується на основі правил цієї граматики.

Граматика визначає, як набір токенів буде пов'язаний між собою для формування мовної конструкції та як перетворити набір токенів у синтаксичне дерево. Граматика визначає структуру мови, описує її за допомогою правил.

Граматика мови складається з:

- Термінальних символів (**терміналів**). Вони не можуть бути замінені іншими символами. Процес заміщення завершується терміналами.
- Нетермінальних символів (**нетерміналів**). Вони представляють синтаксичні класи/групи і заміщуються наступними нетерміналами або терміналами.
- **Породжуючих правил**. Вони відображають сам процес заміни нетерміналів на інші не термінали або термінали.
- **Стартового нетерміналу**.

Таким чином, мова - це набір послідовностей термінальних символів, які, починаючи зі стартового нетермінала, можуть бути згенеровані повторним застосуванням породжуючих правил граматики.

Правила визначені у формі **Бекуса-Наура (Backus Naur Form - BNF)** [9]. У цій нотації для представлення повторення того чи іншого символу/правила використана рекурсія. **Розширена форма Бекуса-Наура** [10] має дві додаткові конструкції для представлення повторень та опціональних елементів, а також дозволяє виразам бути огорненим у дужки. Детальніше ви дізнаєтесь на лекціях та за посиланням [3].

Синтаксичне дерево (дерево розбору)

- є формою представлення програми відповідній граматиці вихідної мови (наприклад, БНФ або РБНФ);
- будується вручну, або генераторами парсерів, такими як *Yacc*;
- є найбільш детальним структурним представленням програми.
- кожний вузол дерева є нетерміналом чи терміналом в граматиці.

Наприклад, для того, щоб побудувати ***дерево розбору*** для будь-якого арифметичного виразу, де застосовуються чотири змінні ***a***, ***b***, ***c*** та ***d*** спочатку будується граматика, один з варіантів правил якої має вигляд:

$$\mathbf{S \rightarrow T \mid T + S \mid T - S \mid T * S \mid T / S}$$
$$\mathbf{T \rightarrow (S) \mid a \mid b \mid c \mid d}$$

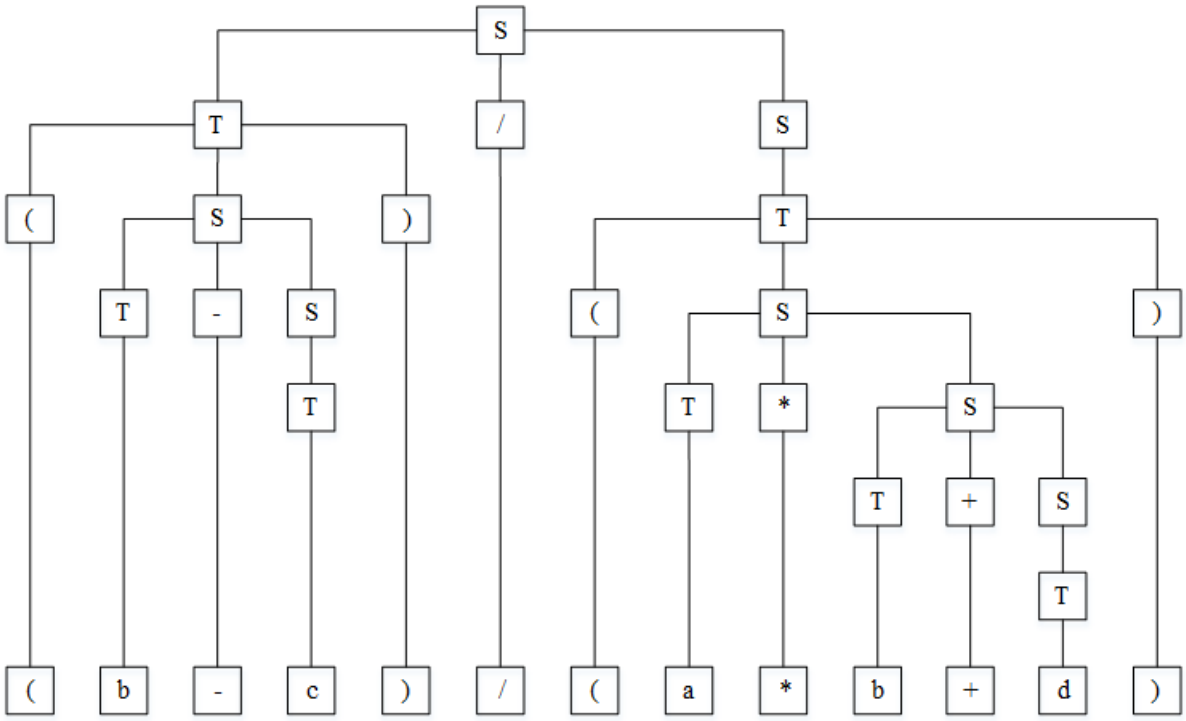
де ***a***, ***b***, ***c, d,*** **(,),+, -, *, /** - термінальні символи;

S, T – нетермінальні символи;

S – стартовий нетермінал.

Тепер на основі цих правил побудуємо ***дерево розбору***, наприклад, для виразу

$$(\boldsymbol{b - c}) / (\boldsymbol{a * b + d})$$



При побудові цього дерева розбору використовувався **низхідний рекурсивний розбір** [11-13], при якому потрібне правило граматики визначається його правою частиною. Якщо у виразі залишаться частки, для яких не буде знайдено відповідного правила граматики, то робиться висновок, що вираз не належить до даної граматики. Парсер з рекурсивним спуском є найбільш простим та ефективним.

Такий розбір потребує граматики, що належить до виду **LL(1)**, що означає виконання аналізу згори до низу зліва направо, перевіряючи лише один символ попереду. Цей символ повинен підказати компілятору, яким шляхом йти далі. Це аналогічно аналізу програми людиною.

Примітка: *належність до виду **LL(1)** важливо для аналізу токенів під час парсингу, а не для проходу символів під час лексингу, де можна заглядати на декілька символів вперед у пошуках крапки, обробляючи числа з плаваючою комою тощо.*

Для аналізу та обробки виразів також може бути побудовано ***абстрактне синтаксичне дерево*** (Abstract Syntax Tree, AST), яке вважається більш зручним. **AST** – це інший спосіб представлення структури програми, яке має наступні відмінності від дерева розбору:

- видалення більшості нетермінальних вузлів, що мають одного нащадка;
- заміна частини термінальних символів атрибутами вузлів;
- модифікація групуючих вузлів.

AST є структурним представленням вихідної програми, яке очищене від елементів конкретного синтаксису. Відкидається непотрібна структурна/граматична інформація, яка не несе в собі семантику програми. Коренем **AST** є уся програма, а кожний вузол має дочірній, що представляє

складові частини. Кожний вузол має в собі властивості, що описують ту чи іншу ізольовану частину дерева.

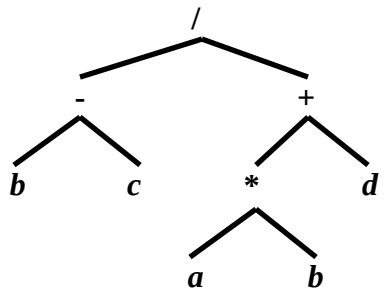
В якості вузлів у **AST** виступають оператори, до яких приєднуються їх аргументи, які у свою чергу також можуть бути складеними вузлами. У багатьох мовах програмування такі мовні конструкції, як розгалуження, об'явлення функцій тощо складаються з простіших елементів – змінних, констант. **AST** відображає цей зв'язок.

Приклад 1.1

Побудуємо **AST** для виразу

$$(b - c) / (a * b + d)$$

Спочатку визначимо операцію, яка буде виконуватися останньою - це ділення. Воно буде коренем, який має двох нащадків, які ототожнюються з виразами у дужках. У других дужках спочатку повинна виконуватися операція множення, а потім – операція віднімання, що також знаходить відображення у побудованому **AST**:



Псевдокод для цього дерева:

```
{
     type: 'Program',      // Starting at the top level of the AST
     body: [{
          type: 'CallExpression',    // Moving to the first element
          name: 'div',               // of the Program's body
          params: [{
               type: 'CallExpression',    // Moving to the first element
               name: 'sub',               // of CallExpression's params
               params: [{
                    type: 'NumberLiteral', // Moving to the first element
                    value: 'b',            // of CallExpression's params
               },
               {
                    type: 'NumberLiteral',// Moving to the second element
                    value: 'c',           // of CallExpression's params
               }]
          },
          {
               type: 'CallExpression',    // Moving to the second element
               name: 'add',               // of CallExpression's params
               params: [{
                    type: 'CallExpression', //Moving to the first element
```

```
                    name: 'mul',
                    params: [{
                         type: 'NumberLiteral',//Moving to the first one
                         value: 'a',
                    },
                    {
                         type: 'NumberLiteral',//Moving to the second one
                         value: 'b',
                    }]
               },
               {
                    type: 'NumberLiteral', //Moving to the second one
                    value: 'd',
               }]
          }]
     }]
}
```

У випадку необхідності реалізації синтаксичного аналізатора досить складної формальної мови слід здійснювати розробку від низу до верху, починаючи від елементарних виразів до усе більш складних синтаксичних конструкцій, з тестуванням проміжного результату на кожному кроці. Так допрацьовувати останній розглянутий проект можна було б таким чином:

- додати складений оператор;
- додати оператори порівняння і логічні оператори;
- додати умовний оператор;
- додати цикли;
- додати опис і виклик функцій;
- і т.д.

При цьому на кожному кроці виходив би працездатний парсер, який міг би розбирати якусь підмножину необхідної формальної мови.

Розглянемо ще один приклад побудови AST для виразу

```
if (a < b) {
    c = 2;
    return c;
} else {
    c = 3;
}
```

Коренем AST буде "***if statement***". В нього буде три дочірніх вузли:

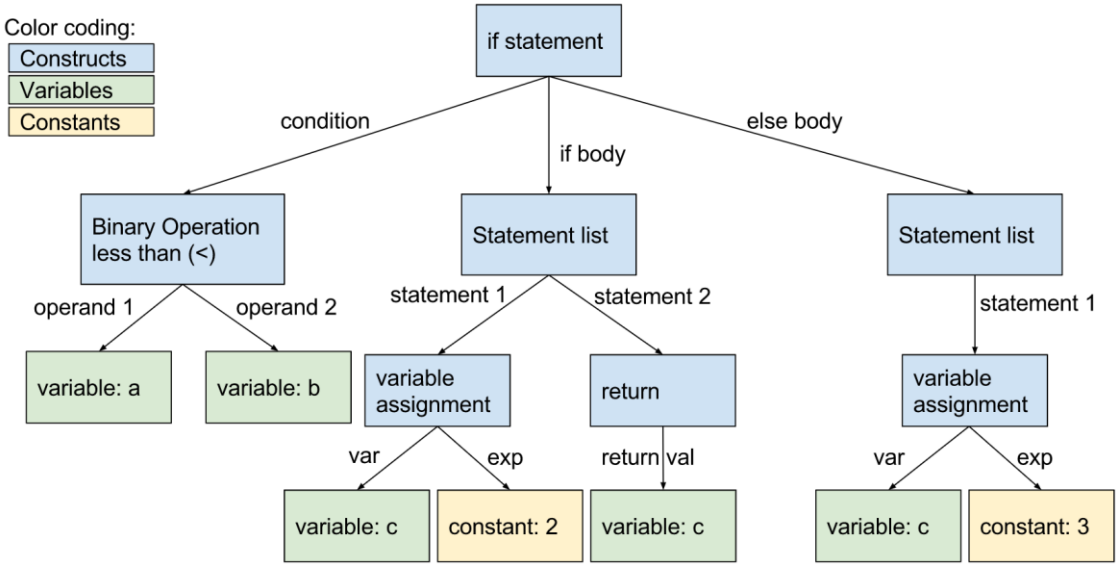
- умова **a < b**
- тіло **if** **c = 2; return c**;
- тіло **else** **c = 3**;

Кожен з цих компонентів, в свою чергу, також може мати нащадків.

Умова – це бінарний оператор "<" з двома "нащадками":

- змінна ***a***
- змінна ***b***

Тіло if може мати довільну кількість дочірніх вузлів. Кожний вираз є новим вузлом. Повне AST для цього коду:



Псевдокод для побудови цього **AST**:

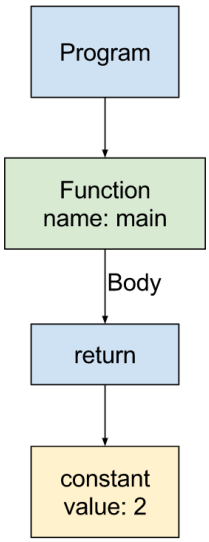
```
//create if condition
cond = BinaryOp(op='>', operand_1=Var(a), operand_2=Var(b))
//create if body
assign = Assignment(var=Var(c), rhs=Const(2))
return = Return(val=Var(c))
if_body = [assign, return]
//create else body
assign_else = Assignment(var=Var(c), rhs=Const(3))
else_body = [assign_else]
//construct if statement
if = If(condition=cond, body=if_body, else=else_body)
```

Псевдокод для процедури для парсингу:

```
Node parseNode() {
    Token current = consume();
    switch (current.lexeme) {
    case "var":
        return parseVariableNode();
    // ...
    }
    panic("unrecognized input!");
}
Node n = parseNode();
if (n != null) {
    // append to some list of top level nodes?
```

```
    // or append to a block of nodes!
}
```

Проте на цьому етапі будуть задіяні лише вузли **program, function declaration, statement, expression**. Її структура має вигляд:




```
program = Program(function_declaration)
function_declaration = Function(string, statement)     //string is
                                                       // the function name
statement = Return(exp)
exp = Constant(int)
```

Теж саме **AST** мовою **C**:

```
typedef enum ExprKind {
      EXPR_NONE,
      EXPR_INT,
      EXPR_CHAR,
      EXPR_FLOAT,
      EXPR_STR,
} ExprKind;

typedef struct Expr {
      ExprKind kind;
      struct Type* type;
      union {
            int64_t int_val;
            int8_t char_val;
            double float_val;
            const char* str_val;
      };
};

typedef enum StmtKind {
      STMT_NONE,
      STMT_RETURN,
```

```
} StmtKind;

struct Stmt {
	StmtKind kind;
	union {
		Expr* expr;
		Decl* decl;
	};
};

typedef enum DeclKind {
	DECL_NONE,
	DECL_FUNC,
} DeclKind;

struct Decl {
	DeclKind kind;
	const char* name;
	union {
		struct {
			Stmt statement;
		} func;

	};
};
```

Хоча на перший погляд здається, що в цьому коді дуже багато зайвих конструкцій, вони знадобляться в подальшому та приведені заради прикладу архітектури.

Зараз програма складається лише з функції **main.** У подальшому на основі цього будуть оброблятися більш складні функції.

Функція має ***назву***, ***тіло*** (може бути також список аргументів). Тіло функції – це лише один вираз, але їх може бути скільки завгодно.

return має лише один дочірній вузол – значення, що буде повернене.

Граматика у формі ***Бекуса-Наура***:

```
<program> ::= <function>
<function> ::= "int" <id> "(" ")" "{" <statement> "}"
<statement> ::= "return" <exp> ";"
<exp> ::= <int> | <float> | <char> | <string>    [добавлены разные типы]
```

Зараз граматика дуже проста. В ній є лише одне правило для кожного нетермінального символу. У майбутньому нетермінали будуть мати багато породжуючих правил.

Аби перетворити список токенів у **AST**, використаний вище описаний рекурсивний низхідний аналізатор (***recursive descent parser***). Оголошується функція для парсингу кожного нетермінального символу в граматиці і повернення

відповідного вузла **AST**. Рекурсивно-нисхідний аналізатор складається з набору рекурсивних функцій/процедур.

Правила граматики, за якими будується AST є рекурсивними (бо у регулярних мовах вкладені структури можуть бути виражені лише рекурсивним чином), отже функції парсингу теж рекурсивні. Звідси і назва рекурсивний нисхідний аналізатор.

Функція для парсингу символу **S** повинна пропускати лексеми з початку списку, доки не досягне правила для **S**. Якщо до завершення парсингу вона виявить токен, якого немає у правилах для **S**, функція повинна аварійно завершитись. Якщо правило **S** містить інші нетермінальні символи, функція повинна викликати інші функції (або саму себе рекурсивно) для їх обробки.

Псевдокод для парсингу виразу:

```
def parse_statement(tokens):
    tok = tokens.next()
    if tok.type != "RETURN_KEYWORD":
        fail()
    tok = tokens.next()
    if tok.type != "INT" | “FLOAT” | “CHAR”
        fail()
    exp = parse_exp(tokens) //parse_exp will pop off more tokens
    statement = Return(exp)
    tok = tokens.next()
    if tok.type != "SEMICOLON":
        fail()
    return statement
```

Під час рекурсивного спуску за деревом, функція парсингу «входить» у якусь гілку, після чого доходить до кінця (термінального символу). На наступному кроці треба повернутися на початок цієї гілки (підйом).

Отже, спускаючись функція парсингу заходить у кожен вузел, а піднімаючись – виходить.

Розглянемо цей процес на основі **Прикладу 1.1**.

```
→ Program (enter)
     → CallExpression (enter)
          → CallExpression (enter)
               → NumberLiteral (enter)
               ← NumberLiteral (exit)
               → NumberLiteral (enter)
               ← NumberLiteral (exit)
          ← CallExpression (exit)
          → CallExpression (enter)
               → CallExpression (enter)
                    → NumberLiteral (enter)
                    ← NumberLiteral (exit)
                    → NumberLiteral (enter)
```

```
                    ← NumberLiteral (exit)
               ← CallExpression (exit)
               → NumberLiteral (enter)
               ← NumberLiteral (exit)
          ← CallExpression (exit)
     ← CallExpression (exit)
← Program (exit)
```

Окрім синтаксичних відмінностей, змінні та функції відрізняються за типами. Ці відмінності є **семантичними**, а відповідні правила мають назву **семантичні правила.** Вони є правилами порівняння між типами операторів та операндів і визначають мову разом з синтаксичними правилами. Процес порівняння типів – **семантичний аналіз**.

При цьому буде застосовуватися наступний підхід:

- на етапі лексичного аналізу повинні оброблятись типи, які визначені за варіантом;
- на етапі синтаксичного аналізу та перевірки типів усі типи, відмінні від **`int`**, повинні бути приведені до нього. Якщо тип не може бути приведений до **`int`**, компіляція аварійно завершується;
- на етапі генерації коду обробляються лише значення типу **`int`**.

Для розрізнення типів вузли **AST** повинні мати відповідні поля, що відповідають за тип елементу. Під час парсингу їх вміст порівнюється з очікуваним типом. Якщо вони однакові, або можуть бути приведені один до одного, відбувається приведення та подальший парсинг.

float – int: відкидається дробова частина

char – int: обробляється ASCII код символу. Інші кодування дозволяється не підтримувати.

str – int: error.

В мові Python заради навчальних цілей типи приводяться так саме, як і в C.

CODE GENERATION

Після побудови **AST**, генерується вихідний асемблерний код. Обходити **AST** треба у зворотному порядку. А саме:

1. Ім’я функції
2. Значення, що повертається (***return value***)
3. ***return statement***

return value треба згенерувати перед посиланням на нього у ***return statement***.

У майбутньому операнди арифметичних виразів також будуть генеруватись перед операцією, що виконується над ними.

Приклад:

Генерація функції **main:**

```
main:
     <function body here>
```

Генерація **return** виразу:

```
mov eax, 2
ret
```

Не бажано записувати згенерований код до файлу рядок за рядком. Задля зменшення кількості доступу до диску та, як наслідок, збільшення швидкості компіляції рекомендується робити запис у тимчасову структуру даних, а потім все одразу заносити до файлу.

Перевірка згенерованого коду відбувається шляхом його виконання та отримання результату. Це можна зробити у середовищі MASM32 [5], за допомогою асемблерної вставки C++ в Visual Studio чи у будь-якому іншому середовищі.

Приклад Visual Studio:

```
#include <iostream>
#include <string>
#include <stdint.h>
using namespace std;
int main()
{
	uint8_t b;
	__asm {
		mov eax, 3
		mov b, eax
	}
	cout << b << endl;
}
```

Для виводу результату створюється додаткова змінна за межами асемблерної вставки.

У випадку використання середовища MASM32 треба написати набагато більше коду. Його можна додати або безпосередньо в генератор коду, або написати вручну. Робляться стандартні налаштування, підключаються необхідні бібліотеки, створюється функція переведення цілого числа будь-якої системи числення у рядок для коректного виведення. Цю функцію можна занести в окремий файл і потім просто підключати її за допомогою директиви **include**. Аналогічно можна по-експериментувати з іншим «допоміжним» кодом для спрощення генерації. На наступних етапах за необхідністю використовувати один і той самий код декілька разів буде влучно використовувати макроси. При цьому слід пам'ятати про необхідність відмінності міток.

Приклад коду у MASM32:

```
.386	; ця директива встановлює набір інструкцій 386. Можна використовувати набір
	;інструкцій 486 чи 586, але 386 є найбільш сумісним набором інструкцій.

.model flat,stdcall		;встановлює модель пам’яті програми. Модель flatпризначена для
				;Windows програм. stdcall встановлює конвенцію виклику функцій -
				;параметри функції заносяться  справа наліво.
```

```
option casemap:none       ;розрізнювати великі та маленькі букви.

; Підключає файли та бібліотеки, необхідні для роботи програми.

include     \masm32\include\windows.inc
include     \masm32\include\kernel32.inc
include     \masm32\include\masm32.inc
includelib  \masm32\lib\kernel32.lib
includelib  \masm32\lib\masm32.lib

NumbToStr   PROTO :DWORD,:DWORD
main        PROTO

.data                     ; оголошення статичних даних (наприклад, глобальних змінних)

buff        db 11 dup(?)  ; найбільше число без знаку, яке може зберігати
                          ; DWORD = 4294967295 ( 10 символів )
                          ; розмір буфера = 10+1, останній символ - NULL terminator

.code

start: ; точка входу у програму
       ; директива INVOKE означає виклик процедури замість call для виклику процедур
       ; з параметрами, які передаються через стек командами push.

    invoke  main
    invoke  NumbToStr, ebx, ADDR buff
    invoke  StdOut,eax
    invoke  ExitProcess,0        ; завершення роботи програми

main PROC
    mov ebx, 11b          ; NumbToStr використовує регістр eax, тому тут задіян ebx.
    ret
main ENDP

NumbToStr PROC uses ebx x:DWORD,buffer:DWORD
    mov     ecx,buffer
    mov     eax,x
    mov     ebx,10              ; base (2 - binary, 8 – octal, 10 – decimal, 16 - hex)
    add     ecx,ebx             ; ecx = buffer + max size of string
@@:
    xor     edx,edx
    div     ebx
    add     edx,48              ; convert the digit to ASCII
    mov     BYTE PTR [ecx],dl   ; store the character in the buffer
    dec     ecx                 ; decrement ecx pointing the buffer
    test    eax,eax             ; check if the quotient is 0
    jnz     @b
    inc     ecx
    mov     eax,ecx             ; eax points the string in the buffer
    ret
NumbToStr ENDP
END start
```

Запуск програми у MASM32 використовується або через командний рядок, або з редактору **qeditor.exe**, який включений у пакет MASM32.

Завдання

1. Ознайомтесь з теоретичними відомостями, додатковою літературою, дайте відповідь на контрольні питання. Це допоможе краще розібратися зі завданням.
2. Розробить лексер, що обробляє вхідний файл з розширенням **.py** чи **.c** в залежності від варіанту та повертає список токенів. Ця функція повинна працювати для **всіх** контрольних прикладів етапу 1, навіть для "**invalid**", але, звичайно, обробляти тільки ті типи та системи числення, що дані за варіантом. Програма повинна розбити заданий текстовий рядок на окремі лексеми та побудувати таблицю лексем. Ця таблиця повинна бути відображена у вигляді пар "лексема – тип лексеми", або мати дещо інший вигляд, при умові, що містить ту же інформацію.
   Зчитування з файлу рекомендується робити у бінарному представленні (функція **open()** з модифікатором **rb** замість **r**)для коректного сприйняття.
   Від'ємне число не може бути токеном, бо мінус – це окрема унарна операція, що може бути використана з додатнім числом.
3. Розробить функцію для парсера, що приймає список токенів та повертає **AST**. За бажанням виводити його на екран. Це допоможе краще розуміти структуру компілятора. При цьому можна використовувати будь-який з існуючих способів представлення **AST** в коді. Це може бути клас, тип даних, структура тощо. Функція повинна будувати **AST** для всіх допустимих прикладів (***valid***) та видавати помилку для неприпустимих (***invalid***). Під час приведення типів на екран виводиться позиція приведення. Якщо тип не може бути приведений до `int`, виводиться помилка. При помилці повинна вказуватись позиція цієї помилки в коді (номер рядка та номер символу).
4. Розробити функцію для генерації асемблерного коду, та записувати згенерований код до файлу.
5. Перевіріти згенерований код шляхом асемблерной вставки або виконананням у середовищі MASM32. Правильний код повинен генеруватись для всіх припустимих (*valid*) прикладів даного етапу та виводити правильний результат.
6. Протестувати програму на декількох своїх контрольних прикладах.

Звіт повинен містити:

- тему, мету, сутність варіанта завдання що необхідно виконати у завданні;
- лістинг програми компілятора;
- власні контрольні приклади, на яких тестувалась програма з порахованими відповідями;
- скріншоти з результатами тестування програми;

До звіту додаються вхідні та вихідні файли, початковий та скомпільований файли програми компілятора.

Варіанти завдання для самостійної роботи:

Варіант	Мова	Система числення значення, що повертається	Типи, що обробляються
1	C	Decimal, Bin	int, float
2	Python	Decimal, Bin	int, str
3	C	Decimal, Hex	int, float
4	Python	Decimal, Hex	int, float
5	C	Decimal, Hex	int, float
6	Python	Decimal, Octal	int, str
7	C	Decimal, Octal	int, float
8	Python	Decimal, Bin	int, str
9	C	Decimal, Bin	int, char
10	Python	Decimal, Bin	int, float
11	C	Decimal, Hex	int, char
12	Python	Decimal, Hex	int, str
13	C	Decimal, Octal	int, char
14	Python	Decimal, Octal	int, float
15	C	Decimal, Bin	int, char

Варіант	Мова	Система числення значення, що повертається	Типи, що обробляються
16	Python	Decimal, Hex	int, str
17	C	Decimal, Bin	int, float
18	Python	Decimal, Bin	int, float
19	C	Decimal, Hex	int, float
20	Python	Decimal, Hex	int, str
21	C	Decimal, Octal	int, float
22	Python	Decimal, Octal	int, str
23	C	Decimal, Bin	int, float
24	Python	Decimal, Hex	int, float
25	C	Decimal, Bin	int, char
26	Python	Decimal, Bin	int, str
27	C	Decimal, Hex	int, float
28	Python	Decimal, Octal	int, str
29	C	Decimal, Octal	int, char
30	***Python***	Decimal, Hex	int, float

Типи, відмінні від **int**, представляти тільки у десятковій системі числення.

Контрольні питання

1. В чому суть і Incremental Approach?
2. На які відмінності мови ***C*** та ***Python*** необхідно звернути увагу при виконанні завдання?
3. Чому не слід використовувати складні регулярні вирази при написанні компілятора?
4. На які етапи поділяється розробка компілятора?
5. Що таке токен?
6. В чому різниця між граматикою та AST?

Етап 2
Унарні та бінарні операції

Мета етапу: розширення функціоналу компілятора за рахунок реалізації обробки унарних та бінарних операцій.

Необхідні теоретичні відомості

На попередньому етапі розглядалися базові принципи роботи компілятора, де оброблялися програми, що повертають цілі числа. На цьому етапі перейдемо до математики. Спочатку розберемось з **унарними операторами**.

Унарний оператор – це оператор, що працює лише з одним операндом і призначений для унарних операцій.

Унарна операція – операція що потребує одного операнду. Наприклад: **-5, !1, ~0b100**.

Унарний мінус (-)

-5 = 0 – 5. Тобто, **унарний мінус** – це операція, що робить зміну знаку числа. Ось чому на першому етапі від'ємні числа не виділялися як окремі токени. Від'ємне число не є лексемою. Це результат операції з унарним мінусом.

Якщо у випадку з десятковим числом для зміни знаку достатньо лише відняти число від нуля, то з двійковим не обов'язково робити саме так. Можна просто інвертувати число і додати одиницю.

Побітове доповнення (~)

Його ще називають **побітовим запереченням**. **Побітовою** називають **операцію**, що виконує дії над кожним бітом числа. **Побітове заперечення** «перегортає» кожен біт числа, внаслідок чого отримуємо від'ємний код.

x	**~x**
0	1
1	0

Наприклад, число 4:

4 = 0b100

~0b100 = 0b011

0b011 = 3

Отже, ~4 = 3.

Логічне заперечення (!)

Логічним запереченням є булева операція «**НІ**». Значення нуля сприймається як ***false***, все інше – ***true***. **!0 = 1, !5 = 0, !1 = 0, !3 = 0** і так далі.

x	**!x**
0	1
будь-яке інше число	0

Окрім унарних операторів існують ще **бінарні**.

Бінарний оператор – оператор, що працює з двома операндами. **Операція** з бінарним оператором називається **бінарною**. Її ще називають **двоелементною**. 2+2, 4/(-5), -8*3.

На цьому етапі будуть розглянуті найпростіші бінарні операції для елементарних математичних дій. Для них знадобиться розібратися в **пріоритетності операторів** та **асоціативності**.

- Додавання
- Віднімання
- Множення
- Ділення

Аби додати всі ці операції до компілятора, треба користуватись результатами минулого етапу, тобто використовувати так званий інкрементний підхід.

Відносно типів все залишається так саме: якщо операнди або результат операції можуть бути представлені типом **int** незважаючи на точність, компіляція відбувається безперешкодно. У противному випадку, виводиться повідомлення про помилку і компіляція завершується. Це обумовлено необхідністю генерувати асемблерний код лише для чисел типу **int**.

Буде простіше, якщо повністю зробити спочатку унарні операції, а потім бінарні. Таким чином виконати роботу в 2 кроки, а не робити лексичний аналіз і для того, і для того, потім так само парсинг і генератор коду.

Також необхідно звертати увагу на хід думок в методичці оскільки це знадобиться в наступних роботах.

LEXING

Спочатку треба розділити вхідний текст програми на лексеми, тож все що треба зробити – лише додати обробку нових операторів.

- **-** – negation
- **~** – bitwise complement
- **!** – logical negation
- **+** – addition
- ***** – multiplication
- **/** – division

Лексема для віднімання відсутня, бо в нас вже є токен унарного мінуса, що також може виступати бінарним мінусом.

Список лексем вище приведений для мови ***C***, тому треба пам’ятати про синтаксичні відмінності ***C*** та ***Python***.

Арифметичні вирази можуть мати дужки, але вони вже є у вигляді токенів з минулого разу.

PARSING

Унарні операції.

У минулій етапі були розглянути **AST** вузли, включаючи **expressions** та **statements**. Сутність цих понять різна тому спочатку розглянемо різницю між **expressions** і **statements**.

Якщо можна вивести який-небудь вираз на екран, присвоїти його якійсь змінній тощо, то це **expression**. Якщо це неможливо, то це **statement**. **Statement** є повноцінним рядком коду, що визначає якусь дію, в той час як **expression** являє собою ділянку коду, що може бути представлена у вигляді якогось значення. **Expressions** можуть поєднуватись «горизонтально» у більші вирази використовуючи оператори. **Statements** можуть поєднуватись «вертикально» у блоки, або просто знаходячись один за одним.

Expression – це будь-яка допустима комбінація констант, змінних, операторів та викликів функції, що може бути представлена значенням якогось типу, наприклад, **int**. Якщо всі елементи **expression** є типу **int**, **expression** теж буде типу **int**.

Statements модифікують змінну, або ж мають якісь інші «побічні» ефекти, під час обрахунку **expression**. Як тільки виконання **statement** завершилось, виконується наступний за порядком **statement** (якщо він є). [26-27].

Раніше було розглянуто лише один тип **expressions** – константи. **Statement** "**return**" міг повертати лише **expression** константу.

Іншим типом **expressions** є унарні операції.

Тепер вузел **AST** що виражає **expressions** буде мати наступний вигляд:

```
program = Program(function_declaration)
function_declaration = Function(string, statement)
statement = Return(exp)
exp = UnOp(unary_operator, exp) | Constant(int)
```

Як можна побачити, тепер **expression** може бути або унарною операцією, або константою. Унарна операція складається з оператора та операнда. Операнд, в свою чергу, теж є **expression**. Отже, визначення **expression** є ***рекурсивним***. **Expression** може включати в себе інші **expressions**.

Наприклад:

```
!3
!!3
!~-4
!!!!!!!-~~-!10
```

Для того, щоб розрізняти яким саме є **expression**, треба оновити формальну граматику.

```
program ::= function
function ::= "int" id "(" ")" "{" statement "}"
statement ::= "return" exp ";"
<exp> ::= <unary_op> <exp> | <int> | <char> | <float>
```

Якщо **expression** розпочинається числом, символом, чи рядком в Python, це - константа, якщо ж унарним оператором, то унарна операція. Але спочатку треба додати **визначення** унарних операторів:

```
<unary_op> ::= "!" | "~" | "-"
```

Тепер парсер знає за яким шляхом треба йти при проході набору лексем. Якщо зустрінеться не число/символ/рядок і не будь-який унарний оператор, то це помилка.

Визначення **expression** рекурсивне, тому і функція парсингу теж повинна бути рекурсивною.

Бінарні операції.

Для них необхідно ще раз змінити вузел **expressions**. Додавання бінарних операцій відбувається за аналогією з унарними, окрім того, що вони працюють не з одним операндом, а з двома (`BinOp(binary_operator, left_exp, right_exp)`). І саме тут виникає різниця між унарним та бінарним мінусом, про яку згадувалось на етапі лексичного аналізу. Вони будуть відноситись до різних вузлів в **AST**.

Приклад вузла AST для унарних та бінарних операцій на мові **C** та парсинг деяких з них:

```
typedef enum ExprKind {
	EXPR_NONE,
	EXPR_INT,
	EXPR_FLOAT,
	EXPR_UNARY,
	EXPR_BINARY,
} ExprKind;

struct Expr {
	ExprKind kind;
	struct Type* type;
	union {
		int64_t int_val;
		double float_val;
		char char_val;
		struct {
			TokenKind op;
			Expr* expr;
		} unary;
		struct {
			TokenKind op;
			Expr* left;
			Expr* right;
		} binary;
	};
};

Expr* parse_expr_unary(void) {
	if (is_unary_op()) {
```

```
            TokenKind op = token.kind;
            next_token();
            return expr_unary(op, parse_expr_unary());
      }
      else {
            return parse_expr_base();
      }
}

Expr* parse_expr_mul(void) {
      Expr* expr = parse_expr_unary();
      while (is_mul_op()) {
            TokenKind op = token.kind;
            next_token();
            expr = expr_binary(op, expr, parse_expr_unary());
      }
      return expr;
}
```

Приклад: AST для виразу `2 - (-3)`

```
two = Const(2)
three = Const(3)
neg_three = UnOp(NEG, three)
exp = BinOp(NEG, two, neg_three)
```

Тепер змінимо визначення **expressions** в граматиці:

```
<exp> ::= <exp> <binary_op> <exp> | <unary_op> <exp> | "(" <exp> ")" | <int>
```

Додали бінарні операції та **expression**, обернений в дужки, оскільки бінарна, унарна операція, або просто число можуть бути в дужках.

Але в цій граматиці є декілька проблем, які доведеться послідовно вирішувати:

- **Не враховано пріоритетність операторів**. Пріоритетність (або черговість) операцій – правило для однозначного трактування порядку виконання операцій в математичному виразі. Визначається порядок, в якому виконуються оператори. Множення виконується раніше додавання, ділення після операції в дужках і таке інше.

  Що стосується пріоритетності, мови **C** та **Python** значно відрізняються. Операція логічного заперечення в **Python** відрізняється від **C** не тільки синтаксично («**not**» проти «**!**»), а й нижчим пріоритетом.

  Тож граматика мови **Python** дещо відрізняється від **C**. Також нижче приведені таблиці пріоритетності операторів.

Пріорітетність виконання операторів у мові С

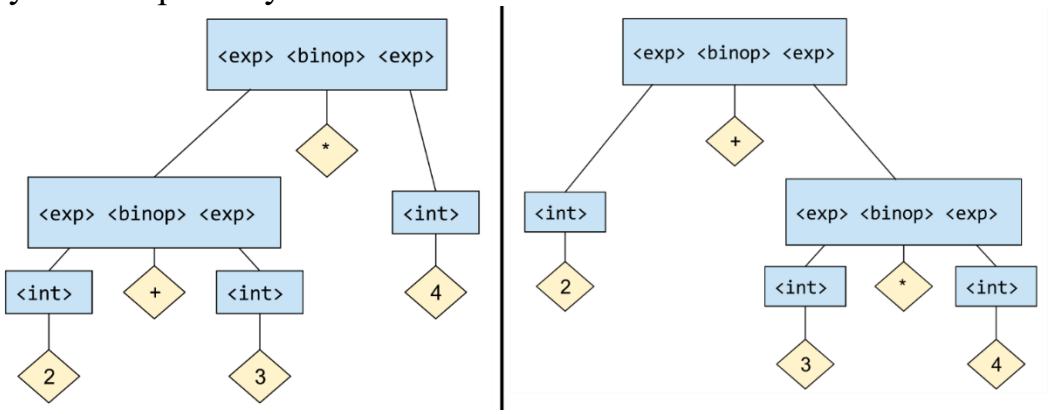
Пріор	Оператор	Сутність	Порядок виконання
1	**( )** **[ ]** **.** **- >** **++ - -**	Круглі дужки або звернення до функції Дужки або індекс масиву Крапка або оператор вибору складової Вибір елементу через покажчик Операторі постфіксного збільшення /зменшення	зліва направо
2	**++ - -** **+ -** **! ~** **(type)** ***** **&** **sizeof**	Операторі префіксного збільшення/зменшення Одномісний плюс і мінус Логічне та побітове "не" Перетворення типів Оператор пере направлення або розіменування Оператор отримання адреси Операторa визначення розміру в байтах	справа наліво
3	*** / %**	Операторі множення, ділення та остачі	зліва направо
4	**+ -**	Операторі додавання та віднімання	зліва направо
5	**<< >>**	Операторі побітового зсуву вліво та вправо	зліва направо
6	**< < =** **> > =**	Операторі "менше", "менше або дорівнює" Операторі "більше", "більше або дорівнює"	зліва направо
7	**= = ! =**	Операторі "дорівнює", "не дорівнює"	зліва направо
8	**&**	Оператор "побітове І"	зліва направо
9	^	Оператор "побітове виключне АБО"	зліва направо
10	|	Оператор "побітове АБО"	зліва направо
11	&&	Оператор "логічне І"	зліва направо
12	||	Оператор "логічне АБО"	зліва направо
13	**? :**	Термальний оператор	справа наліво
14	**=** **+ = - =** ***= /=** **%= &=** **^= |=** **<<= >>=**	Оператор присвоєння Оператор присвоєння суми та різниці Оператор присвоєння добутку та частки Оператор присвоєння остачі та побітового "І" Оператор присвоєння виключного "АБО" та побітового "АБО" Оператор присвоєння побітового зсуву вліво та вправо	справа наліво
15	**,**	Оператор "кома"	зліва направо

Пріорітетність виконання операторів у мові Python

Пріор	Оператор	Сутність
1	**(expressions...),** **[expressions...],** **{key: value...},** **{expressions...}**	Вираз у дужках Відображення списку Відображення словника Відображення множини
2	**x[index]** **x[index:index]** **x(arguments...)** **x.attribute**	Доступ до елемента Зріз Виклик Доступ до атрибуту
3	**await x**	Вираз await
4	******	Піднесення до ступеня
5	**+x**	Одномісний плюс

	-x **~x**	Одномісний мінус Побітове "не"
6	***** **@** **/** **//** **%**	Множення Матричне множення Ділення Ділення з округленням до низу Залишок
7	**+** **-**	Додавання Віднімання
8	**<<** **>>**	Зсув вліво Зсув вправо
9	**&**	Побітове І
10	^	Побітове виключне АБО
11	|	Побітове АБО
12	**in** **not in** **is** **is not** **<** **<=** **>** **>=** **!=** **==**	Порівняння, перевірка входження, перевірка ідентичності
13	**not**	Логічне НЕ
14	**and**	Логічне І
15	**or**	Логічне АБО
16	**if – else**	Умовні вирази
17	**lambda**	lambda-вираз

Розглянемо вираз **`2+3*4`**. Згідно з наведеною граматикою, його можна інтерпретувати по-різному:



Згідно з деревом ліворуч **`(2+3)*4 = 24`**, а згідно з деревом праворуч – **`2+(3*4) = 14`**. Оскільки насправді у первісному виразі дужок немає, то множення виконується перед додаванням, тож праве дерево вірне. Але граматикою це не визначено, тож прийдеться це врахувати.

- **Не враховано асоціативність**. Асоціативність – порядок, в якому обробляються оператори з однаковим пріоритетом. **`1-2-3`** повинно рахуватись як **`(1-2)-3`**. Але з нашою граматикою вийде **`1-(2-3)`**.
- **Ця граматика ліво-рекурсивна**, оскільки містить правіло **`<exp> ::= <exp> <binary_op> <exp>,`** що може привести до нескінченного циклу. Низхідний рекурсивний парсер не може працювати з лівою рекурсією, бо прохід правила відбувається зліва направо і парсер назавжди застрягне на першому ж нетерміналі.

Якщо в граматиці є нетермінал **A**, для якого **A ::= Aa**, де **a** - непустий ланцюжок, граматика має **ліву рекурсію**. Тобто з нетермінала **A** породжується якесь правило і це правило має нетермінал **A** на першому місці. Також існує **права** (**A ::= aA**) та **серединна** (**A ::= aBAc**) рекурсія.

Відсутність лівої рекурсії також необхідна умова приналежності граматики до виду **LL(1)** граматик, які будуть згодом розглядатися.

LL(1) граматика – це така граматика, при якій аналіз відбувається згори до низу та зліва направо, перевіряючи лише один символ попереду. Цей "направляючий" символ підказує компілятору, яке саме наступне правило граматики треба застосовувати.

Почнемо виправляти з ***пріоритетності***. Унарний оператор повинен застосовуватися до всього виразу лише в тому випадку, якщо:

- вираз є лише одним числом: **`~4`**;
- вираз обернений в дужки: **`~(1+1)`**;
- вираз сам по собі є унарною операцією: **`~!8, -~(2+2)`**.

Аби це реалізувати, нам знадобиться додати нове правило в граматиці, яке буде посилатись на вираз, до якого можна застосувати унарний оператор. Назвемо його **`factor`**. Тепер граматика має вигляд:

```
<exp> ::= <exp> <binary_op> <exp> | <factor>
<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int>
```

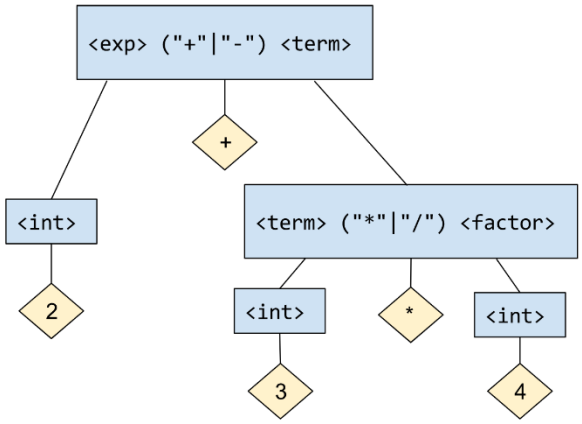
Тепер існує 2 рівня пріоритету: один для бінарних операцій, інший (більш високий) для унарних. Так само робимо і з множенням та діленням, які вище за пріоритетом, ніж додавання та віднімання.

Додамо правило для **`term`**, що представляє ці операнди:

```
<exp> ::= <exp> ("+" | "-") <exp> | <term>
<term> ::= <term> ("*" | "/") <term> | <factor>
<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int>
```

Як можна побачити, зникло правило **`<binary_op>.`** А точніше, розклалося на дві частини: одна частина – це правило **`<term>`** для множення та ділення, а друга частина – термінали додавання та віднімання.

Ця граматика вже не буде мати проблем з пріоритетністю при аналізі виразу: **2+3*4**. Дерево розбору для нього має вигляд:



На черзі ***асоціативність*** та ***ліва рекурсія***. Почнемо здалеку. Це необхідно аби ви зрозуміли хід думок та використовували ці принципи в наступних завданнях.

Якщо б ми НЕ використовували парсер з рекурсивним спуском, могли б обійтись ліво-рекурсивними правилами для ліво-асоціативних операцій і право-рекурсивними правилами для право-асоціативних. В такому випадку можна переписати правило для **expressions** наступним чином:

```
<exp> ::= <exp> ("+" | "-") <term> | <term>
```

Тепер додавання та віднімання ліво-асоціативні: не можна розпарсити **1-2-3** як **1-(2-3)**, бо **2-3** не є **term**.

Але ми використовуємо парсер з рекурсивним спуском, тому не можемо використовувати ліво-рекурсивне правило. Аби зрозуміти чому, давайте спробуємо розпарсити вираз по цьому правилу.

```
def parse_expression(tokens):
    //determine which of two production rules applies:
    //  * <exp> ("+" | "-") <term>
    //  * <term>
    if is_term(tokens): //how do we figure this out???
        return parse_term(tokens)
    else:
        //recursively call parse_expression to handle it
        e1 = parse_expression(tokens) //recurse forever
```

Граматика з лівою рекурсією не дає вибрати наступне правило (а також врахувати асоціативність) без заглядання наперед. Аби з'ясувати яке правило використовувати, недостатньо подивитись на першу лексему – необхідно знати, чи є у виразі **+** або **–**. Тобто це не **LL(1)** граматика у якої для вибору правіла достатньо аналізу лише **першої** лексеми.

Якщо ж все таки з'ясувалось, що цей вираз сума чи різниця, функція застрягне у вічному циклі через ліву рекурсію.

Спробуємо уникнути лівої рекурсії, переставивши **<term>** та **<exp>:**

```
<exp> ::= <term> ("+" | "-") <exp> | <term>
```

Ліва рекурсія зникла, але знову сплила проблема правої асоціативності. **1-2-3** тепер **1-(2-3).**

Необхідно вибрати між непрацездатною ліво-рекурсивною граматикою та неправильною право-асоціативною. На щастя є третій варіант: ввести в граматику

«повтори». В **розширеній формі Бекуса-Наура** [36] обернення у фігурні дужки **{** та **}** говорить про те, що вміст цих дужок може повторюватись декілька разів або бути відсутнім взагалі.

```
<exp> ::= <term> ("+" | "-") <term>
```

В цьому виразі після **<term>** обов'язково має йти **+** або **–**. Якщо парсер не зустріне **+** чи **–**, то видасть помилку.

```
<exp> ::= <term> { ("+" | "-") <term> }
```

В цьому випадку після **<term>** може нічого не бути, або бути **("+" | "-") <term>,** або

```
<exp> ::= <term> ("+" | "-") <term> ("+" | "-") <term> ("+" | "-") <term> …
```

Отже, наша фінальна граматика для обробки **expressions**:

```
<exp> ::= <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/") <factor> }
<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int>
```

Вона правильно обробляє пріоритетність операцій, не ліво-рекурсивна та не право-асоціативна.

Псевдокод для її реалізації:

```
def parse_expression(toks):
    term = parse_term(toks) //pops off some tokens
    next = toks.peek() //check the next token, but don't pop it off the list yet
    while next == PLUS or next == MINUS: //there's another term!
        op = convert_to_op(toks.next())
        next_term = parse_term(toks) //pops off some more tokens
        term = BinOp(op, term, next_term)
        next = toks.peek()
    return t1
```

Такий самий принцип парсингу можна використати в **parse_term**.

```
def parse_factor(toks)
    next = toks.next()
    if next == OPEN_PAREN:
        //<factor> ::= "(" <exp> ")"
        exp = parse_exp(toks) //parse expression inside parens
        if toks.next() != CLOSE_PAREN: //make sure parens are balanced
            fail()
        return exp
    else if is_unop(next)
        //<factor> ::= <unary_op> <factor>
        op = convert_to_op(next)
        factor = parse_factor(toks)
        return UnOp(op, factor)
    else if next.type == "INT":
        //<factor> ::= <int>
        return Const(convert_to_int(next))
    else:
        fail()
```

Ще раз треба наголосити, що граматика мови **C** та **Python** відрізняється через пріоритетність оператора логічного заперечення.

Обробка типів відбувається згідно варіанту. Якщо варіантом передбачена обробка типу, або тип вже оброблявся у минулих завданнях, він обробляється і зараз. Усі значення, відмінні від типу **int**, приводяться до нього (навіть із втратою точності). У разі неможливості приведення, компіляція завершується з помилкою. У разі успішного приведення вводиться повідомлення про це.

Перевірка семантики проходить одразу з перевіркою синтаксису під час етапу парсингу.

CODE GENERATION

Унарний мінус і побітове доповнення реалізуються дуже просто. На них достатньо однієї інструкції. А от логічне заперечення буде по-складніше. Але давайте по черзі.

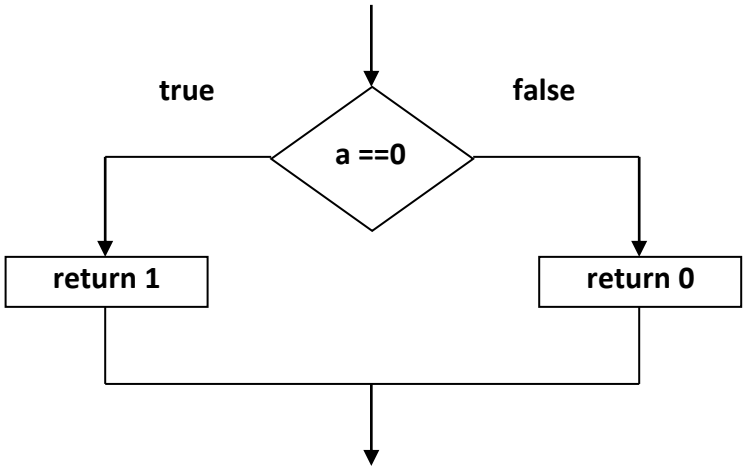
Припустимо що в регістрі **EAX** лежить значення **5**. Тоді, аби отримати число **-5**, необхідно виконати наступну операцію:

```
neg   eax
```

Рахується доповняльний код числа, що дорівнює від'ємному значенню. Слід пам'ятати, що AST проходиться у зворотному порядку **Parent → Child** [23]. Дочірній вузол що представляє операнди обробляється першим, а вже потім виконується сама операція. Спочатку необхідно порахувати значення **EAX**, а потім виконати інструкцію **neg**. Тобто, якщо в вас вираз **`-!5`**, то першим кроком буде виконання логічного заперечення, а лише потім унарного мінуса: **`-!5 = -(!5).`**

Побітове доповнення реалізується так само, як і унарний мінус, тільки командою **not**. Виконується побітова інверсія – нульові біти замінюються на 1, а одиничні стають 0.

Рядок коду **return !a** можна представити у вигляді алгоритму



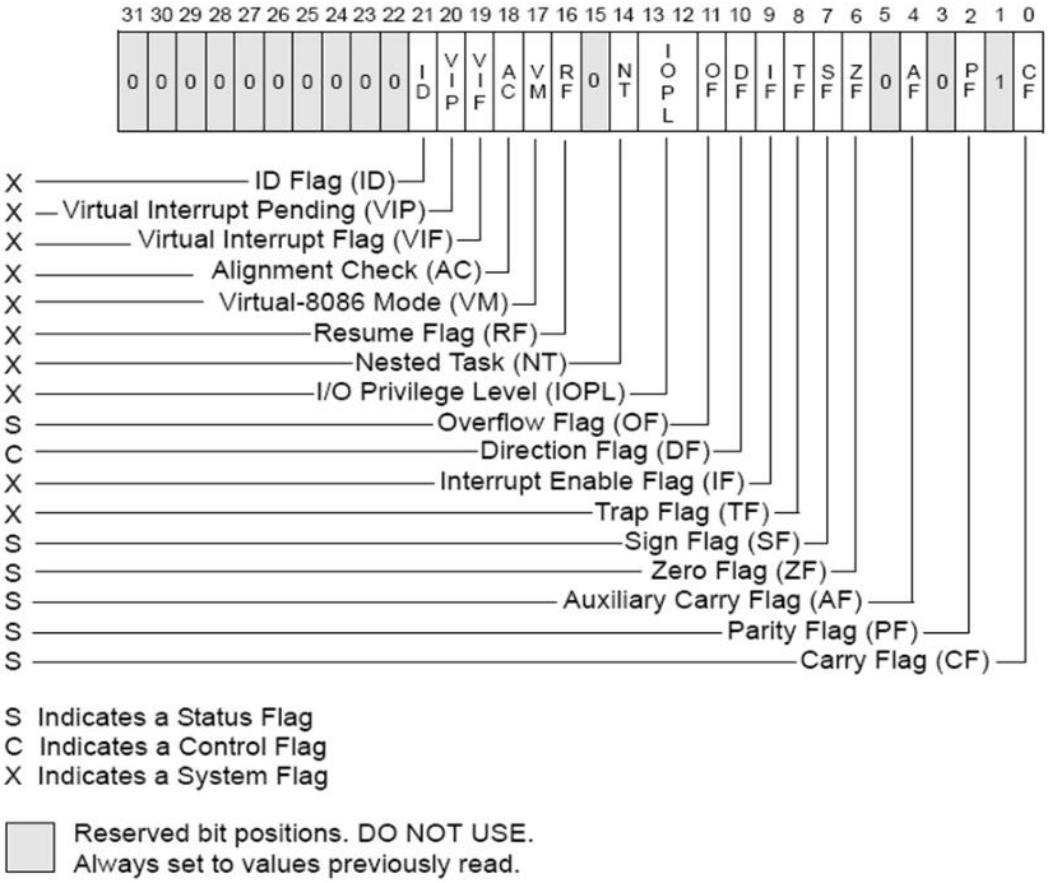

```
if (a == 0) {
    return 1;
} else {
```

```
    return 0;
}
```

Неважко здогадатись, що однією інструкцією тут не обійдешся. Аби реалізувати умовну логіку, знадобляться інструкції **cmp** та **sete**.

cmp порівнює два значення. Але результат порівняння зберігається неявно. **sete** (***set if equal***) встановлює свій операнд в **1** якщо результат останнього порівняння був «рівно», в іншому випадку **0**.

Обидві ці інструкції (як і всі умовні та порівняльні інструкції) неявно посилаються на **EFLAGS регістр**. (**FLAGS** на 16-бітних машинах, **EFLAGS** на 32-бітних і **RFLAGS** на 64-бітних. Проте прапорці, які нам потрібні, знаходяться в молодших 16 бітах, тож це не принципово). Які біти за що відповідають можна подивитись у [28].



EFLAGS регістр зберігає набір бітів, що використовуються процесором для виявлення результатів логічних та арифметичних операцій, для відображення статусу та стану процесора.

Нас цікавить **zero flag (ZF)**, який стає **1**, якщо результат операції **0** і навпаки.

```
    cmp a, b
```

рахує **(b - a)**. Якщо **a = b**, то **b – a = 0**, тому **ZF** встановлюється в **1**.

```
    sete
```

використовує **ZF** для перевірки рівності. **sete** встановлює свій операнд в **1**, якщо **ZF = 1** і в **0**, якщо **ZF = 0**. **setz** – це інша мнемоніка для цієї ж інструкції.

Ще один момент стосовно **sete**: вона працює лише з байтом, не з цілим словом, а отже з регістром **AL**, а не з **EAX**. **AL** – це молодший байт **EAX**.

Спочатку треба обнулити **EAX**, бо старші розряди можуть мати якесь значення та будуть заважати. Хоча інструкція для обнулення

```
mov eax, 0
```

найбільш очевидна, вона працює досить повільно. Тому краще використовувати [31]

```
xor eax, eax
```

Логічне заперечення може бути реалізоване наступним чином:

- порівняти значення **EAX** з нулем
- обнулити **EAX**
- якщо **ZF = 1**, то встановити **AL** в **1**,

Приклад !5:

- записуємо **5** в **EAX**
- порівнюємо **0** і **5**. **5 != 0**, отже **ZF = 0**
- обнуляємо **EAX**.
- оскільки **ZF = 0**, то **AL = 0**.

Оскільки значення константи 5 у регістрі **EAX** через його обнуління не зберігається, то для використання у подальших розрахунках треба попередньо його скопіювати у інші регістри.

Для обробки бінарного виразу **e1 + e2** необхідно:

- порахувати **e1** та зберегти результат
- порахувати **e2**
- додати **e1** та **e2** і зберегти результат в **EAX**

Отже, необхідно зберегти перший операнд. Можна для цього використати регістр, але це не зовсім раціонально, бо таких регістрів може не вистачити на усі проміжні результати. Тому краще використовувати стек.

Реальні компілятори зазвичай зберігають проміжні результати в регістрі, бо це значно швидше ніж стек.

Для цього необхідно відстежувати кількість вільних регістрів, яких не так вже й багато, та стан кожного регістру аби знати, чи можна його використовувати, чи ні. Також необхідно слідкувати, аби обчислення не зіпсували якісь інші дані в регістрах.

Розглянемо, яким чином можна більш раціонально використовувати наявні регістри. Наприклад, маємо вираз:

$$y = (a + b) * (c * (d + e)) * (f + g * h) * (j + k) * (o * p) + (q * r)$$

1. **a + b -> EAX**
2. **d + e -> EBX**
3. **c * (d + e) = c * EBX -> EBX**
4. **g * h -> ECX**
5. **f + g * h = f + ECX -> ECX**
6. **j + k -> EDX**.

Все, вільні регістри закінчилися. Звичайно, регістрами загального призначення також вважаються **ESP**, **EBP**, **ESI**, **EDI**, але використовувати їх не рекомендується, бо в них інше призначення.

7. Необхідно проаналізувати, чи можна якимось чином звільнити хоча б два регістри.
   (a + b) * (c * (d + e)) * (f + g * h) * (j + k) = EAX * EBX * ECX * EDX.
   ECX * EDX -> ECX
   EAX * EBX -> EAX
   EAX * ECX -> EAX
   Звільнено три регістри.
8. **o * p -> EBX**
9. **EAX * EBX -> EAX**
10. **q * r -> EBX**
11. **y = EAX + EBX -> EAX**
12. **EAX -> stack**.

Звичайно, при більш складному виразі, можливо, знадобиться більш детальний аналіз та більш пильніше відслідковування регістрів. Для відстеження «життєздатності» змінних можна використовувати метод розфарбування графів, або сканувати ліінйно.

Процес та стек.

В моделі, що зазначена нижче, програміст пише код на мові високого рівня. Компілятор перетворює цей код в набір машинних команд і зберігає їх в **executable image** файл. Компілятор також визначає статичні дані, такі як початкові значення і включає їх в ***executable image***.

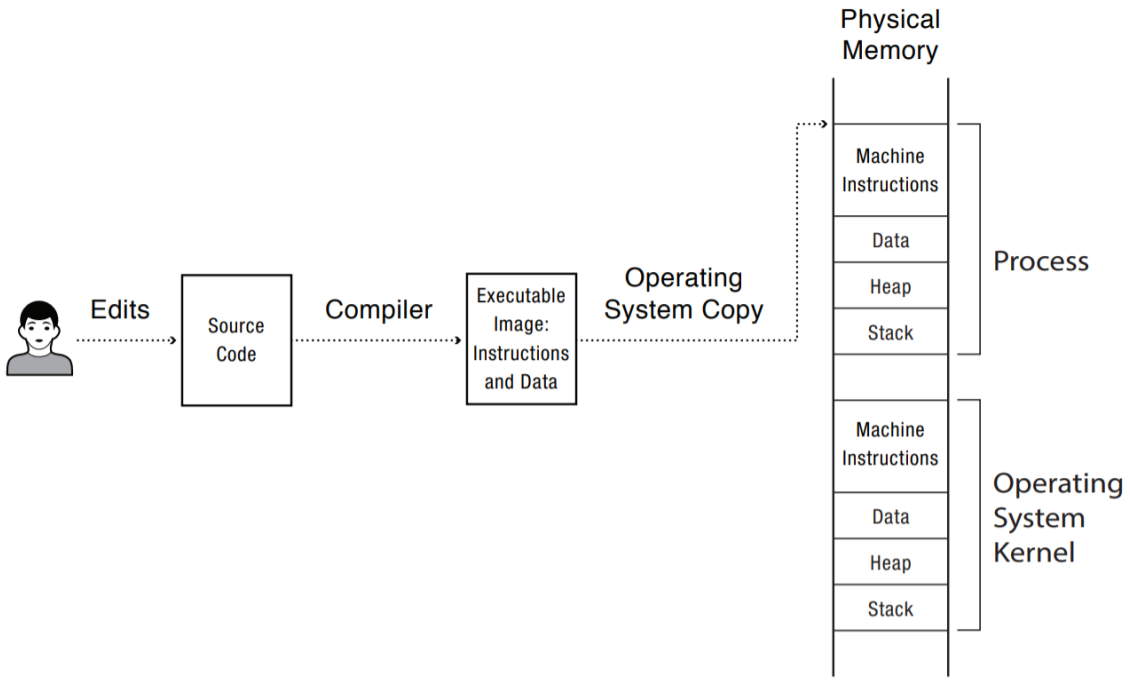
Аби запустити програму, операційна система копіює інструкції і дані з ***executable image*** у фізичну пам’ять. Виділяється область пам’яті **execution stack** для зберігання локальних змінних протягом виконання функцій. Також виділяється **heap** для динамічно виділених структур даних.

Попередньо операційна система сама повинна бути завантажена в пам’ять, зі своїми власними стеком і хіпом.

Як тільки програма завантажена в пам’ять, ОС може запустити її, встановивши вказівник стеку на першу інструкцію програми і почавши виконання з неї. Компілятор сам по собі є програмою і він також проходить всі ці етапи.

Отже, **процес** – це «екземпляр» програми, так само як об’єкт є екземпляром класу в ООП. Кожна програма може мати 0, 1, 2, … скільки завгодно процесів, що її одночасно виконують. Таким чином, для кожного екземпляру програми існує відповідний процес зі своє копією програми в пам’яті.

Треба розрізняти поняття **потік** та **процес**. Процес може складатись з декількох потоків. Більше дізнатись про процеси, потоки, паралельне програмування та як взагалі працює ОС можна у [29].



Як було зазначено, кожний процес займає деяку пам'ять. Ця пам'ять ділиться на декілька сегментів, один з яких стек.

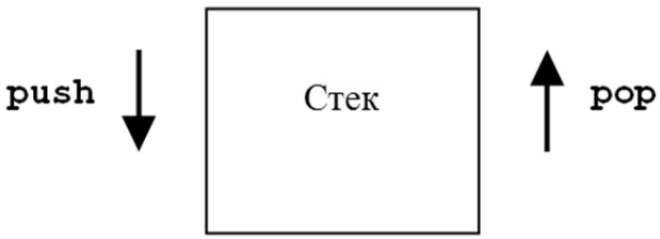
Адреса вершини стеку зберігається в регістрі **ESP** (stack pointer).

Інструкція **push** – покласти значення в стек, а **pop** – дістати значення зі стеку.

Нижче проілюстрована сутність команд **push** і **pop** для кращого розуміння їх роботи, де **ss** – селектор сегменту, в якому розміщено стек:

```
; -- push EAX ---
sub esp, 4
mov ss:[esp], eax

; -- pop EAX ----
mov eax, ss:[esp]
add esp, 4
```



Особливістю є напрямок, в якому росте стек. Він збільшується в сторону менших або нижніх адрес пам'яті на величину що дорівнює розміру типу даних в байтах. Оскільки в навчальних цілях компілятор здатен генерувати код лише для значень типу **int**, стек зменшується на чотири. При виконанні команди **push** зміст регістру **ESP** зменшується, а при виконанні **pop** - збільшується.

Ассемблерний код для **e1 + e2** :

```
<CODE FOR e1 GOES HERE>
push eax                ; save value of e1 on the stack
<CODE FOR e2 GOES HERE>
pop ecx                 ; pop e1 from the stack into ecx
add eax, ecx            ; add e1 to e2, save results in eax
```

Множення виконується так само, тільки використовується команда множення зі знаком **imul**.

У відніманні порядок операндів грає роль. **sub dst, src** рахує **dst – src** і зберігає результат в **dst**. Тому спочатку необхідно забезпечити, щоб **e1** знаходилося в **dst**, а **e2** в **src**.

Ділення найскладніше, оскільки **idiv dst** сприймає **EDX** та **EAX** як єдиний 64-бітний регістр і рахує **[EDX:EAX] / dst**. Потім зберігає частку в **EAX**, остачу в **EDX**. Тому спочатку необхідно помістити **e1** в **EAX** і розширити його знак (**sign-extend**) в **EDX** користуючись командою **cdq**. Вже після цього використовується **idiv**.

Для випадків ділення чи множення на число кратне ступені **2** процес можна оптимізувати. Ділення на **2** представляє собою зсув числа вправо на **1** розряд, ділення на **4** – на **2** розряди, ділення на **2^n** – на **n** розрядів. При множенні зсувати необхідно вліво. Для арифметичного зсуву використовуються команди

```
SAR <register>, <number of bits>
```

та

```
SAL <register>, <number of bits>.
```

Більш детальну інформацію по цим та іншим командам мови **Assembler** можна отримати з ***Intel Software Developer's Manual*** [30].

Завдання

1. Ознайомтесь з теоретичними відомостями, додатковою літературою, дайте відповідь на контрольні питання. Це допоможе краще розібратися в завданні.
2. Удоскональте свою функцію лексера з минулого етапу, щоб вона могла обробляти токени для унарного мінуса, або для побітового доповнення, або для логічного заперечення та для бінарних операцій (згідно варіанту).
3. Оновить функцію парсера для обробки унарної та бінарної операції, яку треба реалізувати за варіантом.
4. Додайте в генерацію асемблерного коду відповідні команди. Результатом унарних операцій повинно бути або числове значення з протилежним знаком, або число складене з біт що обернені до початкових, або результат булевої операції, тобто **1** або **0**. Розрахунки робляться для значень в тій системі числення, що була використана у попередньому етапі.
5. Протестуйте програму на своїх контрольних прикладах, та внесіть їх результати у звіт. Програма повинна працювати для всіх коректних прикладів та видавати помилку для некоректних, що містять помилку. Обробка помилок повинна вказувати місце в коді де сталась помилка.

Звіт повинен містити:

- тему, мету, сутність варіанта завдання що необхідно виконати у завданні;

- лістинг програми компілятора;
- власні контрольні приклади, на яких тестувалась програма з порахованими відповідями;
- скріншоти з результатами тестування програми;

До звіту додаються вхідні та вихідні файли, початковий та скомпільований файли програми компілятора.

Варіанти завдання для самостійної роботи:

Варіант	Унарна операція	Бінарна операція
1	-	/
2	-	/
3	-	/
4	-	/
5	-	/
6	-	/
7	-	/
8	-	/
9	~	-
10	~	-
11	~	-
12	~	-
13	~	+
14	~	+
15	~	+

Варіант	Унарна операція	Бінарна операція
16	~	+
17	!	+
18	!	+
19	!	+
20	!	+
21	!	-
22	!	-
23	!	-
24	!	-
25	-	*
26	~	*
27	!	*
28	!	*
29	-	*
30	~	*

Контрольні питання

1. В чому суть Incremental Approach і як він використовується в цій роботі?
2. Чому використовується тільки одна лексема «-», хоча є 2 різні операції з цією лексемою?
3. В чому різниця між expressions та statements?
4. Як саме парсер визначає наступну операцію?
5. Що таке ліво-рекурсивна граматика і в чому її недоліки?
6. Як вирішується проблема пріоритетності операцій та асоціативності?
7. Опишіть концепцію процесу.
8. Призначення регістру ESP та особливості роботи з ним.
9. Як виконується операція ділення на мові асемблера?

Етап 3
Обробка змінних. Локальні змінні

Мета етапу: побудова компілятору з використанням локальних змінних.

Необхідні теоретичні відомості

На цьому етапі нарешті буде додана обробка локальних змінних. Аби не ускладнювати, введемо деякі обмеження:

- розглядаємо лише локальні змінні;
- не використовуємо жодних модифікаторів, таких як **short**, **long**, **unsigned**, **static**, **const**.
- розглядаємо лише одну змінну на вираз. Поки що ніяких **int a, b**;

Що можна буде робити зі змінною:

- оголошення: **float a**;
- визначення: **a = 3.14**;
- ініціалізація при оголошенні: **float a = 2**;
- посилання на змінну у виразі: **a + 2**.

Також введемо деякі нові бінарні операції:

булеві (логічне І, логічне АБО, …),
побітові (**&**, |, …),
оператори порівняння (<, ==, …)

та ще декілька.

Оскільки вже розглядалися обробка бінарних операцій, тож додати нові не буде проблемою.

LEXING

Лише одна нова лексема знадобиться для визначення змінної. Це «`=`».

Слід зауважити, що існує суттєва різниця між оголошенням змінної в мові С та мові **Python**.

Мова **C** має статичну типізацію. Це означає, що змінна пов'язується з типом в момент оголошення і цей тип не може бути змінений пізніше, лише приведенням типів. Тип змінної (а також тип значення, що повертає функція) необхідно вказувати явно: `int a`. Будь-яка програма, в якій типи порушують правила мови, вважається некоректною. Наприклад, `int a = 2 + 3.14;` - це помилка. «`2`» є цілим числом, «`3.14`» - числом з плаваючою комою. Отже, після додавання цих двох чисел ми отримаємо дробове число, що суперечить типу змінної «`a`».

У мови **Python** динамічна типізація. Змінна пов'язується з типом у момент присвоєння значення, а не під час її оголошення. Таким чином, на різних ділянках програми одна й та сама змінна може приймати значення різних типів.

Оголошувати змінну окремо не потрібно. Це робиться під час її визначення: `a = 3`. Вираз `a = 2 + 3.14` не дасть помилки. У змінну «a» буде занесене дробове число.

Нові бінарні операції:

- Логічне І – `&&`
- Логічне АБО – `||`
- Дорівнює – `==`
- Не дорівнює – `!=`
- Менше – `<`
- Менше або дорівнює – `<=`
- Більше – `>`
- Більше або дорівнює – `>=`
- Остача від ділення – `%`
- Побітове І – `&`
- Побітове АБО – `|`
- Побітове виключне АБО (XOR) – `^`
- Побітовий зсув вліво – `<<`
- Побітовий зсув вправо – `>>`

Ще раз треба нагадати, що треба враховувати синтаксичні відмінності між **C** та **Python**.

PARSING

Ось приклад програми на мові **C**, яку можна буде скомпілювати наприкінці:

```
int main() {    // тип функції залишається int в незалежності від варіанту
    float b;          // оголошення змінної
    float a = 1;      // оголошення змінної та її ініціалізація
    a = a + 1;        // посилання на змінну
    return a;
}
```

Приклад на мові **Python**:

```
def main():
     a = 1      # оголошення змінної «a»; оголошення змінної “b” відсутнє,
                # бо в мові Python немає оголошень як таких.
     a = a + 1  # посилання на змінну
     return a
```

Зараз AST має вигляд:

```
program = Program(function_declaration)
function_declaration = Function(string, statement)
statement = Return(exp)
exp = UnOp(unary_operator, exp) | BinOp(binary_operator, left_exp, right_exp)
                                | Constant(int, float, char)
```

Необхідно оновити **function_declaration** в **AST** аби функція могла мати в собі декілька виразів, а не тільки один.

```
function_declaration = Function(string, statement list) //string is function name
```

Додамо ще декілька окрім ***return***:

```
statement_list = Return(exp)
     | Declare(type, string, exp option)   // type is variable type
                                           //string is variable name
                                           //exp is optional initializer
     | Exp(exp)                            // e.g.: 2+2
```

exp option – змінну можна одразу ініціалізувати, якщо необхідно. У варіанті з мовою **Python** є обов'язковим, а поле **type** має бути відсутнім.

AST для **int a**:

```
decl = Declare("int", "a", None) //None because we don't initialize it
```

AST для **int a = 3**:

```
init_exp = Const(3)
decl = Declare("int", "a", init_exp)
```

Тепер необхідно оновити **exp** для обробки оператора присвоювання. Здається, що «=» є черговим бінарним оператором: **a = b** так схоже на **a + b**. Проте це абсолютно неправильно: ліва і права частини бінарного оператору можуть бути будь-якими виразами, а от ліва частина оператора присвоювання – ні. Вираз **2 = 2** або **3 = 6** не має жодного сенсу. Замість цього визначимо оператор присвоювання «=» як новий тип **expression**:

```
exp = Assign(string, exp)  //string is variable, exp is value to assign
    | BinOp(binary_operator, exp, exp)
    | UnOp(unary_operator, exp)
    | Constant(int, float, char)
```

AST для **a = 2**:

```
assign_exp = Assign("a", Const(2))
assign_statement = Exp(assign_exp)
```

Зараз ми можемо оголошувати змінні та змінювати їх значення, але не можемо посилатися на ці змінні, тобто їх використовувати. Необхідно додати ще один тип **expression**:

```
Var(string) //string is variable name
```

AST для "**return a;**" має вигляд:

```
return_exp = Var("a")
return_statement = Return(return_exp)
```

Також необхідно оновити граматику для можливості створити декілька виразів в тілі функції.

```
<function> ::= "int" <id> "(" ")" "{" { <statement> } "}"
```

Дужки «**{ }**» позначають повторення. <**statement**> у тілі функціїї може повторюватись безліч разів, або бути відсутнім взагалі.

Оголошення змінної складається з типу змінної (в мові **C**), її ім’я, та (опціонально) значення. Необов’язкові елементи позначаються квадратними дужками.

```
<statement> ::= "return" <exp> ";"
              | <exp> ";"
              | <type> <id> [ = <exp> ] ";"
```

У варіанті з мовою **Python** поле **`type`** має бути відсутнім, але зате присвоювання обов’язкове.

Присвоювання є оператором з найнижчим пріоритетом на даний момент, тож він буде найпершим в **<exp> expression**. Також він право-асоціативний, тож його буде легше представити.

```
<exp> ::= <id> "=" <exp> | <add>
<add> ::= <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/") <factor> }
<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int>
```

Трохи зміниться <**factor**> щоб можна було посилатися на змінні так само, як і на костанти:

```
<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int> | <id>
```

Для обробки нових бінарних операцій необхідно змінити граматику за аналогією до етапу 2. На кожний рівень пріоритетності необхідне окреме правило граматики. Для бінарних операцій **AST** залишається з минулого етапу.

Логічні операції «**or**» («||») та «**and**» («**&&**») відрізняються в мові **C** та **Python** пріоритетністю так само, як і операція «**not**». «**not**», «**or**» та «**and**» мають однакову пріоритетність і вона нижче ніж пріоритетність цих самих операторів в мові **C**.

CODE GENERATION

Загальний підхід до генерації коду бінарних операцій залишається таким самим:

1. Порахувати **`e1`**
2. Покласти результат в стек
3. Порахувати **`e2`**
4. Дістати значення e1 зі стеку в регістр
5. Виконати операцію над **`e1`** та **`e2`**

Насамперед зміни стосуються саме пункту 5.

Оператори порівняння

Як і у випадку з логічним заперечення **`NOT`**, оператори порівняння повертають **`1`** для дійсних результатів (**`true`**) і **`0`** для негативних (**`false`**). Також вони майже ідентичні до «**`!`**» за винятком того, що вираз порівнюється не з нулем, а з іншим виразом.

Модифікувавши код логічного заперечення з 2 етапу, отримаємо **`реалізацію для «==»:`**

```
    <CODE FOR e1 GOES HERE>
    push   eax        ; save value of e1 on the stack
    <CODE FOR e2 GOES HERE>
    pop    ecx        ; pop e1 from the stack into ecx - e2 is already in eax
    cmp    ecx, eax   ; set ZF on if e1 == e2, set it off otherwise
    mov    eax, 0     ; zero out EAX (doesn't change FLAGS)
    sete   al         ; set AL register (the lower byte of EAX) to 1 if ZF is on
```

Окрім інструкції **sete** існує ще багато команд з цієї серії: **setne** (set if not equal – біт встановлюється, якщо не рівні), **setge** (set if greater than or equal - біт встановлюється, якщо більше, або рівне) і т.д.

Тож для імплементації інших операторів порівняння можна використати вище приведений код, замінюючи **sete** на відповідні команди. Однак не можливо використовувати **ZF** для визначення більшого операнду. Для цього потрібен знаковий прапорець (**SF** - sign flag), який встановлюється в **1**, якщо результат операції негативний.

```
    mov eax, 0 ;zero out EAX
    mov ecx, 2 ;ECX = 2
    cmp ecx, 3 ;compute 2 - 3, set flags
    set al     ;set AL if 2 < 3, i.e. if 2 - 3 is negative
```

Логічне І та АБО

Для цих операторів знадобиться поняття обчислень за скороченою схемою.

Обчислення за скороченою схемою (***short-circuit evaluation***) полягає в тому, що якщо відомий результат після обчислення першої частини, друга частина рахуватись не буде.

Наприклад:

```
return 0 && foo();
```

Перша частина **false**. Нам не потрібно знати значення **foo()**, тому **foo()** взагалі не буде виконуватись. Але не все так просто. Хоч **foo()** ніяк не вплине на результат цього рядка, ця функція може вплинути на щось інше (виконати ввід/вивід, оновити глобальні змінні, …). Тож виконання логічного **І** та **АБО** за скороченою схемою є не лише питанням оптимізації продуктивності, а й вимогою деяких програм для правильного виконання.

Логічне АБО (OR)

Для гарантування «скороченого обрахунку» логічного **І** є наступний алгоритм:

1. Обчислення **e1**;
2. Якщо результат **0**, перехід на пункт **4**;
3. Встановлення регістру **EAX** в **1,** перехід в пункт **6**;
4. Обчислення **e2;**
5. Якщо результат 0, встановлення **EAX** в **0**. В іншому випадку **EAX = 1**;
6. Кінець.

Другий пункт потребує нового типу інструкції, **умовний перехід (conditional jump)**. Ці інструкції переходять на задану позицію в коді, що позначена спеціальною позначкою (**label**).

Приклад інструкції **je** (**jump if equal**):

```
    cmp    eax, 0     ; set ZF if EAX == 0
    je     _there     ; if ZF is set, go to _there
    mov    eax, 1
    ret
_there:
    mov    eax, 2
    ret
```

Якщо **EAX** з самого початку дорівнює **0,** після виконання цього коду повернеться **2**; в противному випадку – **1**. Розглянемо цей код більш детально.

Припустимо, що **EAX == 0**:

1. **cmp eax,0** – **EAX == 0**, тому цей рядок встановить **ZF** в одиницю.
2. **je _there** – **ZF == 1**, тому стрибок на мітку **_there** відбудеться.
3. **mov eax,2** – цей рядок виконається наступним, бо він перший після рядка **_there:** . Він занесе **2** в **EAX**.
4. **ret** – значення, що повертається, дорівнює **2**.

Тепер **EAX == 1**:

1. **cmp eax,0** – **EAX == 1**, тому **ZF = 0**.
2. **je _there** – **ZF == 0**, тому ця інструкція просто не виконується
3. **mov eax,1** – виконується рядок, який йде після **je _there**.
4. **ret** – значення, що повертається, дорівнює **1**.

Окрім інструкції умовного переходу знадобиться ще й безумовний перехід **jmp**.

```
     mov   eax, 0     ; обнуляємо EAX
     jmp   _there     ; стрибок на мітку _there
     movl  eax, 5     ; цей рядок ніколи не виконається, тому завжди
                      ; перестрибуємо через нього
    _there:
     ret              ; завжди повертає 0
```

Тепер можливо реалізувати операцію логічного **АБО**:

```
    <CODE FOR e1 GOES HERE>
    cmp    eax,  0    ; check if e1 is true
    je     _clause2   ; e1 is 0, so we need to evaluate clause 2
    mov    eax,  1    ; we didn't jump, so e1 is true and therefore result is 1
    jmp    _end
_clause2:
    <CODE FOR e2 GOES HERE>
    cmp    eax,  0    ; check if e2 is true
    mov    eax,  0    ; zero out EAX without changing ZF
    setne  al         ; set AL register (the low byte of EAX) to 1 iff e2 != 0
_end:
```

Звичайно, мітки повинні відрізнятись одна від одної на той випадок, якщо в програмі зустрінеться декілька однакових операцій, що їх використовують. Необхідно запрограмувати генерацію унікальних міток, наприклад, інкрементуючи числове значення для кожної нової мітки.

Логічне І (AND)

Реалізація майже ідентична логічному **АБО**, за винятком обчислень за скороченою схемою при **e1 == 0**. Використовується інструкція **jne** (jump if not equal). В цьому випадку не потрібно заносити якесь значення в **EAX**, бо **0** – це те **число, що нам потрібне.**

```
    <CODE FOR e1 GOES HERE>
    cmp eax, 0    ; check if e1 is true
    jne _clause2  ; e1 isn't 0, so we need to evaluate clause 2
    jmp _end
_clause2:
    <CODE FOR e2 GOES HERE>
    cmp eax, 0    ; check if e2 is true
    mov eax, 0    ; zero out EAX without changing ZF
    setne al      ; set AL register (the low byte of EAX) to 1 iff e2 != 0
_end:
```

Побітові операції, остача від ділення

Операції остача від ділення, побітове **І**, побітове **АБО**, побітове **XOR**, побітовий зсув вліво, побітовий зсув вправо на всіх етапах роботи обробляються так само, як і вже продемонстровано і використовують ті самі принципи, тож ви можете реалізувати їх за аналогією.

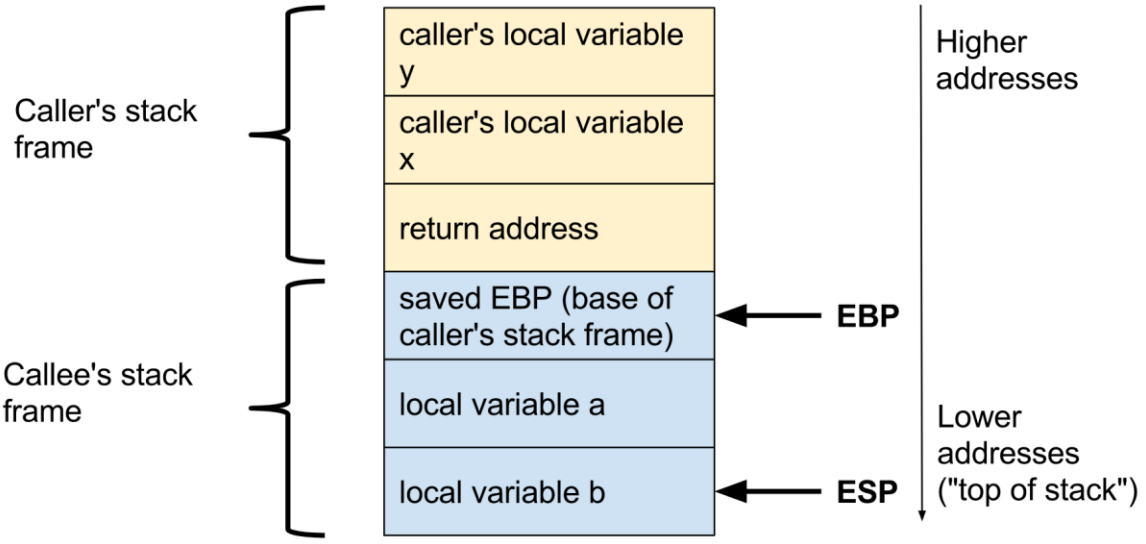
Як вже відомо з попереднього етапу, локальні зміні зберігаються в стеку. Для подальшого посилання на змінну необхідно точно знати її розташування в стеку. Це можна зробити через пари «ім’я - адреса».

Будемо зберігати зсув адреси змінної від регістру **EBP** (**Extended Base Pointer)**, який вказує на вершину, початок стеку, адресу його першого елементу.

Стекові кадри (Stack Frames)

Кадр стеку – область пам’яті, що виділяється на вершині стеку під час виклику функції. Структура даних на стеку з вільною пам’яттю для одного виклику функції, що зберігає локальні змінні, аргументи функції, адресу повернення куди треба перейти після завершення функції.

Ми знаємо, що **ESP** вказує на вершину стеку, яка одночасно є і вершиною поточного стекового кадру. Якщо якесь значення записується в стек, значить від **ESP** віднімається розмір цього значення **(`ESP -= sizeof(variable_type)`)**, а **EBP** залишається незмінним, тому значення **EBP** завжди буде більше ніж **ESP**.



Назвемо функцію, що викликає іншу ***ініціатором***. Коли функція, скажімо, **foo** завершує виконання і повертає значення (повертається), ініціатор повинен знати де він зупинився і з якого місця продовжувати обчислення. Тобто кадр стеку і регістри **ESP** та **EBP** повинні бути такими ж самими якими були до виклику функції **foo**.

Перше що повинна зробити **foo** після виклику – це створити для себе новий кадр стеку за допомогою наступних інструкцій:

```
push ebp                ; save old value of EBP
mov  ebp, esp           ; current top of stack is bottom of new stack frame
```

Це **пролог** функції. Одразу перед поверненням функції виконується **епілог** аби видалити кадр стеку та повернути все як було перед прологом:

```
mov  esp, ebp   ; restore ESP; now it points to old EBP
pop  ebp        ; restore old EBP; now ESP is where it was before prologue
ret
```

Пролог та епілог потрібні для того, щоб:

- представляти адреси змінних як зсув відносно **EBP**. Точно відомо, що нічого немає над **EBP** (створено новий пустий кадр стеку в пролозі) і **EBP** не буде змінюватись до епілогу.
- безпечно класти локальні змінні в стек без зміни кадру ініціатора.

Пролог необхідно створювати одразу після мітки функції, а епілог перед **ret**.

Окрім пари «***змінна – розташування***» треба відстежувати **індекс стеку**, який відповідає за наступне вільне місце в стеку, зсув відносно **EBP**. Наступна вільна позиція завжди після **ESP**, на **ESP - sizeof(variable_type**). Одразу після прологу **EBP** та **ESP** однакові, отже початкове значення індексу буде **-4**. Як тільки змінна записується в стек, значення **ESP** зменшується на 4. (у випадку змінної типу **int**)

Оголошення змінної

Для оголошення змінної вона зберігається в стек і додається до таблиці змінних (**variable map**). Також вона може називатись **symbol table**. Хоча це зовсім не відповідає тому, як працюють реальні компілятори, цей підхід простіший, тому і будемо його використовувати (в учбових цілях). На практиці, компілятори виділяють місце одразу для всіх змінних у пролозі функції або в регістрах.

Variable map зберігає пари «змінна - адреса» щоб можна було знайти змінну в стеку. Поле «змінна» також повинна зберігати тип змінної. Ця інформація необхідна для порівняння типів. Порівняння типів реалізується за рахунок порівння імен типу (**int == int, int != float, ...**). При написанні на мові **C/C++** таке порівняння є доволі ефективним при використанні техніки string interning (описана в ЛР 1), адже необхідно порівняти лише вказівники, а не рядки посимвольно.

Не можна оголошувати змінну декілька разів в одному локальному блоці, як показано нижче:

```
int a;
int a;
```

Тож програма повинна видавати помилку, якщо змінна вже присутня у таблиці змінних.

Ось алгоритм того, як згенерувати асемблерний код для **int a = expression**:

```
if var_map.contains("a"):  // якщо змінна вже є у таблиці
    fail()                       // помилка
generate_exp(expression)      // обчислення виразу …
emit "push eax"             // … та переміщення результату в eax
var_map = var_map.put("a", stack_index)    // запис змінної «a» в таблицю
stack_index = stack_index - 4          // наступна адреса на 4 байти нижче
```

Якщо змінна не ініціалізована, можна присвоїти її будь-яке значення, бо воно все одно не буде використане.

Призначення змінної

Спочатку необхідно знайти змінну в таблиці (якщо вона там взагалі є). Аби надати їй нове значення, треба помістити це значення у відповідну комірку пам’яті.

Розглянемо на прикладі a = expression:

```
generate_exp(expression)
emit "push eax"
var_offset = var_map.find("a")
emit "mov [offset+ebp], eax"
```

Посилання на змінну відбувається через копіювання значення зі стеку в регістр.

Завдання

1. Оновить функцію лексера для обробки лексеми присвоювання та нових бінарних операцій за варіантом.
2. Додати у парсер оголошення змінної, присвоювання їй значення та посилання на цю змінну (її використання в програмі). Також додайте нові бінарні операції.
3. Асемблерний генератор повинен створювати пролог та епілог функції; генерувати код для оголошення, присвоювання та посилання на змінну, нові бінарні операції, та виводити все це в окремий файл. Надалі зміст цього файлу повинен перевірятися на виконання.
4. Протестувати програму на всіх контрольних прикладах з попередніх робіт, нових прикладах, та внесіть їх результати у протокол. Програма повинна працювати для всіх коректних прикладів та видавати помилку для некоректних, що містять помилку. При обробці помилок повинно вказуватися місце в коді де сталась помилка.

Варіанти завдання для самостійної роботи:

Вар	Нова бінарна операція 1	Нова бінарна операція 2	Тип змінних
1	*	&&	int, float
2	*	or	int, str
3	*	==	int, float
4	*	!=	int, float
5	*	<	int, float
6	*	<=	int, str
7	*	>	int, float
8	*	>=	int, str
9	+	%	int, char
10	+	&	int, float
11	+	|	int, char
12	+	^	int, str
13	-	<<	int, char
14	-	>>	int, float
15	-	%	int, char

Вар	Нова бінарна операція 1	Нова бінарна операція 2	Тип змінних
16	-	and	int, str
17	-	||	int, float
18	-	==	int, float
19	-	!=	int, float
20	-	<	int, str
21	+	<=	int, float
22	+	>	int, str
23	+	>=	int, float
24	+	%	int, float
25	/	&	int, char
26	/	or	int, str
27	/	^	int, float
28	/	<<	int, str
29	/	>>	int, char
30	/	%	int, float

Контрольні питання

1. Яка основна відмінність обробки функцій на цьому етапі?
2. Чи вірний цей вираз: a = 2 * (b = 2)? Обґрунтуйте свою відповідь.
3. Чому присвоювання не обробляється як звичайний бінарний оператор?

4. Поясніть порядок розташування операторів в граматиці.
5. Навіщо потрібен стек, стекові кадри та пари «ім’я – адреса»? Яка інша назва цих пар?
6. Яку роль виконує регістр EBP?
7. Що таке пролог та епілог функції?
8. Як змінюється індекс стеку?
9. Який алгоритм використання таблиці змінних?
10. Як обробляється пріоритетність?
11. Чому **AST** залишилось незмінним?
12. Що означає «встановити біт» (**set a bit**)?
13. Чому використовується прапорець **SF**?
14. Що таке **short-circuit evaluation** і чому це важливо чи не важливо?
15. В чому різниця між je та **jmp** ?

Етап 4
Вкладені конструкції. Оператори розгалуження. Тернарний оператор

Мета етапу: застосування у компіляторі обробки вкладених конструкцій, ***conditional statements*** (таких як **if**) та ***conditional expressions*** (тернарних операторів типу **a ? b : c**).

Необхідні теоретичні відомості

Вкладені конструкції

В мові С вкладеним виразом (конструкцією) є набір виразів, обернених у фігурні дужки.

В мові Python блоки усередині складової конструкції виділяються шляхом зсуву на однакову кількість пробілів (зазвичай чотири). Ознакою кінця блоку є зсув наступної інструкції вліво щодо останньої.

Такі блоки можна зустріти в **if, while** виразах тощо.

```
// C
if (flag) {
    //this is a compound statement!
    int a = 1;
}

# Python
if flag:
    #this is a compound statement!
    a = 1
```

В мові С вони також можуть бути самостійними:

```
int main() {
    int a;
    {
        //this is also a compound statement!
        a = 4;
    }
}
```

Також в С існують глибоко вкладені вирази:

```
int main() {
    //compound statement #1 (function bodies are compound statements!)
    int a = 1;
    {
        //compound statement #2
        a = 2;
        {
            //compound statement #3
            a = 3;
```

```
            if (a) {
                //compound statement #4
                a = 4;
            }
        }
    }
}
```

В Python два попередніх твердження невірні.

В мові **C** є **лексична область видимості (lexical scoping)**. Область видимості змінної обмежена блоком, де вона оголошена. Це та ділянка програми, в межах якої ви можете посилатись на цю змінну.

До цього моменту тіло функції було одним єдиним блоком, тож змінну можна було використовувати з будь-якої точки в межах **`main`** після оголошення. Тепер все буде трохи інакше і складніше (а тим і цікавіше).

Подивимось, як працюють області видимості в **C**.

Якщо змінна оголошена у фігурних дужках (у внутрішній області видимості), до неї НЕ можна доступитись за дужками (з зовнішньої області):

```
// here is the outer scope
{
    // here is the inner scope
    int foo = 2;
}
// now we're back in the outer scope
foo = 3; // ERROR - foo isn't defined in this scope!
```

Однак із внутрішньої області можна доступитись до зовнішньої:

```
int a = 2;
{
    a = 4;      // this is okay
}
return a;       // returns 4 - changes made inside the inner scope are
                   reflected here
```

В мові Python не має поняття самостійних областей видимості, таких як в прикладі нижче з мови C:

```
     int main() {
         int a;
         {
             //this is also a compound statement!
             a = 4;
         }
     }
```

Вони є або тілом функції, або тілом циклу, або тілом виразу **if-else**.

В цій мові ви можете доступитись із зовнішнього блоку до змінної, що визначена у внутрішньому блоці:

```
if condition:
  # inner scope 1
  x = 'something'
else:
  # inner scope 2
  x = 'something else'
# outer scope
use(x)
```

Зазначимо, що змінна **x** не була визначена до цієї ділянки коду.

Але не можна з однієї функції доступатись до змінної іншої функції.

В мові С не можна мати дві змінні з однаковим ім'ям в одній області видимості:

```
int foo = 0;
int foo = 1; //This will throw a compiler error
```

Цей факт не заважає мати дві змінні з однаковим ім'ям у різних областях видимості. Як тільки змінна внутрішньої області оголошена, вона буде перекривати зовнішню змінну; зовнішня змінна буде недоступною, доки не закінчиться вкладена область видимості.

```
int foo = 0;
{
    int foo; //this is a TOTALLY DIFFERENT foo, unrelated to foo from
earlier
    foo = 2; //this refers to the inner foo; outer foo is inaccessible
}
return foo; //this will return 0 - it refers to the original foo, which is
                //unchanged
```

Ідея в тому, що це абсолютно різні змінні, яким не пощастило мати однакові імена.

```
int foo = 0;
{
    foo = 3;    //changes outer foo
    int foo = 4;      //defines inner foo, shadowing outer foo
}
return foo;     //returns 3
```

У Python тут теж є свої відмінності. Ця мова дозволяє доступатись до змінних з будь-якого блоку (як зазначено вище), а також переписувати змінні (як розповідалось на етапі номер три). Отже,

- можна мати 2 однакові змінні в одному блоці: змінна вище буде переписана змінною нижче,

- а також мати однакові змінні в різних блоках: змінна теж буде переписана.

```
def main():
    # outer scope
    for i in range(0, 5):
        # inner scope
        a = 1
    print(a)
    if a == 1:
        # inner scope
        b = 2
    print(a + b)
    return 0
```

Оператори розгалуження

Оператор розгалуження (його ще називають ***умовним оператором***) дозволяє програмісту обрати подальшу поведінку програми залежно від якоїсь умови, стану тощо.

Існують дві основні умовні конструкції: **`if`** та **`if-else`**.

Вираз (***statement***) **`if`** складається з умови, першого виразу та необов'язкового другого виразу. Умова невірна (**`false`**), якщо прирівнюється до **`0`**, та вірна (**`true`**) в іншому випадку. Перший вираз виконується тільки якщо умова задовільна, другий вираз – якщо ні. Квадратними дужками показана необов'язкова частина.

```
if (condition) {
  statement1;
  …

}
[else{
  statement2;
  …

}]
```

У свою чергу під-вираз може бути як окремим виразом, …

```
if (flag)
  return 0;
```

…так і вкладеним блоком у фігурних дужках

```
if (flag) {
  int a = 1;
  return a*2;
}
```

Також слід врахувати, що в мові **Python** відрізняється синтаксис і поняття зони видимості. Видимість змінних обмежена лише ***функціями, класами чи***

модулями. Умовні конструкції не враховуються. Наприклад, такий код буде виконано, хоча з'явиться попередження «**Name 'a' can be not defined**». Тут все знаходиться в одній локальній зоні видимості.

```
if cond:
    a = 0
    print(cond)
print(a)
```

В мові **C** немає явної конструкції **else-if**. Але в той же час такий код

```
if (flag)
    return 0;
else if (other_flag)
    return 1;
else
    return 2;
```

еквівалентний наступному:

```
if (flag)
    return 0;
else {
    if (other_flag)
        return 1;
    else
        return 2;
}
```

У Python існує конструкція **elif**:

```
if flag:
    return 0
elif other_flag:
    return 1
else:
    return 2
```

Conditional expressions дозволяють простим умовам бути представленим у вигляді простих expressions. Вони мають наступний вигляд: [изменено]

```
a ? b : c
```

Якщо **a == true**, виразу буде присвоєне значення **b**, в іншому випадку – значення **c**. Наприклад,

```
int a = flag > 0 ? 2 : 3;  // При flag > 0, a = 2. При flag <= 0, a = 3
```

Зауважимо, що необхідно виконувати тільки той вираз, який насправді потрібен. В наступному коді

```
0 ? foo() : bar()
```

функція **foo()** ніколи не повинна викликатись. Можна спробувати викликати обидві функції, а потім відмінити дії однієї із них, але це неправильно, тому що

foo() може щось вивести на пристрій виводу, доступитись до мережі, або розіменувати **null pointer** і привести до аварійної зупинки програми.

В Python **conditional expressions** теж трохи відрізняються:

```
a if b else c
```

Тут умовою виступає **b**. Якщо **b == true**, виразу буде присвоєне значення **a**, в іншому випадку – значення **c**. Також відмінний синтаксис. Зверніть на це увагу при лексичному аналізі.

Тернарні оператори та **if** вирази дуже схожі, але важливо пам'ятати, що ***statements*** та ***expressions*** використовуються та обробляються у різний спосіб. Різниця між ними вже розглядалась у минулих етапах.

Приведемо приклад: ***expression*** має значення, а ***statement*** – ні.

```
int a = flag ? 2 : 3; // це вірно

// це помилка:
int a = if (flag) 2;
        else 3;
```

З іншого боку, ***statement*** може включати в себе інші ***statement***, а ***expression*** не може містити ***statement***. Наприклад, ***return statement*** в ***if statement***:

```
if (flag)
    return 0;
```

return statement не може міститись в ***conditional expression***:

```
flag ? return 1 : return 2;// помилка
```

LEXING

Вкладені конструкції не потребують нових лексем, а для операторів розгалуження знадобляться наступні:

- Ключове слово **if** – **if**
- Ключове слово **else** – **else**
- Двокрапка – **:**
- Знак питання – **?**

Пам'ятаємо про відмінності синтаксису C та Python.

PARSING

Існує певна проблема. Справа у тому, що минулого разу оголошення змінної було позначено як ***statement***. Але оголошення змінних – це не ***statement***, а ***expression***. Як доказ, у вас не скомпілюється такий код:

```
//this will throw a compiler error!
if (flag)       // фігурні дужки мають бути відсутніми
  int i = 0;    // statement. Має бути expression
```

Код, який компілюється:

```
int i;
if (flag) i = 0;
```

Чому ж ця помилка була навмисно допущена минулого разу? На той момент не було важливим, буде оголошення виразом, чи не буде. Все одно б вийшло те ж саме, тільки складніше. Але тепер складність конструкції компілятору підвищується і таке спрощення буде напряму впливати на те, що і як парситься, і чи можна взагалі щось таке робити. Крім того, зміна граматики сьогодні спростить виконання завдань на наступних етапах.

Потрібно окремо винести оголошення змінної **`Declare (string, exp option)`**. Такий крок створює нову проблему: тіло функції визначено як набір виразів, але якщо оголошення змінної не є виразом, в тілі функції не можна оголошувати змінні. Отже, треба змінити спосіб оголошення функцій. Введемо нову термінологію:

- **block item** – вираз чи оголошення нової змінної.
- **block** або **compound statement** – набір вкладених конструкцій у фігурних дужках чи відступах.

Тіло функції – це лише особливий тип блоку, воно складається з виразів та оголошень змінних.

Отже, нове AST і нова граматика повністю з усім, що розглядалося у попередніх етапах:

```
program = Program(function_declaration)
function_declaration = Function(string, block-item list)
block-item list = Statement(statement) | Declaration(declaration)
declaration = Declare(type, string, exp option)
statement = Return(exp) | Exp(exp)
exp = Assign(string, exp) | Var(string) | UnOp(unary_operator, exp) |
  BinOp(binary_operator, left_exp, right_exp) | Constant(int, float, char)
```

```
<program> ::= <function>
<function> ::= "int" id "(" ")" "{" { <block-item> } "}"
<block-item> ::= <statement> | <declaration>
<statement> ::= "return" <exp> ";" | <exp> ";"
<declaration> ::= <type> <id> [ = <exp> ] ";"
<exp> ::= <id> "=" <exp> | <log_or>
<log_or> = <log_and> { "||" <log_and> }
<log_and> ::= <bit_or> { "&&" <bit_or> }
<bit_or> ::= <xor> { "|" <xor> }
<xor> ::= <bit_and> { "^" <bit_and> }
<bit_and> ::= <equals> { "&" <equals> }
```

```
<equals> ::= <not_equals> { ("==" | "!=") <not_equals> }
<not_equas> ::= <shift> { ("<" | ">" | "<=" | ">=") <shift> }
<shift> ::= <add> { ("<<" | ">>") <add> }
<add> ::= <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/" | "%") <factor> }
<factor> ::=  "(" <exp> ")" | <unary_op> <factor> | id | int | float | char
<unary_op> ::= "!" | "~" | "-"
<type> ::= int | float | char
```

Нарешті можна приступити до безпосередньої реалізації вкладених конструкцій. Вкладена конструкція – це набір виразів та оголошень, який є частиною **statement**.

```
Compound (block_item list)
```

Граматика:

```
"{" { <block-item> } "}
```

Нагадуємо, що "**{**" "**}**" – це фігурні дужки, а **{ }** – повторення чогось. Таким чином <**block-item**> може бути безліч.

Процес парсингу ***conditional expressions*** та ***if statements*** значно відрізняється.

Почнемо з ***if.***

На даний момент в нас є 3 типи ***statement***: **return**, **expressions**, **compound**.

```
statement = Return(exp)
          | Exp(exp)
          | Compound (block_item list)
```

Треба додати вираз **if**, який складається з трьох частин: **expression** (умова), гілка **if** та опціональна гілка **else**:

```
Conditional(exp, block-item list, block-item list option)
/*exp is controlling condition, first statement (or a lot of statements) is
'if' branch, second statement (or a lot of statements) is optional 'else'
branch*/
```

Тепер граматика. Її складовими є ключове слово **if**, обернений в дужки умовний вираз, вираз, що виконується при **true**, необов'язковий вираз, що виконується при **false**.

```
"if" "(" <exp> ")" <block-item> [ "else" <block-item> ]
```

Пам'ятаємо про відмінності синтаксису C та Python.

Визначення ***statement*** стало рекурсивним, але не ліво-рекурсивним, тож все гаразд.

Зараз давайте розберемося з ***conditional expressions***.

Актуальне **AST** для ***expressions***:

```
exp = Assign(string, exp)
    | Var(string) //string is variable name
    | BinOp(binary_operator, exp, exp)
    | UnOp(unary_operator, exp)
    | Constant(int, float, char)
```

Дуже просто додати ***conditional***:

```
CondExp(exp, exp, exp)
/*the condition, 'if' expression and 'else' expression, respectively*/
```

Умовний оператор має вищу пріоритетність, ніж оператор присвоєння «=», але нижчу за логічне **АБО** «||»:

```
<conditional-exp> ::= <log_or> "?" <exp> ":" <conditional-exp>
```

Чому цей запис має саме таку форму? Для зручності будемо посилатись на ці вирази як **e1, e2, e3** (**e1 ? e2 : e3**).

Вираз **e1** має бути **<logical-or-exp>**, бо він не може бути ні ***assignment expression***, ні ***conditional expression***, які йдуть перед ним за пріоритетністю. **e1** не може бути ***assignment expression***, тому що присвоєння має пріоритетність нижчу, ніж умовний оператор. Інакше кажучи, вираз **a = 1 ? 2 : 3**; повинен парситись як **a = (1 ? 2 : 3)**;

Тож зараз в нашій граматиці немає неоднозначності. Але якщо ми замінимо **<logical-or-exp>** на **<exp>**, цей вираз зможе парситись ще й інакше: **(a = 1) ? 2 : 3**, тому виникне неоднозначність;

Це також є правильним, але для цього необхідно огорнути **a = 1** в дужки.

e1 не може бути ***conditional expression***, бо оператор **?** є право-асоціативним. Якщо замінити **<logical-or-exp>** на **<conditional-exp>**, вираз

flag1 ? 4 : flag2 ? 6 : 7

буде парситись як

(flag1 ? 4 : flag2) ? 6 : 7,

хоча повинен парситись як

flag1 ? 4 : (flag2 ? 6 : 7).

e3 може бути ще однією тернарною конструкцією, як в попередньому прикладі, але не може бути присвоєнням. Розглянемо приклад:

```
flag ? a = 1 : a = 0
```

Якщо спробувати скомпілювати цей вираз на реальному компіляторі (наприклад, **gcc**), ми отримаємо щось таке:

```
error: expression is not assignable
    flag ? a = 1 : a = 0;
    ~~~~~~~~~~~~~~~~ ^
```

Це вийшло тому, що **gcc** спробував розпарсити цей вираз як

```
(flag ? a = 1 : a) = 0
```

Очевидно, це не спрацювало, бо ліва частина виразу **(flag ? a = 1 : a)** не є змінною. Виникає питання, чому не можна використати таке правило граматики:

```
<conditional-exp> ::= <log_or> "?" <exp> ":" <exp>
```

Тоді можна розпарсити вираз таким чином:

```
flag ? (a = 1) : (a = 0)
```

З цим правилом немає жодних проблем. І більш того, саме так визначено тернарний оператор в **C++.** Але з якоїсь причини, воно не таке у мові **C**.

На щастя, можна уникнути цих проблем, якщо в самій програмі брати все в дужки так само як показано вище:

flag ? (a = 1) : (a = 0).

Змінній **a** буде присвоєно **0** без усяких помилок та проблем.

Також необхідно враховувати вирази, що не є умовними, тож «умовна частина» нашого правила не є обов'язковою.

```
<conditional-exp> ::= <logical-or-exp> [ "?" <exp> ":" <conditional-exp> ]
```

CODE GENERATION

Вкладені конструкції

Як показано раніше, можливо мати дві різні змінні з однаковим ім'ям в різних зонах видимості, які зберігаються у різних комірках стеку:

```
int foo = 3;
{
  int foo = 4;
}
```

Тож згенерований код повинен доступатись до «правильної» змінної в стеку, або завершитись помилкою, якщо змінна за межами блоку. Необхідно приділити основу увагу таблиці змінних (**variable map**).

Пам'ятаємо про відмінності мови Python: у коді вище змінна «3» буде перезаписана змінною «4», друга змінна створена не буде.

Один із способів реалізації: кожний блок має свою таблицю змінних. Таким чином оголошення змінної всередині блоку не буде конфліктувати з «зовнішньою стороною».

Як тільки програма доходить до створення нового блоку, створюється нова таблиця змінних, а при виході з блоку повертаємось до старої (тож не треба її видаляти). Ще одна причина не видаляти стару таблицю – необхідність перевірити існування змінної при посиланні на неї. Для цього шукаємо наявність змінної в попередніх, вже існуючих таблицях.

Після повернення до старої таблиці можна видалити нову. Вона більше не знадобиться, бо до змінної внутрішньої області не можна доступитись із зовнішньої (але у Python можна, майте це на увазі).

Реалізувати збереження таблиць змінних можна за принципом стеку.

```
int main(){
    // POINT 1
    // create variable map 1, work with var_map_1
    int a = 1;
    int b = 2;
    {
        // POINT 2
        // create variable map 2, work with var_map_2
        int c = 3;
        {
            // POINT 3
            // create variable map 3, work with var_map_3
            int d = 4;
        }
        // POINT 4
        // delete variable map 3, work with var_map_2
        a = 0;
        d = 9; // ERROR!
    }
    // POINT 5
    // delete variable map 2, work with var_map_1
    b = 8;
    return 0;
}
```

Стан структури даних, що зберігає таблиці змінних на всіх етапах за принципом стеку.

POINT 1

POINT 2
var_map_1

POINT 3
var_map_2
var_map_1

POINT 4
var_map_1

POINT 5

Нагадаємо, що таблиця змінних складається з декількох таблиць, кожна для окремого блоку. Ці таблиці можуть бути організовані у вигляді стеку. Нова таблиця створюється на вершині стеку коли парсер зустрічає новий блок, і видаляється, коли робота з блоком завершена. Для імплементації самих таблиць можна використати бінарні дерева, хеш-таблиці, і навіть лінійні списки, масиви. Хеш-таблиці дуже ефективні для великих блоків, але для блоків в яких не більше 10 змінних підійдуть і звичайні лінійні структури даних.

Для генерації асемблеру умовних виразів знадобляться умовні та безумовні переходи, які вже розглядалися в одній з минулих робіт. Наприклад, для

e1 ? e2 : e3

можна згенерувати код наступним чином:

```
<CODE FOR e1 GOES HERE>
  cmp eax, 0
  je   _e3               ; if e1 == 0, e1 is false so execute e3
<CODE FOR e2 GOES HERE>  ; we're still here so e1 must be true. execute e2.
  jmp  _post_conditional ; jump over e3
 _e3:
<CODE FOR e3 GOES HERE>  ; we jumped here because e1 was false. execute e3.
  _post_conditional:     ; we need this label to jump over e3
```

Код для **if** генерується аналогічно, хоча може бути складніше через необов'язковий **else**.

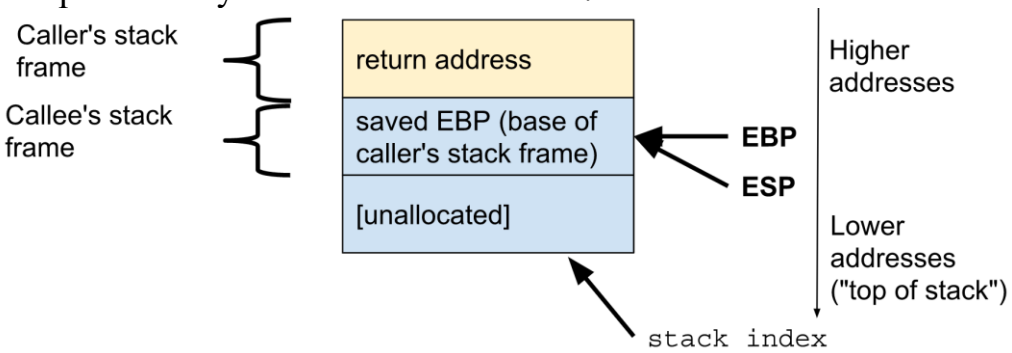
Також не можна забувати про унікальність міток.

Деалокація (видалення) змінних

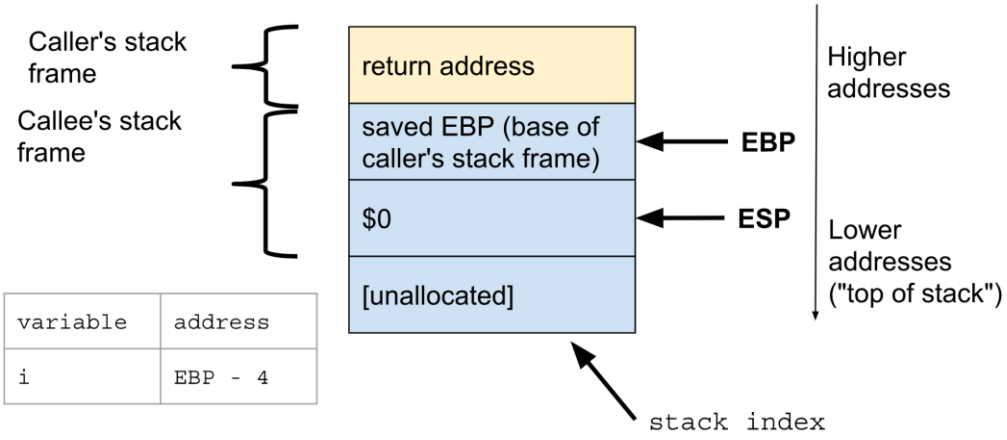
При створенні (оголошенні) змінної змінюється вказівник стеку. Це необхідно врахувати при роботі з внутрішніми областями видимості.

```
int main() {
    {
      int i = 0;
    }
    int j = 1;
    return j;
}
```

На початку маємо порожню таблицю змінних та **`stack_index = -4`**, бо перша порожня комірка в стеку на 4 байти нижче **`EBP`**.

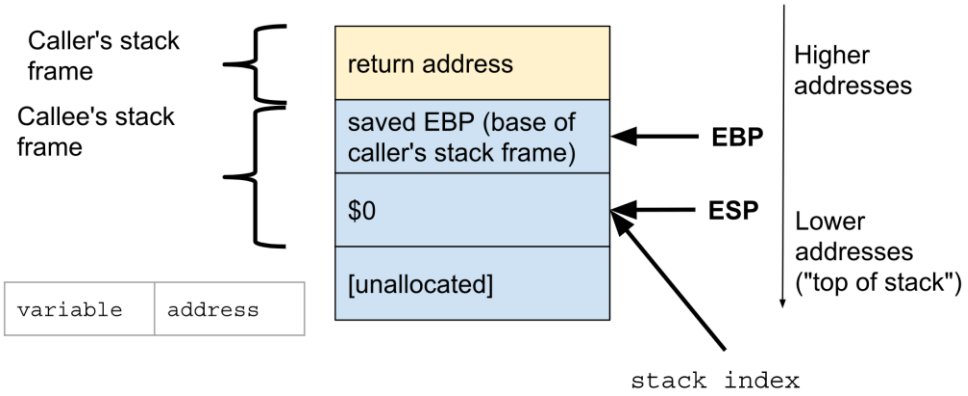


Після виконання **`int i = 0;`** в стек буде занесена змінна **`i`**. **`ESP`** вказує на **`EBP-4`** та **`stack_index`** дорівнює **`-8`**

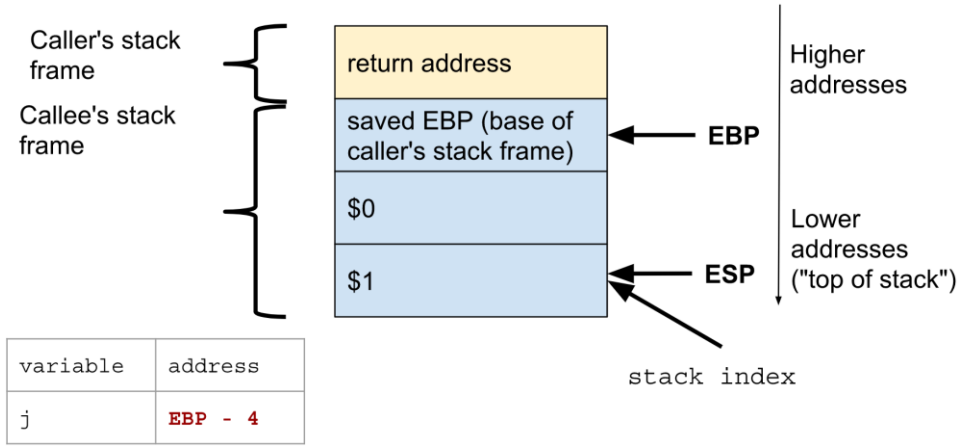


Після виходу з блоку змінна **`i`** більше не потрібна, вона видаляється з таблиці змінних та **`stack_index`** повертається значення **`-4`**.

Проблема в тому, що змінна не видаляється зі стеку.



Після занесення змінної **`j`** в стек, вона повинна бути під **`i`**, на **`EBP - 8`**.



Але оскільки **`stack index == -4`**, посилання на **`j`** будуть насправді посилатись на позицію **`i`**.

Рішенням є видалення змінних зі стеку наприкінці блоку. Кількість змінних для видалення можна дізнатись через розмір таблиці змінних цього блоку.

Завдання

6. Оновити лексер для обробки вкладених конструкцій, областей видимості та операторів розгалуження згідно варіанту.

7. Поновити парсер.
8. Удосконалити генератор коду.
9. Протестувати програму на всіх контрольних прикладах з попередніх робіт, нових прикладах, та внесіть їх результати у протокол. Програма повинна працювати для всіх коректних прикладів та видавати помилку для некоректних, що містять помилку. При обробці помилок повинно вказуватися місце в коді де сталась помилка.

Звіт повинен містити:

- тему, мету, сутність варіанта завдання що необхідно виконати у завданні;
- лістинг програми компілятора;
- власні контрольні приклади, на яких тестувалась програма з порахованими відповідями;
- скріншоти з результатами тестування програми;
- письмові відповіді на контрольні запитання;
- висновки щодо отриманих результатів.

До звіту додаються вхідні та вихідні файли, початковий та скомпільований файли програми компілятора.

Варіанти завдання для самостійної роботи:

Варіант	Умовний оператор
1	if else
2	if else
3	ternary
4	ternary
5	if else
6	if else
7	ternary
8	ternary
9	if else
10	if else
11	ternary
12	ternary
13	if else
14	if else
15	ternary

Варіант	Умовний оператор
16	ternary
17	if else
18	if else
19	ternary
20	ternary
21	if else
22	if else
23	ternary
24	ternary
25	if else
26	if else
27	ternary
28	ternary
29	if else
30	if else

Контрольні питання

1. Що таке область видимості і в чому їх відмінності в різних мовах?
2. Як працюють області видимості?
3. Навіщо введені поняття **block item** та **block**?

4. Як обробляється доступ до «правильної» змінної?
5. У чому різниця парсингу між ***conditional statements*** та ***conditional expressions***?
6. Як умовні оператори та вкладені конструкції відрізняються у мовах ***C*** та ***Python***?

Етап 5
Функції

На цьому етапі ви навчитесь обробляти виклики функцій. Функції є найбільш вживанішими конструкціями для структуризації коду, а тому їх імплементація повинна бути як можна ефективнішою.

Звичайно, наш компілятор вже може працювати з визначенням функцій, адже в наших програмах є функція **`main`**. Але все ще є неможливим викликати інші, власноруч написані функції:

```
int three() {
    return 3;
}

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

Також неможливо передати функції якісь параметри:

```
int sum(int a, int b) {
    return a + b;
}

int main() {
    return sum(1, 1);
}
```

І робити попереднє оголошення в мові **C**:

```
int sum(int a, int b);

int main() {
    return sum(1, 1);
}

int sum(int a, int b) {
    return a + b;
}
```

Все це з’явиться в компіляторі після виконання цієї роботи.

Деяка термінологія:

- **Оголошення** (declaration) функції визначає *ім’я функції*, *тип значення що повертається* (у нашому випадку завжди **`int`**, у мові Python відсутній), *набір параметрів* (якщо він є, у Python типи параметрів не вказуються):

```
int foo(int a);
```

Оголошення говорять компілятору, що функція буде визначена пізніше, можливо в іншому файлі і дозволяє використовувати функція в коді до того, як вона була визначена.

Окрім цього, оголошувати функцію можна вже після визначення, але це навряд чи має сенс:

```
int foo() {
    return 4;
}

int foo();
```

В мові Python немає оголошень як таких. Вони йдуть разом з тілом функції, а це вже *визначення функції* (дивіться нижче).

- **Визначення** (definition) функції – це оголошення плюс тіло функції:

```
int foo(int a) {
      return a + 1;
}
```

Оголошувати функцію можна скільки завгодно разів, але визначати – лише один раз.

- **Аргументами** функції є значення, що передаються функції при виклику.
- **Параметри** функції – це список значень в оголошенні функції.

```
int foo(int a) {         // parameter
    return a + 1;
}

int main() {
    return foo(3); // argument
}
```

Обмеження:

- Підтримуються функції лише типу **`int`.** Параметри типу **`int, float, char/str`**.
- Порожній список параметрів буде інтерпретуватись, як функція без параметрів.
- Імена параметрів функції є обов'язковими. Наступне підтримуватись не буде:
  ```
  int foo(int, int);
  ```
- Жодних модифікаторів: **`extern, static, const`** тощо.

LEXING

Новою лексемою є кома для розділення аргументів та параметрів функції.

Також необхідно додати деякі нові операції. Їх реалізація базується на основі тих операцій, які вже були реалізовані, тож це не буде проблемою:

- +=
- -=
- /=
- *=
- %=
- <<=
- >>=
- &=
- |=
- ^=
- Префіксний інкремент ++
- Префіксний декремент --
- Постфіксний інкремент ++
- Постфіксний декремент --

PARSING

Визначення функції

У нинішньому визначенні функція має лише ім’я та тіло. Нижче приведено повне **AST**:

```
program = Program(function_declaration)
function_declaration = Function(string, block-item list)
block-item list = Statement(statement) |
                  Declaration(declaration)
declaration = Declare(type, string, exp option)
statement = Return(exp) |
            Exp(exp) |
            Compound (block-item list) |
            Conditional(exp, block-item list, block-item list option)
exp = Assign(string, exp) |
      Var(string) |
      UnOp(unary_operator, exp) |
      BinOp(binary_operator, left_exp, right_exp) |
      Constant(int, float, char) |
      CondExp(exp, exp, exp)
```

Тепер необхідно додати список параметрів та оголошення, що не мають тіла функції. Визначимо єдине правило **function_declaration** з необов'язковим тілом функції для представлення і оголошень, і визначень:

```
function_declaration = Function(string, // function name
                                types list, // parameters types
                                 string list, // parameters
                                block_item list option) // body
```

Але можна розробити свої правила для функцій (як і для всього іншого). Це лише приклад.

Звернемо увагу, в цьому правилі не вказується тип повертаємого функцією значення, бо ми маємо лише тип **int**.

Ось повна граматика на даний момент:

```
<program> ::= <function>
<function> ::= "int" id "(" ")" "{" { <block-item> } "}"
<block-item> ::= <statement> | <declaration>
<statement> ::= "return" <exp> ";" |
          <exp> ";" |
          "if" "(" <exp> ")" <block-item> [ "else" <block-item> ]
<declaration> ::= <type> <id> [ = <exp> ] ";"
<exp> ::= <id> "=" <exp> | <conditional-exp>
<conditional-exp> ::= <log_or> ["?" <exp> ":" <conditional-exp>]
<log_or> = <log_and> { "||" <log_and> }
<log_and> ::= <bit_or> { "&&" <bit_or> }
<bit_or> ::= <xor> { "|" <xor> }
<xor> ::= <bit_and> { "^" <bit_and> }
<bit_and> ::= <equals> { "&" <equals> }
<equals> ::= <not_equals> { ("==" | "!=") <not_equals> }
<not_equas> ::= <shift> { ("<" | ">" | "<=" | ">=") <shift> }
<shift> ::= <add> { ("<<" | ">>") <add> }
<add> ::= <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/" | "%") <factor> }
<factor> ::=  "(" <exp> ")" |
                <unary_op> <factor> | id | int | float | char
<unary_op> ::= "!" | "~" | "-"
<type> ::= int | float | char
```

Правило для функцій оновиться на таке:

```
<function> ::= "int" <id> "(" [ <type> <id> { "," <type> <id> } ] ")"
( "{" { <block-item> } "}" | ";" )
```

Оголошення функції завершується або її тілом (у випадку визначення), або крапкою з комою.

В мові **Python** відсутнє поняття оголошення функції, є лише визначення – оголошення з тілом. Також не вказуються типи параметрів.

Виклик функції

Виклик функції – це вираз, що виглядає наступним чином:

```
foo(arg1, arg2)
```

Він складається з імені функції та списку аргументів. Аргументи можуть бути будь-якими expressions:

```
foo(arg1 + 2, bar())
```

Тож оновімо **AST** для expressions:

```
exp = ...
    | FunCall(string, exp list) // string is the function name
    ...
```

Граматика:

```
<factor> ::= <function-call> | "(" <exp> ")" |
<unary_op> <factor> | <int> | <id>
<function-call> ::= id "(" [ <exp> { "," <exp> } ] ")"
```

Виклик функцій має найвищий пріоритет з можливих. Відповідно, це правило граматики буде знаходитись у **<factor>**.

Ще декілька змін. Зараз програма, яку можна компілювати, може мати при собі лише одне визначення функції. Треба дозволити купу таких визначень для різних функцій. **AST** і граматика:

```
program = Program(function_declaration list)
```

```
<program> ::= { <function> }
```

Перевірка

Необхідно перевірити, чи є визначення функцій та їх виклики допустимими в програмі. Ці перевірки можна робити під час генерації коду, або додати новий прохід компілятора між парсингом і генерацією. Останнє є не дуже гарною ідеєю, бо це може виявитись доволі громіздким та заплутаним.

Компілятор повинен видавати помилку:

- Два визначення функції з одним і тим самим іменем:

```
int foo(){
    return 3;
}

int foo(int a){
    return a + 1;
}
```

Python дозволяє перевантаження функцій, актуальною буде функція, що визначена останньою. За допомогою оператора розгалуження **`if`** можна вибирати необхідне визначення функції з однаково назвою, але різною реалізацією.

- В мові **C** два оголошення функції мають різне число параметрів:

```
int foo(int a, int b);

int foo(int a){
    return a + 1;
}
```

Але різні імена параметрів дозволяються:

```
int foo(int a);

int foo(int b){
    return b + 1;
}
```

- Функція викликана з різним числом аргументів:

```
int foo(int a){
    return a + 1;
}

int main() {
    return foo(3, 4);
}
```

Задля виведення корисних повідомлень про помилки можна обходити дерево і обробляти таблицю, яка містить інформацію про число аргументів кожної функції та визначена вона, чи ні.

CODE GENERATION

Конвенції виклику функцій

У більшості прикладів вище функція визначена і викликана в одному й тому файлі. Але для виклику функції, наприклад, зі стандартної бібліотеки необхідно слідувати певним правилам. При використанні бібліотеки загального користування (***shared library***) вона не компілюється заново. Аби доступитись до неї, необхідно згенерувати код, який може взаємодіяти з об’єктними файлами (***object files***) створеними сторонніми компіляторами. Іншими словами, необхідно слідувати відповідній конвенції виклику функцій.

Введемо деякі позначення:

- Функція, що викликає іншу функцію (caller) – функція **A**.
- Викликана функція (callee) – функція **B**.

Конвенція виклику функцій відповідає на питання:

- Як передаються аргументи функції **B**? Вони передаються до регістрів, чи до стеку?
- Функція **A** чи **B** відповідальна за видалення аргументів зі стеку після виконання функції **B**?
- Як повернені значення (return values) передаються назад до функції **A**?
- Які регістри є caller-saved, а які callee-saved?

Якщо регістр є ***caller-saved***, ***callee*** (функція **B**) може переписувати його. Для цього ***caller*** (функція **A**) повинна заносити значення цього регістра в стек, а потім повертати до регістра після завершення виконання функції **B**.

Якщо регістр є ***callee-saved***, функція **A** може припустити, що регістр не зміниться після виконання функції **B**. Отже, функція **B** повинна зберегти значення регістра в стек і відновити його перед поверненням до функції **A**.

У цій роботі прикладом буде конвенція **`cdecl`**, але можна використовувати будь яку іншу:

- Аргументи передаються в стек справа наліво (перший аргумент в найменшій адресі пам’яті)
- Функція **A** видаляє аргументи зі стеку.
- Значення що повертаються потрапляють у регістр **EAX** (у загальному випадку не все так просто, але оскільки ми працюємо лише з **`int`** на етапі генерації коду, цього буде достатньо).
- Регістри **EAX**, **EBX**, **ECX** та **EDX** є ***caller-saved***, всі інші ***callee-saved***. В наступному розділі ми побачимо, що функція B повинна відновлювати **EBP** та **ESP** перед поверненням та відновлює EIP інструкцією **ret**. Ми не використовуємо регістри **ESI**, **EDI**, а також одразу заносимо значення **EAX**, **EBX**, **ECX** та **EDX** в стек. Відповідно, взагалі не потрібно піклуватися відносно збереження та відновлення регістрів.

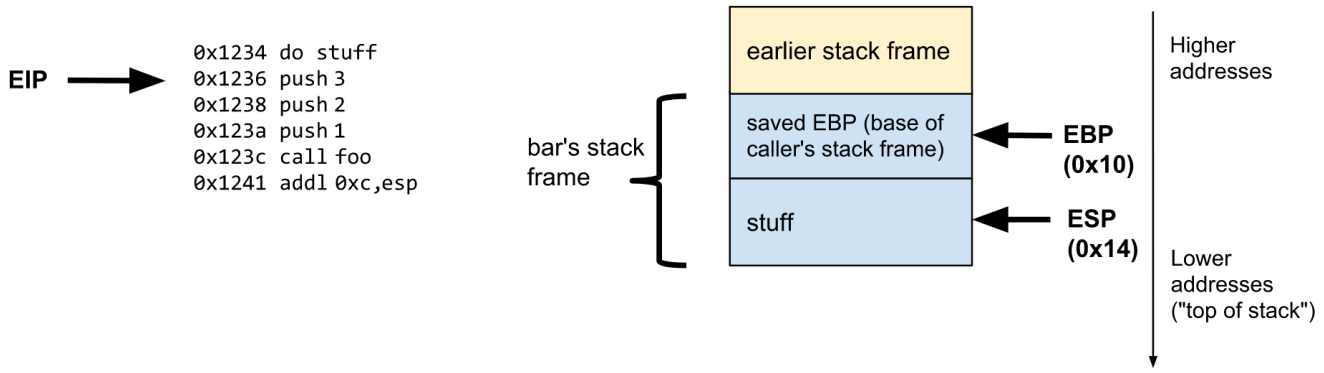
`cdecl`-виклик функції в деталях

```
foo(1, 2, 3);
```

Функція **`foo`** викликана іншою функцією **`bar`** і цей рядок перетвориться у наступний асемблер:

```
push 3
push 2
push 1
call _foo
add esp, 0xc
```

Спочатку подивимось на стан стеку перед викликом **`foo`**:



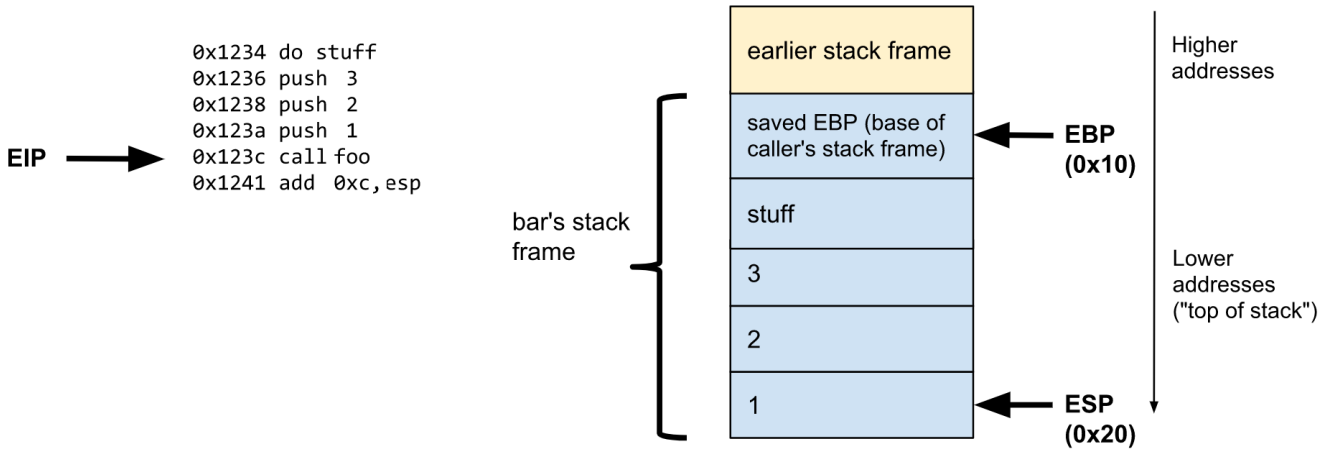
Зауважимо, що тут вказані, звичайно, не фактичні адреси пам'яті.

Одна ділянка пам'яті зберігає стековий кадр (stack frame). **`EBP`** та **`ESP`** вказують на межі стекового кадру, щоб процесор міг виявити знаходження стеку.

Інша ділянка містить **CPU** інструкції, що виконуються. Регістр **`EIP`** (*extended instruction pointer)* вказує на адресу поточної команди. Значення **`EIP`** збільшується на величину, рівну довжині інструкції, для отримання наступної інструкції. Усі команди переходу та команда **`call`** змінюють це значення. На картинках **`EIP`** вказує на інструкцію, яку збираємось виконувати.

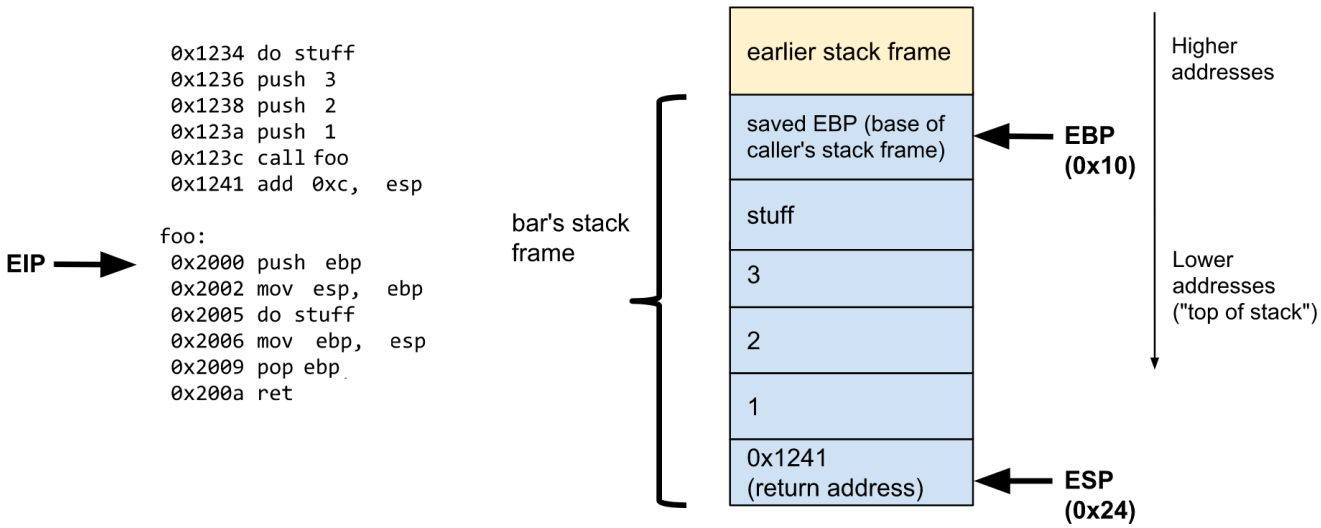
Коли **`bar`** викликає **`foo`**, першим кроком є занесення аргументів функції в стек, де **`foo`** може їх знайти. (Насправді, перший крок має бути збереження значення ***caller-saved*** регістра у стеку, але як вже зазначалося, завдяки обраному способу роботи з регістрами, можна це ігнорувати). Аргументи заносяться до стеку в оберненому порядку (це спрощує обробку функцій зі змінним числом аргументів: локація першого аргументу відома завжди, не зважаючи на кількість аргументів):

```
push 3
push 2
push 1
```



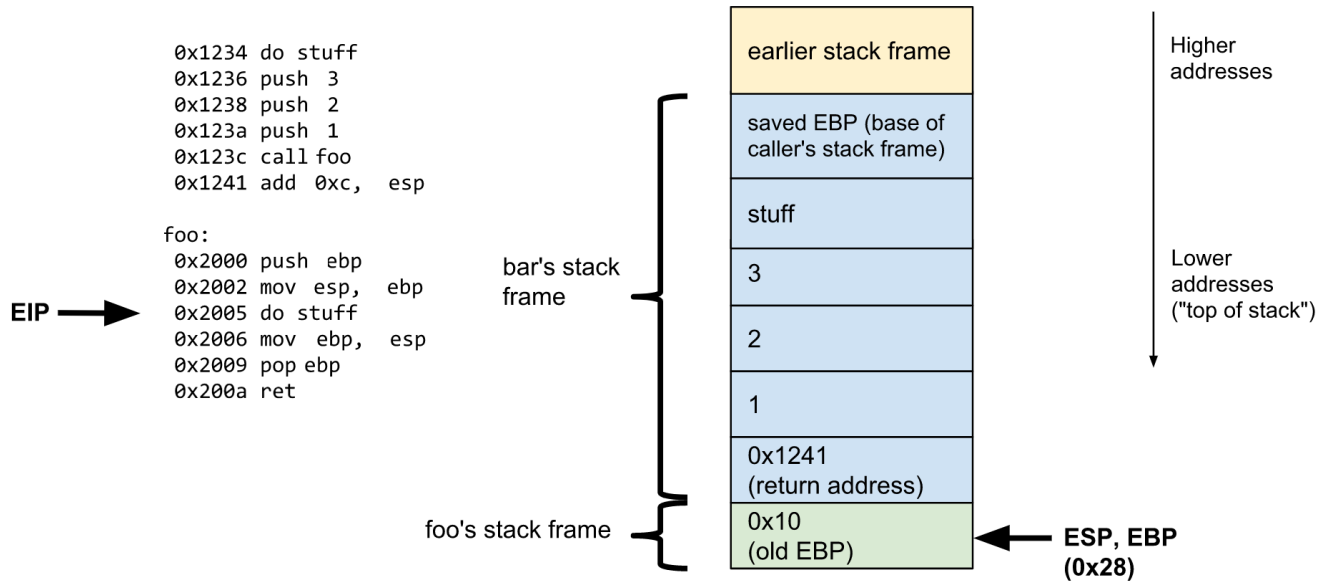
Далі **bar** виконує інструкцію **call**, яка робить дві речі:

- Заносить адресу інструкції, що йде після **call** (return address);
- Переходить на **foo**, заносячи адресу **foo** до **EIP**.



Наступний крок – пролог функції для встановлення нового стекового кадру:

```
push ebp
mov ebp, esp
```



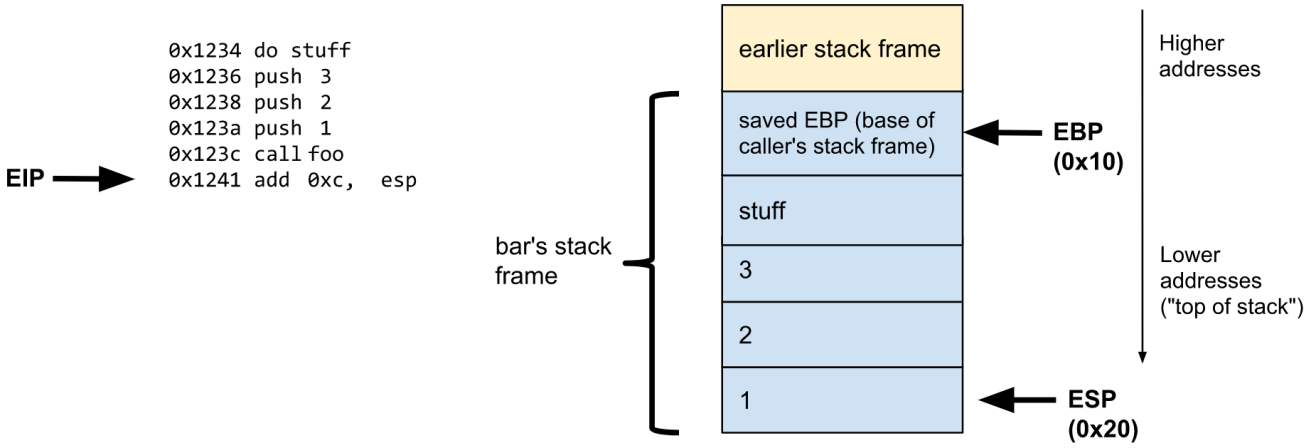
Тепер можливо виконати тіло функції **foo**. Доступитись до параметрів легко завдяки їх відносному розташуванні відносно **EBP:**

EBP + 0x8, EBP + 0xc, EBP + 0x10.

Після того як **foo** було опрацьовано і return value було занесено у **EAX**, повертаємось до функції **bar**, що викликала **foo**. Необхідно, щоб все було так, як до виклику **foo**. Для цього першим ділом запустимо епілог і відновимо старий стековий кадр:

```
mov esp, ebp ;deallocate any local variables on the stack
pop ebp      ;restore old EBP. return address is on top of the stack again
```

Виконується інструкція **ret**, яка бере верхнє значення стеку і переходить на нього безумовним переходом (копіює його до **EIP**).



Залишилось лише видалити аргументи функції зі стеку. Для цього достатньо лише змінити значення **ESP.**

```
add esp, 0xc
```

Отже, можна закінчити виконання функції **bar**.

Перейдемо безпосередньо до генерації коду.

Визначення функцій

Так само як і з **main**, саморобну функцію треба відмітити міткою:

```
foo:
```

Нам вже відомо, як робиться пролог та епілог функцій. Необхідно лише додати параметри функції до **var_map** та **current_scope**. Перший параметр розташується на **ebp + 8**, а кожний наступний на 4 (бо на етапі генерації коду ми працюємо лише з **int**) байти вище попереднього. Псевдокод:

```
param_offset = 8 // first parameter is at EBP + 8
for each function parameter:
    var_map.put(parameter, param offset)
    current_scope.add(parameter)
    param_offset += 4
```

Після цього параметри обробляються як звичайні змінні у тілі функції.

Для **оголошень функції** не генерується жодного асемблеру, оскільки це відбувається при визначенні.

Виклики функцій

Функція, що викликає іншу, повинна:

1. Покласти аргументи в стек в оберненому порядку:
```
for each argument in reversed(function_call.arguments):
     generate_exp(arg) // puts arg in eax
     emit ('push eax')
```
2. Згенерувати інструкцію **call**:
```
emit ('call %s', function_name)
```
3. Видалити аргументи зі стеку після завершення роботи викликаної функції:
```
bytes_to_remove = 4 * number of function arguments
emit (‘add %d, %esp', bytes_to_remove)
```

Завдання

7. Ознайомиться з теоретичними відомостями, додатковою літературою, надати відповідь на контрольні питання. Це допоможе краще розібратися в завданні.
8. Оновити лексер для нових операцій за варіантом.
9. Оновити парсер для обробки функцій та нових операцій. При помилці повинна вказуватись позиція цієї помилки в коді (номер рядка та номер символу).
10. Оновити функцію для генерації асемблерного коду. Вона повинна приймати **AST** та записувати згенерований код до файлу.
11. Протестувати програму на своїх контрольних прикладах, як правильних, так й з помилками.

Протокол повинен містити:

- Тему, мету, завдання що необхідно виконати.
- Лістинг програми.
- Свої контрольні приклади, на яких тестувалась програма з порахованими відповідями.
- Скріншоти з результатами тестування програми та згенерованих кодів.
- Відповіді на контрольні питання та висновки.

Варіанти завдання для самостійної роботи:

Варіант	Операція
1	/=
2	*=
3	/=
4	*=
5	/=
6	*=
7	/=
8	*=
9	%=
10	&=
11	|=
12	^=
13	<<=
14	>>=
15	%=

Варіант	Операція
16	-=
17	Prefix ++
18	+=
19	Postfix ++
20	+=
21	Prefix --
22	-=
23	Postfix --
24	-=
25	&=
26	|=
27	^=
28	<<=
29	>>=
30	%=

Контрольні питання

1. Чим відрізняються функції у **C** та **Python.**
2. В чому різниця між оголошенням та визначенням функції?
3. У якому порядку виконуються оголошення та визначення функції?
4. Чим відрізняються аргументи функції від її параметрів?
5. Коли на етапі парсингу повинна виникати помилка?
6. В чому сутність конвенції виклику функцій, для чого вони потрібні, чим відрізняються?
7. Чим відрізняються ***caller-saved*** та ***callee-saved*** регістри?
8. Який порядок занесення аргументів у стек та чим це пояснюється?

Етап 6
Цикли

Мета етапу: навчитися обробляти одну з найважливіших ідей програмування – **for**, **while**, **do-while** цикли. Їх ще називають **ітераційні вирази** (***iteration statements***).

Теоретичні відомості

`for` loops

Заголовок циклу складається з ***ініціалізації*** (початкового виразу), ***умови роботи циклу*** (контролюючого виразу), ***дії наприкінці кожної ітерації*** (реініціалізації):

```
for (int i = 0; // initial clause
     i < 10;    // controlling expression
     i = i + 1  // post-expression
     ) {
        // do something
}
```

Початковий вираз може бути оголошенням змінної, або звичайним виразом.

```
for (int i = 0; i < 10; i = i + 1) {
    // do something
}
int j;
for (j = 0; j < 10; j++) {
    //do something
}
```

Будь-який вираз в заголовку циклу може бути порожнім. Цикл стане нескінченним:

```
for (;;) {
    //do something
}
```

Але в такому випадку компілятор замінить порожній контролюючий вираз на невід'ємну константу:

```
for (;1;) {           // умова буде завжди true
    //do something
}
```

У Python `for`-цикли реалізуються інакше:

```
for i in range(a, b [,c]):
    # do something
```

Оператор **in** перевіряє входження **i** у діапазон від **a** до **b** з кроком **c**.

range – це функція, яка генерує набір чисел у заданому діапазоні.

a – початок, **b** – кінець, **c** – крок, з яким проходиться цикл. Він може бути будь-яким числом. Наприклад, з кроком **2** результатом наступного коду буде набір чисел «**0**, **2, 4**»:

```
for i in range(0, 5, 2):
    print(i)
```

Для проходження у зворотньому напрямку крок повинен бути від'ємним, а також **a > b**.

a, b, c можуть бути як числом, так і змінною (але вже ініціалізованою). Крок не є обов'язковим. Якщо він не вказаний, він автоматично дорівнює одиниці.

Хоча **range** і є функцією, на це можна не зважати уваги і обробляти її як ключову конструкцію. Аналогічно з мовою **C**, **a** буде початковим виразом, **b** - умовою роботи циклу (контролюючим виразом), а **c** – дією наприкінці кожної ітерації (реініціалізацією). Отже, відмінності від мови **C** будуть лише синтаксичні.

Або можна за власним бажанням підключити функцію зі стандартної бібліотеки, адже ви вже вмієте працювати з функціями т конвенціями їх викликів.

while & do loops

```
// while loop
while (i < 10) {// expression
    i = i + 1;  // statement
}

// do-while loop
do {
    j = j + 1;  // statement
} while (j < 10); // <- the semicolon is required!          expression
```

В мові **Python** конструкція **do-while** відсутня.

break & continue

Вони називаються **стрибковими виразами** або безумовними стрибками.

Вираз **break** всередині циклу призводить до переходу виконання циклу на кінець циклу. Використовується для виходу з циклу.

```
while (1) {
    break; // go to end of loop
}
// statement after break will go here
```

Вираз **continue** переносе виконання на кінець тіла циклу, перед реініціалізацією. У прикладі нижче цикл буде виконуватись **10 разів**, але **do something** виконається тільки для непарних значень змінної **i**.

```
for (int i = 0; i < 10; i = i + 1) {
    if (i % 2)
        continue;
    // do something

    //continue statement will jump here
}
```

LEXING

З’явилось 5 нових лексем для циклу, а також є нові лексеми для нових унарних чи бінарних операцій:

- **for**
- **while**
- **do**
- **break**
- **continue**

Лексеми для мови **Python** можуть відрізнятись.

PARSING

Ініціалізація та оновлення значення в `for` опціональні. Нижче приведене повне **AST** з оновленими частинами.

```
program = Program(function_declaration list)
function_declaration = Function( string,
                                 string list,
                                 types list,
                                 block_item list option)
block-item list =     Statement(statement) |
                      Declaration(declaration)
declaration = Declare(type, string, exp option)
statement =      Return(exp) |
                 Exp(exp option) |
                 For(exp option, exp, exp option, statement) |
                 ForDecl(declaration, exp, exp option, statement)|
                 While(expression, statement) |
                 Do(statement, expression)    |
                 Break |
                 Continue |
                 Compound (block-item list) |
                 Conditional(exp, block-item list, block-item list option)
exp =       Assign(string, exp) |
            Var(string) |
            FunCall(string, exp list) |
            UnOp(unary_operator, exp) |
            BinOp(binary_operator, left_exp, right_exp) |
```

```
            Constant(int, float, char) |
            CondExp(exp, exp, exp)
```

Звернемо увагу, що **AST** припускає наявність **`break`** та **`continue`** за межами циклів, хоча це невірно. Ця помилка оброблятиметься на етапі генерації коду, а не парсингу.

В граматиці для представлення необов'язкового **expression** визначимо нове правило <**exp-option**>:

```
<exp-option> ::= <exp> | ""
```

А далі оновити граматику буде нескладно. Ось повна граматика:

```
<program> ::= { <function> }
<function> ::= "int" <id> "(" [ <type> <id> { "," <type> <id> } ] ")" ( "{"
{ <block-item> } "}" | ";" )
<block-item> ::= <statement> | <declaration>
<statement> ::= "return" <exp> ";" |
                <exp> ";" |
                <exp-option> ";" |
                "for" "(" <exp-option> ";" <exp-option> ";" <exp-option>
")" <statement>|
                "for" "(" <declaration> <exp-option> ";" <exp-option> ")"
<statement> |
                "while" "(" <exp> ")" <statement> |
                "do" <statement> "while" <exp> ";" |
                "break" ";" |
                "continue" ";"|
                "if" "(" <exp> ")" <block-item> [ "else" <block-item> ]
<declaration> ::= <type> <id> [ = <exp> ] ";"
<exp> ::= <id> "=" <exp> | <conditional-exp>
<conditional-exp> ::= <log_or> ["?" <exp> ":" <conditional-exp>]
<log_or> = <log_and> { "||" <log_and> }
<log_and> ::= <bit_or> { "&&" <bit_or> }
<bit_or> ::= <xor> { "|" <xor> }
<xor> ::= <bit_and> { "^" <bit_and> }
<bit_and> ::= <equals> { "&" <equals> }
<equals> ::= <not_equals> { ("==" | "!=") <not_equals> }
<not_equas> ::= <shift> { ("<" | ">" | "<=" | ">=") <shift> }
<shift> ::= <add> { ("<<" | ">>") <add> }
<add> ::= <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/" | "%") <factor> }
<factor> ::=    <function-call> |
                "(" <exp> ")" |
                <unary_op> <factor> |
                id | int | float | char
<function-call> ::= id "(" [ <exp> { "," <exp> } ] ")"
<unary_op> ::= "!" | "~" | "-"
<type> ::= int | float | char
```

Можна помітити, що після першого <**exp-option**> в першому `for` правилі є крапка з комою, а після першого <**declaration**> в другому правилі `for` її немає. Це тому, що правило <**declaration**> саме по собі має крапку з комою і ще один символ не потрібен.

Парсити <**exp-option**> не так просто, як здається на перший погляд. Порожній рядок не є лексемою. Обійти це можна подивившись наперед і перевіривши, чи є наступний токен закриваючою дужкою (у випадку третьої частини заголовку **for**), або крапкою з комою (у випадку виразу/третьої частини циклу/умови). Якщо це так, вираз порожній, якщо ні – то ні.

Але такий підхід спричиняє деякі формальні проблеми стосовно контекстно-вільних граматик та ліво-рекурсивних парсерів: з метою парсингу правила <**exp-option**> , треба дивитися не на перший символ правила, а на другий. Це можна обійти, переписав граматику наступним чином:

```
<exp-option-semicolon> ::= <exp> ";" | ";"
<exp-option-close-paren> ::= <exp> ")" | ")"
<statement> ::= ...
                | <exp-option-semicolon> // null statement
                | "for" "(" <declaration> <exp-option-semicolon> <exp-
option-close-paren> ")" <statement>
                ...
```

Проте, таке рішення прибирає одну проблему, але створює іншу. Тепер маємо невідповідність між граматикою та **AST**. Граматика дозволяє обробляти порожні вирази в заголовку **for**, але **AST** ні.

Як було сказано раніше, порожні вирази повинні замінюватись на невід'ємну константу. Отже, підхід до парсингу контролюючого виразу циклу **for** буде мати вигляд:

```
match parse_optional_exp(controlling_expression) with
    | Some e -> e
    | None -> Const(1) // construct a constant nonzero expression
```

Також можна зробити обробку на етапі генерації коду замість парсингу.

CODE GENERATION

Цикли `while`

```
while (expression)
    statement
```

Алгоритм такої конструкції можна описати наступним чином:

1. Розрахунок **expression**
2. Якщо **false** – перехід на пункт 5
3. Виконання **statement**

4. Перехід на пункт 1
5. Кінець

Тут не буде приведено безпосередньо прикладу коду, бо все що треба вже розглядалось на минулих етапах.

Також треба пам’ятати, що цикл є окремою областю видимості.

Цикли `do`

Вони майже не відрізняються від **`while`** окрім обрахунку **`expression`** після **`statement.`**

Цикли `for`

```
for (init; condition; post-expression)
    statement
```

1. Обрахунок **init**
2. Обрахунок **condition**
3. Якщо false – перехід на пункт 7
4. Виконання **statement**
5. Виконання **post-expression**
6. Перехід на пункт 2
7. Кінець

init та **post-expression** можуть бути порожніми, тоді асемблер не генерується для пунктів 1 і 5.

Заголовок циклу та його тіло – це два різні блоки.

```
int i = 100;     // scope 1
// scope 2 - variable i shadows previous i
for (int i = 0; i < 10; i = i + 1) {
    int i;       //scope 3 - this variable i shadows BOTH previous i's
}
```

`break` & `continue`

Вони дуже просто реалізуються за допомогою інструкції **`jmp`**. Питанням є, куди саме стрибати. **`break`** зупиняє поточний ітераційний блок і переходить на позицію одразу після циклу. Вже існує мітка **`end_of_loop`**, тож можна використати її.

Знадобиться ще одна мітка для **`continue`**.

ЗАВДАННЯ

1. Поновіть лексер для обробки нових лексем.
2. Удоскональте парсер для обробки різних видів циклу та нових операцій.
3. Оновить генератор коду для команд циклів.
4. Протестуйте програму на нових контрольних прикладах та контрольних прикладах з попередніх робіт, занесіть їх результати у протокол. Програма повинна працювати для всіх коректних прикладів та видавати помилку для некоректних. Обробка помилок повинна вказувати місце в коді де сталась помилка.
5. Дайте відповідь на контрольні питання.

Варіант обирається за номером у списку групи:

Варіант	Цикл	Операція
1	for	+
2	while	+
3	do-while	&&
4	for	||
5	while	%
6	for	&
7	do-while	|
8	while	^
9	for	/
10	while	*
11	do-while	/
12	for	*
13	do-while	%
14	while	&
15	For	||

Варіант	Цикл	Операція
16	while	*
17	do-while	/
18	for	~
19	do-while	~
20	While	~
21	for	~
22	while	~
23	do-while	~
24	for	~
25	do-while	+
26	While	-
27	for	+
28	while	-
29	do-while	+
30	for	-

Контрольні питання

1. Назвіть ключові частини циклу ***for*** та особливості генерації їх асемблеру.
2. У чому відмінність циклів мови **Python**?
3. В чому особливість ***стрибкових виразів***?
4. Що таке ***реініціалізація***?
5. Яка проблема відносно контекстно-вільних граматик зустрічається на цьому етапі і в чому полягає рішення?
6. Які конструкції асемблеру використовуються в циклах?
7. Яка помилка парсингу виправляється при генерації коду?

Література:

1. Соломатин Д.И., Копытин А.В., Другалев А.И. Основы синтаксического разбора, построение синтаксических анализаторов. Воронеж, 2014. — 57 с.
2. Орлов С.А. Теория и практика языков программирования. Учебник для вузов. — СПб.: Питер, 2013. — 688 с.
3. Niklaus Wirth - Compiler Construction. URL: https://people.inf.ethz.ch/wirth/CompilerConstruction/
4. Compiler Construction. The Art of Niklaus Wirth. URL: https://www.researchgate.net/profile/Hanspeter_Moessenboeck/publication/221350529_Compiler_Construction_-_The_Art_of_Niklaus_Wirth/links/0deec5213875d84735000000.pdf
5. Windows Assembly Programming Tutorial. URL: https://doc.lagout.org/operating%20system%20/Windows/winasmtut.pdf

Додаткова література:

6. Abdulaziz Ghuloum. An Incremental Approach to Compiler Construction. URL: http://scheme2006.cs.uchicago.edu/11-ghuloum.pdf
7. Making Compiler Design Relevant for Students who will (Most Likely) Never Design a Compiler. URL: https://www2.cs.arizona.edu/classes/cs453/fall20/DOCS/teaching_compilers.pdf
8. Regular expressions in lexing and parsing. URL: https://commandcenter.blogspot.com/2011/08/regular-expressions-in-lexing-and.html
9. Backus–Naur form. URL: https://en.wikipedia.org/wiki/Backus–Naur_form
10. Extended Backus–Naur form. URL: https://en.wikipedia.org/wiki/Extended_Backus–Naur_form
11. Parsing expressions by precedence climbing. URL: https://eli.thegreenplace.net/2012/08/02/parsing-expressions-by-precedence-climbing
12. A recursive descent parser with an infix expression evaluator. URL: https://eli.thegreenplace.net/2009/03/20/a-recursive-descent-parser-with-an-infix-expression-evaluator/
13. Top-Down operator precedence parsing. URL: https://eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing/
14. Kinga Baxter – Clean Code. URL: https://kingadesign.com/clean-code-poster-free-download?fbclid=IwAR2p9y7Rf5gyH586sXYxp2dgoA3kWvRlLO5jfgeHlj77k1Y94qM1grjdW84
15. Hints for Computer System Design. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/acrobat-17.pdf
16. GitHub. Sean Barrett. URL: https://github.com/nothings/stb/blob/master/stretchy_buffer.h

17. International Standart ISO/IEC 9899:201x. Programming languages C. 2011. — 711 c. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
18. The Python Language Reference. URL: https://docs.python.org/3/reference/
19. A Brief And Brisk Overview of Compiler Architecture. URL: https://felixangell.com/blogs/compilers-brief-and-brisk.
20. The Super Tiny Compiler. URL: https://github.com/jamiebuilds/the-super-tiny-compiler
21. YouTube – Bitwise. URL: https://www.youtube.com/user/pervognsen
22. GitHub - pervognsen/bitwise. URL: https://github.com/pervognsen/bitwise
23. Wikipedia - Tree traversal. URL:https://en.wikipedia.org/wiki/Tree_traversal#Post-order
24. Daniel Kusswurm - Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX, 1st Edition. URL: https://www.amazon.com/Modern-X86-Assembly-Language-Programming/dp/1484200659
25. Thomas H. Cormen - Introduction to Algorithms, 3rd Edition (The MIT Press). URL: https://www.amazon.com/Introduction-Algorithms-3rd-MIT-Press/dp/0262033844
26. What's the difference between a statement and an expression in Python? URL: https://www.quora.com/Whats-the-difference-between-a-statement-and-an-expression-in-Python-Why-is-print-%E2%80%98hi%E2%80%99-a-statement-while-other-functions-are-expressions
27. What's the difference between a statement and an expression in C? URL: https://www.quora.com/What-is-the-difference-between-a-statement-and-an-expression-in-C
28. X86 Assembly/X86 Architecture. URL: https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture#EFLAGS_Register
29. Thomas E. Anderson, Michael Dahlin. Operating Systems: Principles and Practice. — Recursive Books, Ltd, 2012. — 469 p. URL: https://www.amazon.com/Operating-Systems-Principles-Thomas-Anderson/dp/0985673524
30. Intel Software Developer’s Manual. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
31. What is the best way to set a register to zero in x86 assembly: xor, mov or and? URL: https://stackoverflow.com/questions/33666617/what-is-the-best-way-to-set-a-register-to-zero-in-x86-assembly-xor-mov-or-and/33668295#33668295
32. Parsing expressions by precedence climbing. URL: https://eli.thegreenplace.net/2012/08/02/parsing-expressions-by-precedence-climbing
33. Top-Down operator precedence parsing. URL: https://eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing

34. Pratt Parsers: Expression Parsing Made Easy. URL: http://journal.stuffwithstuff.com/2011/03/19/pratt-parsers-expression-parsing-made-easy
35. Pratt Parsing and Precedence Climbing Are the Same Algorithm. URL: https://www.oilshell.org/blog/2016/11/01.html
36. Розширена нотація Бекуса-Наура. URL: https://uk.wikipedia.org/wiki/%D0%A0%D0%BE%D0%B7%D1%88%D0%B8%D1%80%D0%B5%D0%BD%D0%B0_%D0%BD%D0%BE%D1%82%D0%B0%D1%86%D1%96%D1%8F_%D0%91%D0%B5%D0%BA%D1%83%D1%81%D0%B0-%D0%9D%D0%B0%D1%83%D1%80%D0%B0
37. Some problems of recursive descent parsers. URL: https://eli.thegreenplace.net/2009/03/14/some-problems-of-recursive-descent-parsers
38. Recursive descent, LL and predictive parsers. URL: https://eli.thegreenplace.net/2008/09/26/recursive-descent-ll-and-predictive-parsers

ДОДАТОК

КОРИСНІ ПОРАДИ

Ось декілька принципів, яких слід дотримуватись при написанні компілятора (та й взагалі будь-якої програми) [4, 14-15]:

1. Використовуйте прості технології.
2. Прості компілятори є більш надійними. Одна людина повинна розуміти весь дизайн компілятора. Компілятор, який не піддається в розумінні одній людині, не гарантує свою правильність і не може підтримуватись без подальших помилок.
3. Приділяйте багато уваги коду, що використовується часто. Не оптимізуйте виняткові випадки. Повністю використовуйте потенціал архітектури, на якій ви працюєте та для якої генеруєте код.
4. Пишіть та генеруйте гарний код з самого початку. Не працюйте з поганим кодом, щоб потім його оптимізувати.
5. Код повинен бути передбачуваним.
6. Те що складно скомпілювати компілятору, також складно зрозуміти людині.
7. Намагайтесь як найбільше скоротити час компіляції. Важлива не тільки генерація компілятором ефективного коду, а й ефективна робота самого компілятора.

Why?	*Functionality* Does it work?	*Speed* Is it fast enough?	*Fault-tolerance* Does it keep working?
Where?			
Completeness	Separate normal and worst case	Shed load End-to-end Safety first	End-to-end
Interface	Do one thing well: Don't generalize Get it right Don't hide power Use procedure arguments Leave it to the client Keep basic interfaces stable Keep a place to stand	Make it fast Split resources Static analysis Dynamic translation	End-to-end Log updates Make actions atomic
Implementation	Plan to throw one away Keep secrets Use a good idea again Divide and conquer	Cache answers Use hints Use brute force Compute in background Batch processing	Make actions atomic Use hints

***

Для зберігання і подальшого виводу токенів на екран використовується структура даних з динамічним розміром (вектор, зв'язний список тощо).

Наприклад, в подібній структурі зберігається вся необхідна інформація про токен, яка потім заноситься в масив.

```
typedef struct Token {
      TokenKind kind;
      const char* start;
      const char* end;
      union {
            int64_t int_val;
            double float_val;
            char ch_val;
      };
} Token;
```

Якщо у якості мови програмування обрана **C**, динамічний масив можна реалізувати у вигляді **Stretchy Buffer**, який запропонував Sean Barrett [16].

Замість динамічних масивів у деяких випадках краще використовувати хеш-таблиці (hash tables). Пошук по ним набагато швидший за лінійний перебір масиву.

***

Задля оптимізації коду і кращої продуктивності використовуйте макроси. Найкраще це робити, якщо код використовується дуже часто і він не надто складний, адже макроси доволі підступні.

***

Для оптимізації роботи з рядками рекомендується використовувати техніку **String Interning**. При порівнянні двох рядків необхідно обійти їх посимвольно і перевірити, чи символи співпадають і чи однакової довжини ці рядки.

Ідея полягає в тому, що існує функція **`str_intern()`**, у якої є наступна властивість:

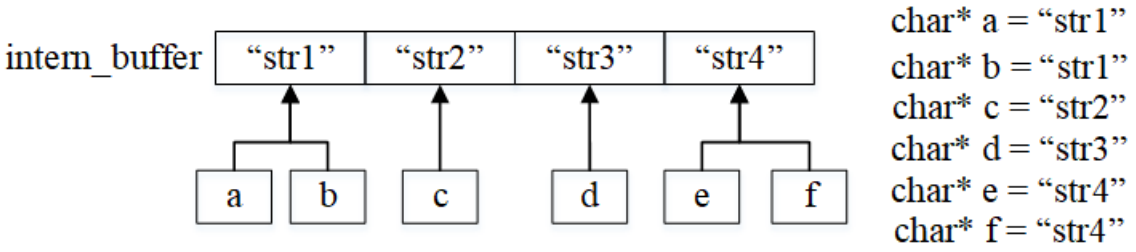
`str_intern(x) == str_intern(y)` тоді і тільки тоді, коли рядки **`x`** та **`y`** однакові.

`str_intern(x)` повертає вказівник на рядок **`x.`** Цей вказівник є повним представленням рядка, бо рядком в мові **C** є вказівник на перший символ рядка.

`str_intern(x)` перевіряє наявність рядка `x` у спеціальному буфері. Якщо ні, виділяє пам'ять у буфері, копіює туди вміст рядка **`x`** і повертає вказівник на цю пам'ять.

При наступному виклику **`str_intern(x)`** пам'ять виділятись не буде. Замість цього повернеться вже існуючий вказівник на вже існуючу ділянку пам'яті.

Отже, два різні об'єкти, що представляють собою один і той самий рядок, насправді будуть представлені одним об'єктом. І коли однаковий рядок розміром, скажімо, 4 байти зустрічається 100 разів, замість створення 100 різних об'єктів і виділення на них 400 байт буде створено один об'єкт розміром 4 байти. Це дуже скоротить використання пам'яті і дозволить порівнювати рядки набагато швидше. Річ у тому, що робити посимвольне порівняння знадобиться лише один раз. Після цього рівність рядків підтверджується рівністю вказівників.



Слід пам’ятати, що описаний вище метод дійсний лише для незмінних, константних рядків. Також реалізація методу відрізняється від мови до мови, в мові С його необхідно реалізовувати самостійно.

***

Перед тим як парсити програму, треба мати список ключових слів, щоб відрізняти їх від звичайних імен. Можна просто ініціалізувати незмінний масив з цих слів і послідовно проходити його, перевіряючи токен на відповідність ключовому слову. Цей спосіб найпростіший, але й найповільніший.

У нагоді стане вище описаний метод **String Interning**. При ініціалізації ключових слів вони послідовно додаються у буфер рядків, послідовно займаючи неперервну ділянку пам’яті. Після сканування токену він додається до буфера, або вказівник посилається на існуючу ділянку пам’яті. Якщо вказівник лежить в межах ділянки ключових слів, токен є ключовим словом. Отже, достатньо лише порівняти вказівники, а не рядки посимвольно.

***

Не рекомендується виділяти пам’ять за вимогою, тобто виділяти її окремо при кожному новому створенні об’єкту тощо. Достатньо виділити задовільний об’єм пам’яті один раз, а потім розширювати його якщо це буде необхідним. Наприклад, необхідно створити 10 об’єктів розміром 1024 байти. Краще один раз виділити 1 – 2 Кб, ніж 10 разів виділяти по 1024 байти. Це значно скоротить час виконання програми, зменшить фрагментацію пам’яті та дозволить використовувати техніки доступу до окремих об’єктів, які самі по собі позитивно вплинуть на продуктивність (наприклад, string interning).

***

Обробка помилок не повинна "смітити" в коді, не повинна складатись з величезної кількості виразів та уповільнювати компілятор, а повинна бути елегантно вбудована в основну логіку парсингу.

***

Хеш-таблиця або хеш-карта (Hash Table or Hash Map) зберігає дані парами ключ – значення.

Хеш-функції приймають ключ і повертають значення, що є унікальним і єдиним для цього ключа. Цей принцип відомий як хешування. Хеш-функції повертають унікальну адресу пам’яті необхідних даних.

Хеш-таблиці створені для оптимізації пошуку, вставки та видалення. Хеш-колізія – це коли хеш-функція повертає одне й те саме значення для двох різних вхідних даних. Усі хеш-функції мають цю проблему. Зазвичай причиною цьому є дуже великий розмір таблиці.

Часова складність:

Індексування: O(1).

Пошук: O(1).

Вставка: O(1).

При написанні програми не обов’язково одразу реалізовувати хеш-таблиці. Спочатку можна реалізувати необхідний функціонал якимись більш простими методами (буферами, динамічними масивами тощо), навіть якщо вони працюють повільніше, а вже на фінальній стадії оптимізувати код використанням хешування.