Реалізація компілятора базової частини адресної мови

Текстова частина до курсової роботи

за спеціальністю «Інженерія Програмного Забезпечення»

121

Керівник курсової роботи кандидат фізико-математичних наук, старший викладач Ю.Ющенко

_____

(підпис)

“___” _____ 2022 року

Виконала студентка ІПЗ-4

Чередник К.В.

“___” _____ 2022 року

Київ 202

Календарний план виконання курсової роботи

N° п/п	Назва етапу курсової роботи	Термін виконання	Примітка
1	Отримання завдання на курсову роботу	15.10.2021	
2.	Пошук тематичної наукової літератури	10.12.2021	
3.	Ознайомлення з літературою з побудови компілятора	18.12.2021	
4.	Ознайомлення з джерелами з Адресної мови	10.01.2022	
5.	Реалізація базової частини компілятора адресної мови	10.02.2022	
6.	Реалізація основного функціоналу для компіляції адресної мови	01.03.2022	
7.	Опис результатів	10.04.2022	
8.	Реалізація додаткового функціоналу для компіляції адресної мови	20.04.2022	
9.	Опис результатів	30.04.2022	
10.	Остаточне тестування програми	01.05.2022	
11.	Надання роботи керівнику для перевірки	05.05.2022	
12.	Коригування роботи відповідно до зауважень керівника	10.05.2022	

Зміст

Анотація

Мета даної курсової роботи - реалізація базової частини Адресної мови програмування. Це завдання не є новим, оскільки у 1955 році вже було створено перший у світі компілятор Адресної мови програмування - Програмуючу Програму Адресної мови для ЕОМ "Київ". В цій роботі компілятор реалізовується з використанням високорівневої мови програмування загального призначення C++. Використовуються лише базові бібліотеки, такі як <vector> та <string>. Описується процес створення всіх класів, що відповідають за компіляцію та обробку скомпільованого коду.

Вступ

Адресна мова програмування є придатною для роботи з завданнями у різних сферах, пов’язаних з виконанням арифметичних та інформаційно-логічних завдань, таких як розпізнавання образів, програмуючі програми, завдання зі сфери економічної кібернетики тощо. Вона може бути використана для опису електронно-обчислювальних цифрових машин та систем автопрограмування.

Адресна мова програмування застосовувалася для розширення можливостей комп’ютера “Київ”. Самі по собі групові операції комп’ютера “Київ” є засобом для організації циклічних процесів, які присутні у високорівневих мовах програмування (заголовок циклу та тіло циклу).

Варто зауважити, що система команд вище згаданого комп’ютера мала більше засобів та можливостей, ніж перша мова програмування високого рівня «Планкалькюль» Конрада Цузе (1944 р.). Це було обумовлено тим, що комп’ютер “Київ” з його системою команд окрім операцій “засилання” (присвоєння), умовних та безумовних переходів мав також засоби організації циклічних процесів на кшталт операторів циклів імперативних мов програмування високого рівня, а також засоби опосередкованої адресації (вказівники).

З використанням адресної мови програмування на комп’ютері “Київ” було створено першу у світі табличну базу даних реляційного типу. Саме завдяки присутності у Адресній мові програмування засобів декларативного програмування, цією мовою можна було зафіксувати результати виконання задач, що записуються основними конструкціями мови SQL - JOIN. Також адреси даної

мови програмування дозволяють зв’язати згруповані дані та є альтернативою FOREIGN KEY та PRIMARY KEY, які використовуються у базах даних.

Не слід оминати увагою факт, що саме на Адресній мові програмування вперше в світі було реалізовано задачі розпізнавання образів методом машинного навчання, а зроблено це було 1950-х - 1960-х роках, що значно випередило свій час. Завдяки реалізації можливості розпізнавання образів комп’ютер “Київ” мав змогу розпізнавати друковані та рукописні літери та цифри.

Перша у світі смарт техніка також була розроблена з використанням даної мови. Це була розумна домна у місті Дніпродзержинськ. Ідея полягала у тому, що комп'ютер, який знаходився у Києві, міг керувати процесом виплавлення сталі на цій домні, яка знаходилася приблизно за 500 кілометрів від нього. Дана ідея була реалізована з використаням сигналів датчиків доменої печі, які передавалися телеграфними лініями на комп’ютер “Київ”, який у свою чергу через ті самі лінії передавав керівні сигнали про вимкнення, підливання чи підмішування чогось до цеху. Вже в ті часи з використанням Адресної мови програмування та комп’ютера “Київ” було реалізовано щось на кшталт технології “Internet of Things”.

Розділ 1. Дослідження предметної області

1.1 Основні поняття та засоби адресної мови

1.1.1 Позначення

Адресна мова близька до мови звичайних формул. В ній наявні всі символи, якими в математиці визначають величини, вектори, функції та множини, тобто дужки, цифри, літери з індексами та без них, а також всі математичні операції, такі як +, -, *, :, $\sqrt{}$, ln і т.д.

В адресну мову вводяться спеціальні поняття та відповідні символи, завдяки яким вона стає найбільш пристосованою для опису алгоритмів.

Адресний запис алгоритму складається із рядків.
У кожному рядку записується одне чи кілька алгоритмічних дій. Запис кожної дії називається формулою.

Рядки можна записувати як використовуючи перехід на інший рядок, тобто один під іншим, так і лінійно, поставивши між ними крапку. Дії алгоритму відбуваються послідовно у порядку їх запису, проте цей порядок може бути змінено, застосовуючи певні формули.

Опис програми може супроводжуватися коментарями, які беруться в спеціальні квадратні дужки [].

1.1.2 Мітки та відмічені рядки

Міткою вважається цифра, літера, слово або кілька слів, складених з цифр та літер з індексами та без.
Мітками є 214, А31, $К_1$

Після мітки ставиться трикрапка, що позначає, що рядок, най якому розташована мітка, відмічен цією міткою.
Коли після мітки немає трикрапки, вона є міткою безумовного переходу. Виконання програми переходить на рядок, помічений цією міткою.

Один рядок може бути помічний багатьма мітками, після кожної мітки ставиться трикрапка.

1.1.3 Послідовність виконання дій

В одному рядку може бути записано кілька формул засилки або обміну. В такому випадку між ними ставить крапка с комою або кома. У другому випадку дозволяється змінювати послідовність виконання формул місцями.

В кінці списку формул в одному рядку після крапки з комою може стояти формула обчислюваного переходу. Такий рядок називається списком формул з переходом та визначає виконання всіх перерахованих у списку дій та переходом за останньою формулою.

1.2 Операції

1.2.1 Штрих-операція та адресне відображення

Основним поняттям адресної мови є штрих-операція, що задає відображення множини адресів А на множину Б, що називається множиною вмісту цих адресів.
Операція записується як **‘а = б**, де а належить множині адресів, б є вмістом адреси а. Такий запис можна читати як:

1. Штрих а дорівнює б.
2. а - адреса б.

3. а містить б.
4. б - вміст а.

Штрих-операція є функціональним відображенням, тобто одному адресу в один момент часу відповідає лише один вміст. Штрих-операція вважається алгоритмічно виконуваною, тобто за однією адресою завжди можна визначити її вміст.

Якщо а не належить множині адресів, то запис "а не має змісту.

Штрих-операцію можна застосовувати повторно кілька разів, поки її операнд належить множині адрес.

Запис ^{н}а = б читається як а - є адресою н рангу для б.

Для кожного елементу б б є адресою 0 рангу для б.

1.2.2 Операція засилання

Адресне відображення також можна встановити через операцію засилання **б => а**, що рівнозначно "а = б.

1.2.3 Операція слідування

Для множини упорядкованих елементів вводиться операція слідування, що позначається як С. Са позначає наступний елемент після а в упорядкованій множині. В загальному **$С^{н}$ а** позначає н-тий елемент після а в упорядкованій множині. **$С^{-н}$α** означає елемент, що передує елементу а

В кожному конкретному випадку операція слідування повинна мати алгоритмічний запис. Наприклад, в множині натуральних чисел Н, С може бути визначена як Са = а + 1.

1.2.4 Адресна функція

Вираз побудований з використанням штрих-функції.

Окремим випадком адресної функції вважається вираз, що не містить штрих-операції. У такому випадку вважається, що штрих-операція 0-го рангу.

Ранг штрих функції **R** визначається найбільшим рангом штрих-операції, що в ній знаходяться.

1.3 Формули, що не впливають на хід виконання програми

1.3.1 Формули зупину

Формула безумовного зупину записується як **!** та позначає безумовне завершення виконання програми.

Формула відносного зупину позначається як визначає кінець алгоритму. Якщо перед виконанням була визвана формула входження, програма повертається до виконання наступного після формули входженням рядка. У цьому разі еквівалентно інструкції return з функції, що була викликана через **П**.

1.3.2 Формула засилання

Аналогічно операції засилання, формула засилання **f1 => f2** визначає наступні алгоритмічні дії: значення адресної функції f1 засилається за адресою, що обраховує f2.

Наприклад, нехай "a = 2; `b = 0; Тоді у результаті виконання "a + 3 =>  c + `б, c буде містити значення 5.

Допускається запис a=> b; => "c;=> d + 2, що є еквівалетним до a=> b; a=> "c; a=> d + 2

1.3.3 Формула обміну

Вираз виду **f1 <=> f2**, де f1, f2 - адресні функції є формулою обміну. У результаті її виконання проводиться обмін вмістом вирахованих адрес.

Виконання цієї формули еквівалентно наступній послідовності дій:

(1)f1 => temp1 ; "f2 => temp2; "f1 => f2; temp2 => temp1,

що може не бути еквівалентним до

(2) "f1 => temp ; "f2 => f1; "temp => f2

у випадку, коли f2 не залежить від f1.

Наприклад:

"a = 3; b = "a + 1; "4 = 8

Після виконання формули обміну у випадку (1) отримуємо:"a = 8; "4 = 3

у випадку (2) отримуємо "a => temp, "b => "("f + 1) = 8 => temp2, "a = 3 => b = 4; "temp2 = 8 => a; `a = 8, a "9 = 3.

1.4 Формули, що впливають на хід виконання програми

Такі формули завжди записуються з нового рядка.

1.4.1 Формула обчислюваного переходу

Запис адресної функції, значеннями, якої можуть бути лише мітки визначає окрему алгоритмічну дію, що має назву обчислюваним переходом. При цьому виконання програми переходить до рядку, поміченому міткою, отриманою у результаті обчислення.

Наприклад, якщо є рядки, помічені мітками 4 та 6, та допустимими значеннями для "а є 3 та 5, тоді допустимою буде формула обчислюваного переходу "а + 1.

1.4.2 Формула відносного переходу

Формула виду ↕**н**, де н - адресна функція, значенням якої є будь-яке ціле число. При виконанні формули відбувається перехід на визначену кількість рядків алгоритму вниз (якщо значення додатнє) або вгору (якщо від"ємне).

1.4.3 Формула входження

Формули входження застосовуються для переходу до виконання перетворення, що знаходиться у іншому місці запису програми. Саме перетворення називається підпрограмою. В описі підпрограми повинні бути вказані

1) мітка, що присвоєна даній підпрограмі
2) точка закінчення обрахунків (формула відносного зупину)
3) впорядкований список вхідних та вихідних параметрів

Підпрограми в адресній мові починаються з рядка, типу

∅ => c_1 , … , … ∅ => c_n

Замість символу ∅ під час виконання програми ∅ замінюють на елементи списку, що входять в формулу входження.

Сама формула входження має вигляд

П а {a_1, a_2 … $a_н$}б, де

а - мітка початку підпрограми, куди треба перейти

б - мітка куди потрібно перейти після закінчення виконання підпрограми

а1 … ам (м <= н) 0 - вхідні аргументи підпрограми, стають на місце ∅ під час її виконання.

а(м+1) … ан - адреси, куди після завершення виконання підпрограми засилається впорядкований ряд результатів.

Якщо після виконання підпрограми, потрібно перейти на наступний після формули входження рядок мітка б опускається.

1.4.4 Предикатна формула

Відповідна до предикатної формули **P {L} a ↓ b** алгоритмічна дія полягає в виконання рядка **a**, якщо вираз **L** дійсний, або рядка **b** інакше.

Допускаються такі види формули:

P {L} a ↓ b

P {L} a - рядок **a** виконується якщо значення **L** дійсне

P {L} ↓ b - рядок **b** виконується якщо значення **L** хибне

1.4.5 Формула заміни

Алгоритмічна дія визначена формулою заміни **З{a_1 > c_1, … , a_n -> c_n} a,b, c** полягає в виконанні рядків алгоритму, що обмежені мітками **a** та **b** з попередньою заміною всіх a_i на c_i.

Можливо вийти с формули заміни у результаті її вичерпання або в результаті переходу через формули переходу. В обох випадках після виходу з області дії формули заміни початковий запис алгоритму відновлюється.

c - мітка рядку переходу після виконання формули заміни, якщо цей рядок є наступним після формули, то ця мітка опускається.

Як **a** та **b** можуть використовуватися як символи адрес, так і символи знаків операції. Запис виду **З{- > +} a,b, c** є валідним та

замінює всі операції віднімання на додавання в межах дії формули заміни.

1.4.6 Формула циклювання

Формула **Ц {a, C ∅, P {L} => □ } b, l** визначає циклічне виконання алгоритму обмеженого цією формулою та міткою **b** з перебором елементів та розміщені їх за адресою **□.** Після завершення циклу програма переходит до виконання рядка, позначеного міткою **l.**

a - перший елемент циклу

C ∅ - операція слідування

P{L} - вираз, при хибності якого цикл завершується

Мітка **l** може бути опущена, якщо по завершенні циклу потрібно перейти на наступний після мітки **b** рядок. Мітка **b** також може бути опущена, якщо тіло циклу складається з одного рядка, при цьому кома перед міткою **l** зберігається. ЯКщо обидва мітки опущені, кома також прибирається.

Рядки алгоритму

Ц {a, C ∅, P {L} => □ } b, l

Ф("□)

a …

еквівалентні наступному запису:

a => □

b … P{L} ↓ l

Ф("□)

C"(□) => □

b

Якщо в тілі циклу містяться формули входження, заміни та циклювання, то їх область дії входить в область дії циклу.

Є різні видозміни запису циклу:

1) Замість циклювання по адресу => □ використовують -> □, коли □ є параметром.
2) Цикл можна обмежувати, як вказавши предикатну формулу P{L}**,** так і зазначиви останній елемент множини.

   Ц {a_1, С ∅, a_n => □ } b, l

3) Для множини з операцією слідування, визначеною співвідношенням С ∅ => ∅ + с застосовують запис

   Ц { a (c) P=>□$_1$ }b, l
4) Можна одночасно перебирати кілька різних множин, розділивши їх точкою з комою

   Ц{ a_1 (b_1) P(L_1)=>□$_1$ }; a_2 (b_2) c_2 => □$_2$} b, l;
5) Допускається циклювання за однією адресою □ по множині, в якій для початкової групи елементів визначено одну операцію слідування, а для інших груп - інші.

   Ц{ a_1 (b_1) P(L_1); a_2 (b_2) c_2 => □$_2$} b, l;
6) Елементи множини значень параметру можуть задаватися безпосередньо перебором цих елементів.

   Ц{ a_1 ; a_2 ; .. ; a_n => □} b, l;
7) Для циклювання по параметру, що приймає значення 1,2 … n , де n - адресна функція, значення якої є цілими додатними числами використовують запис

   Ц{ (n) -> □} b, l;

   Для перебору натуральних чисел в оберненому порядку (тобто n, n -1 … 0) використовують запис

Ц{ -(n) -> □} b, l;

Розділ 2. Видозміна синтаксису

Оскільки в синтаксисі адресної мови багато символів та літер, що не доступні при на клавіатурі, для пришвидшення написання програм на адресній мові на комп'ютері було проведено певні заміни в позначеннях.

Назва формули	**Оригінальний синтаксис**	**Його заміна**	**Обгрунтування**
Формула відносного зупину	□	B	Схожість символів
Формула відносного переходу	↕n	|n	Схожість символів
Формула входження	П a {… }б	P a {… }b	Program
Предикатна формула	P {L} a ↓ b	Pr {L} a | b	Predicate
Формула заміни	З{ …}	R{ …}	Replacement

Формула циклювання	`Ц{ …}`	`L{ …}`	Loop
Параметр підпрограми	∅	`NIL`	Читання символу

Розділ 3. Основна концепція побудови компілятору для мови програмування

3.1 Сканування тексту

Сканування тексту проводить паралельно з виконанням програми.
Сканується по одному токену за раз за потреби.
Кожному токену відповідає один з елементів переліку `TokenType`

TokenType	Представлення токену в тексті
NEW_LINE	перехід на новий рядок \n
INLINE_DIVIDER	, або ;
MINUS	-
PLUS	+
SLASH	/
STAR	*
DOT	.
L	L
B	B
PR	Pr

R	R
LEFT_PAREN	(
RIGHT_PAREN	)
LEFT_CURLY	{
RIGHT_CURLY	}
HORIZONTAL	|
BANG	!
BANG_EQUAL	!=
EQUAL	=
EQUAL_EQUAL	==
GREATER	>
MINUS_GREATER	->
GREATER_EQUAL	>=
LESS	<
LESS_EQUAL	<=
SINGLE_QUOTE	`
EQUAL_GREATER	=>
DOTS_3	…
LESS_EQUAL_GREATER	<=>
IDENTIFIER	послідовність з літер та цифр
NUMBER	раціональне число
ERROR	символ, якого не має в переліку допустимих

EOF	кінець файлу/вводу
TRUE	true
FALSE	false
PRINT	print

Для сканування застосовується свій класс `Scanner`

При роботі з ним основним методом є scanToken - передає наступний токен в тексті, що вдалося обробити. Для обробки використовуються функції , що допомогають посимвольно визначити тип токену.

```
char advance()
```

перейти на наступний символ в тексті та повернути поточний

```
bool match(char expected)
```

перейти на наступний символ в тексті, якщо поточний символ збігається з параметром

```
char peek()
```

повернути поточний символ без просування по тексту

```
char peekNext()
```

повернути наступний символ без просування по тексту

Якщо було визначено, що токеном є послідовність з літер та цифр, то такий токен перевіряється на приналежність до одного з ключових слів (true, false, L, Pr і т.д.), а вже потім повертається токен зі встановленим для нього типом. Сам токен повертається у вигляді структури `Token` з полями

```
TokenType type
```

тип токену (один з перелічених)

`const char* start{nullptr}`

вказівник на початок токену

`int length{0}`

довжина символа (лише роспізнання для ідентифікаторів та чисел)

`int line{-1}`

номер рядку, де цей токен зустрівся

3.2 Парсинг

Парсинг відбувається з використанням структури `Parser,` що покладається на роботу від Scanner. Парсер містить низку корисних методів

`Token advance()`

просканувати наступний токен та повернути поточний

`bool match(TokenType type)`

метод використовується для перевірки на тип токену. Якщо тип поточного токену збігається з параметром, просканувати наступний токен, повернути чи збігається тип.

`void consume(TokenType type, const char*errMsg)`

метод використовується, коли існує лише один допустимий тип поточного токену, інакше повідомляється про виявлення помилки

`bool peek(TokenType type) const` – метод перевіряє, чи збігається тип поточного токену з даним, наступний токен не сканується

3.3 Компіляція в байткод

Компілятор має два завдання. Він аналізує вихідний код користувача, щоб зрозуміти, що він означає. Потім він бере ці знання і виводить низькорівневі інструкції, які виробляють потрібну семантику.

В багатьох мовах програмування ці завдання розділяються на дві частини. Спочатку будується AST дерево, а потім генератор коду проходить це AST дерево та видає результуючий код.

При написанні цієї роботи, було обрано інший підхід до побудови компілятора. Обробка вхідного тексту та вивід коду буде здійснена за один прохід. Такий підхід має назву single pass compilation.

Як і при написанні компілятора, що працює за AST-деревом, для будь-якої мови програмування спочатку слід визначити її граматику.

Для логічної мови програмування, граматика має наступний вигляд:

```
program ->          oneline ("\n" oneline)*  EOF | OEF

oneline ->                  (identifier …)* oneline_body

oneline_body ->             complex         |
                            simple ( ";" simple )*
```

```
complex_formula ->      program     |
                        replacement |
                        predicate   |
                        loop        |
                        print       | return

simple_formula ->       exchange    |
                        refer       |
                        pointer     |
                        expression

expression ->           logic_or

logic_or ->             logic_and  ( "or" logic_and )*

logic_and ->            equality ( "and" equality )*

equality ->             comparison ( ("!="| "==") comparison )*

comparison ->           term ( (">" | ">=" | "<"| "<=") term)*

term ->                 factor ( ("*"|"/") factor )*

factor ->               unary ( ("-"|"+") unary )*

unary ->                ("-"| "\"") unary | primary

primary ->              "true" | "false" | number | identifier | "(" expression ")"

program ->              "P" identifier "{" (expression (";" expression)* )? "}" identifier?

replacement ->          "R" "{" ((primary -> primary)|(symbol -> symbol) )* "}" identifier
identiifer    identifier

predicate ->            "P" "{" expression "}" expression "|" expression
```

loop -> "L" "{" "}" identifier identifier |
loop_header -> expression "(" expression ")" expression ("=>" identifier)*
expression "(" expression ")" "L" "{" expression "}" " =>" identifier

exchange -> expression <=> expression
refer -> expression => expression
pointer -> " ’ ”expression

Для того, щоб можна було побачити результат роботи програми вводить окремий вид statement - print statement. Це єдиний спосіб для програми вивести значення на екран.

При розробці компілятору бралась до уваги як можливість запустити програму для компілювання певного файлу, так і можливість почергово вводити код з консолі.
Компіляція програми починається з методу

```
bool Compiler::compile(const char*source, Chunk* chunk)
```

Він приймає вхідний текст, що потрібно обробити, а також вказівник на структуру Chunk, де після компіляції буде зберігатися байткод.
Структура Chunk має наступний вигляд:

```
struct Chunk {
    std::vector<byte> code;
    std::vector<int> lines;
    std::vector<Value> constants;

    void write(byte val, int line);
    void write(Chunk& chunk);
    int addConstant(Value const_val);
```

```
    inline size_t count(){ return  code.size(); }
    std::map<std::string, size_t> labelMap;
};
```

В векторі code зберігаються інструкції, що буде виконувати віртуальна машина.

Вектор lines зберігає номер рядка, на якому зустрілася відповідна інструкція.

Вектор constants зберігає значення, з якими оперує програма.

Значення можуть бути числові, стрічкові та бульові.

З стрічками програма не оперує, вони використовуються для позначення міток та адресів.

Сама структура Value має наступний вигляд:

```
struct Value {
    ValueType type;
    union {
        bool boolean;
        double number;
        const char* string;
        Value* pointTo;
      }val;
     ... // методи
  };
```

Оскільки для зменшення використання пам’яті застосовується union, то стрічка представляється через вказівник типу чар. Для того, щоб уникнути проблеми з memory leak, перед тим, як передати такий помістити такий вказівник в структуру, застосовується спеціальна функція `const char* addString(const char* start, int length)` , що приймає вказівник на початок стрічки, яку треба скопіювати, а також довжину стрічки. Ця функція зберігає в окремому векторі новостворену стрічку (через оператор new) та

повертає вказівник на неї. По закінченню виконання програма викликає функції `void freeStrings()` , що для кожної стрічки в векторі викликає застосовує delete .

Також у структурі Chunk зберігається `map<std::string, size_t> labelMap`, що зберігає номер інструкції, на яку вказує мітка.

`bool Compiler::compile(const char*source, Chunk* chunk)`
Починає свою роботу з ініціалізації парсеру вхідним текстом.
Потім, відповідно до граматики викликається метод void program(),
з якого далі послідовно викликається низка методів, вдіповідно до граматики.

У простого компілятора є 2 основних завдання. Йому потрібно розпарсити код користувача, щоб виявити дії, що цей код повинен виконувати. Потім ці дії потрібно привести до низько-рівневих інструкції. Багато мов програмування розділяють ці дві функції на дві послідовних дії при імплементації. Зазвичай парсер створює AST дерево, а потім генератор коду проходиться по цьому дереву та створює вихідний код. Для компіляції логічної мови програмування було обрано інший підхід - ці два кроки виконуються паралельно під час аналізу вхідного тексту. Такий підхід називається Single-Pass Compilation.

Оскільки програма компілюється в байткод послідовно з її парсингом, виникаєть труднощі в місцях, коли операції потрібно виконувати не в тій самій послідовності, в якій вони були задані.

Такі труднощі починаються з методу expression (вираз) - перший метод який не тільки виконує операцію, а й повертає (на стек) значення.

Для того, щоб правильно обробити вираз використовується підхід Pratt Parsing.

Його можна зрозуміти на прикладі наступного виразу

1 + 3*a - (-a)

Бажана послідовність дій виглядає наступним чином:

1. Виконати 3*a
2. Виконати 1 + (результат 1)
3. Виконати -a
4. Виконати (результат 2) - (результат 3)

Це досягається за наступним рекурсивним алгоритмом:

Передумова: Для кожної операції, що використовується при створенні виразу, визначаємо пріоритетність: найменша (згідно з гарматикою) у бульового оператора or, найбільша у primary (константне значення, або вираз в лапках). Аби не залежати від конкретного оператора визначаємо 0(none) пріоритетність, з якою і починає роботу expression()

1. Викликаємо parsePrecedence з пріоритетом none
1. Парсимо перший оператор виразу
2. викликаємо unary()

   (випадок, коли унарного оператора не має сюди також враховується)
3. Якщо є унарний оператор,

   викликаємо parsePrecedence unary,

   записуем оператор

Інакше

записуем значення

4. Парсимо бінарний оператор parsePrecedence з
5. Поки пріоритет оператору більше чи рівний пріоритету з якою була викликана parsePrecedence,

   викликаємо binary()
6. З binary викликаємо parsePrecedence з пріоритетом на 1 більше за минулий оператор, записуем оператор, через який була викликана binary

Алгоритм в коді має назву `parsePrecedence(Precedence precedence)`

На прикладі 1 + 3*a - (-a) буде здійснена наступна послідовність дій:

1. `parsePrecedence` запущено з пріоритетністю none
1. Виявлено 1, просуваємось, записуєм 1 в code
2. Виявлено +, просуваємось
3. Викликається функція binary з оператором +
4. Для виявлення другого операнду `parsePrecedence` запущено з пріоритетністю term+1 = factor
5. Виявлено 3, просуваємось - записуєм 3 в code
6. Виявлено *, просуваємось
7. Викликається функція binary з оператором *
8. Для виявлення другого операнду `parsePrecedence` запущено з пріоритетністю factor+1 = unary
9. Виявлено a, просуваємось - записуєм a в code
10. Виявлено -

    Пріоритетність “-” (term) менша за пріоритетність unary, повертаємось з функції
10. Записуємо “*” в code

11. Пріоритетність наступного токену “-” менша за factor, повертаємось

12. Записуємо “+” в code (функція зі станом 3-5, пріоритетність factor)

12. Пріоритетність наступного токену “-” більша за none, просуваємось

13. Викликається функція binary з оператором -

13. `parsePrecedence` запущено з пріоритетністю term+1 = factor

14. Виявлено a, просуваємось - записуєм в code

15. Виявлено (- унарний оператор

16. Викликаємо unary з (

17. Функція викликає `parsePrecedence none`

18. Виявлено “-” - унарний оператор

19. Викликаємо unary з -

20. Функція викликає `parsePrecedence unary`

21. Виявлено a, записуємо a в код, повертаємось

22. Записуємо - в код, поветаємось

23. Для (нічого не записуєм, просуваємось через), повертаємось

24. Записуємо “-” повертаємось

25. Аналіз тексту закінчено

Для виразу 1 + 3*a - (-a) код на виконання буде мати наступний вигляд:

code: [1, 3, a, *, +, a, -, -]

Розрахунок значення буде відбуватися з використанням стеку за допомогою оберненої польської нотації.

Пріоритет всіх операцій наступний (від меншого до більшого)

```
enum Precedence {
 PREC_NONE,
```

```
    PREC_OR,                    or
    PREC_AND,               and
    PREC_EQUALITY,          ==
    PREC_COMPARISON,        > < >= <=
    PREC_TERM,              + -
    PREC_FACTOR,       * /
    PREC_UNARY,             - (
    PREC_PRIMARY            value
          };
```

Оскільки Chunk, що буде використовуватися для виконання коду, для його збереження містить лише вектор байтів, то
для кожного оператору виділяється окрема інструкція довжиною в один байт,
всі числові/рядкові значення зберігаються в окремому векторі constants, а вектор code, записується лише інструкція OP_CONSTANT та їх індекс в constants.
Повний список інструкцій:

```
enum OpCode {
      OP_RETURN,
      OP_CONSTANT,
      OP_NEGATE,
      OP_ADD,
      OP_SUBTRACT,
      OP_MULTIPLY,
      OP_DIVIDE,
      OP_NOT,
      OP_OR,
      OP_AND,
      OP_LESS,
      OP_EQUAL,
      OP_GREATER,
      OP_TRUE,
      OP_FALSE,
      OP_PRINT,
```

```
    OP_POP,
    OP_POP_AND_JUMP, //a.k.a OP_JUMP_TO_LABEL
    OP_SET_ADDRESS_CONTENT,
    OP_SET_ADDRESS_CONTENT_INVERSE,
    OP_GET_ADDRESS_CONTENT,
    OP_PART_END,
    OP_JUMP_IF_FALSE,
    OP_JUMP,
    OP_RELATIVE_JUMP,
    OP_EXCHANGE,
    OP_JUMP_IF_FALSE_TO_LABEL,
    OP_GET_LABEL

};
```

Деякі з них будуть пояснені в окремих розділах. Також з метою оптимізації для літералів true, false було виділено окремі інструкції.

3.4 Створення машинного коду за допомогою віртуальної машини

Для виконання скомпільованих інструкцій використовується клас Vm - Virtual machine. Віртуальна машина є частиною внутрішньої архітектури нашого інтерпретатора. Ми передаємо йому шматок коду(Chunk) — і він запускає його.

Об’єкт Vm починає свою роботу з публічного методу

InterpretResult interpret(const char* source)

В ньому він компілює вхідний текст в чанк коду та запускає свій основний метод для обчислень `InterpretResult run()`

Принцип роботи методу `run` простий - він послідовно оброблює кожну байтову інструкцію даного chunk.

Якщо інструкція `OP_CONSTANT` - в стек заноситься значення Value з `chunk.constants`

Якщо інструкцією є бінарна операція - з стеку дістається два останні значення, над ними проводиться бінарна операція та її результат заноситься в стек. Оскільки програма обробляє не один expression (що повертає значення), а сукупність expression та statement, то після обробки expression компілятором додається інструкція OP_POP - результуюче значення виразу повертається зі стеку. Таким чином після виконання всього chunk стек має виявитися порожнім.

Код функції має наступний вигляд:

```
InterpretResult Vm::run() {

#define BINARY_OP(op) \
    do {                            \
                double b = pop().val.number;  \
                double a = pop().val.number;  \
                push(Value(a op b));         \
    } while(false)

    for(ip = 0; ip < chunk->count() - 0 ){

        switch (readByte()) {
            case OP_RETURN: return InterpretResult::OK;
            case OP_PRINT: ...
            case OP_POP: ....

            case OP_SUBTRACT: BINARY_OP(-); break;
            case OP_MULTIPLY:   BINARY_OP(*); break;
           (...код для всіх інших інструкцій було опущено)

    }
    runtimeError("No return statement");
    return InterpretResult::RUNTIME_ERROR;
}
```

Для роботи використовується змінні екземпляру stack та stackCount, а також допоміжні методи

```
Value Vm::pop() { return stack[--stackCount];}
void Vm::push(Value val) { stack[stackCount++] = val;}
```

3.5 Дебаггінг

Під час написання коду для цього проекту також важливу роль мала можливість подивитися скомпільовані в байткод інструкції, щоб з'ясувати чи виник баг на етапі компіляції, чи інтерпретації.

Для цього було розроблено окрему функцію для дебагінгу, що на вхід приймає chunk, та послідовно виводить на екран кожну інструкцію, що була туди записана. При цьому для інструкції OP_CONSANT поряд з нею замість індекса підставляється значення по цьому індексу.

Код функції має наступний вигляд:

```
void disassembleInstructions(const Chunk* chunk){
...
#define OP_CASE(name)  case(name): {cout << #name << '\n'; break;}
    for(int i = 0; i < chunk->code.size(); i++){
        cout << '[' << i << "]\t";
        switch (chunk->code[i]) {
            OP_CASE(OP_RETURN)
            OP_CASE(OP_NEGATE)
           (...всі інші інструкції так само як і попередні дві)
            case OP_CONSTANT:{
                cout << "OP_CONSTANT \t";
                chunk->constants.at(chunk->code[++i]).printValue();
                cout << endl;
                break;
            }
            default: cout << "Unknown OP :\t"  << chunk->code[i] <<
endl;;
        }
    }
#undef OP_CASE
}
```

Лаконічність коду була досягнута через використання директиви define.

Окремо на початку функції відбувається вивід міток та інструкцій, на які вони посилаються

```
    cout << "Labels:" << endl;
    for(auto    i    =    chunk->labelMap.begin();    i    !=    chunk-
>labelMap.end(); i++){
        cout << i->first << ":\t" << i->second << endl;
    }

    cout << " ---" << endl;
```

Розділ 4. Реалізація базової частини адресної мови

4.1 Штрих-операція

Штрих-операція є однією з унарних операцій.

На етапі компіляції спочатку до байткоду записується вираз, над яким проводиться операція. Потім, якщо за ним слідує “=”, це є сигналом, що відбувається засилання значення за адресою, до байткоду заноситься скомпільований вираз, з правої сторони від “=”, потім заноситься інструкція `OP_SET_POINTER`. Якщо “=” не слідує заноситься інструкція `OP_GET_POINTER`. Відповідний код знаходиться у методі void pointer().

Віртуальна машина працює з адресами наступним чином:

У якості змінної екземпляру віртуальної машини зберігається вектор `memory`. Він містить значення типу Value. Також у віртуальної машини є мапа `map<std::string , Value*> pMap`.

Коли оголошується нова адреса, та їй присвоюється певне значення, тобто при дії (`OP_SET_POINTER`), відбуваються наступні дії.

1. Два рази виклик `pop()`, що повертає дві Value. Друге Value, що повернулося (тобто те, яке першим зустрілось в коді), має тип String або Pointer.
2. В `memory` додається Value, що відповідає за присвоєне значення. (*)

Якщо ліве значення є типу String:

3. В `memory` додається Value типу Pointer, що відповідає за адресу. Value завжди створюється через конструктор, що приймає вказівник на Value (присвоєне значення),яке зберігається в memory.

4. В `pMap` додається зв’язок між цим Pointer Value та його назвою.

Якщо ліве значення є типу Pointer:

Значення `val.pointTo` замінюється. (**)

(*) - перший пункт виконується не завжди. Інколи адреси можуть містити значення, що самі по собі є адресами. Чи є значення адресою визначається просто - воно представлене String Value.
У такому разі через `pMap` повертається відповідний вказівник на Pointer Value, та він і використовується у пункті 2.
(**) - Зі стеку можна дістати Pointer Value, лише у тому разі, якщо воно до цього вже було збережено в `memory`. Отже, простої заміни значення, яке ця адреса містить є достатньо. Значення, на які жодна адреса не посилається можна було б видаляти через Garbage Collector. В межах цієї роботи така функціональність не передбачена.

При виконанні `OP_GET_POINTER`:

1. 1 раз викликається `pop()`, що повертає Value типу Pointer або String
2. Якщо Value типу String, через `pMap` знаходиться потрібне Pointer Value
3. Значення val.pointTo заноситься до стеку

4.2 Операція засилання

При інтерпретації оброблюється аналогічно штрих-операції. Єдина різниця при компіляції - спочатку до байткоду вноситься код, що відповідає за вміст, а потім - за адресу. Отже, замість інструкції `OP_SET_POINTER` використовується `OP_SET_POINTER_INVERSE` , що є

сигналом, щоб віртуальна машина поміняла 2 Value місцями при виконанні пункту 1 для `OP_SET_POINTER`.

4.3 Мітки

При компіляції мітки перевіряються на рівні методу void `Compiler::oneline()`. Спочатку обробляється перший токен, якщо за ним слідує трикрапка. До Chunk.labelMap додаємо цю мітку та індекс, що буде відповідати наступній інструкції (індекс = кількість вже доданих інструкцій). Продовжуємо так само обробляти наступні для цього рядка мітки. Коли більше немає трикрапки - мітки закінчились.

При цьому кількість байткоду не збільшується.

Коли в рядку зустрічається мітка без трикрапки - це сигнал переходу. До байткоду додаються інструкції `OP_CONSTANT` (+ індекс збереженої стрічки) та `OP_POP`.

4.4 Формула зупину

Коли парсер зустрічає символ “!” до байткоду додається інструкція `OP_RETURN`, що при виконанні коду на віртуальній машині сигналізує про закінчення виконання програми та вихід з методу `Vm::run()`.

Коли парсер зустрічає символ “B” до байткоду додається інструкція `OP_PART_END`, що виконується наступним чином: якщо на стеці нічого не має, програма закінчує своє виконання. Інакше зі стеку вилучається Number Value (якщо значення іншого типу, кидається помилка), та значення ip (instruction pointer) змінюється на значення цього числа - програма переходить до виконання

інструкції, індекс якої було збережено на стеці перед початком виконання підпрограми.

Розділ 5. Реалізація базової частини формул адресної мови

5.1 Формула обчислюваного переходу

Формула обчислюваного переходу компілятором обробляється як додавання інструкції `OP_POP` після обробки виразу. Така інструкція додається у випадку, якщо вираз є останнім з ланцюга з формул засилки, обміну. Для формул в середині цього ланцюга застосовується інструкція `OP_POP_UNCHECK`.

Різниця між цими інструкціями при виконанні байткоду наступна.

Коли зустрічається `OP_POP` зі стеку вилучається останнє значення та його зміст перевіряється до приналежність до списку міток.

Наприклад, в результаті операцій на стек було занесено Number Value 7, а отже при `OP_POP` перевіряється чи є в програмі мітка зі значенням “7”. Якщо так, то програма продовжує своє виконання з рядка, на який вказує ця мітка. Інакше програма продовжує виконувати наступну інструкцію.

При виконанні інструкції `OP_POP_UNCHECK` останнє значення лише вилучається зі стеку, переходу не відбувається.

5.2 Формули безумовного та відносного зупину

Формула безумовного переходу обробляється аналогічно формулі обчислюваного переходу. Компілятор в байткод записує стрічку (а точніше індекс стрічки в векторі константних значень) та

оператор `OP_POP`. Таким чином віртуальна програма при виконанні `OP_POP` розпізнає мітку та здійснює перехід, аналогічно до минулого випадку.

Формула відносного переходу обробляється компілятором як додавання `OP_JUMP` після обробки виразу, що стоїть зправа від символу відносного переходу. Інструкція `OP_JUMP` при виконанні методи `Vm::run()` вилучає зі стеку останнє значення (яке повинно бути числовим), та додає це значення до ip (instruction pointer).

5.3 Формула засилання

Окремо для формули засилання реалізовувати нічого не потрібно, адже при виконанні операції засилання зі стеку вилучаються два останніх значення, що можуть бути як певними літералами, так і результатами виконання певних формул.

5.4 Формула обміну

При компіляції коду, коли зустрічається команда <=> після компіляції правої частини виразу додається інструкція `OP_EXCHANGE`. В віртуальній машині вона обробляється згідно правилу:

`a <=> c` – еквівалентно до низки операцій:

```
1. 'r = a ;
2. 'r1 = 'c;
3. 'c = a;
4. ''r = 'r1
```

Зі стеку вилучаються два значення - обидва повинні належати до адресного простору. Це перевіряються. Якщо ці значення належали типу Number чи String, потрібно з мапи адрес повернути Pointer Value, що відповідає цьому літералу.

Наступні дії проводяться аналогічно до попередньо вказаних:

5.5 Формула входження

Формула входження являє собою окремий рядок користувацького коду.

```
P a {c1, c2, … }b
```

Оброблюється компілятором наступним чином.

Інструкція у байткоді для формули входженя - `OP_PROGRAM`.

Оскільки компіляція коду виконується за один прохід, то при обробці всього цього виразу до байткоду спочатку би заносилось значення a, потім c1, c2 і так до b.

a - мітка початку підпрограми

b - мітка куди слід перейти після її закінчення. Як було сказано в параграфі з формулою відносного зупину, b повинне знаходитися в верхівці стеку перед початком підпрограми.

c1, c2, cн - параметри, починаючи з першого аргументу, закінчуючи останнім значенням, що повертається. Хотілось би, щоб до стеку вони потрапляли в оберненому порядку.

Таким чином при виконанні підпрограми, коли б зустрічалось перше NIL (∅), то до адреси, яка входить з ним в формулу засилалось би значення, що вилучалось з верхівки стеку. Значення, що були перечислені в кінці підпрограми, засилалися до адрес, що були останніми параметрами формули входження, також шляхом вилучення їх зі стеку - пере значення до верхівки стеку і так далі.

Це і значить, що параметри формули входження повинні зберігатися записуватися до байткоду в оберненому виді.

Ілюстрація:

c1, c2 - вхідні параметри

c3, c4 - вихідні параметри

б - мітка переходу після виконання підпрограми

а - мітка підпрограми

Байткод:

```
б c4 c3 c2 c1 a OP_PROGRAM
```

Стек на момент виконання інструкції `OP_PROGRAM`:

```
[....б c4 c3 c2 c1] <- head
```

Після переходу до підпрограми:

1. Зустріч з першим NIL:

   Зі стеку вилучається верхівка c1, та засилається за вказаною адресою.

2. Зустріч з другим NIL:

   Зі стеку вилучається верхівка c2, та засилається за вказаною адресою.

3. Вказується перша адреса в1, що повертається:

   Зі стеку вилучається верхівка c3, її змістом стає зміст в1.

4. Вказується друга адреса в2, що повертається:

   Зі стеку вилучається верхівка c4, її змістом стає зміст в2.

5. Програма закінчує своє виконання чере В - зі стеку вилучається верхівка б та відбувається перехід.

Отже для байткоду потрібно змінити порядок, в якому була записана программа. Компілятор обробляє запис формули входження наступним чином:

0. Зчитується символ `P` , починається обробка входження

1. До констант зберігається стрічка, що відповідає початку підпрограми
2. Зчитується символ `{`
3. Для кожного з виразів, що перелічені через кому, до символу `}` створюється свій об’єкт типу компілятор, що ділить з основним компілятором парсер (саме тому в коді парсер зберігається як відсилка). Кожний з компіляторів обробляє свій відповідний вираз, та в результаті обробки містить свій вектор байткоду та список констант.
4. Зчитується символ `}`
5. До констант зберігається стрічка, що відповідає закінченню підпрограми
6. В байткод основного компілятору додається команда `OP_CONSTANT` з індексом стрічки з (5).
7. В байткод основного компілятору почергово додається код, що ,було згенеровано кожним з компілятором в оберненому до їх створення порядку. Також при цьому на кожному кроці до констант заносяться константи, що були виявлені цими компіляторами.
8. В байткод основного компілятору додається команда `OP_CONSTANT` з індексом стрічки з (1).
9. Додається інструкція `OP_PROGRAM`.

В реалізації методу `Vm::run()` при обробці інструкції `OP_PROGRAM` зі стеку вилучається верхівка (стрічка 1), та здійснюється перехід по цій мітці, як було вказано раніше.

Розділ 6. Реалізація додаткової частини формул адресної мови

6.1 Предикатні формули

6.1.1 Основная форма

Предикатна формула виконується за допомогою раніше зазначеної інструкції `OP_JUMP` а також `OP_JUMP_IF_FALSE`. В цьому розділі використання цих інструкцій буде детально пояснено.

Формула має вигляд `Pr {L} a | b`.

Компілятором обробляється наступним чином:

1. Зчитується символ `Pr`
2. Зчитується символ `{`
3. Викликається `Compiler::expression()`, що обробляє вираз в лапках
4. Зчитується символ `}`
5. До байткоду заноситься інструкція `OP_JUMP_IF_FALSE` - перехід до виконання b. Після `OP_JUMP_IF_FALSE` по аналогії з `OP_JUMP` повинне вказуватися число, на скільки інструкцій вперед (чи назад) віртуальна машина потрібно продвинутись.

   Це число в даному випадку є індексом інструкції, з якої починається вітка b. Оскільки це число наперед невідомо замість нього залишаемо 2 байти нулів. Потім їх потрібно замінити на вірне значення, тому зберігаємо це місце.

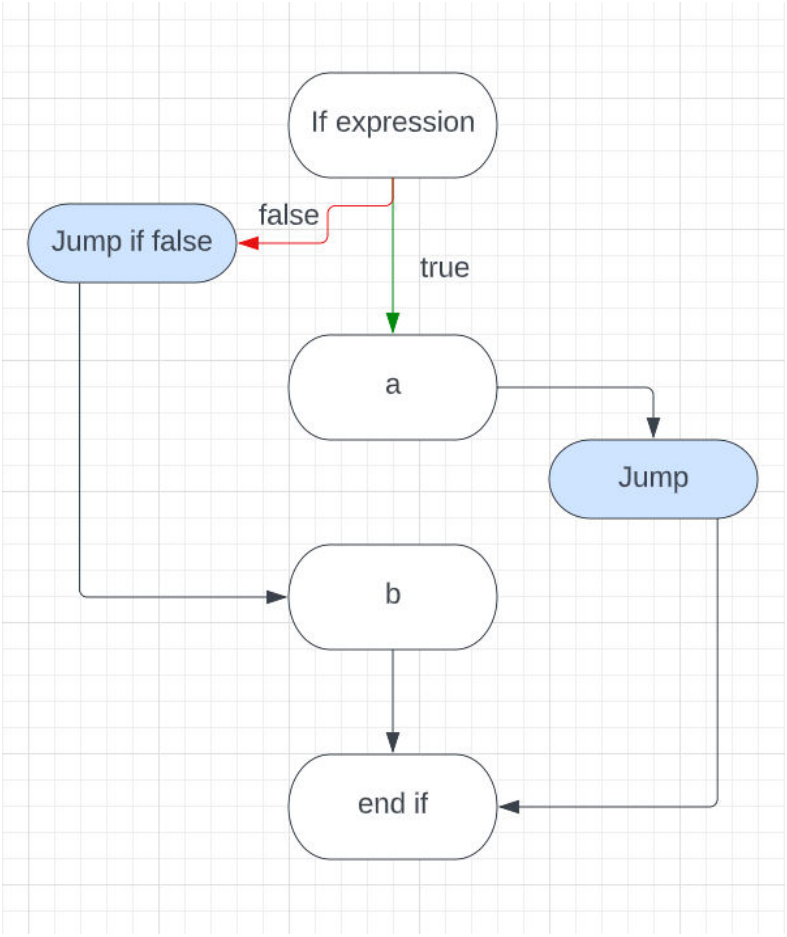
   Ці дії відбуваються в методі `size_t Compiler::writeJump(byte instruction)`. Оскільки перехід умовний у якості параметру передаємо `OP_JUMP_IF_FALSE`.
6. Оброблюємо в байткод вітку a

7. Оскільки наступна інструкція буде починати вітку б, то саме цей індекс інструкції потрібно записати замість двох байтів нулів.

   Ці дії відбуваються в методі `void Compiler::patchJump(size_t instructionIdx)`.
8. Тепер з вітки a потрібно перейти до кінця виконання предикатної формули, оминувши вітку b. Аналогічно до п.5 викликаємо `Compiler::writeJump`, у цьому разі перехід безумовний у якості параметру передаємо `OP_JUMP`.
9. Оброблюємо в байткод вітку b
10. Викликаємо `void Compiler::patchJump` з індексом інструкції, що повернувся у п.7

При виконанні байткоду, коли зустрічається `OP_JUMP` програма зчитує наступні дві інструкції, що повертають константу та просувається на цю кількість.

Коли зустрічається `OP_JUMP_IF_FALSE` програма зчитує наступні дві інструкції, що повертають константу, а також вилучає зі стеку останнє значення, що має зводитися до бульового типу. Програма просувається на цю кількість інструкцій лише у тому разі, якщо це бульове значення виявилось хибним.



6.1.2 З відкиданням першої мітки

6.1.3 З відкиданням другої мітки

6.1.4 Приклади тестування

6.2 Формули циклювання

6.2.1 Основная форма

Через велику кількість модифікацій формули циклювання в даній курсовій роботі розглядається лише кілька її видів.

Для зберігання частин циклу для їх використання при створенні байткоду використовується наступна структура

```
struct ForLoopParts{

    Chunk initialization, step, endCondition, parameter;
    ForLoopParts* nextPart{nullptr};

};
```

Розглянемо першу можливу форму:

`L{ a(b)L{expression} => p } l1 l2`

Для парсинг відбуваються наступним чином:

1. Зчитується `L`
2. Зчитується `{`
3. Створюється внутрішній компілятор, який і буде використовуватися для обробки тексту, зв’язаного з оголошенням циклу. Компілятор при ініціалізації отримує оригінальний парсер, щоб продовжити роботу з потрібного токену.
4. В цьому компіляторі викликається метод `void Compiler::parseForLoopParts(ForLoopParts* parts)`
5. На початку цього методу створюється новий екземпляр класу ForLoopParts.
6. Зчитується наступний вираз (що відповідає за початкове значення `a`), байткод, що було при цьому створено, записується до initialization chunk.
7. Зчитується `(`
8. Зчитується наступний вираз (що відповідає за значення інкременту `b`), байткод, що було при цьому створено, записується до step chunk.
9. Зчитується `)`
10. Зчитується `PR` та `{`

11. Зчитується наступний вираз (що відповідає за вираз, при хибності якого цикл закінчується), байткод, що було при цьому створено, записується до endCondition chunk.
12. Зчитується `=>`
13. Зчитується наступний вираз (що відповідає за початкове значення a), байткод, що було при цьому створено, записується до parameter chunk.

Для того, щоб додати підтримку обробки формули виду

```
L{ a(b)c => p } l1 l2,
```

слід трохи змінити попередні кроки, а саме:

В пункті 11, якщо не зустрічається токен PR, слід скомпілювати наступний вираз в тимчасову змінну endConditionPart chunk, а в endCondition chunk записати наступне:

1. `parameter`
2. `OP_GET_POINTER`
3. `endConditionPart`
4. `OP_LESS`

Що буде еквівалентно наступному оголошенню:

```
L{ a(b) L{'p < c} => p } l1, l2
```

Оскільки частини коду, що відповідають за оголошення були збережені, можна переходити до створення коду, що буде реалізовувати виконання циклу.

Для двох попередньо вказаних формул в байткод потрібно записати наступні інструкції::

1. Частина ініціалізації:
    a. `parameter`
    b. `initialization`
    c. `OP_SET_POINTER`
    d. `cond`

e. OP_POP

Еквівалентно до `'p=a, cond`

2. Частина інкрементування змінної:

   Спочатку тут слід поставити мітки **incr, та l1**

   a. parameter
   b. parameter
   c. OP_GET_POINTER
   d. step
   e. OP_ADD
   f. OP_SET_POINTER

   Еквівалентно до `'p + b => p`

3. Частина умови виходу з циклу:

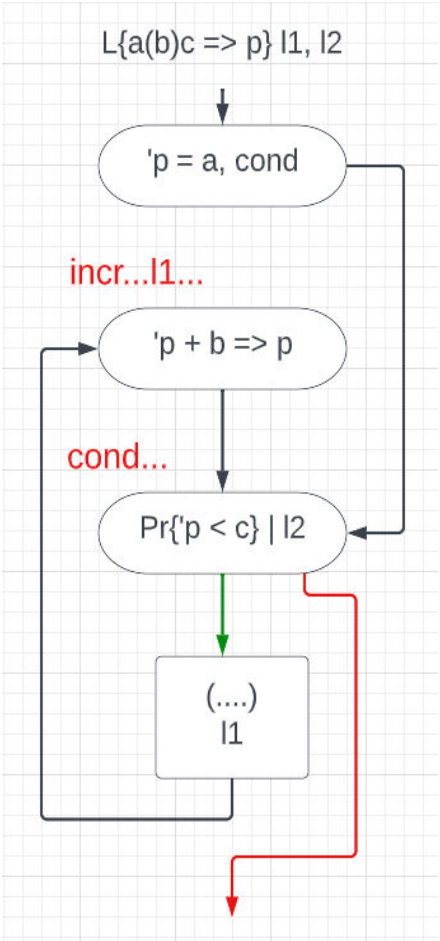
   Спочатку тут слід поставити лейбл **cond**

   a. endCondition
   b. l2
   c. OP_JUMP_IF_FALSE_TO_LABEL

   Еквівалентно до `L{expr}  | l2`

Потім код компілюється, поки не зустрінеться мітка, що відповідає за кінець циклу (l1).В такому разі останньою буде стрічка коду l1, що відповідає за перехід до частини інкрементування.

Діаграма виконання циклу:



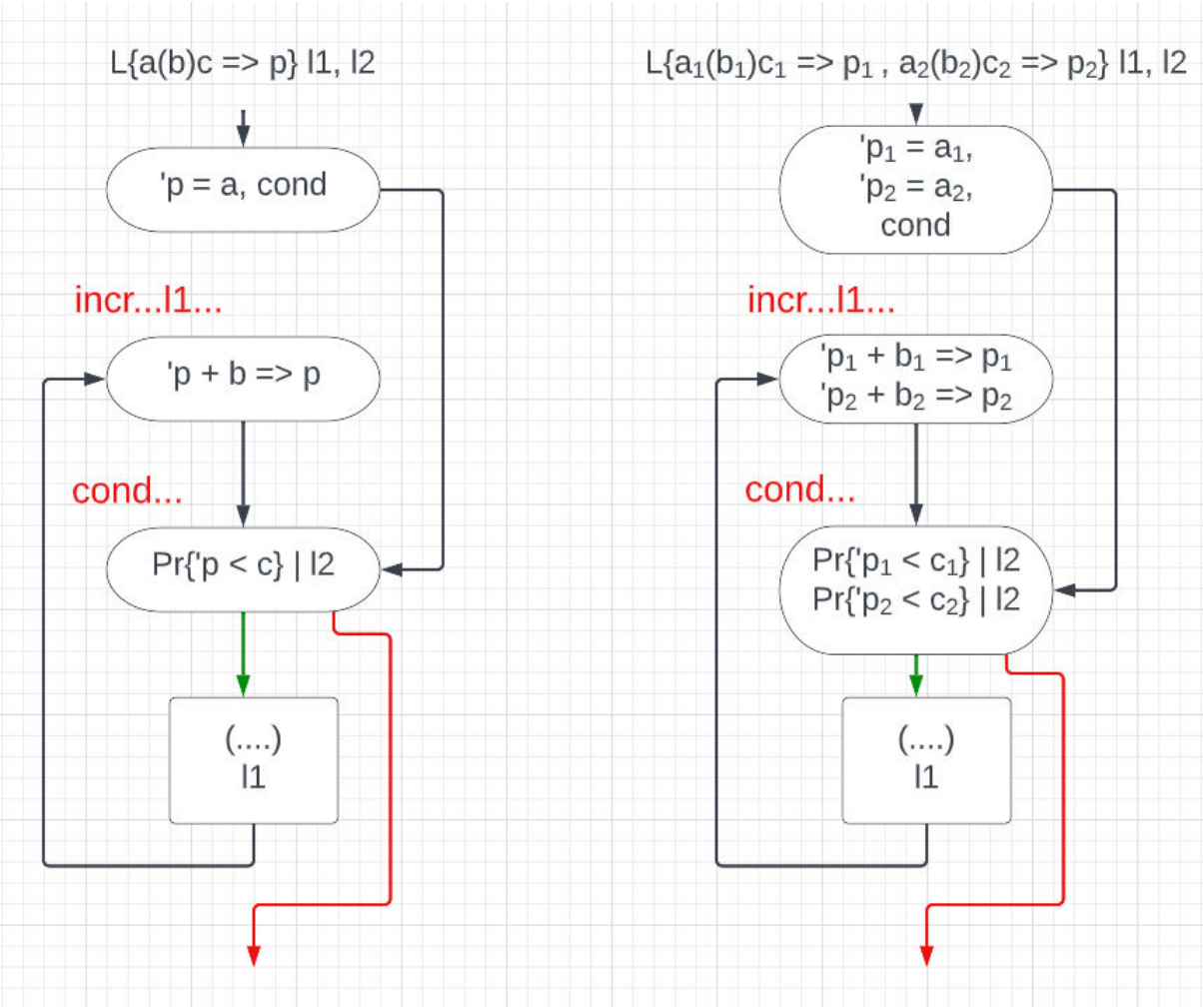
Для того, щоб програма могла обробляти форму виду

$L\{\ a_1(b_1)c_1 \Rightarrow p_1\ ;\ a_2(b_2)c_2 \Rightarrow p_2\ ;\ \dots\ ;\ a_n(b_n)c_n \Rightarrow p_n\}\ l1,\ l2$

при парсингу оголошення формули, замість одного об’єкту ForLoopParts створюється вектор таких об’єктів.

Під час компіляції байткоду це враховується таким чином, що кожна з частин (ініціалізація, інкрементування, умова) є об’єднанням коду частини для кожного елементу цього вектору.

На діаграмі це можна показати наступним чином:



В цьому випадку в частині ініціалізації перехід до частини умови записується один раз в кінці частини, всі інші частини будується шляхом послідовного запису коду для кожного елементу.

Останньою реалізована формула наступного виду

$L\{\ a_{11}(b_{11})c_{11}\ ;...;\ a_{1m}(b_{1m})c_{1m} \Rightarrow p_1\ ;\ \dots\ ;\ a_{n1}(b_{n1})c_{n1}\ ;...;\ a_{nk}(b_{nk})c_{nk} \Rightarrow p_n\}$ `l1, l2`

Для цього випадку в структурі ForLoopParts крім чанків з кодом є ще один член структури - `ForLoopParts* nextPart{nullptr};`

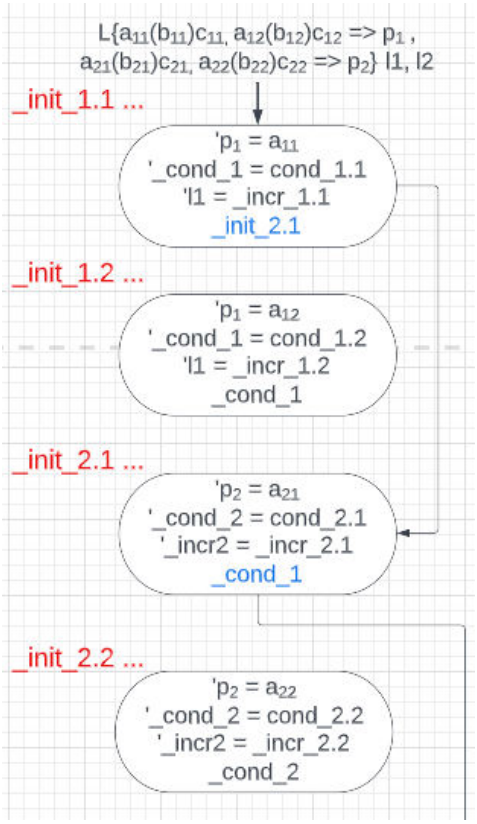
що стає в пригоді при обробці таких послідовностей $a_{11}(b_{11})c_{11}\ ;...;\ a_{1m}(b_{1m})c_{1m} \Rightarrow p_1$

У цьому разі до вектору додається елемент, що відповідає за $a_{11}(b_{11})c_{11}$, а наступні елементи виду $a_{1i}(b_{1i})c_{1i}$ зможна дістати послідовно пройшовши по `nextPart`. При цьому для полегшення

процесу написання інструкцій, коли програма дійла до моменту де за виразом $a_{1i}(b_{1i})c_{1i}$ слідує $=>$ p_1 ,то $p1$ як parameter chunk записується не лише до останнього елементу послідовності, а й до всіх попередніх.

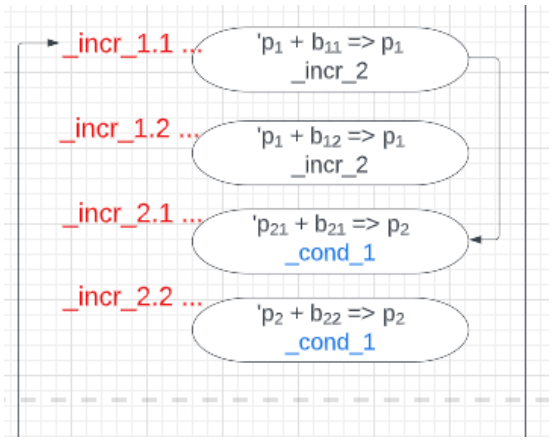
Діаграма має наступний вигляд:

Ініціалізація:



- Нехай вираз $a_{x1}(b_{x1})c_{x1}$;...; $a_{xm}(b_{xm})c_{xm}$ => p_x утворює одну послідовність
  А $a_{xm}(b_{xm})c_{xm}$ - одну підмножину послідовності
- Стрічки виду _aaa_x.y - мітки, виду _aaa_x - адреси, що використовуються для керування на якому етапі знаходиться кожен з циклів.
- Перша ініціалізація останньої послідовності останньою інструкцією переходить до умовних інструкцій.
- Не враховуючи останню послідовність, кожна перша ініціалізація послідовності переходить до ініціалізації наступної послідовності
- Всі інші ініціалізації переходять до виконання умовних інструкцій наступної послідовності
- Оскільки для останньої послідовності немає наступної перед початком тіла циклу ставить мітка останньої послідовності
- Саме в частині ініціалізації міткам переходу до умовної та інкрементувальної частини надається значення.

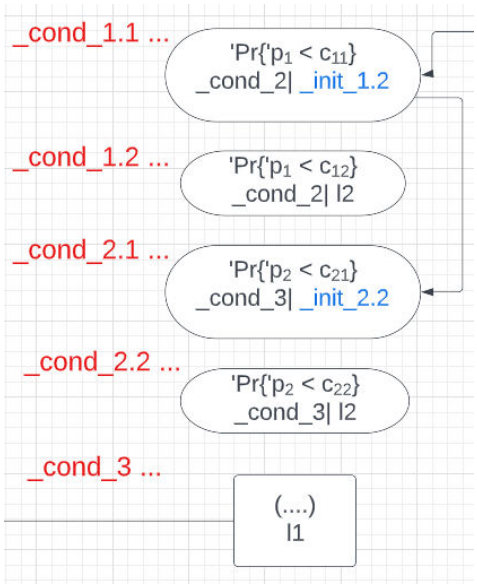
Інкрементування:



- Частина інкрементування для кожної підмножини останньої послідовності закінчується переходом до виконання умовної частини першої послідовності.

- Для всіх інших послідовностей відбувається перехід до виконання інкрементування підмножини наступної послідовності

Умова:



- Для останніх підмножин кожної послідовності при порушені умови відбувається вихід з циклу

- Для інших частин при порушені умови відбувається перехід до ініціалазації наступною підмножиною даної послідовності.

- З тої ініціалізації відбувається перехід назад до перевірки умови для цієї ж послідовності (для випадку коли одразу ж після ініціалізації умова виявилась хибною).

- Якщо умову не було порушено відбувається перехід до перевірки умови наступної послідовності

- Після частини з умовами ставиться мітка cond_(номер останньої послідовності + 1)

Всі три діаграми можна переглянути повністю в додатку.

Для того, щоб отримати значення лейблу вводиться додаткова інструкція `OP_GET_LABEL`. При її виконанні зі стеку вилучається остання string value та в стек додається індекс інструкції, на яку вказує ця мітка.

6.3 Формула заміни

Формула входження являє собою окремий рядок користувацького коду. Її обробка потребує вводу нової структури та видозміни деяких частин існуючого класу Scanner.

Вводить структура:

```
struct ReplaceTokens {
    Token what;
    Token with;
```

`}` - зберігає який токен в програмі потрібно бути замінити та його заміну.

Її обробка виглядає наступним чином:

1. Зчитується символ `R`
2. Зчитується символ `{`
3. Поки не зустрівся токен `}` почергово зберігається список пар `ReplaceTokens` токенів та їх замін, що розділяються через токен `->`
4. Зчитується 2 токена, що є мітками, в межах яких потрібно провести заміну
5.
    a. Оскільки при заміні токенів може змінитися порядок виконання інструкцій, то заміну потрібно проводити ще на

етапі сканування програми. Найпростіший спосіб - замість зчитаних токенів повертати їх заміни, ніяк не редагуючи процес компіляції.

b. Для цього копіюємо частину вхідного тексту, що знаходиться між мітками та передаємо її в новий внутрішній компілятор, що згенерує новий байткод для цієї частини.
c. Створюємо новий внутрішній компілятор.
d. Компілятор ініціалізуємо порожнім парсером, який повинен повертати токени, слідкуючи за замінами.
e. Для цього додаємо в конструктор класу Parser необов'язковий параметр, що приймає список замін.
f. В класі Scanner додаємо мапу з Токену в Токен, при ініціалізації сканеру передаємо список отриманих замін, what частина яких буде слугувати ключами для цієї мапи, а which - значеннями цих ключів.
g. Перед тим як повертатися з методу `Scanner::scanToken()` , перевіряємо чи є просканований токен одним з ключів мапи та робимо заміну.
h. Оброблений в внутрішньому компіляторі байткод додаємо до основного байткоду.

6. Якщо після двох міток є третя, записуємо її разом з інструкцією `OP_POP` в байткод.

Додавання формули заміни не потребує жодних змін імплементації виконання байткоду.

Розділ 7. Тестування

Штрих-операція

Test case:

```
'a = 1
'b = a
print b
print a
```

Compilation disassembly:

```
[0]     OP_CONSTANT     a
[2]     OP_CONSTANT     1.000000
[4]     OP_SET_POINTER
[5]     OP_CONSTANT     b
[7]     OP_CONSTANT     a
[9]     OP_SET_POINTER
[10]    OP_CONSTANT     b
[12]    OP_PRINT
[13]    OP_CONSTANT     a
[15]    OP_PRINT
[16]    OP_RETURN
```

Result:

```
Pointer to :    Pointer to :    1.000000
Pointer to :    1.000000
```

—----

Test case:

```
'a = 1;'b = a;'c = 3
print 'a + ''b + 'c [ 1 + 1 + 3 = 5 ]
```

Result:

```
5
```

—----

Test case:

```
'd = 1; 'b = d; 'e = 14; '14 = 18;
print 'd * 'e + ''b + ''e
```

Result:

```
33
```

—----

Test case:

```
'a = 1; 'b = a;
print 'a + ''b;
```

Result:

```
2
```

Формула засилання

Test case:

```
'a = 3;
'a => b;
a => c;
print b
print c
```

Compilation disassembly:

```
Labels:
 ---
[0]     OP_CONSTANT     a
[2]     OP_CONSTANT     3.000000
[4]     OP_SET_POINTER
[5]     OP_POP_AND_JUMP
[6]     OP_CONSTANT     a
[8]     OP_GET_ADDRESS_CONTENT
[9]     OP_CONSTANT     b
[11]    OP_SET_POINTER_INVERSE
[12]    OP_CONSTANT     a
[14]    OP_CONSTANT     c
[16]    OP_SET_POINTER_INVERSE
[17]    OP_CONSTANT     b
[19]    OP_PRINT
[20]    OP_CONSTANT     c
[22]    OP_PRINT
[23]    OP_RETURN
```

Result:

```
Pointer to :    3
Pointer to :    Pointer to :    3
```

Предикатна формула

Test case:

```
'a = 1;
lab1 ...
PR {'a == 5}  ! | print 'a
'a = 'a + 1
lab1
```

Result:

```
1
2
3
4
```

Test case:

```
'a = 3
PR {false}  | 'a = 2
print a
```

Result:

```
2
```

Формула обміну

Test case:

```
'a = 3; 'b = 1;
a <=> b
print a
```

Result:

```
Pointer to :    1.000000
```

Арифметика адрес

Test case:

```
'a = 1
'(a + 1) = 2;
'(a + 2) = 3
'(a + 3) = 4
print '(a + 3)
```

Result:

```
4
```

Формула заміни

Test case:

```
'a = 10, 'b = 9
l1... 'a = 'a + 3
l2...
R{a->b, +->-}l1,l2
print a
print b
```

Result:

```
Pointer to :    13
Pointer to :    6
```

Циклювання

Test case:

```
L{1 (1) 10 => pi} l1, l2
print 'pi
l1
l2 ...
```

Result:

```
1
```

```
      2
      3
      4
      5
      6
      7
      8
      9
      10
---
```

Test case:

```
      L{1 (0) PR{'pi <= 10} => pi} l1, l2
      print 'pi
      'pi = 'pi + 3
      l1
      l2 ...
```

Result:

```
      1
      4
      7
      10
---
```

Test case:

```
      L{1(1)10 => pi1, 1(1)10 => pi2} l1, l2
      print 'pi1 + 'pi2
      l1
      l2 ...
```

Result:

```
      2
      4
      6
      8
      10
      12
      14
      16
      18
      20

---
```

Test case:

```
      L{1(1)3, 8(2)10 => pi1, 2(1)4, 17(2)24 => pi2} l1, l2
      print 'pi1 + 'pi2
      l1
      l2 ...
```

Result:

```
3
5
7
25
29
```

Формула входження

Test case:

```
'x = 1 ; 'y = 2
main
l1 ...
NIL => a
'a + 3 => a
return a
B
main ...
P l1 {x, y} l2
l2 ...
print y
```

Result:

```
Pointer to :    4
```

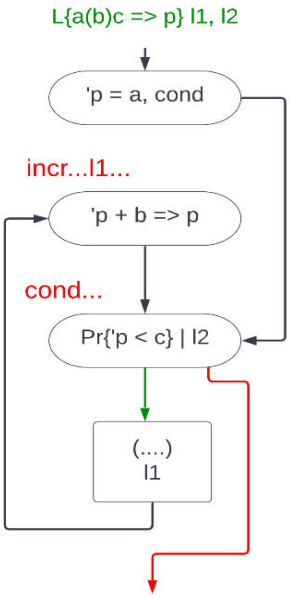
Висновок

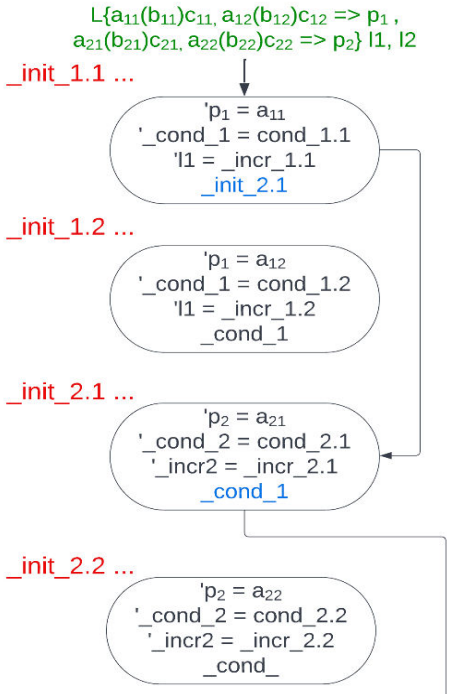
У даній роботі описуються основні моменти побудови компілятора, що можуть бути використані багатьох мов програмування високого рівня. Крім того було реалізовано багато конструкцій, що властиві лише Адресній мові програмування. Описуються унікальні моменти реалізації компіляції Адресної мови, що не можна знайти в реалізаціях компіляторів інших мов програмування.

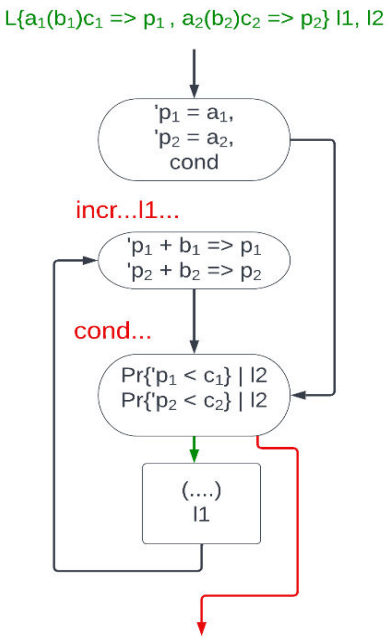
Надалі цей проект можна покращити, реалізувавши відкидання тривіальних міток в формулах. Також є видозміни формули циклювання, для яких не було реалізовано обробку, а саме - перебір натуральних чисел в прямому та зворотному порядку, використання параметру як значення, а не адресу тощо.

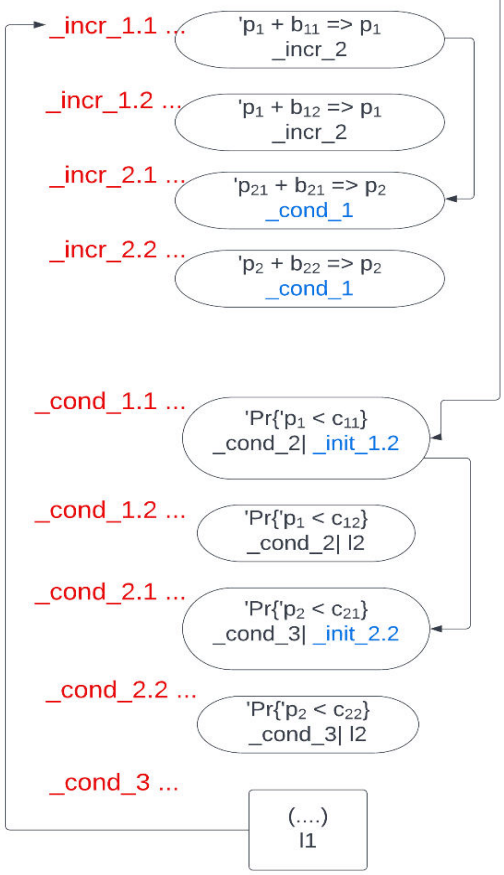
Для зручного користування даним компілятором також необхідно написати більш детальні повідомлення, при виникненні помилок користувача. Поки втілені лише повідомлення з синтаксичними помилками, було б краще, якщо б вказувалось, що саме є неправильним та чим це слід замінити.

Додаток


L{a(b)c => p} l1, l2
'p = a, cond
incr...l1...
'p + b => p
cond...
Pr{'p < c} | l2
(....)
l1


L{a11(b11)c11, a12(b12)c12 => p1 ,
a21(b21)c21, a22(b22)c22 => p2} l1, l2
_init_1.1 ...
'p1 = a11
'_cond_1 = cond_1.1
'l1 = _incr_1.1
_init_2.1
_init_1.2 ...
'p1 = a12
'_cond_1 = cond_1.2
'l1 = _incr_1.2
_cond_1
_init_2.1 ...
'p2 = a21
'_cond_2 = cond_2.1
'_incr2 = _incr_2.1
_cond_1
_init_2.2 ...
'p2 = a22
'_cond_2 = cond_2.2
'_incr2 = _incr_2.2
cond


L{a1(b1)c1 => p1 , a2(b2)c2 => p2} l1, l2
'p1 = a1,
'p2 = a2,
cond
incr...l1...
'p1 + b1 => p1
'p2 + b2 => p2
cond...
Pr{'p1 < c1} | l2
Pr{'p2 < c2} | l2
(....)
l1


_incr_1.1 ...
'p1 + b11 => p1
_incr_2
_incr_1.2 ...
'p1 + b12 => p1
_incr_2
_incr_2.1 ...
'p21 + b21 => p2
_cond_1
_incr_2.2 ...
'p2 + b22 => p2
_cond_1
_cond_1.1 ...
'Pr{'p1 < c11}
_cond_2| _init_1.2
_cond_1.2 ...
'Pr{'p1 < c12}
_cond_2| l2
_cond_2.1 ...
'Pr{'p2 < c21}
_cond_3| _init_2.2
_cond_2.2 ...
'Pr{'p2 < c22}
_cond_3| l2
_cond_3 ...
(....)
l1

Перелік Використаних джерел

1. Ющенко Е. Л. Адресне програмування / Е. Л. Ющенко. – Київ: Державне видавництво технічної літератури УРСР, 1963.
2. Nystrom R. Crafting Interpreters [Електронний ресурс] / Robert Nystrom. – 2015. – Режим доступу до ресурсу: https://craftinginterpreters.com/.
3. Compilers: Principles, Techniques, and Tools / A.Aho, M. Lam, R. Sethi, J. Ullman., 1986.
4. Бурдіна Е. Створила одну з перших у світі високорівневих мов програмування. Історія української науковиці Катерини Ющенко [Електронний ресурс] / Е. Бурдіна, Ю. Ющенко // https://dou.ua/. – 2022. – Режим доступу до ресурсу: https://dou.ua/lenta/interviews/about-kateryna-yushchenko.
5. Ющенко Ю. О. Деревоподібні формати адресного програмування / Ющенко Ю. О..