МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ»

КОНСПЕКТ ЛЕКЦІЙ
ЗА КУРСОМ
"ТЕОРІЯ СИНТАКСИЧНОГО АНАЛІЗУ І КОМПІЛЯЦІЇ"

для студентів напряму підготовки
050103 «Програмна інженерія»
спеціальностей
6.05010301«Програмне забезпечення систем»,
6.05010302 «Інженерія програмного забезпечення»

Покровськ-2016

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ»

КОНСПЕКТ ЛЕКЦІЙ
ЗА КУРСОМ
"ТЕОРІЯ СИНТАКСИЧНОГО АНАЛІЗУ І КОМПІЛЯЦІЇ"

для студентів напряму підготовки
050103 «Програмна інженерія»
спеціальностей
6.05010301«Програмне забезпечення систем»,
6.05010302 «Інженерія програмного забезпечення»

Затверджено на
засіданні навчально-
видавничої ради ДонНТУ
протокол № ___
від __._____.2016 р.

Ухвалено на засіданні
кафедри прикладної
математики і інформатики
протокол № 1
від 29.09.2016 р.

Покровськ-2016

Тема 1

Мета і задачі дисципліни. Основні поняття і визначення

Метою викладення дисципліни є надання знань по основах теорії мов і формальних граматик, теорії автоматів, методів розробки трансляторів, визначення загальних принципів організації процесу трансляції і структури трансляторів, вивчення основ теорії побудови трансляторів.

В даний час штучні мови, що використовують для опису предметної області текстове представлення, широко застосовуються не тільки в програмуванні, але й в інших галузях. З їхньою допомогою описується структура всіляких документів, тривимірних віртуальних світів, графічних інтерфейсів користувача і багатьох інших об'єктів, використовуваних у моделях і в реальному світі. Для того, щоб ці текстові описи були коректно складені, а потім правильно розпізнані й інтерпретовані, використовуються спеціальні методи їхнього аналізу і перетворення. В основі такого аналізу лежить теорія мов і формальних граматик, а також теорія автоматів.

Незважаючи на те, що вже існують тисячі різних мов і їхніх трансляторів, процес створення нових додатків у цій області не припиняється. Це пов’язано як з розвитком технології виробництва обчислювальних систем, так і з необхідністю розв’язання усе більш складних прикладних задач. Крім того, елементи теорії мов і формальних граматик застосовуються й в інших областях, наприклад, при описі структур даних, файлів, зображень, представлених не в текстовому, а в двійковому форматі. Ці методи корисні і при розробці своїх трансляторів навіть там, де вже маються відповідні аналоги. Необхідність нових трансляторів може бути обумовлена різними причинами, зокрема, функціональними обмеженнями, відсутністю локалізації, низькою ефективністю. Наприклад, однією з останніх розробок компанії Microsoft є мова програмування C#, а однією з причин її створення є прагнення до зниження популярності мови програмування Java. Можна привести безліч інших прикладів, коли створення свого транслятора може виявитися актуальним.

Тому основи теорії мов і формальних граматик, а також практичні методи розробки трансляторів лежать у фундаменті інженерної освіти по інформатиці й обчислювальній техніці. Пропонований матеріал торкається основи методів розробки трансляторів і містить відомості, необхідні для вивчення логіки їхнього функціонування, використовуваного математичного апарата (теорії формальних мов і формальних граматик, метамов).

Основні поняття і визначення

Транслятор – *обслуговуюча програма, що перетворює початкову програму, надану вхідною мовою програмування, у робочу програму, представлену об'єктною мовою*.

Трансляція вихідної програми в об'єктну програму, відбувається під час компіляції, а фактичне виконання програми відбувається після генерації коду. Приведене визначення відноситься до всіх різновидів програм, що транслюють. Однак у кожної з таких програм можуть матися свої особливості по організації процесу трансляції. В даний час транслятори розділяються на три основні групи: асемблери, компілятори й інтерпретатори.

Асемблер – *системна обслуговуюча програма, що перетворює символічні конструкції в команди машинної мови*. Специфічною рисою асемблеров є те, що вони здійснюють дослівну трансляцію однієї символічної команди в одну машинну. Таким чином, мова асемблера (ще називається автокодом) призначена для полегшення сприйняття системи команд комп'ютера і прискорення програмування в цій системі команд. Програмісту набагато легше запам'ятати мнемонічне позначення машинних команд, чим їхній двійковий код. Тому основний виграш досягається не за рахунок збільшення потужності окремих команд, а за рахунок підвищення ефективності їхнього сприйняття.

Разом з тим, мова асемблера, крім аналогів машинних команд, містить безліч додаткових директив, що полегшують, зокрема, керування ресурсами комп'ютера, написання повторюваних фрагментів, побудову багатомодульних

програм. Тому виразність мови набагато богаче, ніж просто мови символічного кодування, а це значно підвищує ефективність програмування.

Компілятор – *це обслуговуюча програма, що виконує трансляцію на машинну мову програми, записаної вихідною мовою програмування*. Також як і асемблер, компілятор забезпечує перетворення програми з однієї мови на іншу (найчастіше, у мову конкретного комп'ютера). Разом з тим, команди вихідної мови значно відрізняються по організації і потужності від команд машинної мови. Існують мови, у яких одна команда вихідної мови транслюється в 7-10 машинних команд. Однак є і такі мови, у яких кожній команді може відповідати 100 і більш машинних команд (наприклад, Пролог). Крім того, у вихідних мовах досить часто використовується жорстка типізація даних, здійснювана через їхній попередній опис. Програмування може спиратися не на кодування алгоритму, а на ретельне обмірковування структур чи даних класів. Процес трансляції з таких мов звичайно називається компіляцією, а вихідні мови звичайно відносяться до мов програмування високого рівня (чи високорівневим мовам). Абстрагування мови програмування від системи команд комп'ютера призвело до незалежного створення найрізноманітніших мов, орієнтованих на розв’язання конкретних задач. З'явилися мови для наукових розрахунків, економічних розрахунків, доступу до баз даних і інші.

Інтерпретатор – *чи програма пристрій, що здійснює пооператорну трансляцію і виконання вихідної програми*. На відміну від компілятора, інтерпретатор не породжує на виході програму машинною мовою. Розпізнавши команду вихідної мови, він відразу виконує її. Як у компіляторах, так і в інтерпретаторах використовуються однакові методи аналізу вихідного тексту програми. Але інтерпретатор дозволяє почати обробку даних після написання навіть однієї команди. Це робить процес розробки і налагодження програм більш гнучким. Крім того, відсутність вихідного машинного коду дозволяє не "засміччуватиь" зовнішні пристрої додатковими файлами, а сам інтерпретатор можна досить легко адаптувати до будь-яких машинних архітектур, розробивши його тільки один раз на широко розповсюдженій мові

програмування. Тому, інтерпретовані мови, типу Java Script, VB Script і інші одержали широке поширення. Недоліком інтерпретаторів є низька швидкість виконання програм. Звичайно інтерпретовані програми виконуються в 50-100 разів повільніше програм, написаних у машинних кодах.

Емулятор – *чи програма програмно-технічний засіб, що забезпечує можливість без перепрограмування виконувати на даної ЕОМ програму, що використовує коди чи способи виконання операцій, відмінні від даної ЕОМ*. Емулятор схожий на інтерпретатор тим, що безпосередньо виконує програму, написану на деякій мові. Однак, найчастіше це машинна мова чи проміжний код. І та і інша представляють команди в двійковому коді, що можуть відразу виповнюватися після розпізнавання коду операцій. На відміну від текстових програм, не потрібно розпізнавати структуру програми, виділяти операнди.

Емулятори використовуються досить часто у всіляких цілях. Наприклад, при розробці нових обчислювальних систем, спочатку створюється емулятор, що виконує програми, які розроблюються для ще неіснуючих комп'ютерів. Це дозволяє оцінити систему команд і напрацювати базове програмне забезпечення ще до того, як буде створене відповідне устаткування.

Дуже часто емулятор використовується для виконання старих програм на нових обчислювальних машинах. Звичайно нові комп'ютери мають більш високу швидкодію і мають більш якісне периферійне устаткування. Це дозволяє емулювати старі програми більш ефективно в порівнянні з їхнім виконанням на старих комп'ютерах. Прикладом такого підходу є розробка емуляторів домашнього комп'ютера ZX Spectrum з мікропроцесором Z80. Дотепер знаходяться аматори пограти на емуляторі в застарілі, але які ще й досі не втратили колишньої привабливості, ігрові програми. Емулятор може також використовуватися як більш дешевий аналог сучасних комп'ютерних систем, забезпечуючи при цьому прийнятну продуктивність, еквівалентну молодшим моделям деякого сімейства архітектур. Як приклад можна привести емулятори IBM PC сумісних комп'ютерів, реалізовані на більш могутніх комп'ютерах

фірми Apple. Ряд емуляторів, написаних для IBM PC, з успіхом заміняють різні ігрові приставки.

Емулятори проміжного представлення, як і інтерпретатори, можуть легко переноситися з однієї комп'ютерної архітектури на іншу, що дозволяє створювати мобільне програмне забезпечення. Саме ця властивість визначила успіх мови програмування Java, програма з який транслюється в проміжний код. Виконуюча цей код віртуальна Java машина, є ні чим іншим як емулятором, що працює під керуванням будь-якої сучасної операційної системи.

Перекодувальник – *чи програма програмний пристрій, що перекладає програми, написані машинною мовою однієї ЕОМ у програми машинної мови інший ЕОМ*. Якщо емулятор є менш інтелектуальним аналогом інтерпретатора, то перекодувальник виступає в тій же якості стосовно компілятора. Точно також вихідний (і звичайно двійковий) машинний код чи проміжне представлення перетворяться в інший аналогічний код по одній команді і без якого-небудь загального аналізу структури вихідної програми. Перекодувальники бувають корисними при переносі програм з одних комп'ютерних архітектур на інші. Вони можуть також використовуватися для відновлення тексту програми мовою високого рівня по наявному двійковому коду.

Макропроцесор – *програма, що забезпечує заміну однієї послідовності символів на іншу*. Є різновидом компілятора. Макропроцесор здійснює генерацію вихідного тексту шляхом обробки спеціальних установ, розташованих у вихідному тексті. Ці вставки оформляються спеціальним чином і належать до конструкцій мови, називаної макромовою. Макропроцесори часто використовуються як надбудови над мовами програмування, збільшуючи функціональні можливості систем програмування. Практично будь-який асемблер містить макропроцесор, що підвищує ефективність розробки машинних програм. Такі системи програмування звичайно називаються макроасемблерами.

Макропроцесори використовуються і з мовами високого рівня. Наприклад, вони збільшують функціональні можливості PL/1, C, C++. Особливо широко макропроцесори використовуються в C і C++. Вони дозволяють спрощувати написання програм. Прикладом широкого використання макропроцесорів є бібліотека класів Microsoft Foundation Classes (MFC). Через макро вставки в ній реалізовані карти повідомлень і інші програмні об'єкти. Макропроцесори дозволяють підвищити ефективність програмування без зміни синтаксису і семантики мови.

Синтаксис – *сукупність правил деякої мови, що визначають формування його елементів.* Інакше кажучи, це *сукупність правил утворення семантично значимих послідовностей символів у даній мові*. Синтаксис задається за допомогою правил, що описують поняття деякої мови. Прикладами понять є: змінна, вираз, оператор, процедура. Послідовність понять і їхнє припустиме використання в правилах визначає синтаксично правильні структури, що утворять програми. Саме ієрархія об'єктів, а не те, як вони взаємодіють між собою, визначаються через синтаксис. Наприклад, оператор може зустрічатися тільки в процедурі, а вираз в операторі, змінна може складатися з імені і необов'язкових індексів і т.д. Синтаксис не зв'язаний з такими явищами в програмі як «перехід на неіснуючу мітку» чи «змінна з даним ім'ям не визначена». Цим займається семантика.

Семантика – *правила й умови, що визначають співвідношення між елементами мови, а також інтерпретацію змістовного значення синтаксичних конструкцій мови*. Об'єкти мови програмування не тільки розміщаються в тексті відповідно до деякої ієрархії, але і додатково зв'язані між собою за допомогою інших понять, що утворюють різноманітні асоціації. Наприклад, змінна, для якої синтаксис визначає припустиме місце розташування тільки в описах і деяких операторах, має визначений тип, може використовуватися з обмеженою безліччю операцій, має адресу, розмір і повинна бути описана до того, як буде використовуватися в програмі.

Синтаксичний аналізатор – *компонента компілятора, що здійснює перевірку вихідних операторів на відповідність синтаксичним правилам і семантиці даної мови програмування*. Незважаючи на назву, аналізатор займається перевіркою і синтаксису, і семантики. Він складається з декількох блоків, кожний з який вирішує свої задачі. Більш докладно буде розглянутий при описі структури транслятора.

Розходження мов програмування і трансляторів

Мови досить сильно відрізняються друг від друга по призначенню, структурі, семантичній складності, методам реалізації. Це накладає свої специфічні особливості на розробку конкретних трансляторів.

Мови програмування є інструментами для розв’язання задач у різних предметних областях, що визначає специфіку їхньої організації і розходження по призначенню. Як приклад можна привести Фортран, орієнтований на наукові розрахунки, С, призначений для системного програмування, Пролог, що ефективно описує задачі логічного висновку, Лісп, використовуваний для рекурсивної обробки списків. Ці приклади можна продовжити. Кожна з предметних областей висуває свої вимоги до організації мови. Тому можна відзначити розмаїтість форм представлення операторів і виразів, розходження в наборі базових операцій, зниження ефективності програмування при розв’язанні задач, не пов'язаних із предметною областю. Мовні розходження відбиваються й у структурі трансляторів. Лісп і Пролог найчастіше виконуються в режимі інтерпретації через те, що використовують динамічне формування типів даних у ході обчислень. Для трансляторів з мови Фортран характерна агресивна оптимізація результуючого машинного коду, що стає можливим завдяки відносно простій семантиці конструкцій мови – зокрема, завдяки відсутності механізмів альтернативного іменування змінних через покажчики чи посилання. Наявність же покажчиків у мові С висуває специфічні вимоги до динамічного розподілу пам'яті.

Структура мови характеризує ієрархічні відносини між її поняттями, що описуються синтаксичними правилами. Мови програмування можуть сильно відрізнятися друг від друга по організації окремих понять і по відносинам між ними. Мова програмування PL/1 допускає довільне вкладення процедур і функцій, тоді як у С усі функції повинні знаходитися на зовнішньому рівні вкладеності. Мова С++ допускає опис змінних у будь-якій точці програми перед першим її використанням, а в Паскалі змінні повинні бути визначені в спеціальній області опису. Ще далі в цьому питанні йде PL/1, що допускає опис змінної після її використання Опис змінної в цій мові взагалі можна опустити і керуватися правилами, прийнятими за замовчуванням. У залежності від прийнятого розв’язання, транслятор може аналізувати програму за один чи кілька проходів, що впливає на швидкість трансляції.

Семантика мов програмування змінюється в дуже широких межах. Вони відрізняються не тільки реалізацією окремих операцій, але і парадигмами програмування, що визначають принципові розходження в методах розробки програм. Специфіка реалізації операцій може стосуватися як структури оброблюваних даних, так і правил обробки тих самих типів даних. Такі мови, як PL/1 і APL підтримують виконання матричних і векторних операцій. Більшість же мов працюють в основному зі скалярами, надаючи для обробки масивів процедури і функції, написані програмістами. Але навіть при виконанні операції додавання двох цілих чисел такі мови, як С і Паскаль можуть поводитися по-різному.

Поряд із традиційним процедурним програмуванням, називаним також імперативним, існують такі парадигми як функціональне програмування, логічне програмування й об’єктно-орієнтоване програмування. Структура понять і об'єктів мов сильно залежить від обраної парадигми, що також впливає на реалізацію транслятора.

Навіть та сама мова може бути реалізованою декількома способами тому, що теорія формальних граматик допускає різні методи розбору тих самих

пропозицій. У результаті транслятори різними способами можуть одержувати той самий результат (об'єктну програму) по первісному вихідному тексту.

Тема 2

Вивчення методів аналізу вихідної мови і генерації ефективної об'єктної програми

Короткий огляд процесу компіляції

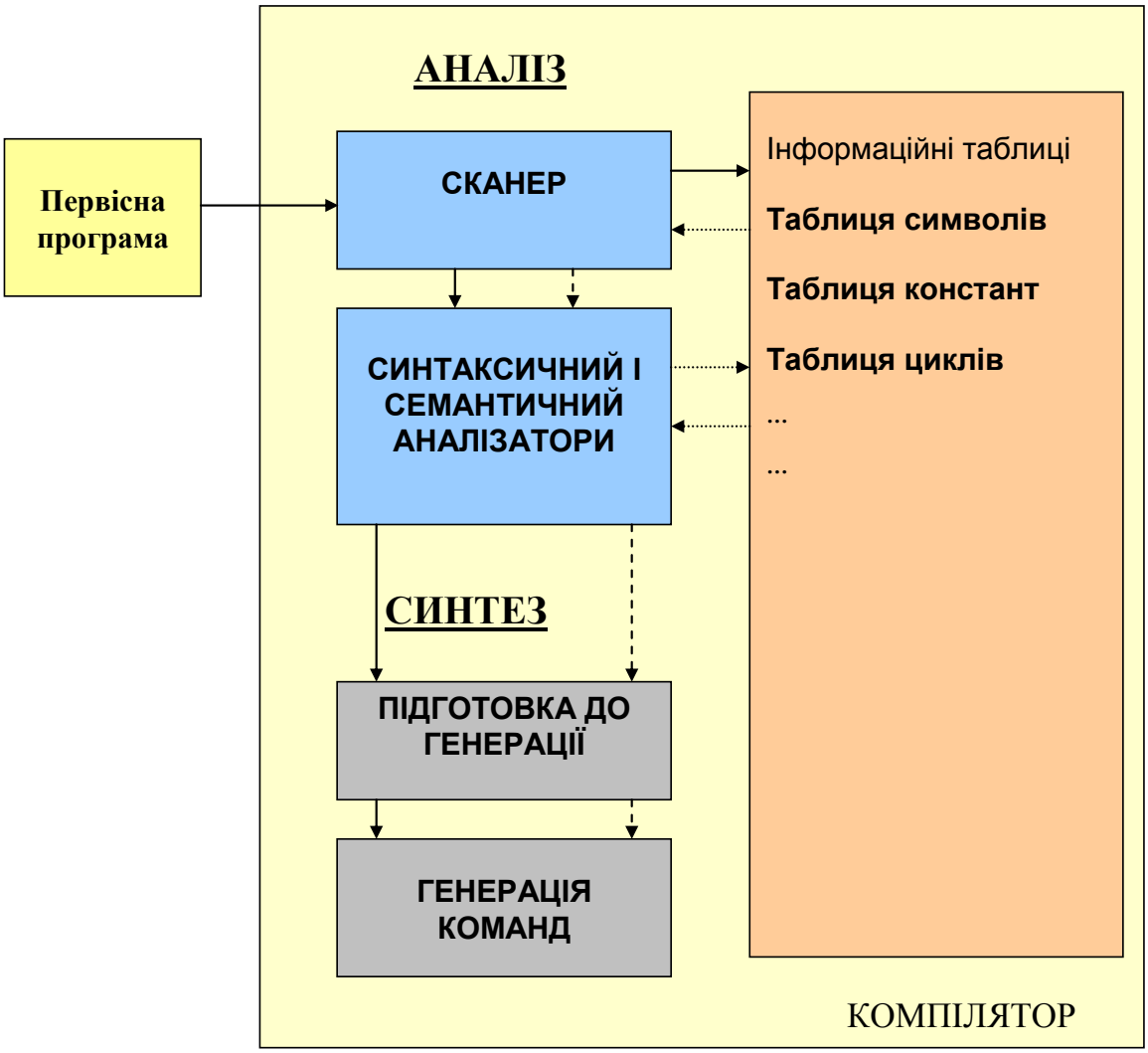


Рисунок 2.1 – Логічні частини компілятора

Інформаційні таблиці. При аналізі програми і запису заголовків процедур, циклів і т.д. витягається інформація і зберігається для наступного використання, ця інформація виявляється в окремих крапках програми й організовується таким чином, щоб до неї можна було звернутися з будь-якої крапки компілятора.

У будь-якому компіляторі в тій чи іншій формі використовуються:

1. Таблиця ***символів*** (ідентифікаторів, імен) – у цій таблиці зберігаються всі ідентифікатори вихідної програми разом з їхніми атрибутами (тип, адреса у вихідній програмі);
2. Таблиця ***констант*** використовуваних у вихідній програмі (константа і її адреса у вихідній програмі);
3. Таблиця ***заголовків for-циклів*** – відображає структуру вкладеності і зберігає змінні циклів.

Сканер (лексичний аналізатор)

Переглядає літери вихідної програми і будує символи вихідної програми. Також може забрукувати ідентифікатори в таблицю символів і не вимагає аналізу вихідної програми. Сканер передає символи аналізатору у внутрішній формі. Кожен роздільник вихідної програми (службове слово, знаки операції і пунктуації) будуть представлені цілим числовим ідентифікатором, а константи будуть представлені парою чисел. Це дозволить в інших частинах компілятора працювати ефективно, оперуючи символами фіксованої довжини.

Синтаксичний і семантичний аналізатор

Виконує роботу з розчленовуванню вихідної програми на складові частини, формуючи її внутрішнє представлення і занесення інформації в таблиці символів і інші таблиці. Як тільки синтаксичний аналізатор довідається конструкцію вихідної мови, він викликає відповідну семантичну процедуру, що контролює семантичну конструкцію і зберігає інформацію про неї в інформаційних таблицях. Внутрішнє представлення вихідної програми

заздрості від її подальшого використання. Це може бути дерево, що відбиває синтаксис вихідної програми. Польський інверсний запис, список тетрад.

Підготовка до генерації команд

Перед генерацією обробляється і змінюється внутрішня структура програми, розподіляється пам'ять під змінні програми, і оптимизируется з метою скорочення часу її роботи.

Генерація команд

На цьому етапі відбувається переклад внутрішнього представлення програми на чи автокод машинну мову.

Усі чотири логічних послідовних процеси можуть виконуватися послідовно відповідно до мал.1 (ці ж можна виконувати паралельно з визначеною взаємною синхронізацією)

Одним із критеріїв якості розробленого компілятора є обсяг займаної пам'яті. Часто вигідно, навіть необхідно, мати кілька проходів роботи компілятора.

Організація таблиць у трансляторах

Перевірка правильності семантики і генерації коду вимагають знання характеристик ідентифікатора використовуваного у вихідній програмі. Ці характеристики накопичуються в таблицях:

Аргумент	Атрибути
…	…
…	…

Рисунок 2.2 Загальний вигляд інформаційної таблиці

Кожен елемент займає в машині звичайно більш одного слова (4-2 байта). Розташувати інформацію можна двома способами:

1. Кожен елемент помістити в до-послідовних слів і мати таблицю з до*n слів.

2. Мати до таблиць, кожна з який складається з n слів.

Для побудови таблиць трансляторів аргументами таблиць є чи символи ідентифікатори, а значеннями характеристики. Т.к. кількість літер в ідентифікаторі не постійно в аргументі часто поміщають замість самого ідентифікатора покажчик. Це зберігає розмір аргументу фіксованим. Ідентифікатори характеризуються в спеціальному списку рядків, число літер у кожнім ідентифікаторі може характеризуватися, як частина аргументу, чи в списку ідентифікаторів прямо перед ними.

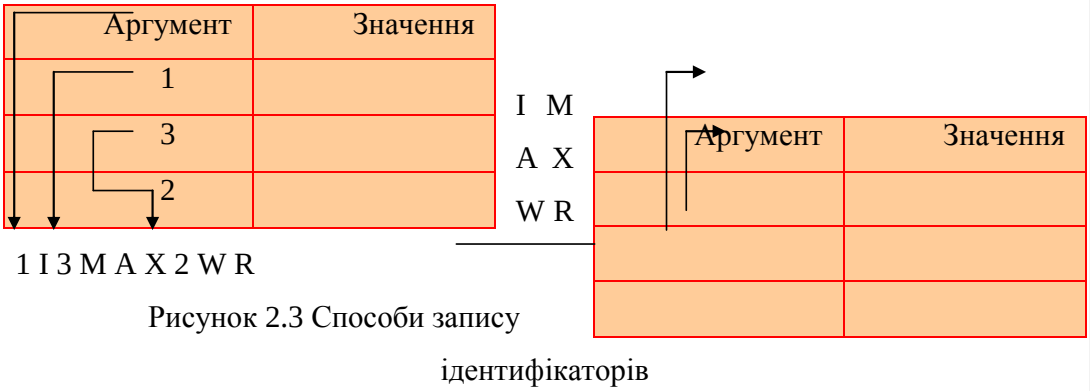


Рисунок 2.3 Способи запису ідентифікаторів

Коли компілятор починає роботу з вихідною програмою, таблиця символів порожня чи містить кілька елементів для службових чи слів функцій. У процесі компіляції для кожного нового ідентифікатора елемент додається один раз, але пошук здійснюється щораз, як тільки зустрічається цей ідентифікатор.

Уведемо поняття очікуваного числа порівнянь (яке буде залежати від кількості записів у таблиці) - ***коефіцієнта завантаження*** (load factor), і являє собою відношення числа елементів таблиці до можливого числа:

$$lf = \frac{n}{N} \qquad E = \varphi(lf)$$

Таблиці компіляторів:

1. ***Таблиці лексем***: як тип – основний символ, константа, ідентифікатор, ключове слово.

Тип	Довжина

Тип	Значення

2. ***Таблиці ідентифікаторів*** (імен символів)

Ім'я / ідентифікатор	Значення / довжина	Тип \ атрибути

3. Таблиця констант

Константа	Атрибути

При найпростішому способі організації таблиць елементи для аргументів додаються в порядку їхнього надходження, пошук у цьому випадку вимагає порівняння з кожним елементом таблиці, поки не буден знайдений необхідний; у цьому випадку E=n/2, для бінарного пошуку $E=1+\log_2 n$.

Хеш-адресація.

При лінійному рехешуванні - $E = (1 - \frac{lf}{2}) * (1 - lf)$

Випадкове рехешування - $E = -\frac{1}{lf} * \log_2 (1 - lf)$

Також мається рехешування додаванням, квадратичне рехешування.

Тема 3

ЛЕКСИЧНИЙ АНАЛІЗ. ПОБУДОВА СКАНЕРА

Призначення і необхідність фази лексичного аналізу

Лексичний аналіз – перша фаза процесу трансляції, призначена для угруповання символів вхідного ланцюжка в більш великі конструкції, називані лексемами. З кожною лексемою зв'язано два поняття:

- ***клас лексеми***, що визначає загальну назву для категорії елементів, що володіють загальними властивостями (наприклад, ідентифікатор, ціле число, рядок символів і т.д.);
- ***значення лексеми***, що визначає підрядок символів вхідного ланцюжка, що відповідають розпізнаному класу лексеми. У залежності від класу, значення лексеми може бути перетворене у внутрішнє представлення вже на етапі лексичного аналізу. Так, наприклад, числа преутворюють у двійкове машинне представлення, що забезпечує більш компактне збереження і перевірку правильності діапазону на ранній стадії аналізу.

У принципі, задачі, розв'язувані сканером, можна покласти на синтаксичний аналізатор. Але такий підхід незручний по наступним причинах:

1. Заміна, ланцюжків символів, що представляють елементарні конструкції мови, робить внутрішнє представлення програми більш зручним для подальшого аналізу розпізнавачем. Останній маніпулює не окремими символами, а закінченими елементарними конструкціями, що полегшує їхнє загальне сприйняття і подальший семантичний аналіз. Крім цього, при побудові лексем може здійснюватися найпростіша семантична обробка. Наприклад, перетворення і перевірка числових констант.
2. Зменшується довжина програми, що надходить у синтаксичний аналізатор, за рахунок усунення з її несуттєвих для подальшого аналізу пробілів, коментарів, ігнорованих символів. Зменшення розміру тексту

підвищує продуктивність розпізнавачей, особливо для багатопрохідних компіляторів.

3. Одна й і та ж сама мова програмування може мати різні зовнішні представлення елементарних конструкцій. Тому, наявність декількох лексичних аналізаторів, що породжують на виході такі саме безлічі лексем, дозволяє не переписувати синтаксичний аналізатор. Написати новий лексичний аналізатор набагато простіше, ніж синтаксичний. Прикладом мови, що має різні вхідні набори символів, є PL/1. Для неї визначені 48-символьний і 60-символьний алфавіти.

4. Лексичний аналізатор використовує більш прості, у порівнянні із синтаксичним аналізатором, методи розбору. Отже, на тих самих ланцюжках і при виконанні розбору тих самих конструкцій його продуктивність буде вище.

5. Блок лексичного аналізу природним образом вписується в ієрархічну структуру компілятора, що теж важливо при системному підході до розробки трансляторів.

Транслітератор

Незважаючи на те, що лексичний аналізатор обробляє вхідний ланцюжок, зручніше на його вхід подавати не просто окремі символи, а символи, згруповані по категоріях. Тому, перед лексичним аналізатором здійснюється додаткова обробка, яка зіставляє з кожним символом його клас, що дозволяє сканеру маніпулювати єдиним поняттям для цілої групи символів, іноді досить великої. Пристрій, що здійснює зіставлення класу з кожним окремим символом, називається **транслітератором**. Найбільш типовими класами символів є:

- ***буква*** – клас, з яким зіставляється безліч букв, причому необов'язково тільки одного алфавіту;
- ***цифра*** – безліч символів, що відносяться до цифр, найчастіше від 0 до 9;
- ***роздільник*** – пробіл, закінчення рядка, повернення каретки закінчення формату;

- ***ігнорований*** – може зустрічатися у вхідному потоці, але ігнорується і тому просто фільтрується з нього (наприклад, невидимий код звукового сигналу й інші аналогічні коди);
- ***заборонений*** – символи, що не відносяться до алфавіту мови, але зустрічається у вхідному ланцюжку;
- ***інші*** – символи, що не ввійшли в жодну з визначених категорій.

Пари, клас символу і його значення, надходять на вхід сканера, що вибирає для аналізу той її елемент, що найбільш зручний у даний момент. Наприклад, при аналізі ідентифікатора зручніше маніпулювати поняттям «буква», тоді як при аналізі дійсного числа можна відразу дивитися значення букви **«E»** чи **«e»**, що означає початок порядку. Слід також зазначити, що у всіх подальших міркуваннях будемо вважати, що поняття «буква» і «цифра» є терміналами, як отримані в транслітераторі до початку лексичного аналізу. У попередньому трактуванні ці поняття вважалися нетермінальними символами.

У будь-якій мові програмування є послідовності символів, що утворять єдині синтаксичні об'єкти, що починаються лексемами: ключові слова, ідентифікатори, константи, операції і т.п. Виділення лексем у вихідному тексті програми є основною задачею лексичного аналізатора чи сканера.

Усі лексеми в трансляторі приводяться до єдиного формату і представляються дескриптором, що містить два поля (тип лексеми, значення).

Перше поле визначає тип об'єкта, що зв'язується з лексемою: ідентифікатор, константа і т.д. Друге поле містить інформацію чи покажчик на інформацію про дану лексему в таблиці, що визначається типом лексеми.

Які послідовності символів вважати лексемами, і які типи лексем виділяти, залежить не тільки від мови програмування, але і від обраних алгоритмів трансляції.

Синтаксис лексем описується автоматними граматиками. У такий спосіб сканер можна розглядати як кінцевий автомат, входом якого служить ланцюжок символів вихідної програми, а виходом - послідовність лексем і

таблиці лексем, послідовність лексем називають лексичною згорткою програми. У лексичну згортку програми не включаються коментарі і пробіли.

Приклад. Розглянемо, які лексеми виділить сканер в операторі присвоювання мови ПАСКАЛЬ.

Внесок := 1,02 * Внесок;

Лексична згортка цього оператора представлена на рис. 2.1.

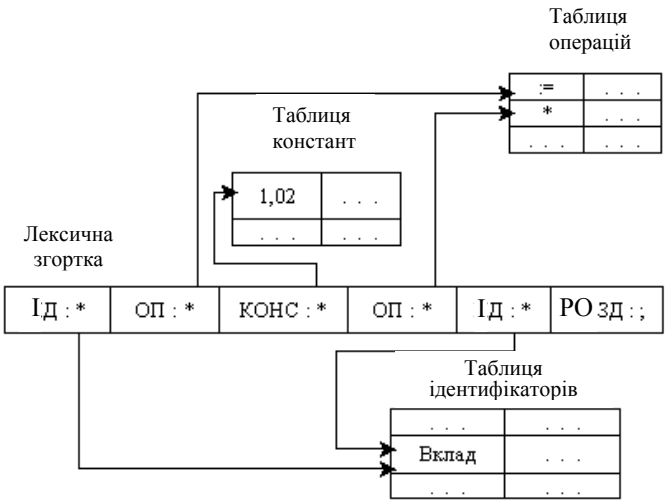


Рисунок 3.1 – Лексична згортки оператора присвоювання.

ЗАУВАЖЕННЯ. Використано наступні позначення класів лексем:

ІД – ідентифікатор; ОП - операція; КОНС – константа; РОЗД – роздільник. У мові ПАСКАЛЬ пробіли є роздільниками (крім вживання усередині текстової константи). Тому в лексичній згортці програми пробіли відсутні.

Лексичний аналіз програми можна реалізувати у виді окремого перегляду усього вихідного тексту програми.

У цьому випадку необхідна пам'ять для лексичної згортки всієї програми. Звичайно сканер програмується як модуль, до якого звертається синтаксичний аналізатор усякий раз, коли йому потрібна наступна лексема.

Обов'язковими вхідними даними для сканера є послідовність символів тексту вихідної програми і пояснення покажчика, що фіксує поточний символ. Виділяють два крайніх способи організації роботи лексичного аналізатора: прямий і непрямий. На практиці може використовуватися комбінація цих способів.

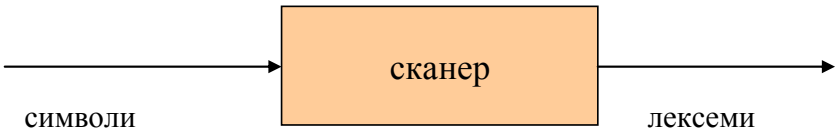
1) Сканер працює прямо, якщо він визначає лексему, лівий символ якої є поточним символом, і зрушує покажчик на перший символ, що не входить у лексему.

2) Сканер працює непрямо, якщо при звертанні до сканера задається тип очікуваної лексеми, у цьому випадку сканер перевіряє, чи утворять символи вхідного ланцюжка, починаючи з поточного символу, лексему заданого типу. Якщо так, то покажчик зрушується за цю лексему.

Час роботи сканера значно позначається на загальному часі трансляції, оскільки сканер обробляє вихідний текст програми по одному символу. Тому на практиці програму лексичного аналізатора часто оптимізують.

Робота лексичного аналізатора полягає в тому, щоб згрупувати визначені термінальні символи в єдині синтаксичні об'єкти – лексеми. Лексема – це ланцюжок термінальних символів, з яким зв'язують лексичну структуру, що складається з виду: ***тип лексеми – атрибути***.

Для будь-якої алгоритмічної мови число типів лексем звичайно. У такий спосіб лексичний аналізатор має на вході ланцюжок символів, що представляють вихідну програму, а виходом є послідовність лексем.



Вихід сканера являє собою вхід синтаксичного аналізатора. Лексичний аналізатор важливий для процесу компіляції по наступним причинах:

1. Заміна символів програми (ідентифікаторів, роздільників, службових слів ...) лексемами робить представлення програми більш зручним для подальшого процесу обробки.

2. З програми виключаються коментарі і множинні пробіли, що зменшує довжину програми.

Функції лексичного аналізатора (сканера)

1. Перекодування вихідної програми і приведення її до стандартної вхідної мови.

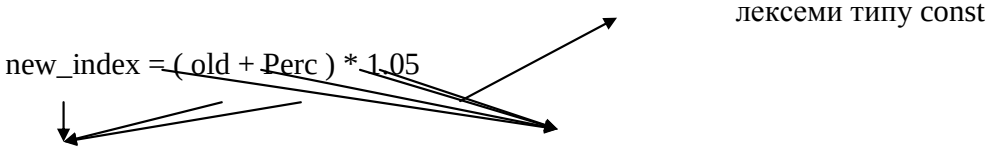
2. Здійснення лексичного контролю (контроль алфавіту і символів вхідної мови).

3. Виділення і зборка лексем з окремих знаків.

4. Виділення, зборка і переклад у машинний вид числових констант.

5. Видалення коментарів і пробілів.

Приклад



Виходом лексичного аналізатора буде послідовність лексем виду:

1) $Л_1 = (Л_2 + Л_3) * Л_4$

2) $ЛІ_1 = (ЛІ_2 + ЛІ_3) * ЛС_1$

3) $ЛІ_1\ ЛЗ_1\ ЛЗ_2\ ЧІ_2\ ЛЗ_3\ \ ЧІ_3\ ЛЗ_4\ ЛЗ_5\ ЛС_1$

Більшість відомих способів кодування засновані на прямому і непрямому методах. Існують способи організації сканерів, засновані на комбінації цих методів.

Говорять, що сканер працює прямо, якщо для даного вхідного тексту (ланцюжка) і положення покажчика в цьому тексті, аналізатор розпізнає

Формат: Список

лексему, розташовану безпосередньо праворуч від цього місця і зрушує покажчик вправо від частини тексту, що утворює цю лексему.

Лексичний аналізатор працює непрямо (посередньо), якщо для даного тексту, положення покажчика в цьому тексті і типу лексеми, він визначає, чи утворять знаки, розташовані безпосередньо праворуч від покажчика лексему заданого типу, якщо «так», то покажчик пересувається вправо від частини тексту, що утворює цю лексему.

Теоретичні передумови проектування сканера

Синтаксис будь-якої мови програмування краще класифікувати з позиції граматики кінцевих автоматів.

Граматика – це математична система, що визначає мову

$$G = (N, \sum , P, S)$$

N – кінцева безліч нетермінальних символів

$\sum$ – непересічна з N безліч термінальних символів, з яких утворяться слова і ланцюжки мови

P – кінцева безліч правил чи утворення продукції, що описують процес породження ланцюжків мови ($\alpha \rightarrow \beta$)

S – виділений символ з N – початковий (вихідний) символ

Кінцеві автомати – мають кінцеву безліч станів, частина з яких є кінцевими. В міру зчитування кожної літери з рядка, контроль передається від стану до стану відповідно до заданої безлічі переходів. Якщо після зчитування останньої літери, автомат знаходиться в одному з заключних станів, то говорять, що рядок належить мові, прийнятим автоматом.

S = (K, W, V, k, F)

K – кінцева безліч станів

W – кінцевий вхідний алфавіт

V – безліч переходів

k – початковий стан

F – безліч кінцевих станів

Велику частину того, що відбувається під час лексичного аналізу, можна моделювати за допомогою кінцевих перетворювачів, що працюють послідовно чи паралельно. При цьому лексичний аналізатор може складатися з ряду послідовно з'єднаних кінцевих перетворювачів (конвеєр). При другому способі організації вводиться кінцева безліч перетворювачів, що працюють паралельно, при цьому кожний з них шукає «свою лексичну» конструкцію.

Програмування сканера

Приклад : визначимо в якості основних складових символи вихідної мови

- роздільники = { /, *, + , - , (,) , ;, =, > }
- службові слова = {begin,end,while}
- ідентифікатори, що не можуть бути службовими словами
- цілі числа

Сканер повинний розпізнавати і виключати коментарі. Для кожного символу будується внутрішнє представлення, у більшості цих слів фіксованої довжини (4 байти).

В внутрішнім представленні всі ідентифікатори позначаються тим самим числом, тому що для синтаксичного аналізатора ідентифікатор є термінальним символом, однак при семантичному аналізі приходитися мати справа із самим ідентифікатором, тому його необхідно зберегти. Кращим варіантом є сканер, що видає дві величини: внутрішнє представлення і фактичний чи символ посилання на нього. Іноді сканер містить таблицю всіх різних символів, тоді як вихідну величину видається ціле число фіксованої довжини, що відповідає індексу позиції в цій таблиці. Мнемонічне ім'я може бути зв'язане з отриманим числом або через макровизначення, або, використовуючи змінну, котрої привласнюється це число під час ініціалізації.

ВНУТРІШНЄ ПРЕДСТАВЛЕННЯ	СИМВОЛ	ЛЕКСИЧНЕ ІМ'Я
1	ціле	$LC
2	ідентифікатор	$LI
3	begin	$LBEGIN
4	end	$LEND
5	while	$LWHILE
6	+	$LPLUS
7	-	$LMINUS
8	*	$LSTA
9	/	$LSLASH
10	(	$LLPA
11	)	$LRPA
12	=	$LE
13	>	$LGT

Рисунок 2.4 Зв’язок між внутрішнім представленням і лексичним ім’ям

Після аналізу наступного фрагмента

```
begin
        In = In / 2/* зміна ідентифікатора */
  while ( IL > In )
      begin
         Endw = ( Endw + 1 ) + IL   /* нагромадження */
         IL = IL – 1
      end
end.
```

сканер передасть програмі, що його викликала.

Крок	Результат	Зміст
1	3, ’begin’	begin
2	2, ’In’	ідентифікатор
3	12, ’=’	=
4	9, ’/’	/
5	1, ’2’	2
6	5, ’while’	while
7	10, ’(’	(
8	2, ’IL’	ідентифікатор
9	13, ’>’	>
10	11, ’)’	)
11	2, ’Endw’	іденетифікатор
12	6, ’+’	+
13	1, ’1’	1
14	8, ’*’	*
15	7, ’-’	-
16	4, ’end’	end

Рисунок 2.5 Приклад покрокового сканування

Лексична згортка буде мати вид:

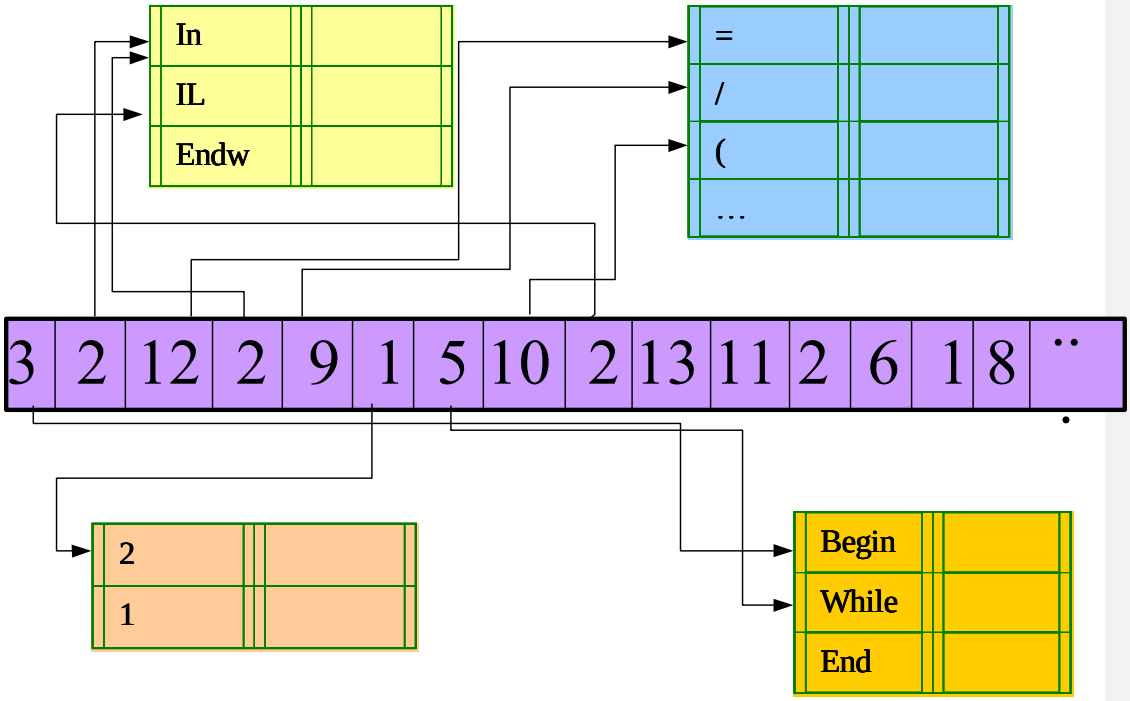


Рисунок 2.6 Лексична згортка фрагменту програми

Тема 4

ПОБУДОВА ПОРОДЖУЮЧИХ ГРАМАТИК ДЛЯ КОНСТРУКЦІЙ МОВ ПРОГРАМУВАННЯ

Опис будь-якої формальної мови здійснюється звичайно на іншій мові, яка зветься **метамовою**. Метамова може описувати або синтаксис (тобто форму конструкцій) формальної мови, або семантику (зміст цих конструкцій), або і те й інше разом.

Способи запису синтаксису мови

Існують різні способи запису синтаксичних правил, що в основному визначається умовним позначкам і обмеженнями на структуру правил, прийнятими у використовуваних метамовах. Метамови використовуються для завдання граматики мов програмування з часів Алгола 60. Ще раніш вони почали використовуватися при описі невеликих мов у статтях, присвячених формальним граматикам.

Для мов програмування найбільш розповсюдженою метамовою для опису синтаксису є нормальна форма Бэкуса (чи форма Бэкуса-Наура), скорочено - БНФ. Виділяють основні поняття і конструкцій цієї метамови:

- **термінальний символ** – символ, що складається тільки з букв алфавіту описуваної мови. У загальному випадку символом може бути одна буква чи кілька букв, що мають разом певне значення (наприклад, ключове слово *END*);

- **нетермінальний символ** – сформульоване на російській (чи будь-якій іншій) мові поняття описуваної мови програмування. Нетермінальні символи беруться в кутові дужки: < >. Наприклад, <програма>, <символічне ім'я>, <арифметичний вираз>. Іноді нетермінальні символи називають металінгвістичними змінними (тобто змінними метамови).

Для того, щоб розкрити поняття мови, що позначається нетермінальним символом, використовуються так називані правила підстановки чи металінгвістичні формули.

У загальному випадку правило підстановки записується у виді:

U ::= u ,

де U, u – довільні (кінцеві) послідовності (ланцюжка) нетермінальних і термінальних символів. А знак ::= означає “є по визначенню” чи “являє собою”.

При описі синтаксису мов програмування звичайно U – один нетермінальний символ, а u – кожна (у тому числі порожня) послідовність нетермінальних і термінальних символів, що розкриває (можливо не до кінця) сутність нетермінального символу, що стоїть в лівій частині.

Наприклад, правило підстановки, що визначає синтаксис умовного оператора мови ПЛ/1:

<умовний оператор> ::= IF <скалярне вираження> THEN

<оператор>; ELSE <оператор>;

Тут u у правій частині складається з 4-х термінальних символів: *IF*, *THEN*, *ELSE*, *;* і 2-х нетермінальних: <скалярне вираження>, <оператор>. Для того, щоб цілком визначити умовний оператор, необхідно розкрити ці нетермінальні символи, причому робити це доти, поки в правих частинах правил підстановки не залишиться невизначених нетермінальних символів.

Якщо права частина правила підстановки може приймати кілька різних форм, для скорочення запису використовується символ | , що означає “чи”. Наприклад:

<колір> ::= червоний | синій | жовтий

<оператор> ::= <оператор присвоювання> |

<умовний оператор> | <оператор циклу>

У першому прикладі слова “*червоний*”, ”*синій*”, ”*жовтий*” записані як термінальні символи.

Ще одним способом формального опису мови програмування є метамова Хомського, яка вийшла з надр математичної логіки. Мається наступна система позначень:

* символ «→» відокремлює ліву частину правила від правої (читається як "породжує" і "це є");

* нетермінали позначаються буквою **A** с індексом, що вказує на номер;
* термінали - це символи використовувані в описуваній мові;
* кожне правило визначає породження одного нового ланцюжка, причому той самий нетермінал може зустрічатися в декількох правилах ліворуч.

Опис ідентифікатора на метамові Хомского буде виглядати в такий спосіб:

1. $A_1 \to A$	**23. $A_1 \to W$**	**45. $A_1 \to s$**
2. $A_1 \to B$	**24. $A_1 \to X$**	**46. $A_1 \to t$**
3. $A_1 \to C$	**25. $A_1 \to Y$**	**47. $A_1 \to u$**
4. $A_1 \to D$	**26. $A_1 \to Z$**	**48. $A_1 \to v$**
5. $A_1 \to E$	**27. $A_1 \to a$**	**49. $A_1 \to w$**
6. $A_1 \to F$	**28. $A_1 \to b$**	**50. $A_1 \to x$**
7. $A_1 \to G$	**29. $A_1 \to c$**	**51. $A_1 \to y$**
8. $A_1 \to H$	**30. $A_1 \to d$**	**52. $A_1 \to z$**
9. $A_1 \to I$	**31. $A_1 \to e$**	**53. $A_2 \to 0$**
10. $A_1 \to J$	**32. $A_1 \to f$**	**54. $A_2 \to 1$**
11. $A_1 \to K$	**33. $A_1 \to g$**	**55. $A_2 \to 2$**
12. $A_1 \to L$	**34. $A_1 \to h$**	**56. $A_2 \to 3$**
13. $A_1 \to M$	**35. $A_1 \to i$**	**57. $A_2 \to 4$**
14. $A_1 \to N$	**36. $A_1 \to j$**	**58. $A_2 \to 5$**
15. $A_1 \to O$	**37. $A_1 \to k$**	**59. $A_2 \to 6$**
16. $A_1 \to P$	**38. $A_1 \to l$**	**60. $A_2 \to 7$**
17. $A_1 \to Q$	**39. $A_1 \to m$**	**61. $A_2 \to 8$**
18. $A_1 \to R$	**40. $A_1 \to n$**	**62. $A_2 \to 9$**
19. $A_1 \to S$	**41. $A_1 \to o$**	**63. $A_3 \to A_1$**
20. $A_1 \to T$	**42. $A_1 \to p$**	**64. $A_3 \to A_3A_1$**
21. $A_1 \to U$	**43. $A_1 \to q$**	**65. $A_3 \to A_3A_2$**
22. $A_1 \to V$	**44. $A_1 \to r$**	

Метамова Хомського-Щутценберже

Приведений у попередньому розділі приклад опису ідентифікатора показує громіздкість метамови Хомського, що дозволяє ефективно

застосовувати його тільки для опису невеликих абстрактних мов. Більш компактний опис можливо з застосуванням метамови Хомського-Щутценберже, що використовує наступні позначення метасимволів:

- символ «=» відокремлює ліву частину правила від правої (замість символу «→»);
- нетермінали позначаються буквою A с індексом, що вказує на номер;
- термінали - це символи використовувані в описуваній мові;
- кожне правило визначає породження декількох альтернативних ланцюжків, відокремлюваних друг від друга символом «+», що дозволяє, при бажанні, використовувати в лівій частині тільки різні нетермінали.

Уведення можливості альтернативного перерахування дозволило скоротити опис мов. Опис ідентифікатора буде виглядати в такий спосіб:

1. A_1=A+B+C+D+E+F+G+H+I+J+K+L+M+N+O+P+Q+R+S+T+ U+V+W+X+Y+Z+a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+ r+s+t+u+v+w+x+y+z
2. A_2=0+1+2+4+5+6+7+8+9
3. A_3=A_1+A_3A_1+A_3A_2

Метамови Хомського і Хомського-Щутценберже використовувалися в математичній літературі при описі простих абстрактних мов.

Синтаксис усієї мови (чи його частини – декількох конструкцій) визначається послідовністю правил підстановки. Одне з цих правил, що визначає (найбільш загальне) поняття мови, стоїть першим; нетермінальний символ, що позначає це поняття, називається початковим символом. Згадана послідовність правил підстановки називається граматикою, що породжує, тому що вона описує процедуру одержання (породження) правильних пропозицій мови. Граматика, що породжує, має наступне формальне визначення.

Формальна граматика G, що породжує, являє собою четвірку об'єктів (кортеж)

$$G = (N, T, \Sigma, П)$$

де N – безліч нетермінальних символів, що позначають конструкцію описуваної мови;

T – безліч термінальних символів, що складаються з букв алфавіту описуваної мови;

Σ – початковий символ граматики; $\Sigma \in N$;

П – набір правил підстановки.

Таким чином, щоб описати синтаксис формальної мови, досить задати N, T, Σ, П.

БНФ - не єдина метамова опису синтаксису формальних мов, однак його терміни і поняття стали загальновживаними і використовуються звичайно при визначенні граматик.

Приклад 1. Опис синтаксису (граматика) символічного імені.

<символічне ім'я> ::= <буква> | <символічне ім'я> <бц>

<бц> ::= A | B | C | D | ... | Z

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

У цьому прикладі використаний один дуже простий прийом – рекурсивне визначення. У першому правилі підстановки <символічне ім'я> визначається спочатку найпростішим образом як <буква>, а потім, після роздільника | (“чи”), нетермінальний символ <символічне ім'я> використовується з визначеним значенням. Після того, як <символічне ім'я> одержало значення <буква> <бц>, на наступному кроці воно може бути визначене як <буква> <бц> <бц> і так далі. Цей покроковий процес і є процес породження конструкцій мови за допомогою граматики, що породжує. У теорії формальних мов такий процес називається *висновком*.

Приклади висновку символічних імен:

1) <символічне ім'я> => <символічне ім'я> <бц> => <символічне ім'я> <бц> <бц> => <буква> <бц> <бц> => *X*<бц> <бц> => *X* <буква> <бц> => *XP*<бц> => *XP*<цифра> => *XP3*

2) <символічне ім'я> => <буква> => *A*

Зверніть увагу на те, що на будь-якому кроці висновку можна по-різному розкривати нетермінальні символи.

Це пов'язано з тим, що в прикладі 1 у правих частинах усіх правил підстановки присутні роздільники | (“чи”), що дозволяють довільний вибір. Висновок закінчується, коли в отриманому ланцюжку всі символи термінальні.

Використання рекурсивного визначення нетермінальних символів дозволяє одержувати як завгодно довгі речення мови, тому що рекурсія природним образом забезпечує повторюваний (циклічний) процес.

Узагалі, якщо необхідно описати синтаксис як завгодно довгої послідовності об'єктів, варто застосовувати рекурсію:

<послідовність об'єктів> ::= <об'єкт> | <послідовність об'єктів> <об'єкт>

У якості об'єктів можуть виступати, наприклад, букви і цифри в іменах і константах, імена з комами в операторах опису, списках параметрів процедури, списках введення-висновку й інші конструкції.

Нижче приведений приклад застосування БНФ для повного опису синтаксису простої мови програмування. У цьому описі можна знайти практично всі особливості використання БНФ, зокрема, опис синтаксису арифметичних і логічних виразів, що є вже класичним.

Приклад 2. Граматика мови програмування.

Спочатку дамо його короткий неформальний опис.

Мова дозволяє оперувати з даними цілого (integer) і логічного (logical) типів. Логічні значення: true, false. Дане може бути константою, змінною чи одномірним масивом цілих чисел. Масив описується в такий спосіб: *array* <ім'я> [n_1 : n_2], де n_1, n_2 – цілі константи, що визначають нижню і верхню

границі зміни індексу. Елемент масиву (перемінна з індексом) записується у виді: [індексний вираз], наприклад: A[i], B[k*m+1].

Оператори мови:

- оператор присвоювання: X := E;
- умовний оператор: *if* <умова> *then* <оператор> *else* <оператор> *fi* чи *if* <умова> *then* <оператор> *fi*;
- оператор циклу: *while* <умова> *do* <*оператор*> *od*;
- порожній оператор.

де S_i – будь-який з перерахованих вище операторів.

Формальний опис синтаксису мови БНФ:

<програма> ::= <опису> <оператор>

<опис> ::= <оператор опису>; | *<опис> <оператор опису>;*

<оператор опису> ::= <оператор опису типу> | *<оператор опису масиву>*

<оператор опису типу> ::= <тип> <список типу>

<список типу> ::= <ім'я> | *<список типу> , <ім'я>*

<оператор опису масиву> ::= <елемент списку> | *<список масивів> , <елемент списку>*

<елемент списку> ::= <ім'я> [<ціле> : <ціле>]

<оператор> ::= <оператор присвоювання> | *<умовний оператор>* | *<оператор циклу>* | *<порожній оператор>* | *<оператор>; <оператор>*

<оператор присвоювання> ::= <перемінна> := <вираз>

<умовний оператор> ::= if <логічна умова> then <оператор> else <оператор> fi | *if <логічна умова> then <оператор> fi*

<оператор циклу> ::= while <логічний вираз> do <оператор> od

<порожній оператор> ::=

<змінна> ::= <проста змінна> | *<змінна з індексом>*

<проста змінна> ::= <ім'я>

<змінна з індексом> ::= <ім'я> [<арифметичний вираз>]

<вираз> ::= <арифметичний вираз> | <логічний вираз >

<арифметичний вираз> ::= <доданок> | + <доданок> | - <доданок> | <арифметичний вираз> | + <доданок> | <арифметичний вираз> - <доданок>

*<доданок> ::= <множник> | <доданок> * <множник> | <доданок> \ <множник>*

*<множник> ::= <ціле> | <змінна> | (<арифметичний вираз>) | <множник> ** <множник>*

<логічний вираз> ::= <логічний доданок> | <логічний доданок> ∪ <логічний вираз>

<логічний доданок> ::= <логічний множник> | <логічний множник> ∩ <логічний доданок>

<логічний множник> ::= <логічна константа> | <проста змінна> | <арифметичний вираз> <операція відносини> <арифметичний вираз> | (<логічний вираз>) | ¬ <логічний множник>

<логічна константа> ::= true | false

<операція відносини> ::= = | ≠ | > | < | ≥ | ≤

<ім'я> ::= <буква> | <ім'я> <буква> | <ім'я> <цифра>

<ціле> ::= <цифра> | <ціле> <цифра>

<буква> ::= A | B | C | D | ... | Z | a | b | c | ... | z

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

У багатьох випадках для скорочення і підвищення наочності опису синтаксису використовують так називану модифіковану БНФ. Модифікація полягає в додаванні нових символів метамови. Найбільше часто застосовує квадратні і фігурні дужки.

Частина речення, укладена в квадратні дужки, може або бути відсутньою, або бути присутньою.

Приклад 3. Використання квадратних дужок у БНФ.

1) <умовний оператор> ::= if <логічна умова> then <оператор> [else <оператор>] fi

2) <дійсна константа> ::= [<знак>] <ціле>.<ціле> [E [<знак>] <ціле>]

< знак > ::= + | -

<ціле> ::= <цифра> | <ціле> <цифра>

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Частина речення, укладена у фігурні дужки, може або бути відсутньою, або повторюватися будь-яке число разів.

Приклад 4. Використання фігурних дужок у БНФ.

<символічне ім'я> ::= <буква> { <буква> | <цифра> }

<ціле> ::= <цифра> { <цифра> }

<список імен> ::= <ім'я> { , <ім'я> }

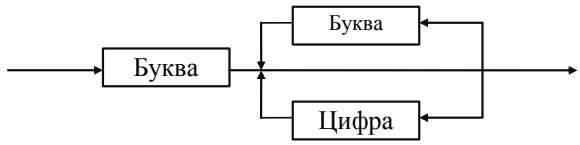
Число повторень частини речення, укладеної у фігурні дужки, може бути обмежено знизу і зверху підрядковими і надрядковими індексами. Це записується в такий спосіб: $\{\ldots\}_m^n$, де m – мінімальне, n – максимальне число повторень. Наприклад, ланцюжок символів $<буква>\{<буква>|<цифра>\}_0^5$ визначає символічне ім'я довжиною не більш 6-ти символів.

Інший розповсюджений спосіб опису синтаксису мов програмування – синтаксичні діаграми, що є фактично графічним зображенням нормальної форми Бэкуса. Нетермінальні символи записуються на діаграмі в прямокутниках, термінальні – у кружках (довгі термінальні символи можуть зображуватися в овалах). Символи з'єднуються стрілками, що вказують послідовність читання символів, що утворюють ланцюжок. Символ | (“or”) на діаграмі зображується петлею.

Приклад 5. Визначення синтаксису символічного імені.

Символічне ім'я

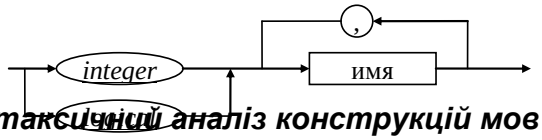
Буква Цифра



Приклад 6. Опис синтаксису оператора опису в мові з приклада 2.

Оператор опису

Не термінальний символ <Ім'я> визначене в прикладі 5.



Синтаксичний аналіз мовних конструкцій являє собою задачу, протилежну задачі породження (висновку). Задача синтаксичного аналізу (задача розбору) формулюється в такий спосіб: визначити, чи відповідає дана конструкція деякої мови граматиці цієї мови або, іншими словами, чи є дана конструкція правильним (не утримуючим синтаксичних помилок) реченням мови. Задача розбору має широке практичне застосування: кожен транслятор мови програмування має у своєму складі блок синтаксичного аналізу.

Перш ніж розглядати алгоритми синтаксичного аналізу, введемо поняття **синтаксичного дерева** як графічного способу зображення висновку конкретного речення мови.

Дерево являє собою ієрархічну структуру, що складається з вузлів різних рівнів; кожний з вузлів деякого рівня (крім останнього) зв'язаний з декількома вузлами наступного рівня і не більш, ніж з одним вузлом попереднього рівня.

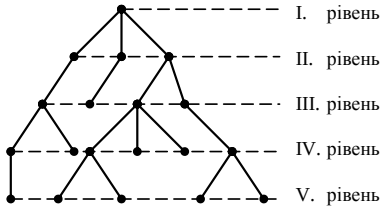


Рисунок 4.1 Приклад синтаксичного дерева

На І рівні може бути тільки один вузол, який називається **коренем**. Вузли останнього рівня називаються **листами**. **Кущ** вузла – безліч підлеглих йому вузлів нижніх рівнів. У синтаксичних деревах вузли являють собою нетермінальні і термінальні символи, причому листи є термінальними символами, а інші вузли – нетермінальними. Корінь дерева – початковий символ граматики. Кущ являє собою праву частину правила підстановки для даного нетермінального символу.

Приклад 7. Синтаксичне дерево для арифметичного виразу A1*(A+B).

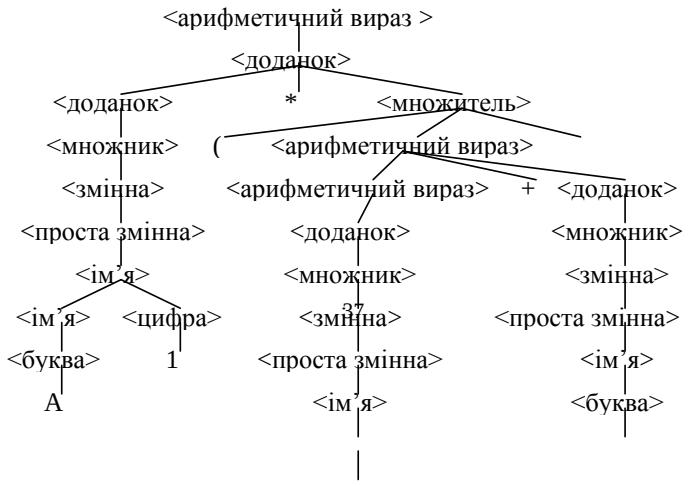


Рисунок 4.2 Синтаксичне дерево низхідного розбору

Достоїнство синтаксичних дерев є наочність опису синтаксису як самого речення так і його окремих складових. З іншого боку, для реальних конструкцій мов програмування дерево виявляється дуже громіздким. Тому якщо дерево використовується для одержання наочності опису, то або опускаються деякі проміжні рівні, несуттєві для цього опису, або воно складається для простих речень. Сама ідея побудови дерев широко використовується в трансляторах, тому що існують ефективні методи відображення дерев у пам'яті ЕОМ.

Синтаксичний аналіз речень, записаних на мовах програмування, фактично зводиться до побудови синтаксичних дерев різними методами. Існують два базових алгоритми синтаксичного аналізу (рішення задачі розбору):

- низхідний розбір (розгорнення);
- висхідний розбір (згортка).

При низхідному розборі синтаксичне дерево будується від кореня до листів. Його відмітна риса – цілеспрямованість, тому що, відправляючись від нетермінального символу, ми прагнемо знайти таку підстановку, що привела б до частини необхідного ланцюжка термінальних символів. У найпростіших випадках синтаксичний аналізатор (розпізнавач) намагається досягти цього шляхом спрямованого перебору різних варіантів. Розпізнавач розглядає дерево зверху вниз і зліва направо і вибирає перший нетермінальний символ, що не має підлеглих вузлів (що є “листом”). У списку правил підстановки відшукується правило (чи набір правил), що у лівій частині містить цей нетермінальний символ, а в правій частині (у правих частинах) – чергові

(рахуючи зліва направо) символи аналізованого речення. Якщо таке правило є, дерево нарощується й описаний процес повторюється. Якщо правило не знайдене, розпізнавач повертається на один чи кілька кроків назад, намагаючись змінити вибір зроблений раніше; процес розбору закінчується в одному з двох випадках:

1) Побудовано дерево, усі листи якого є термінальними символами, і при читанні зліва направо утворюють аналізоване речення. У цьому випадку результат розпізнавання позитивний, тобто синтаксис розглянутого речення відповідає граматиці мови.

2) Розпізнавач переробив усі можливі варіанти послідовностей підстановок, але так і не прийшов до дерева, яке описане в П.1. Це означає, що аналізоване речення не належить даній мові (тобто містить помилки). Варто підкреслити, що помилка вважається виявленою тільки в тому випадку, коли розглянуті всі припустимі варіанти підстановки.

Розглянемо цей процес на прикладі.

Приклад 8. **Провести низхідний синтаксичний розбір числа +2.4 по граматиці.**

(1) *<константа> ::= <КФТ> | <знак> <КФТ>*

(2) *<КФТ> ::= .<ціле> | <ціле>. | <ціле>.<ціле>*

(3) *<ціле> ::= <цифра> | <цифра> <ціле>*

(4) *<знак> ::= + | –*

(5) *<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*

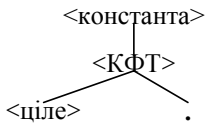
Процес розбору покажемо у виді послідовності кроків, на кожнім з який будується якась частина дерева.

Крок 1.

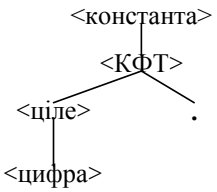
<константа> – корінь дерева.

<u>Крок 2.</u> Оскільки вираз (1) не містить у правій частині термінальних символів, обираємо перший зліва ланцюжок символів: <константа> ::= <КФТ>. Одержуємо

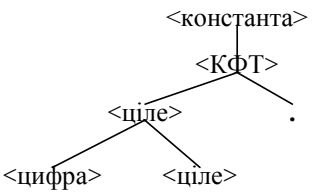
<u>Крок 3.</u> У правилі (2) перший ланцюжок починається з термінального символу ".", а тому що перший символ аналізованого речення – "+", цей ланцюжок не підходить. Вибираємо наступну.



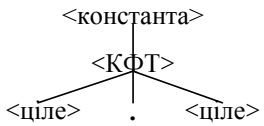
<u>Крок 4</u>. У правилі (3) вибираємо перший ланцюжок <цифра>:



<u>Крок 5.</u> Для визначення нетермінального символу <цифра> за правилом (5) є 10 термінальних символів, але жоден з них не збігається із символом "+". Тому здійснюємо повернення на один крок назад і, знову користуючись правилом (3), одержуємо:

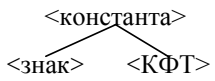


<u>Крок 6.</u> З двох гілок дерева вибираємо, відповідно до алгоритму, ліву і знову намагаємося застосувати правило (5). Тому що це знову не приводить до успіху, повертаємося назад до дерева, яке отримали на етапі 2 (оскільки спроби застосування правила (3) вичерпані. У правилі (2) вибираємо ланцюжок <ціле>.<ціле>. Одержуємо дерево:



На цьому шляху ми, мабуть, зробимо ще 2 кроки, перш ніж переконатися в його хибності. Проробіть їх самостійно.

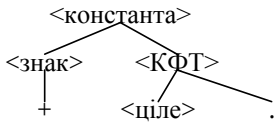
<u>Крок 9</u>. Вертаємося до правила (1) і вибираємо для підстановки другий ланцюжок <знак><КФТ>.



<u>Крок 10.</u> Застосовуємо правило (4) до нетермінального символу <знак>. Одержуємо.

Отриманий термінальний символ збігається з першим символом аналізованого речення, отже, галузь отриманого дерева правильна. Далі намагаємося одержати наступний символ –"2".

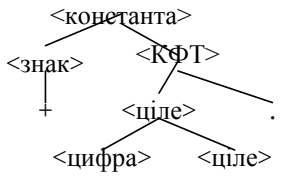
<u>Крок 11.</u> Розкриваємо нетермінальний символ <КФТ> за правилом (2), при цьому вибираємо другий ланцюжок, тому що в першому символ "." очевидно не збігається із символом "2".



Кроки 12, 13. Застосовуємо правила (3) і (5).

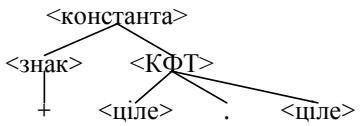
Одержали відразу два термінальних символи аналізованого речення "2" і ".". Однак дерево вже побудоване, тому що всі листи є термінальними символами. Якби усі варіанти підстановок були вже випробувані, можна було б зробити висновок, що в реченні +2.4 допущена помилка. Оскільки це не так, робимо два кроки назад, повертаючи до дерева, отриманого на кроці 11.

Крок 14. Снову застосовуємо правило (3), але його другий ланцюжок.

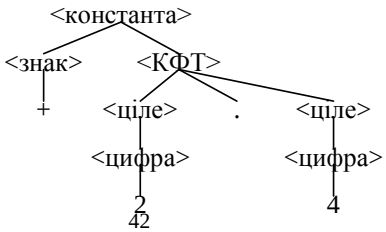


Очевидно, що на цьому шляху ми зробимо ще 3 кроки (переконаєтеся в цьому!), перш ніж зайдемо в тупик. Пропускаємо ці три кроки і повертаємося, нарешті, до дерева отриманому на кроці 10.

Крок 18. Застосовуємо до символу <КФТ> на цьому дереві правило (2) (третій ланцюжок).



Кроки 19, 20, 21, 22. Розкриваємо послідовно лівий і правий символи <ціле>.



Листи цього дерева (зліва направо) утворюють вихідне речення +2.4, отже воно відповідає заданій граматиці, тобто не містить помилок.

Завжди варто мати на увазі, що тільки неухильне, точне проходження алгоритму низхідного розбору дозволяє гарантувати правильність розбору, особливо для складних граматик. Спроби пропустити кілька кроків як очевидні чи зробити їх "у розумі" часто приводять до помилок.

Примітка. Описаний алгоритм низхідного розбору не можна застосовувати, якщо граматика є ліво-рекурсивною, тобто містить правила підставлення виду

U ::= Uu

оскільки при цьому спроби розкрити самий лівий нетермінальний символ приведуть до нескінченного породження нетермінального символу U. У такому випадку необхідно видозмінити алгоритм, зробивши його, наприклад, правостороннім, чи змінити граматику.

В алгоритмі низхідного розбору не обов'язково будувати дерево. На кожнім кроці можна записувати ланцюжок нетермінальних і термінальних символів, отриманих після чергової підстановки.

При висхідному розборі дерево будується від листів до кореня, тобто алгоритм відправляючись від заданого рядка, намагається, застосовуючи правила підстановки справа наліво, привести її до початкового символу граматики.

Частина рядка, яку можна привести до нетермінального символу, називається **фразою**. Якщо приведення здійснюється застосуванням одного правила підстановки, фраза називається безпосередньо **зводимою**. Сама ліва фраза, що безпосередньо зводиться, називається **основною.**

Алгоритм висхідного розбору полягає в наступному. У вихідному реченні відшукується основа (тобто перегляд, як і в низхідному розборі, йде зліва направо) і зводиться до нетермінального символу. Ця операція застосовується

доти, поки або отриманий єдиний символ, що є початковим символом граматики (у цьому випадку розбір закінчується і робиться висновок про правильність речення, що розбирається,), або в отриманому ланцюжку не може бути знайдена фраза. В останньому випадку робиться повернення на один чи кілька кроків назад і вибирається інша основа. Якщо всі можливі варіанти перебрані, а корінь дерева так і не отриманий, робиться висновок про наявність помилки в реченні, що розбирається, і алгоритм припиняє роботу.

Таким чином, що висхідний розбір являє собою перебір варіантів, однак не цілеспрямований, тому що тут немає можливості відкидати свідомо не правильні підстановки, як це має місце в низхідному розборі (ланцюжок ".<ціле>" у правилі (2), невідповідні цифри в правилі (5) у прикладі 9).

Процес висхідного розбору записують або у виді побудови дерева від листів до кореня, або у виді послідовності ланцюжків термінальних і нетермінальних символів, що починається з аналізованого речення, що закінчується (якщо речення не містить помилок) початковим символом граматики.

Головним недоліком як низхідного, так і висхідного алгоритмів розбору є велика кількість кроків, що визначається переборним характером алгоритмів.

Аналіз методів компіляції арифметичних виразів

Алгоритм розбору зверху-вниз

Основна проблема розбору, що передбачає алгоритм, - визначення правила виводу, яке потрібно застосувати до нетермінала. Процес розбору, що передбачає (зверху-униз) з погляду побудови дерева розбору можна проілюструвати рис. 1.4. Фрагменти недобудованого дерева відповідають сентенціальным формам виводу. Спочатку дерево складається тільки з однієї вершини, відповідній аксіомі S. У цей момент по першому символі вхідного потоку аналізатор, що передбачає, повинен визначити правило S->X1 X2 ..., що повинне бути застосоване до S. Потім необхідно визначити правило, що повинне бути застосоване до X1, і т.д., доти, поки в процесі такої побудови сентенціальной форми, що відповідає лівому виводу, не буде застосовано правило Y->a .... Цей процес потім застосовується для наступного самого лівого нетермінального символу сентенціальной форми.

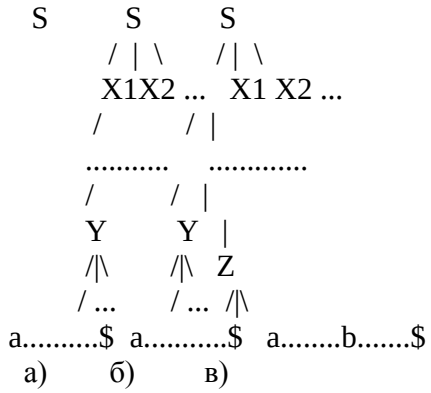


Рисунок 4.3

На рис. 4.4 наведена структура аналізатора, що передбачає визначення чергового правила з таблиці. Таку таблицю можна побудувати безпосередньо із граматики.

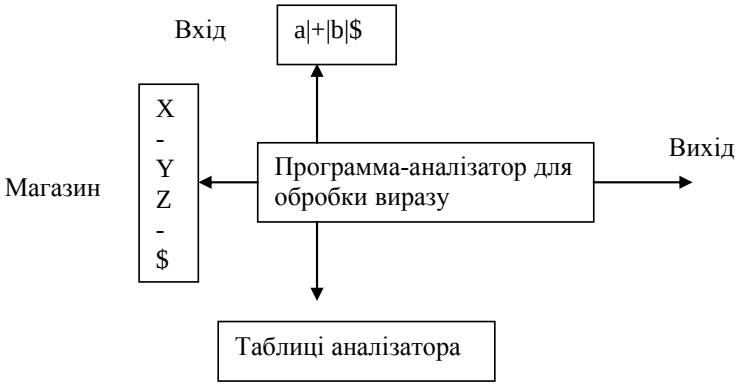


Рисунок 4.4 Структура аналізатора

Таблично-керуємий аналізатор,має вхідний буфер, таблицю аналізу й вихід. Вхідний буфер містить розпізнаваний рядок, за яким чергує $ - правий кінцевий маркер, ознака кінця рядка. Магазин містить послідовність символів граматики з $ на дні. Спочатку магазин містить початковий символ граматики на верхівці й $ на дні. Таблиця аналізу - це двовимірний масив M[A,a], де A - нетермінал, і a - термінал або символ $. Аналізатор управляється програмою, що працює у такий спосіб. Вона розглядає X - символ на верхівці магазина й a - поточний вхідний символ. Ці два символи визначають дія аналізатора. Є три можливості.

1. Якщо X=a=$, аналізатор зупиняється й повідомляє про успішному закінченні розбору.

2. Якщо X=a#$, аналізатор видаляє X з магазина й просуває покажчик входу на наступний вхідний символ.

3. Якщо X - нетермінальний символ, відбувається програмне звертання до таблиці M[X,a]. По цьому вході зберігається або правило для X, або помилка. Якщо, наприклад, M[X,a]={X->UVW}, аналізатор заміняє X на верхівці магазина на WVU {на вершині стека U}. Будемо вважати, що аналізатор як вихід просто друкує використані правила виводу. Якщо M[X,a]=error, аналізатор звертається до підпрограми аналізу помилок. Поводження аналізатора може бути описане в термінах конфігурацій автомата розбору.

Алгоритм нерекурсивного предсказуючого аналізу

```
repeat X:=верхній символ магазина;
   if X - термінал або $
   then if X=InSym
     then видалити X з магазина;
       InSym:=черговий символ;
     else error()
     end
   else /*X = нетермінал*/
     if M[X,InSym]=X->Y1Y2...Yk
     then видалити X з магазина;
       помістити Yk,Yk-1,...Y1 у магазин
       (Y1 на верхівку);
       вивести правило X->Y1Y2...Yk
     else error() /*вхід таблиці M порожній*/
   end end
until X=$ /*магазин порожній*/
```

Рисунок 4.5

Спочатку аналізатор перебуває в конфігурації, у якій магазин містить S$, (S - початковий символ граматики), у вхідному буфері w$ (w - вхідний ланцюжок), змінна InSym містить перший символ вхідного рядка. Програма, що використає таблицю аналізатора M для здійснення розбору, зображена на Рис. 4.5.

Алгоритм Рутисхаузера

Алгоритм Рутисхаузера є одним з найбільш ранніх. Його особливістю є припущення про повну скобкову структуру виразу. Під повним скобковим записом виразу розуміється запис, у якій порядок дій задається розміщенням дужок. Неявне старшинство операцій при цьому не враховується. Наприклад:

D:=((C-(B*L))+K)

Обробляючи вирази з повною скобковою структурою, алгоритм привласнює кожному символу з рядка номер рівня за наступним правилом:

1). якщо це дужка, що відкривається, або змінна, то значення збільшується на 1;

2). якщо знак операції або дужка, що закривається, то зменшується на 1. Для вираження (A+(B+3)) присвоювання значень рівня буде відбуватися, як описано в таблиці

N симв.	1 2 3 4 5 6 7 8 9
Символи рядка.	(A + (B * C))
Номера рівней	1 2 1 2 3 2 3 2 1

Алгоритм складається з наступних кроків:

1). виконати розміщення рівнів;

2). виконати пошук елементів рядка з максимальним значенням рівня;

3). виділити трійку - два операнда з максимальним значенням рівня й операцію, що укладена між ними;

4). результат обчислення трійки позначити допоміжної змінної;

5). з вихідного рядка видалити виділену трійку разом з її дужками, а на її місце помістити допоміжну змінну, що позначає результат, зі значенням рівня на одиницю менше, ніж у виділеної трійки;

6). виконувати п.п.2 - 5 доти, поки у вхідному рядку не залишиться одна змінна, що позначає загальний результат вираження.

Приклад розбору в таблиці

Генеруємі трійки	Вирази
	((((A+B) *3) /D)-E) 01234545434323210
+ A B - > R	(((R * C) / D) - E) 012 3 4 34 3 23 2 10
|* R C -> S	((S / D) - E) 012 3 23 2 1 0

З ранніх стекових методів розглядається алгоритм Бауэра й Замельзона. Алгоритм використовує два стеки й таблицю функцій переходу. Один стек використовується при трансляції виразу, а другий - під час інтерпретації. Позначення: Т - стек транслятора, Е - стек інтерпретатора. У таблиці переходів задаються функції, які повинен виконувати транслятор при розборі виразу. Можливі шість функцій при читанні операції із вхідного рядка:

f1 - заслати операцію із вхідного рядка в стек Т; читати наступний символ рядка;

- f2 - виділити трійку - взяти операцію з вершини стека Т и два операнда з вершини стека Е; допоміжну змінну, що позначає результат, занести в стек Е; заслати операцію із вхідного рядка в стек Т; читати наступний символ рядка;
- f3 - виключити символ зі стека Т; читати наступний символ рядка;
- f4 - виділити трійку - взяти операцію з вершини стека Т и два операнда з вершини стека Е; допоміжну змінну, що позначає результат, занести в стек Е; по таблиці визначити функцію для даного символу вхідного рядка;
- f5 - видача повідомлення про помилку;
- f6 - завершення роботи.

Таблиця переходів для алгебраїчних виражень буде мати вигляд (символ $ є ознакою порожнього стека або порожній рядок).

	Операція із вхідного рядка.	
	$ (+ - * /)	
Операція на вершині стека	$	6 1 1 1 1 1 5
	(	5 1 1 1 1 1 3
	+	4 1 2 2 1 1 4
	-	4 1 2 2 1 1 4
	*	4 1 4 4 2 2 4
	/	4 1 4 4 2 2 4

Алгоритм переглядає зліва направо вираз й циклічно виконує наступні дії: якщо черговий символ вхідного рядка є операндом, то він безумовно переноситься в стек E; якщо ж операція, то по таблиці функцій переходу визначається номер функції для виконання. Для виразу A+(3)*D приводиться наступна послідовність дій алгоритму.

Стік E	стік T	Вхідний символ	Номер функції	Трійка
$	$	A		
$A	$	+	1	
$A	$+	(	1	
$A	$+(	B		
$AB	$+(	-	1	
$AB	$+(-	C		
$ABC	$+(-	)	4	- B C ->R
$AR	$+(	)	3	
$AR	$+	*	1	
$AR	$+*	D		
$ARD	$+*		4	* R D ->Q
$AQ	$+		4	+ A Q ->S
$S	$		Кінець	

Зворотня польська нотація

Нехай задано простий арифметичний вираз виду:

(A+B)*(C+D)-E

Представимо цей вираз у вигляді дерева, у якому вузлам відповідають операції, а гілкам - операнди. Побудову почнемо з кореня, у якості якого обирається операція, що виконується останньою. Лівої гілці відповідає лівий операнд операції, а правої гілці - правий. Дерево виразу показано на рис. 1.7. Проведемо обхід дерева, під котрим будемо розуміти формування строки символів із

символів вузлів і галузей дерева. Обхід будемо проводити від самої лівої вітки вправо й вузол записувати у вихідну строку тільки після розсомотрення всіх його галузей. Результат обходу дерева має вигляд:

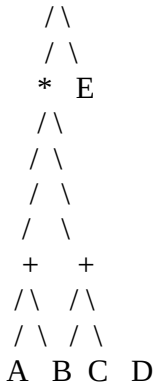


Рисунок 4.6 Дерево побудови зворотної польської нотації для виразу

(A+B)*(C+D)-E

Характерні особливості виразу складаються в проходженні символів операцій за символами операндів й у відсутності дужок. Такий запис називається обратним польським записом. Обратний польський запис володіє рядом чудових властивостей, которі перетворюють його в ідеальну проміжну мову при трансляції. По-перше, обчислення виразу, записаного в зворотьному польському записі, може проводиться шляхом одноразового просмотру, що є досить зручним при генерации об'єктного коду программ. Наприклад, обчислення виразу може бути проведено наступним образом:

Тут r1, r2 - допоміжні замінні. По-друге, одержання зворотьного польського запису з вихідного виразу може здійснюватися досить просто на основі простого алгоритма, запропонованого Дейкстрой. Для цього вводиться поняття стекового приорітету операцій

№	Аналізований рядок	Дія
0	A B + C D + * E -	r1=A+B
1	r1 C D + * E -	r2=C+D
2	r1 r2 * E -	r1=r1*r2
3	r1 E -	r1=r1-E
4	r1	Обчислення кінчене

Операція	Пріоритет
( ) + - / * **	0 1 2 3 4

Розглядається вихідна строка символів зліва направо, операнди переписуються у вихідну строку, а знаки операцій заносяться в стек на основі наступних міркувань:

а) якщо стік порожній, то операція із вхідної строки переписується в стек;

б) операція виштовхує зі стека всі операції з більшим або рівним пріорітетом у вихідну строку;

в) якщо наступний символ з вихідної строки є видкриваюча дужка, то він записується в стек;

г) закриваюча кругла дужка виштовхує всі операції зі стека до найближчої відкриваючої дужки, самі дужки у вихідну строку не переписуються, а знищують друг друга. Процесс одержання обратной польського запису виразу схематично представлен на рис. 4.7.

Символ, що аналізується	1	2	3	4	5	6	7	8	9	10	11	12	13	
Вхідний рядок	(	A	+	B	)	*	(	C	+	D	)	-	E	
Стан стека	(	(	+ (	+ (		* 	(*	(*	+ (*	+ (*	* 	- 	- 	
Вихідний рядок		A		B	+			C		D	+	*	E	-

Рисунок 4.7 Формування вихідного рядка

Можливі способи паралельної обробки арифметичних виразів. Для паралельної обробки вираження потрібне дотримання таких факторів:

1) Можливість поділу вираження на шматки довільної або рівної довжини і їх незалежна обробка.

2) Можливість збору отриманих результатів у єдину відповідь

3) Незалежність результату від черговості обробки частин вираження.

Не всі способи подання виражень мають такі можливості. Наприклад, зворотну польську нотацію буде досить нелегко розбити на незалежні частини. У той час як алгоритм Рутисхаузера й алгоритм Бауэра й Замельзона дозволяють розбити вираження на шматки й проводити обчислення кожної частини незалежно. Одна з переваг цих алгоритмів у тім, що вирази можна дробити на шматки, не звертаючи уваги на вміст і робити обчислення. Звичайно, існує ймовірність, що при такому підході не кожна частина виразу буде логічно завершена. Але ці алгоритми мають найціннішу властивість, навіть у логічно незавершеному виразі вони здатні обчислити логічно завершені ділянки. Якщо після обчислення кожної частини виразу, цю частину замінити на результат обчислення, то одержимо вираз, спрощений в порівнянні з вихідним. Отриманий вираз тепер можна знову розподілити між обчислювальними елементами для подальшого спрощення. У підсумку за певну кількість кроків ми одержимо вираз, що складається з одного елемента, що і буде результатом обчислень.

Тема 5

ГРАМАТИКИ-РОЗПІЗНАВАЧИ

Граматики-розпізнавачи (граматики, що розпізнають чи просто розпізнавачи) являють собою принципово інший спосіб завдання синтаксису формальних мов. Побудова і застосування цих граматик цілком орієнтовані на рішення задачі розбору, а не породження мови.

Кінцевий автомат - розпізнавач для мов классу 3 за Хомським.

Визначення. Кінцевий автомат-розпізнавач К – це п'ятірка;

$$K = (A, Q, \delta, q_1, z),$$

де A – вхідний алфавіт (включаючи порожній символ λ);

Q – множина станів керуючого пристрою;

δ – функція переходів автомата; $q'_j = \delta(q_j, a_i)$;

$q_1 \in Q$ – початковий стан керуючого пристрою;

$z \subseteq Q$ – множина заключних станів керуючого пристрою.

Множина Z складається з 2-х елементів: $\{q_{z_1}, q_{z_2}\}$; q_{z_1} – відповідає успішному закінченню розбору, q_{z_2} – закінчення розбору помилкового речення.

Кінцевий автомат – розпізнавач задається звичайно таблицею переходів. Заключні стани в клітках цієї таблиці звичайно позначаються: q_{z_1} – знак “+”; q_{z_2} – знак “–” чи порожня клітка.

Приклад 1. Побудувати розпізнавач для мови, що складається з ланцюжків 0 та 1, які обов’язково містять два підряд стоячих нуля.

Таблиця переходів має вигляд:

Таблиця 4.1 – Таблиця переходів.

	a		
	0	**1**	**λ**

	q_1	q_2	q_1	–
	q_2	q_3	q_1	–
	q_3	q_3	q_3	+

Тут A = {0,1}; Q = {q_1, q_2, q_3, “+”, “–”}; Z = {“+”, “–”}.

λ – порожній символ, який фактично позначає кінець ланцюжка. Припустимо, на вхід автомата поступає ланцюжок 1011001λλ…**.** Послідовність станів керуючого пристрою для цього ланцюжка: $q_1q_1q_2q_1q_1q_2q_3q_3$+. Для ланцюжка 01101λ… послідовність станів: $q_1q_2q_1q_1q_2q_3q_3$–.

Одним з найважливіших принципів, яким варто керуватися при побудові кінцевих автоматов-розпізнавачей, є запам'ятовування інформації про аналізоване речення за допомогою станів пристрою керування. Оскільки КА не може записувати проміжну інформацію на стрічку, єдиним способом запам'ятовування інформації є перехід у новий стан при виявленні у вхідному рядку символу (групи символів), наявність якого необхідно запам'ятати. Розглянемо на прикладі.

Приклад 2. Побудувати розпізнавач для дійсних констант, які записуються в однієї з наступних форм:

[±]<цифри>E[±]<цифри>,

[±]<цифри>.<цифри>,

[±]<цифри>.<цифри>E[±]<цифри>.

Наприклад: 5.32, 7E-2, 17.0E10.

При прогляданні вхідного рядку необхідно запом’ятовувати наявність цифр цілої частини (можливо зі знаком), крапки, цифр дрібної частини, символу “E”, знака, цифр порядку. Тому з урахуванням початкового стану q_1 КА буде мати 8 станів (не рахуючи двох заключних). Построїмо таблицю переходів КА.

Таблиця 5.2 – Таблиця переходів.

	a				
	“0”…“9”	**“.”**	**“E”**	**“±”**	**λ**

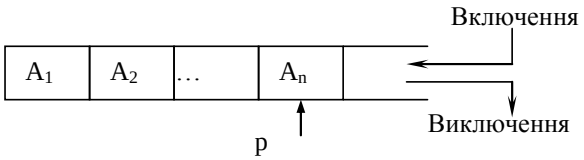
q	**q_1**	**q_3**			**q_2**	
	q_2	**q_3**				
	q_3	**q_3**	**q_4**	**q_6**		
	q_4	**q_5**				
	q_5	**q_5**		**q_6**		+
	q_6	**q_8**			**q_7**	
	q_7	**q_8**				
	q_8	**q_8**				+

В таблиці q_{z_1} відповідає знак "+" (число записано без помилок), q_{z_2} – порожня клітинка (запис помилкова).

Стан q_2 "зберігає" інформацію про наявність на стрічці "±"; стан q_3 – цифр цілої частини; стан q_4 – про наявність крапки; стан q_5 – цифр дрібної частини; стан q_6 – символу "Е"; стан q_7 – знак порядку. Робота КА закінчується або при зчитуванні порожнього символу, або при виявленні помилки.

Автомат з магазинною пам'яттю для розпізнавання мов классу 2 за Хомським

МП-автомат має робочу пам'ять, організовану за принципом стека чи магазина автоматичної зброї. Магазин (стік) являє собою лінійну структуру даних, відкриту для доступу тільки з одного кінця:



Комірка, що містить a_n, називається вершиною магазина, змінна p – покажчиком вершини. При включенні елемента p збільшується на 1, при виключенні зменшується на 1. Робота магазина описується формулою: "останнім прийшов – першим пішов".

Визначення – МП-автомат – це сімка:

$$M = (A, Q, P, \delta, q_1, p_1, z),$$

де A, Q, q_1, z – те ж, що й у КА;

δ– функція переходів і запису в робочу пам'ять, що задає відображення $Q*A*P \rightarrow Q*P^*$;

P – алфавіт робочої пам'яті;

P^* – множина слів в алфавіті P, включаючи порожнє;

p_1 – символ, що находиться в робочій пам'яті в початковий момент (початковий символ).

Функціонування МП-автомата являє собою послідовність кроків (тактів), у кожнім з яких керуючий пристрій виконує наступні елементарні операції:

1) читає символ у вершині магазина і можливо (але не обов'язково) на вхідній стрічці;

2) по q_j, a_i і p_l переходить у новий стан q'_j;

3) виробляє слово $\mu = p^{(1)}p^{(2)}\ldots p^{(n)}$(можливо порожнє);

4) виключає p_l з магазина і записує в магазин слово μ так, що б $p^{(1)}$ виявився на вершині магазина;

5) пересувається по вхідній стрічці на одну комірку вправо чи залишається на місці.

МП-автомат вважається заданим, якщо визначені множини Q, A, P, Z, символ p_1 і набір виразів, що задають функцію δ і записуваних у виді: $\delta(q_j, a_i, p_l) = (q'_j, \mu)$(*****). Якщо ліва частина виразу (*****) має вид $\delta(q_j, \lambda, p_l)$, це значить, що керуючий пристрій не оглядає символ на вхідній стрічці і не пересувається. МП-автомат **допускає** вхідний рядок, якщо він зупиняється в заключному стані q_z і при цьому магазин порожній; вхідний рядок вважається помилковим, якщо автомат зупиняється при не порожньому магазині, чи серед виразів (*) не знайшлося жодного, ліва частина якого відповідала б поточної конфігурації автомата.

У МП-автоматі, на відміну від кінцевого автомата, нагадування інформації про аналізоване речення виробляється як переходом у новий стан, так і записом слів у магазинну пам'ять. Особливістю мов класів 2 (контекстно-вільних мов), що відрізняє їх від мов класу 3, є наявність вкладених

конструкцій з довільною глибиною вкладеності. Прикладами таких конструкцій у мовах програмування є арифметичні вирази загального виду, вкладені умовні оператори, оператори циклу, складені оператори, блоки. Обробку вкладених конструкцій зручно здійснювати за допомогою магазина (стека), при цьому у вершині магазина знаходиться інформація про внутрішніх, оброблюваних у даний момент, елементах цих конструкцій, а в глибині – про зовнішніх, що включають у себе оброблювані внутрішні. Після завершення обробки внутрішнього елемента, інформація про нього виключається з магазина.

В арифметичних виразах такого типу вкладеними конструкціями, що вимагають для своєї обробки магазинної пам'яті, є підвирази, ув'язнені в дужки. Оскільки інші конструкції не можуть бути вкладеними, алфавіт магазинної пам'яті може складатися усього лише з одного символу, запис якого в магазин здійснюється, коли зустрічається черговий підвираз, укладений в дужки.

Приклад 3. Побудувати МП-автомат для розпізнавання арифметичних виразів, синтаксис яких описується наступною граматикою, що породжує:

<вираз> ::= <доданок> | <доданок>+<вираз> | <доданок>–<вираз>

<доданок> ::= <множник> | <доданок><множник> |*

<доданок>/<множник>

*<множник> ::= <имя> | <множник>**<множник> | (<вираз>)*

<і'мя> ::= A | B | C | D

Правильними в цій граматиці є, наприклад, вирази: A*(B–D)**C; (((B–D)*A+C)/B)*D-C.

В арифметичних виразах такого типу укладеними конструкціями, потребуючеми для своєї обробки магазинної пам'яті, є підвирази, які взяті в лапки. Оскільки інші конструкції не можуть бути вкладеними, алфавіт магазинної пам'яті може складатися лише з одного символу, запис якого у магазин робиться, коли зустрічається черговий підвираз, який взят у лапки.

Функція переходів автомату:

$\delta(q_1,b,s)=(q_2,s)$;

$\delta(q_2,z,s)=(q_1,s)$;

$\delta(q_1,"(",s)=(q_1,ss)$;

$\delta(q_2,")",s)=(q_2,\lambda)$;

$\delta(q_2,\perp,s)=(q_z,\lambda)$.

Тут b – одна з букв *A*, *B*, *C*, *D*; z – знак операції +, –, *, /, **; s – символ алфавіту магазинної пам'яті; ⊥ – умовний символ-ознака кінця вхідного рядку. У початковому стані в магазині записан символ S.

Промоделюємо роботу автомата. Візьмемо вираз A*(((B–D)*C–D)/B)+D⊥. В ньому 20 символів, а оскільки керуючий пристрій МП-автомата на кожному кроці зсовується на одну комірку вхідної стрічки, автомат повинен зробити максимум 20 кроків. Наведемо їх в у вигляді таблиці.

Таблиця 5.3 – Моделювання роботи автомату.

№ шага	a_i	q_j	стек	№ шага	a_i	q_j	стек
1.	A	q_1	s	11.	C	q_1	sss
2.	*	q_2	s	12.	–	q_2	sss
3.	(	q_1	s	13.	D	q_1	sss
4.	(	q_1	ss	14.	)	q_2	sss
5.	(	q_1	sss	15.	/	q_2	ss
6.	B	q_1	ssss	16.	B	q_1	ss
7.	–	q_2	ssss	17.	)	q_2	ss
8.	D	q_1	ssss	18.	+	q_2	s
9.	)	q_2	ssss	19.	D	q_1	s
10.	*	q_2	sss	20.	⊥	q_2	s

На останньому, 21-м кроці автомат переходить в q_z, очищає магазин і зупиняється.

Розберемо ще один вираз: ((B+D)*A–C⊥

Таблиця 5.4 – Моделювання роботи автомату.

№ кроку	a_i	q_j	крок	№ кроку	a_i	q_j	крок
1.	(	q_1	s	7.	*	q_2	ss
2.	(	q_1	ss	8.	A	q_1	ss
3.	B	q_1	sss	9.	–	q_2	ss
4.	+	q_2	sss	10.	C	q_1	ss

5.	D	q_1	sss	11.	$\perp$	q_2	ss
6.	)	q_2	sss	12.		q_z	s

Автомат зупинився в заключному стані, але стек не порожній. Значить початкове припущення помилкове (помилка – відсутність закриваючої дужки).

Тема 6

ПРОЕКТУВАННЯ СИНТАКСИЧНОГО АНАЛІЗАТОРА ДЛЯ ГРАМАТИКИ ПЕРЕДУВАННЯ

Синтаксис більшості сучасних мов програмування описується контекстно-вільними граматиками (КВ-граматиками). До цього класу граматик відноситься граматика (простого) передування, що дозволяє конструювати як вручну, так і автоматично досить прості синтаксичні аналізатори (розпізнавачи). Алгоритм розпізнавача для граматики передування використовує лівосторонній висхідний метод діалізу, що дозволяє знаходити в поточній сентенціальній формі основу (тобто саму ліву фразу, що безпосередньо приводиться,) і замінювати її нетермінальним символом. Для того щоб спосіб відшукання основи був однобічним без "тупиків" і "повернень", КВ-граматика повинна мати визначені властивості. Граматика передування, наприклад, відрізняється тим, що в ній:

1) Для кожної упорядкованої пари термінальних і нетермінальних символів (S_i, S_j) виконується не більш ніж одна з трьох відносин передування:

а) $S_i \doteq S_j$, якщо і тільки якщо існує правило

$$U \to xS_iS_jY$$

б) $S_i <\cdot S_j$, якщо і тільки якщо існує правило

$$U \to xS_iDY \text{ і висновок } D \overset{*}{\Rightarrow} S_jZ;$$

в) $S_i \cdot > S_j$, якщо і тільки якщо існує правило

$$U \to xCS_iY \text{ і висновок } C \overset{*}{\Rightarrow} ZS_i, \text{ чи – правило}$$

$$U \to xCDY \text{ і висновки } C \overset{*}{\Rightarrow} ZS_i,\ D \overset{*}{\Rightarrow} S_jW$$

2) Різні правила, що породжують, мають різні праві частини, де U, C, D - нетермінальні символи, x, Y, Z, W - будь-які рядки, можливо порожні; $\overset{*}{\Rightarrow}$ - знак породження рядка.

Ці характерні властивості даної граматики дозволяють на кожному кроці синтаксичного аналізу легко виділяти основу шляхом відшукання в рядку $S_i \ldots S_j$ самої лівої групи символів, зв'язаних відносинами передування виду

$$S_{i-1} < \cdot \underbrace{S_i \doteq \ldots \doteq S_j}_{основа} \cdot > S_{j+1}$$

Висловлена ідея пошуку основи може бути використана для побудови лівостороннього розпізнавача у виді автомата з магазинною пам'яттю (малюнок 5.1).

Програма роботи керуючого пристрою повинна виконувати наступний алгоритм:

а) символи вхідного рядка по черзі листуються в стек доти, поки між символом у вершині стека S_i і черговим символом вхідного рядка S_j не виникне відношення $S_i \cdot > S_j$;

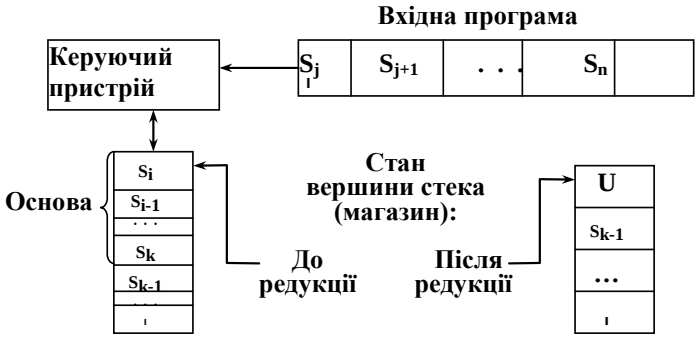


Рисунок 6.1 – Розпізнавач для граматики передування

б) тоді стік проглядається в напрямку від вершини до початку доти, поки між двома черговими символами не з'явиться відношення $S_{k-1} \cdot > S_k$ а з цього випливає, що частина стека від символу S_k до символу у вершині S_i є основа;

в) тепер серед правил, що породжують, знаходиться правило $U \to S_k \ldots S_i$ виду й у стеці основа прямо редукується до нетермінального символу U. При необхідності викликається семантична підпрограма, що обробляє основу,

переводячи її на проміжну чи вихідну мову, і описаний вище процес повторюється.

Якщо при аналізі виявиться, що між сусідніми символами S_i і S_j не існують відносини передування, то це свідчить про синтаксичну помилку.

Для знаходження відносин передування необхідно, насамперед, визначити дві допоміжні множини: множина самих лівих L(U) і множина самих правих R(U) символів щодо нетермінального символу U. Ці множини можна визначити рекурсивно:

$$\begin{aligned} L(U) &= \{S \mid \exists(U \to SZ) \vee \exists(U \to U'Z' \wedge S \in L(U'))\}, \\ R(U) &= \{S \mid \exists(U \to ZS) \vee \exists(U \to Z'U' \wedge S \in R(U'))\} \end{aligned} \qquad (1)$$

Дані множини використовуються для виявлення відносин передування за наступними правилами:

$$\begin{aligned} & S_i \doteq S_j \leftrightarrow \exists(U \to xS_iS_jY) \\ & S_i <\cdot S_j \leftrightarrow \exists(U \to xS_iDY) \wedge S_j \in L(D) \\ & S_i \cdot > S_j \leftrightarrow \exists(U \to xCS_jY) \wedge S_i \in R(C) \\ & \vee \exists(U \to xCDY) \wedge S_i \in R(C) \wedge S_j \in L(D) \end{aligned} \qquad (2)$$

Відносини передування варто записати у виді матриці передування, що представляє собою таблицю з двома входами, входами в таблицю є попередній S і наступний символи рядка, що приводиться, а в її клітках записуються відносини передування.

Приклад 1 Побудувати методом передування розпізнавач для обраної мови програмування на прикладі синтаксичного розбору одного чи декількох операторів, описаних граматикою передування.

1. Для заданих операторів вхідної мови складемо множину правильних конструкцій. Синтаксис одержання цих можливих конструкцій запишемо у виді граматики передування.

Граматика **G = (T, N, P, A)** , де

T={+,-,,/,цф}*

N={<у>,<ав>,<з>,<м>,<ц>}
A = <у>
P - безліч правил, що породжують:

<у> ::= ⊥<ав>⊥
<ав> ::= <з> | + <з> | - <з> | <ав> + <з> | <ав> - <з>
<с> ::= <м> | <з> * <м> | <з> / <м>
<м> ::= <ц> | (<ав>)
<ц> ::= цф | <ц> цф

2. Побудуємо розпізнавач для синтаксичного аналізу трансльованих операторів. З цією метою на початку для всіх нетермінальних символів граматики за допомогою правил (1) визначимо допоміжні множини L(U) і R(U).

Таблиця 6.1 – Множини L(U) і R(U)

U	L(U)	R(U)
<у>	⊥	⊥
<ав>	<с>,+,-, <ав>,<м>,<ц>,(,цф	<с>,<м>,<ц>,),цф
<с>	<м>,<з>,<ц>,(,цф	<м>,<ц>,),цф
<м>	<ц>,(,цф	<ц>,),цф
<ц>	цф,<ц>	цф

3. Складемо матрицю передування, що містить відносини передування для кожної упорядкованої пари символів рядка, що приводиться. Алгоритм заповнення матриці визначається правилами (2).

Таблиця 6.2 – Матриця передування.

	⊥	**<ав>**	**<с>**	**<м>**	**<ц>**	**цф**	**z1**	**z2**	**(**	**)**
⊥		=,<	<	<	<	<	<		<	
<ав>	=						=			=
<с>	>						>	=		>
<м>	>						>	>		>
<ц>	>					=	>	>		>
цф	>					>	>	>		>
z1			=,<	<	<	<			<	
z2				=	<	<			<	
(		=,<	<	<	<	<	<		<	
)	>						>	>		>

…, де z1={- , +} і z2={* , /}

4. Т.к. існують пари символів, для яких відносини передування неоднозначні, то **G** не є граматикою передування. Для усунення неоднозначності потрібно змінити граматику, ми усунемо рекурсивність у визначених <ав> і <з>. Визначимо нову граматику **G'** наступними правилами:

Граматика **G' = (T, N, P, A)** , де

T={+,-,,/,цф}*
N={<y>,<в~><ав>,<з>,<с~>,<м>,<ц>}
A = <y>
P - безліч правил, що породжують :

<y> ::= ⊥<в~>⊥
<в~> ::= <ав>
<ав> ::= <с~> | + <с~> | - <с~> | <ав> + <с~> | <ав> - <с~>
<с~>::=<с>
*<с> ::= <м> | <з> * <м> | <з> / <м>*
<м> ::= <ц> | (<ав>)
<ц> ::= цф | <ц> цф

5. Складемо остаточні безлічі L(U) і R (U).

Таблиця 6.3 – Множини L(U) і R(U).

U	L(U)	R(U)
<y>	⊥	⊥
<в~>	<с~>,+,- <ав>,<м>,<з>,<ц>,(,цф	<с~>,<м>,<ц>,),цф
<ав>	<с~>,+,- <ав>,<м>,<з>,<ц>,(,цф	<с~>,<м>,<ц>,),цф
<с~>	<м>,<з>,<ц>,(,цф	<м>,<ц>,),цф
<с>	<м>,<з>,<ц>,(,цф	<м>,<ц>,),цф
<м>	<ц>,(,цф	<ц>,),цф
<ц>	цф,<ц>	цф

6. Складемо матрицю передування.

Таблиця 6.4 – Матриця передування.

	⊥	**<в~>**	**<ав>**	**<с~>**	**<с>**	**<м>**	**<ц>**	**цф**	**z1**	**z2**	**(**	**)**

⊥		=		<	<	<	<	<	<		<	
<в~>	=											
<ав>									=			=
<с~>	>								>			>
<с>												
<м>	>								>	>		>
<ц>	>							=	>	>		>
цф	>							>	>	>		>
z1				=	<	<	<	<			<	
z2						=	<	<			<	
(			=	<	<	<	<	<	<			
)	>								>	>		>

…, де z1={- , +} і z2={* , /}

У реальних мовах програмування число символів граматики досить велико, тому матриця передування виходить дуже великих розмірів. Для скорочення обсягу пам'яті матрицю варто зберігати в "упакованому" виді (по 2 біти на елемент) чи використовувати функції передування разом з матрицею сполучуваності символів [2].

Тепер розглянемо синтаксичний розбір наступного умовного вхідного рядка:

⊥ (-23)+5*6 ⊥

використовуючи для цього розроблений розпізнавач передування (рис. 5.1), таблицю правил, що породжують, (табл. 5.5) і матрицю передування.

Таблиця 6.5 – Таблиця правил граматики, що породжують.

Номер правила	Правило, що породжує
1	<y> ::= ⊥<в~>⊥
2	<в~> ::= <ав>
3	<ав> ::= <с~> | + <с~> | - <с~> | <ав> + <с~> | <ав> - <с~>

4	<c~>::=<c>
5	<c> ::= <м> | <з> * <м> | <з> / <м>
6	<м> ::= <ц> | (<ав>)
7	<ц> ::= цф | <ц> цф

Таблиця 6.6 – Тестовий приклад. Трансляція рядка методом передування.

Стек розпізнавача	Відношення	Вхідний рядок	Правило, яке використане
⊥	<	(-23)+5*6 ⊥	-
⊥ (	<	-23)+5*6 ⊥	-
⊥ (-	<	23)+5*6 ⊥	-
⊥ (-2	>	3)+5*6 ⊥	цифра в термін. символ
⊥ (- цф	>	3)+5*6 ⊥	7
⊥ (- <ц>	=	3)+5*6 ⊥	-
⊥ (-<ц>3	>	)+5*6 ⊥	цифра в термін. символ
⊥ (-<ц> цф	>	)+5*6 ⊥	7
⊥ (-<ц>	>	)+5*6 ⊥	6
⊥ (-<м>	>	)+5*6 ⊥	5
⊥ (-<з>	>	)+5*6 ⊥	4
⊥ (-<c~>	=	)+5*6 ⊥	-
⊥ (-<c~>)	>	+5*6 ⊥	3
⊥ <ав>	=	+5*6 ⊥	-
⊥ <ав>+	<	5*6 ⊥	-
⊥ <ав>+5	>	*6 ⊥	цифра в термін. символ
⊥ <ав>+цф	>	*6 ⊥	7
⊥ <ав>+<ц>	>	*6 ⊥	6
⊥ <ав>+<м>	>	*6 ⊥	5
⊥ <ав>+<з>	=	*6 ⊥	-
⊥ <ав>+<з>*	<	6 ⊥	-
⊥ <ав>+<з>*6	>	⊥	цифра в термин. символ
⊥ <ав>+<з>* цф	>	⊥	7
⊥ <ав>+<з>* <ц>	>	⊥	6
⊥ <ав>+<з>* <м>	>	⊥	5
⊥ <ав>+<з>	>	⊥	4

⊥ <ав>+<с~>	>	⊥	3
⊥ <ав>	>	⊥	2
⊥ <в~>	=	⊥	1
⊥ <у>		⊥	

Алгоритм розпізнавача універсальний у класі граматик передування. Якщо, допустимо, необхідно змінити вихідну мову, то в трансляторі досить поміняти лише семантичні підпрограми. А якщо змінюється вхідна мова, то необхідно змінити тільки таблицю правил, що породжують, матрицю передування і семантичні підпрограми.

Практичне застосування методу передування утрудняється неоднозначністю відносин передування, що часто зустрічається в граматиках реальних мов програмування. Загального алгоритму приведення вихідної КВ-граматики до граматики передування немає. На практиці застосовуються приватні алгоритми усунення конфліктів передування [2] чи використовується розширене передування [1], що дозволяє забезпечити однозначність відносини шляхом використання контексту не з двох, а з трьох символів граматики.

Тема 7

ПРОЕКТУВАННЯ РОЗПІЗНАВАЧА ДЛЯ ГРАМАТИКИ З ОПЕРАТОРНИМ ПЕРЕДУВАННЯМ

Метод операторного передування застосуємо тільки до граматик з операторним передуванням. Граматикою з операторним передуванням називають граматику класу 2 за Хомським, у якій:

1) праві частини правил, що породжують, не містять поруч стоячих нетермінальних символів, тобто в граматиці немає правил виду

U → X C D Y , де X, Y ∈ T, A, C, D ∈ N;

2) для кожної упорядкованої пари термінальних символів A и B виконується не більш ніж одне з трьох наступних відносин передування:

а) A = У, якщо і тільки якщо існує правило

U → X A B Y чи U → X A C B Y;

б) A < B, якщо і тільки якщо існує правило

U (X A C Y і висновок C(B Z чи C(D B Z;

в) A > B, якщо і тільки якщо існує правило

U (X C B Y і висновок C(Z A чи C(Z A D;

3) різні правила, що породжують, мають різні праві частини;

Де U, C, D - нетермінальні символи; X, Y, Z – будь-які рядки, у тому числі порожні. Цією граматикою можна описати будь-яку реальну мову програмування.

Перша вимога дозволяє обмежитися установленням відносин передування тільки для термінальних символів, що значно скорочує розмір матриці передування і підвищує ефективність алгоритму розбору в порівнянні з методом простого передування. Алгоритм розбору відшукує для редукції не основу, а так називану первинну фразу.

Первинна фраза - це фраза, що не містить ніякої іншої фрази, крім самої себе, і утримуюча, принаймні, один термінальний символ. Алгоритм розбору майже такий ж як у методі простого передування (див. лабораторну роботу 5).

Різниця полягає в тому, що в методі операторного передування, по-перше, використовуються тільки відносини термінальних символів і, по-друге, у знайденому для редукції правилі U (X права частина може відрізнятися від виділеної первинної фрази нетермінальними символами, що не гарантує однозначності даного розбору. Для усунення неоднозначності розбору варто використовувати семантичні підпрограми. Один зі станів стека і вхідний рядок у процесі синтаксичного розбору методом операторного передування показане на рис. 7.1.

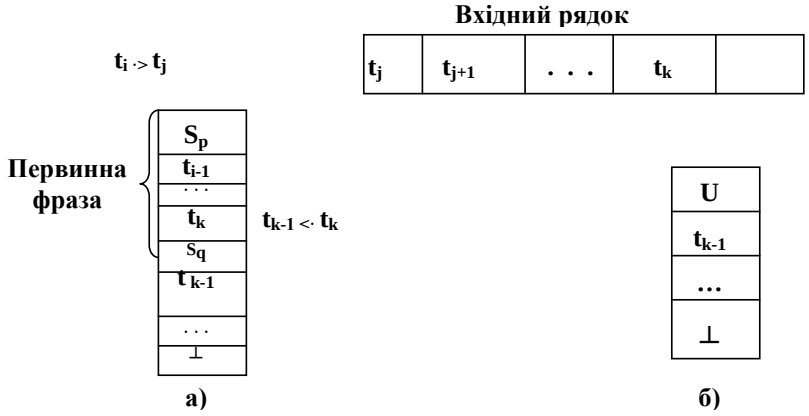


Рисунок 7.1 - Стан вхідного рядка і стека: а) до редукції; б) після редукції.

Для знаходження відносин операторного передування необхідно попередньо визначити множину самих лівих L.(U) і множину самих правих R.(U) термінальних символів щодо нетермінального символу U. Алгоритм відшукання цих множин визначається наступними формулами:

$$L_t(U) = \{t \mid \exists(U \to tZ) \vee \exists(U \to CtZ) \vee (U' \in L(U) \wedge t \in L_t(U'))\} \quad (1)$$

$$R_t(U) = \{t \mid \exists(U \to Zt) \vee \exists(U \to ZtC) \vee (U' \in R(U) \wedge t \in R_t(U'))\} \quad (2)$$

де t – термінальний символ.

За допомогою цих множин і правил граматики, що породжують, тепер можна легко знайти відносини операторного передування за наступними правилами:

$$t_i \doteq t_j \leftrightarrow \exists(U \to xt_i t_j Y) \vee \exists(U \to xt_i Ct_j Y); \qquad (4)$$

$$t_i <\cdot t_j \leftrightarrow \exists(U \to xt_i CY) \,\&\, \exists t_j \in L_t(C); \qquad (5)$$

$$t_i \cdot > t_j \leftrightarrow \exists(U \to xCt_j Y) \,\&\, t_i \in R_t(C). \qquad (6)$$

Алгоритм побудови матриці операторного передування.

1. Для кожного нетермінального символу V граматики G_1 будуються множини L_t(V) та R_t(V). Для цього

а) відшукуються множини L(V) та R(V).

б) для кожного нетермінального символу V :

- відшукуються правила граматики G_1 з правими частинами виду tz та Ctz. Термінальні символи t фіксуються в якості елементів множини L_t(V);

- відшукуються правила з правими частинами виду zt та ztC. Термінальні символи t

фіксуються в якості елементів множини R_t(V).

В результаті отримуємо таблицю 6.1 для граматики G_1:

П::=⊥В⊥

В::=Т | В+Т

Т::=М| Т×М

М::=U|(В)

Таблиця 7.1 - Таблиця множин L_t(V) и R_t(V).

V	L_t(V)	R_t(V)
П	⊥	⊥
В	+	+
Т	×	×
М	U,(	U,)

в) Для кожного нетермінального символу V :

- проглядається множина L(V) та відшукуються вхідні в нього нетермінальні символи V', V'', … . Далі множина L_t(V) доповнюється символами, які входять у L_t(V'), L_t(V''), … та які не входять у L_t(V).

- проглядається множина R(U) та відшукуються вхідні в нього нетермінальні символи V', V'', … . Далі множина R_t(V) доповнюється символами, які входять у R_t(V'), R_t(V''), … та які не входять у R_t(V).

В результаті отримуємо таблицю 6.2.

Таблиця 7.2 - Таблиця множин L_t(V) и R_t(V).

V	L_t(V)	R_t(V)
П	⊥	⊥
В	+, ×, U, (	+, ×,U,)
Т	×, U, (	×, U,)
М	U,(	U,)

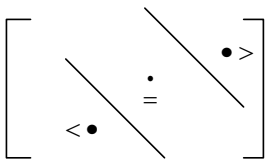
2. Складається матриця операторного передування. Для цього проглядаються праві частини правил граматики G_1, що породжують.

- за правилом (6) визначаються відносини $\dot{=}$.
- за правилом (7) з використанням таблиці 6 визначаються відносини $<\bullet$.
- за правилом (8) з використанням таблиці 6 визначаються відносини $\bullet>$.

Таблиця 7.3 - Матриця операторного передування для граматики G_1.

t_j \ t_i	(	U	×	+	)	⊥
)			$\bullet>$	$\bullet>$	$\bullet>$	$\bullet>$
U			$\bullet>$	$\bullet>$	$\bullet>$	$\bullet>$
×	$<\bullet$	$<\bullet$	$\bullet>$	$\bullet>$	$\bullet>$	$\bullet>$
+	$<\bullet$	$<\bullet$	$<\bullet$	$\bullet>$	$\bullet>$	$\bullet>$
(	$<\bullet$	$<\bullet$	$<\bullet$	$<\bullet$	$\dot{=}$	
⊥	$<\bullet$	$<\bullet$	$<\bullet$	$<\bullet$		$\dot{=}$

Одержана матриця операторного передування має вигляд,



отже, існують функції операторного передування, які дозволяють зберігати матрицю в упакованому вигляді.

Функції передування

У реальних мовах матриця передування має розмірність порядку 300x300. Щоб неї скоротити і заощадити пам'ять для її збереження, уводяться дві целочисленные функції передування f (S_i) і g (S_j) , такі, що

$$f(S_i) = g(S_j) \leftrightarrow S_i \doteq S_j$$

$$f(S_i) < g(S_j) \leftrightarrow S_i <\bullet S_j \qquad (*)$$

$$f(S_i) > g(S_j) \leftrightarrow S_i \bullet> S_j$$

Використання функцій передування скорочує потреби пам'яті з м2 до 2м величин.

Для відшукання значень функцій передування використовуємо наступний алгоритм:

1. Покласти f (S_i) = g (S_j) = 1, i,j = 1,…,m
2. Переглянути кожен рядок матриці передування, починаючи з першої. Якщо $S_i \bullet> S_j$, а f (S_i) ≤ g (S_j), то покласти f (S_i) = g (S_j) + 1.
3. Переглянути кожен стовпець матриці передування, починаючи з 1-го. Якщо $S_i <\bullet S_j$, а f (S_i) ≥ g (S_j) , то покласти g (S_j) = f (S_i) + 1
4. Переглянути всі позиції матриці передування. Якщо $S_i \doteq S_j$, а f (S_i) ≠ g (S_j) , то покласти f (S_i) = g (S_j) = max|f (S_i),g (S_j)|;
5. Повторювати виконання пунктів 2, 3, і 4 доти, поки не будуть знайдені функції, що задовольняють (*), (тобто значення функцій f і g перестануть

змінюватися), чи поки значення однієї з функцій не перевершить величину 2м. У першому випадку задача буде вирішена, а в другому – установлено, що для даної граматики функції передування не існує.

Обчислення функцій передування для $G_1^{"}$

Номер ітерації	S_i чи S_j	(	U	M	×	T	T’	+	B’	)	B	⊥	коментарі
0	f(S_i)	1	1	1	1	1	1	1	1	1	1	1	Пункт 1
	g(S_j)	1	1	1	1	1	1	1	1	1	1	1	
1	f(S_i)	1	2	2	1	2	2	1	2	2	1	1	Пункт 2
	g(S_j)	2	2	2	1	2	2	1	2	1	1	1	
	f(S_i)	1	2	2	2	2	2	2	2	2	1	1	Пункт 3
	g(S_j)	2	2	2	2	2	2	2	2	1	1	1	
2	f(S_i)	1	3	3	2	3	3	2	2	3	1	1	2 – не викон. умови f(S_i) ≤ g(S_j) Пункт 2
	g(S_j)	3	3	3	2	3	2	2	2	1	1	1	
	f(S_i)	1	3	3	3	3	3	2	2	3	1	1	Пункт 3
	g(S_j)	3	3	3	3	3	2	2	2	1	1	1	
3	f(S_i)	1	4	4	3	3	3	2	2	4	1	1	Пункт 2
	g(S_j)	4	4	3	3	3	2	2	2	1	1	1	
	f(S_i)	1	4	4	3	3	3	2	2	4	1	1	Пункт 3
	g(S_j)	4	4	3	3	3	2	2	2	1	1	1	
4	f(S_i)	1	4	4	3	3	3	2	2	4	1	1	
	g(S_j)	4	4	3	3	3	2	2	2	1	1	1	
	f(S_i)	1	4	4	3	3	3	2	2	4	1	1	
	g(S_j)	4	4	3	3	3	2	2	2	1	1	1	

- // - с m^2 до 2m. Але використання функцій передування призводить до зникнення порожніх позицій в матриці передування, які дозволяють здійснювати контроль несумісності символів. Тому або проводиться попередній синтаксичний контроль, або використовується додаткова матриця сумісності, елемент якої = 1, якщо пара сумісна і 0 – інакше.

На підставі отриманих значень функції можно перебудувати матрицю передування у вигляді:

	g(S_j)	4	4	3	3	3	2	2	2	1	1	1
f(S_i)	S_j / S_i	(	U	M	×	T’	T	+	B’	)	B	⊥
4	(				●>			●>		●>		●>
4	U				●>			●>		●>		●>
3	M				●>			●>		●>		●>
3	×	<●	<●	$\overset{\bullet}{=}$								
3	T’				$\overset{\bullet}{=}$			●>		●>		●>
2	T							●>		●>		●>
2	+	<●	<●	<●		<●	$\overset{\bullet}{=}$					
2	B’							$\overset{\bullet}{=}$		●>		●>
2	B									$\overset{\bullet}{=}$		$\overset{\bullet}{=}$
1	(	<●	<●	<●		<●	<●		<●		$\overset{\bullet}{=}$	
1	⊥	<●	<●	<●		<●	<●		<●		$\overset{\bullet}{=}$	

Алгоритм аналізу, що базується на граматиці з операторним передуванням

1. Породження правильних конструкцій заданих операторів вхідної мови програмування описати граматикою з операторним передуванням. При складанні набору синтаксичних правил варто скористатися відділенням граматики з операторним передуванням.

2. Для кожного нетермінального символу U складеної граматики G визначаються множини $L_t(U)$ і $R_t(U)$. Для цього застосовується наступний алгоритм, визначений формулами (1):

2.1 Знаходяться множини L(U) і R(U) за правилами, описаними у лабораторній роботі 5.

2.2 Для кожного нетермінального символу U:

а) відшукуються правила граматики G із правими частинами виду t і Ct, де C - нетермінальний символ, а Z – будь-який рядок, бути може, порожній. Термінальні символи t фіксуються як елементи множини $L_t(U)$;

б) відшукуються правила з правими частинами виду Zt і Zt. Термінальні символи t фіксуються як елементи множини $R_t(U)$.

2.3. Для кожного нетермінального символу U:

а) проглядається множина $L_t(U)$ і відшукуються вхідні в нього нетермінальні символи U', U",... . Потім множина $L_t(U)$ доповнюється символами, що входять у $L_t(U')$, $L_t(U")$... і не вхідними в $L_t(U)$;

б) аналогічно проглядається множина $R_t(U)$ і відшукуються вхідні в неї нетермінальні символи U, U',... . Потім множина $R_t(U)$ доповнюється символами, що входять у $R_t(U')$, $R(U_t)$... і не вхідними в $R_t(U)$.

3. Побудувати матрицю операторного передування.

Для цього необхідно переглянути праві частини правил граматики G, що породжують. За правилом (2) визначаються відносини ≐. За правилом (3) з використанням множини $L_t(U)$ визначаються відносини <• . За правилом (4) і множиною $R_t(U)$ визначаються відносини •>.

4. Скласти таблицю правил, що породжують, розпізнавача стосовно до граматики G. З таблиці виключити правила, що містять у правій частині тільки нетермінальні символи, а в правих частинах всіх інших правил нетермінальні символи замінити символом N. Крім того, таблицю доповнити семантичними підпрограмами обробки трансльованих операторів.

Транслятор

Транслятор для вхідної ямови, що породжується граматикою передування, можна побудувати по класичній схемі (лексичний аналіз, розпізнавач, генератор)

Приклад: Розібрати вираз

а + в + с,

який після роботи лексичного аналізатору було приведено до вигляду

$$U \overset{p_1}{+} U \overset{p_2}{\times} U$$

Відповідно граматиці $G_1^{'}$ и алгоритму розбору

⊥U⊥+ U × U ⊥
⊥M⊥+ b × c ⊥
⊥T'⊥+ b × c ⊥
⊥T ⊥+ b × c ⊥
⊥B'⊥+ b × c ⊥
⊥B'⊥+ M × c ⊥
⊥B'⊥+ T' × c ⊥
⊥B'⊥+ T' × M ⊥
⊥B'⊥+ T'⊥
⊥B'⊥+ T ⊥
⊥B'⊥
⊥B ⊥
П

Список рекомендованої літератури

1. Грис Д. Конструирование компиляторов для ЦВМ.- М.: Мир, 1975. - 544 с.
2. Лебедев В.Н. Введение в системы программирования. - М.: Статистика, 1975. - 312 с.
3. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов, - М.: Мир, 1979.- 654 с.
4. Хантер Р. Проектирование и конструирование компиляторов - М.: Финансы и статистика, 1984.-232 с.
5. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. – М.: Мир, 1978.- т.1 - 612 с., т.2 - 487 с.
6. Касьянов В.Н., Поттосин И.В. Методы построения трансляторов - Новосибирск: Наука, 1986.-344 с.
7. Пратт Т. Языки программирования: разработка и реализация. - М.: Мир, 1979. - 574 с.

8. Болье Л. Методы построения компиляторов.//Языки программирования, - М.: Мир, 1972, с. 87-277.
9. Донован Дж. Системное программирование. - М.: Мир, 1975. - 540 с.
10. Васючкова Т.Л. и др. Языки программирования ДОС ЕС ЭВМ. Краткий справочник. - М.: Статистика, 1977.-152 с.
11. Йенсен К., Вирт Н. Паскаль. Руководство для пользователя и описание языка. - М.: Финансы и статистика, 1982.-151 с.
12. Хопгуд Ф. Методы компиляции. – М.: Мир, 1972. – 160 с.
13. Фостер Дж. Автоматический синтаксический анализатор. – М.: Мир, 1975. – 72с.
14. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979. – 536 с.
15. Ингерман П. Синтаксически ориентированный транслятор. – М.: Мир, 1969. – 176 с.
16. Савинков В.М., Сидоренко Л.А. Синтаксический транслятор анализирующего типа. – М.: Статистика, 1972. – 88 с.
17. Ахо А., Ренди С., Ульман Дж. Компиляция. – М.: Мир, 2002. – 834 с.
18. Василеску Ю. Прикладное программирование на языке Ада. – М.: Мир, 1990. – 352 с.
19. Павловская Т.А. С/С++ Программирование на языке высокого уровня: учебник для вузов. – С.-П.: Питер, 2005. – 272 с.
20. Прата Стивен. Язык программирования С++. Лекции и упражнения. – К.: ДиаСофт, 2005. – 1104 с.
21. Дмитрієва О.А. Методичні вказівки і завдвння до лабораторних робіт за курсом “Теорія синтаксичного аналізу і компіляції”. – Донецьк: ДонНТУ, 2005 р., 72 с. (укр. мовою).