Передмова

До дисципліни «Системне програмування-2» входить цикл з 6 лабораторних робіт, у яких відпрацьовуються основні кроки розробки компілятору:

- ЛР 1 – Розробка та опрацювання базової структури компілятора
- ЛР 2 – Унарні та бінарні операції
- ЛР 3 – Обробка змінних. Локальні змінні.
- ЛР 4 – Вкладені конструкції
- ЛР 5 – Оператори розгалуження, тернарний оператор
- ЛР 6 – Цикли

Таким чином, кожна наступна лабораторна робота базується на знаннях, що отримані під час виконання попередньої, та у свою чергу слугує певним етапом побудови компілятору з більш розширеним функціоналом.

Компілятори відносяться до системного програмного забезпечення. Зазвичай під поняттям «розробка компілятору» розуміється щось дуже складне та незрозуміле. Це дійсно здається непростим завданням, але лише на перший погляд. 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 тощо.

За власним бажанням та за згодою викладача завдання можуть бути виконані з використанням будь якої мові програмування. Пояснення же викладаються на класичній мові С та псевдокодом, причому псевдокод різних форм та формулювань.

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

У мові С 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 С. 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 С з іншим синтаксисом і ця мова обрана лише у навчальних цілях.

Описи лабораторних робіт 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 методичні вказівки до виконання кожної лабораторної роботи мають схожу структуру та складаються з

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

У подальшому отримані знання та навички знайдуть своє застосування під час виконання курсової роботи або РГР.

Лабораторна робота № 1
Розробка та опрацювання базової структури компілятора

Мета роботи: 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.

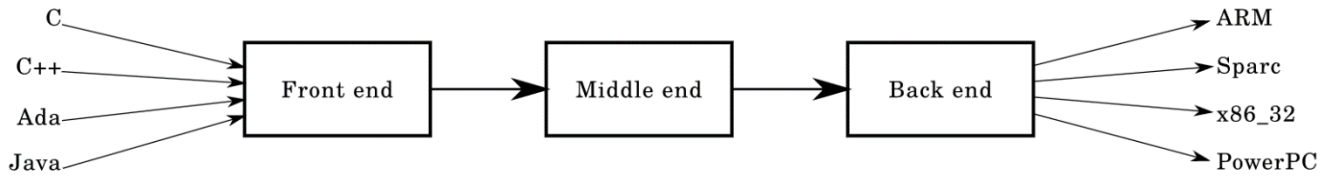
У програмі присутня тільки одна функція **`main`**, що складається лише з одного виразу **`return`**. Єдине що буде відрізнятися – це **тип**, **значення та система числення** числа, що повертається.

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 C 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:

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

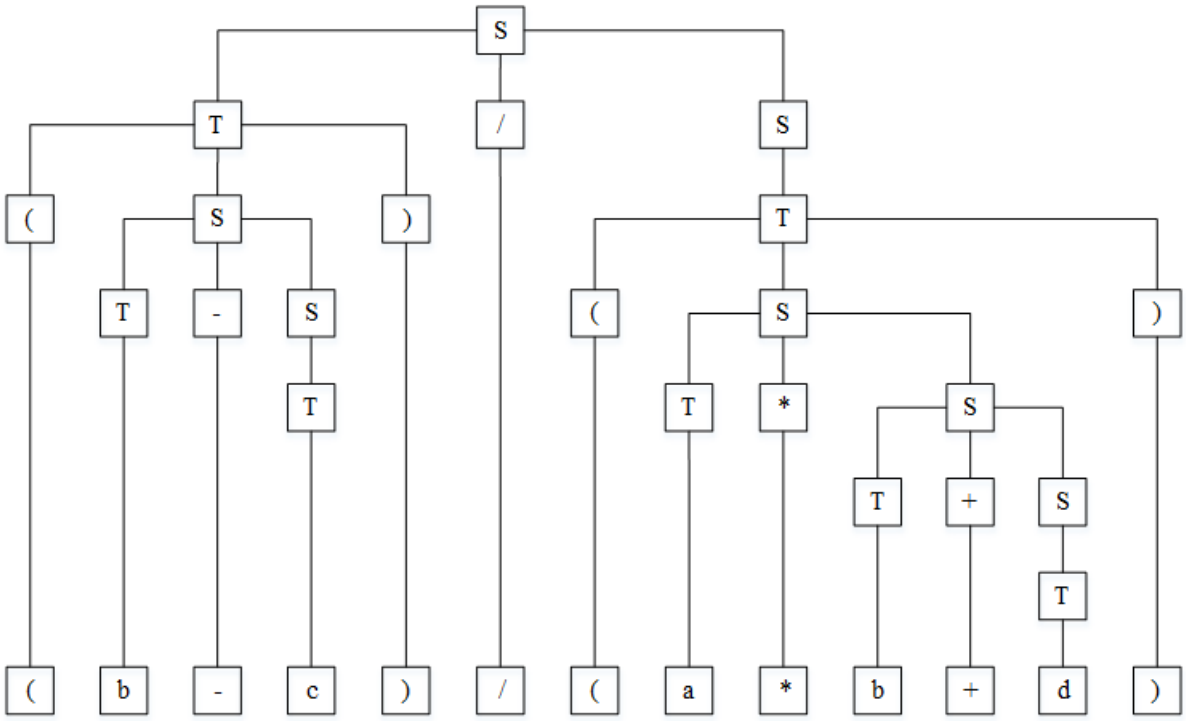
де ***a***, ***b***, ***c, d,*** **(,),+, -, *, /** - термінальні символи;

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.

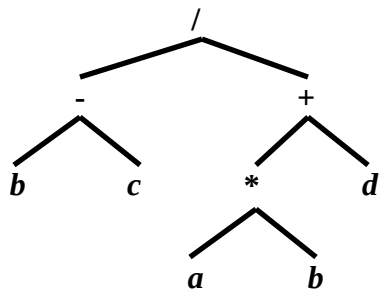
В якості вузлів у **AST** виступають оператори, до яких приєднуються їх аргументи, які у свою чергу також можуть бути складеними вузлами. У багатьох мовах програмування такі мовні конструкції, як розгалуження, об'явлення функцій тощо складаються з простіших елементів – змінних, констант. **AST** відображає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

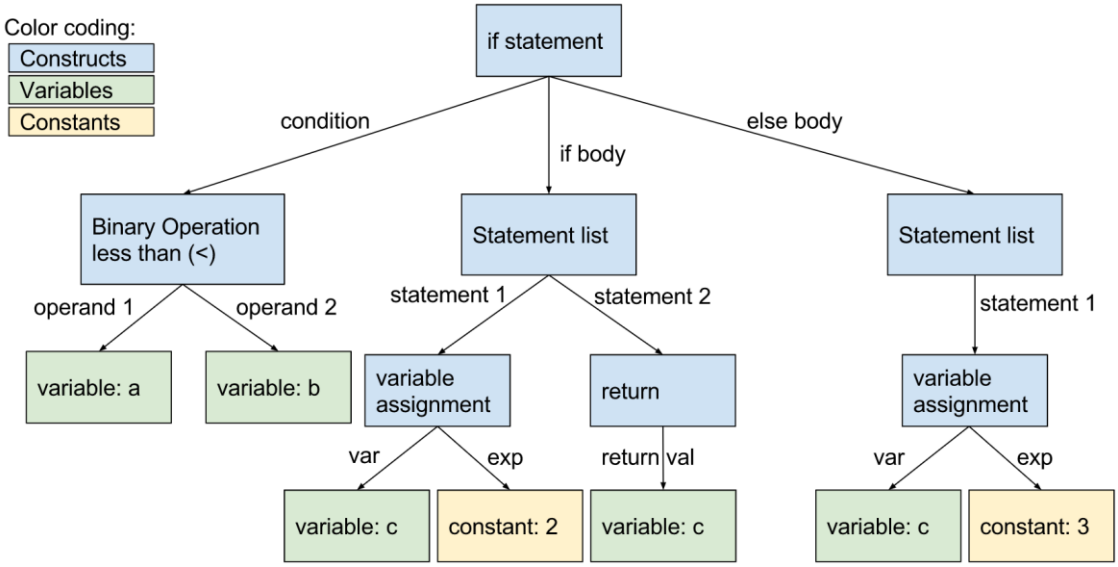
- умова **a < b**
- тіло **if** **c = 2; return c**;
- тіло **else** **c = 3**;

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**:

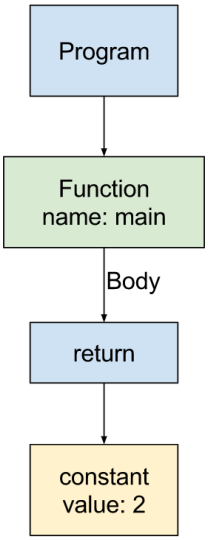
```
//create if condition
cond = BinaryOp(op='>', operand_1=Var(a), operand_2=Var(b))
//create if body
assign = Assignment(var=Var(c), rhs=Const(2))
return = Return(val=Var(c))
if_body = [assign, return]
//create else body
assign_else = Assignment(var=Var(c), rhs=Const(3))
else_body = [assign_else]
//construct 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 у цієї лаб. роботі 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. У наступних роботах 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 в роботі.
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 лабораторної роботи 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 лабораторної роботи та виводити правильний результат.
6. Протестувати програму на декількох своїх контрольних прикладах та відобразити результати тестування у звіті.

Звіт повинен містити:

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

Варіант обирається за порядковим номером у списку групи:

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 роботи?
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?

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:

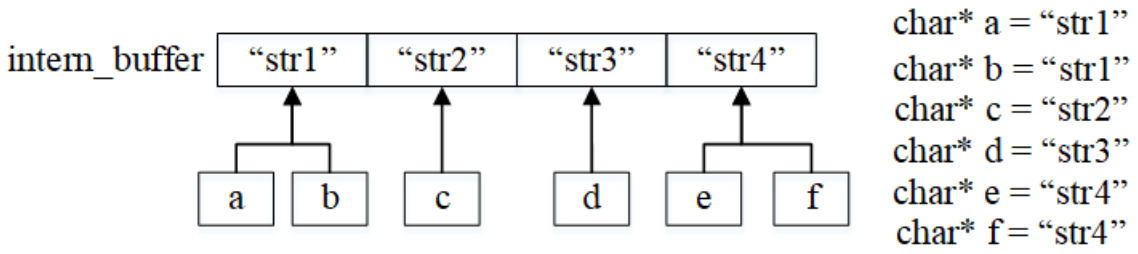
`str_intern(x) == str_intern(y)` тоді і тільки тоді, коли рядки **`x`** та **`y`** однакові.

`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.