Пустоваров Володимир Ілліч 536 ауд.

Курсова робота Стандарт ЄСПД (ESPD)

① Склад поясн. записки: ① Титульний аркуш: Мін., Київ, фак. каф., Пояснювальна записка на тему "…" з курсу "Сист. прог." Робота до захисту допущена Керівник, Робота захищена з оцінкою, виконав студент і підпись. Київ … р.
② Тех. завдання (завдання). З курсу, на тему, Кафедра,
③ Вимоги (до програм) – вхідні дані, форма предст., формати результатів, обмеження на методи, рекомендована література, завдання на роботу видав керівник, завдання прийняв студент + підписи.
④ Вступ. Обґрунтування теми, навіщо потрібно розв'язувати такі задачі, вибір алгоритму чи методу розв'язання. Блок-схема алгоритму, опис роботи алгоритму за блок-схемою. Інструкція з експлуатації програми.
⑤ Методика перевірки програми та контрольні приклади на яких перевіряється програма. Результати експериментів по перевірці прогр.
⑥ Висновки. Чому програма буде працювати правильно, помилки, недоліки, їх виправлення.
⑦ Лістинг у додатку.

Розділ 1. Елементи системних програм. Програми роботи з таблицями, графіками і автоматами

Розділ 2. Основні етапи роботи транслюторів, побудова лексичних, синтаксичних, семантичних аналізаторів, та реалізація інших видів обробки – інтерпретація і генерація кодів.

Розділ 3. Побудова програм для ОС. Основні механізми та інструменти, що вик. в цих програмах.

Класифікація системних програм

Поділяються на 2 групи – системні управляючі (програми ОС), які автоматизують роботу користувача, оператора чи адміністратора ОС; системні обробляючі програми – це програми, що автоматизують працю програміста, намагаються автоматизувати працю системіста системної системи, до цих відносять

транслятор з мов програмування та інших комп'ютерних мов. Транслятори поділяють на 2 осн. класи — компілятори які звичайно формують виконавчі коди у машинному форматі комп. і на інтерпретатори — які формують внутрішнє подання, і на його основі вик. деякі оператори один за одним і також таким чином розв'язують задачу, запрограмовану на якійсь комп'ютерній мові.

Основні компоненти ОС

1) Програми завантаження ОС, — звичайно запускаються BIOS, після завантаження робота ОС. Незалежно від виду ОС, вона ~~ОС~~ виконує наступні функції:

2) Завантаження задач з носіїв, найчастіше у файловому форматі для подальшого вик., для цього після підготовчих програм в регістри CS та IP завант. адреса вхідної точки. Далі вик. програм

повчина знах в ОП (RAM). ~~Для роботи~~

3) Програми управління головною оперативною або віртуальною пам'яттю.

До гол. пам'яті крім ОП може входити постійна пам'ять, Віртуальна пам'ять (уявна оперативна пам'ять).

Задачі працюють з адресами віртуальної пам'яті, а за необхідності сегменти або сторінки, яких немає в ОП, завантажуються або заміщують непотрібні сторінки в ОП. Щоб полегшити цю роботу процесору, у внутр. структуру процесорів сімейства Pentium вбудовані механізми управління заміщення сторінок та сегментів. Підрахунок цих адрес за таблицями сторінок, хоча і включає декілька операцій додавання, вик. практично з найбільшою швидкістю (вик. за 1 такт процесора). Таким чином швидкість роботи проц.

не відрізняється як при роботі з ОП так і з ВП, але звертання до пам'яті у випадку ВП вик. дещо повільніше.
4) Підсистема введення/виведення та відображення даних. Більшість підсистем вв/вив. побудовані за ієрархічним принципом – на верхньому рівні функції в вигляді високого рівня. Фактично вони говорять інтерфейс для взаємодії з системою. На найнижчому рівні знах. драйвера, які обслуговують системні процеси вв/вив. Зараз до складу ОС включають різні засоби захисту від несанкціонованого втручання та захисту інформації. Традиційні елементи ОС – журнали, в яких фіксуються різноманітні події при виконанні сист. програм і користувачів. Вони дозволяють відслідкувати послідовність подій, небезпечних для комп'ютера – проникнення вірусів чи хакер-

шуваних через мережу.

Реєстри створюються для коректного використання для коректного користування програмами. В них зберігається інф. про права, надані користувачу при роботі з відповідними ярликами чи програми. Спроба виконання інші можливості веде до блокування програми.

Основні типи системних обробляючих програм

1) Транслятори - для модульного програмування з вик. різних мов у відповідну систему програмування входять: компілятори з різних комп'ютерних мов, компонувальники, які об'єднують об'єктні модулі, утворені компіляторами, між собою та модулями підключеної бібліотеки. Зручні оболонки створення проектів включають редактори (текстові), для написання чи корекції вхідних програмних

модулів. Ці редактори можуть спец. чином відображувати різними наборами ліній конструкції вхідної мови. Звичайно транслятори і різні компонувальники, формують інф. що зв'язує помилки з вхідним текстом. Після виконання редагування, компіляції та компонування часто виникає потреба перевірити програму у реальній напівгодині. Звичайно програми напівгодинної відпрацьовують напівгодинний набір, який повідомляється з вхідним текстом програми, що дозволяє побачити у вхідному тексті оператор. При виявленні значимих помилок можна повернутись до редактора для вик. корекції. Якщо програ. дає, як правило, коректні результати, то можна перейти до її тестування (перевірка набором тестів).

Сучасні різновиди системних оброблюючих програм

Симміздація програм - машинозалежна та машинонезалежна. Задач крім звичайних мов програмування, створюють мови загальної специфікації. Вони призначені для формального опису проблемних галузей на задач, що вик. програмними модулями і комплексами. Деякі з цих мов (60-70 р.) мали програмну реалізацію. Це META-IV (IBM) PL/I, для побудови семантичної частини компіляторів. PL/I (класичний опис модулів) — специфікація програмних модулів. Поширення не отримали. Задач єхочі мови ~~Ро~~ і Z. Вона розробляється і супроводжується в Оксфордському університеті. Це мова орієнтована на розв'язання задач теоретико-множинного класу, та класу реляційних баз даних.

Вик. для ~~задач~~ цілого задач, як формальна верифікація та автоматизований синтез програм.

Елементи системних програм

Інформаційні бази сист. програм складають реляційні таблиці, подібні до реляційних таблиць баз даних. Наприклад: в ОС створюються таблиці задач, процесів або потоків (процеси, хто запускав, час). Залежно від типу ОС для кожної з задач визн. пріоритети, а також додаткову часову інф. що може бути викор. в сист. реального часу. Ключовими полями для табл... є поля імен процесів або файлів, та ім'я користувача, який їх запускав. В компіляторах створюють декілька типів таблиць.

1) Таблиця ключових слів.
2) Таблиці імен (системні імена, імена користувача).

5) Таблиця константи.

а) Системні таблиці (в деяких мовах).

Відмінність від внутрішніх масс. сист. програм, що вони живуть від час роботи програми в оперативній або віртуальній пам'яті, в той час як таблиці баз даних зберігаються на диску, після вик. комп'ютера залишаються в описній збереженій формі. А таб. сист. програм з'являються при включенні і зникають при виключенні сист. програми комп'ютера. Способи обробки цих таблиць та баз однакові.

Команди опису: Create, Alter, Drop.

Команди обробки: Select, Insert, Delete, Update (вибрати, вставити, видалити, обновити).

Таблиці сист. програм організуються як масиви записів на мові Pascal,

або як масиви об'єктів (C++, Java).
В масивах елементи розміщуються на-
лідовно, і вибираються за номером або
посилаються на цей рядок. Якщо
табл. організована таким чином то
логічно найпростішим способом пошуку -
перечислити всі елементи таблиці і
порівняти всі ключові поля з аргу-
ментом пошуку. Просуватися від
попереднього до наступного елемента
зручніше всього по одному рядку. Щоб
підвищити швидкість такого пошуку,
можна впорядкувати таблицю за
ключами.

Найбільш поширеним порядком вважається
словниковий порядок. Для виконання впоряду-
вання на всіх ключових полях повинні
існувати відношення порядку, які є
транзитивними. Якщо в нас визн.
відношення порядку і таблицю впоря

таблиця за ключами, є лінійний пошук пос. виконувати з одного з країв таблиці; то якщо при порівнянні наступного ключового поле результат відношення порядку змінюється, то це свідчить про те, що у записах таблиці неможливо буде знайти відповідного ключа, і пошук можна припинити. Подальшого прискорення пошуку у впорядкованій таблиці може бути досягнуто методом двійкового пошуку. Метод двійкового пошуку спирається на первинне порівняння за відношенням порядку із середнім елементом, і надалі скорочує можливі порівняння вдвічі. Такий процес закінчиться коли ми знайдемо потрібний ключ або між словами немає потрібного ключа. Кількість операцій пропорційна $]\log_2 N[$ (де N - кількість елементів у таблиці).

Зауваження : найпростіша реалізація

двійкового пошуку спирається на визн. індексів елементів таблиці, з якими не прийдеться порівнювати наші ключі. Тому, якщо індекси змінюються від 0 до N то початковими значеннями індексів для нижнього браузера -1, а для верхнього N+1.

В багатьох випадках, коли бази даних дуже великі (100.000 і більше), методи працюють відносно повільно; зазвичай від них задачі шукають найбільш прості. Таким методом вважається метод пошуку за прямою адресою (якщо знаємо номер відповідного рядка в таблиці, то можемо одразу вийти на цей елемент, додавши до поч. адреси потрібний індекс, помножений на довжину елемента). На мовах високого рівня (Pascal, C) це можна записати так: ім'я

масива [номер елемента] . поле. На мові асемблера для вик. будь-якої дії над індексованим елементами масиву можливе вик. адресації другого операнда з вик. індексного або базового регістра
Індексні: si, di.
Базові: bx, bp. (в 32 б. режимах додається префікс е)
В 32 бітовому режимі можна вик. як індексні і базові будь-які регістри, окрім sp і bp. Коли довжина елементів масиву дор. 1, 2, 4, 8, то можливе бічне відповідного індексного регістра накладання зірку і відповідний коефіцієнт.

Специальні методи організації пошуку в масивах

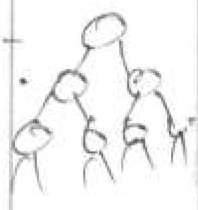
Первинна — операція пошуку за ключем. Подальші проблеми виникають у випадку переповнення масивів, що у випадках дуже великого розміру масивів. Шляхи розв'язання цих проблем можуть бути такі:

1. Сегментація таблиць — розбиття табл. для подальшого зберігання (в пам'яті ОС може розміщуватись в різних місцях). Для побудови різноронних табл. слід вик. показники, які показують на нові сегменти або окремі записи таблиць. Якщо табл. скл. з окремих записів, які можуть бути розміщені в різних сегментах, то крім звичайних полів елементи вони повинні включати додаткові поля показників. Таким чином можна побудувати списки та деревоподібні структури.

```
struct rec {
    struct key k;
    struct fun f;
    struct rec *ptrL;
    struct rec *ptrR; };
```

Списки будують як однозв'язкові та двозв'язкові. В однозв'язковому списку вик. тільки 1 показник (він показує

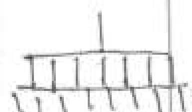
що виконує адр. наст. запису), тупиковий кінець списку виконує нульовий покажчик, що є ознакою закінчення списку (в деревах - закінчення гілки дерева). В списках можливий тільки лінійний пошук, а в деревах (якщо вони впорядковані) можна вик. двійковий пошук та комбінації різних методів пошуку. Найбільш поширені в базах даних - двійкові дерева, де робота з деревом поч. з кореневого вузла і до нього приєднується ієрархія вузлів дерева (вузол - елементи і два покажчика). Впорядкованим буде таке дерево, де для кожного кореневого вузла всі підлеглі вузли в лівому піддереві будуть меншими, ніж ключ кореневого вузла, а всі праві - більше. Вузли, які не мають жодного покажчика наз. термінальними вузлами (листям).



У впорядкованому двійковому дереві

можна організувати двійковий пошук з результатами порівняння з ключем вузлів. Для ефективності цього пошуку необхідно щоб дерево було збалансоване, тобто відстань від кореневого вузла до будь-якого термінального вузла не відрізнялась більше ніж на 1.

Розглядаємо або бі-дерева.



У кожного ключа повинні мати свої підлеглі піддерева. Таблиці на індекси найпродуктивніше працюють з впорядкованими масивами (двійковий пошук) або пошук за прямою адресою.

Проблеми з пошуком за прямою адресою

Якщо діапазон значень аргументу не дуже великий (вимірюється в 1-2 десятки) то і масиви займають не дуже великий об'єм пам'яті (до 64 К елементів), але більшість сучасних компіляторів мають властивість розрізнення імен розм

мірам до 1 кбайта. Теоретично можливе кількість аргументів - 28.1000. Тому було введено ф. hash = h(A) - цілоч. значення, що визн. адресу осо комір початкових елементів.

Вимоги до хеш-функцій

1) Хеш-ф. повинна бути детермінованою (кожен раз дає однакове значення результату).

2) Діапазон значень хеш-ф. відповідає кількості елементів в сегменті пам'яті.

3) Для аргументів з близькими значеннями доцільно мати найбільш відмінні значення хеш-ф. Ситуація коли для різних значень аргументів формується однакове значення ф. наз. колізією.

Якщо виникає колізія, то необхідно мати механізм вирішення і здійснювати роботу. В цьому випадку

вик. ф-ї з розв'язання колізії. Для цього в мові. вик. звичайно додатковий лінійний пошук в декількох сусідніх позиціях. Якщо знайшли - нічого не робимо, якщо ні - перейти до наступного елементу.
З базах даних хеш-ф. вик. для побудови зв'язання таблиць. Для пошуку за прямою адресою ключі можна не зберігати, але коли вик. хеш-пошук, що в наслідок колізії доводиться зберігати всі ключові поля.

Графи, що вик. на різних етапах роботи компілятора

Більшість методів лексичного, синтаксичного та семантичного аналізу спираються на якісь графові моделі. Лексичний аналіз, як один з варіантів, може побудувати на основі принципів скінченних автоматів. Звичайного лексичного аналізатора я розбиття вхідного

тексту на окремі лексеми. Кінцевими станами обробки лексем можуть бути коди типів лексем:

1) Розділювачі та операції; Виділяють білі "розділювачі", що лексемами не вважаються.
2) Ключові слова та імена.
3) Константи (числові та символьні).
4) Коментарі.
5) Інші лексеми специфічні залежно від мови.

Наявність таких станів дозволяє побудувати автомат класифікації лексем, як автомат, який проходить визначені стани до стану кінцевої, коректної або помилкової лексеми. Для цього при проектуванні автомату до кінцевих станів слід додати проміжні стани, які в незакінченому вигляді будуть не помилковими, але стани помилкові

лексем. Спочатку дає автомату будувати літери вхідного тексту, що надходять до лексичного аналізатора. Результатом роботи автомата повинен бути один коректний або помилковий сигнал на одну лексему.

Програмна реалізація кінцевого автомата

Граф можна зобразити графічно або таблично (матриця переходів). В цій матриці видно попередній стан та символ. Щоб один з розмірів не був надмірно великим, то попередньо можна класифікувати символи, що надходять до автомата. З точки зору синтаксису нам вигідно яка цифра чи літера надходить, важливий знак. Тому зручно вхідні літери прокласифікувати за допомогою таблиці класів, яка може займати до 256 байти.

В цій табл. на місці, що відп. якійсь літері, розміщується код класу літери.

Класи:

1) Роздільники;
2) Літери, що припустимі в іменах.
3) Літери спеціального призначення.
4) Літери - спеціальні ознаки.
5) Спеціальні роздільники реалізовані константи.

Для визн. станів автомату і вхідних станів, доцільно вик. перенумеровані шини, що і будуть визн. елементи матриці. Кожен елемент матр. характеризує наступне значення автомата.

st = Mt[Sg][St];

Sg = Cls[Lt];

В асемблері - команда Xlat.

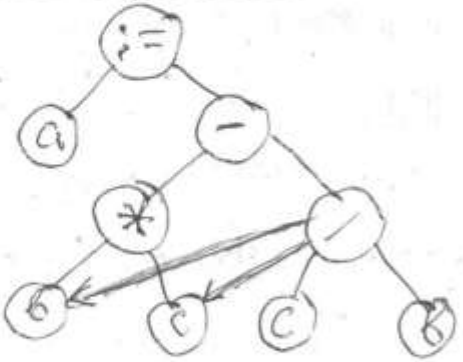
Вик. спрямований граф загального виду (поч. вузол, кінц. вузол).

Графи для представлення виразів та операторів в компіляторах

Граф (дерево) ініціалізації операцій

В двійковому дереві початковим вказівником є вузол лівої операції, що вик. синтаксичного, проміжними вершинами є операції, що вик. для одержання проміжних результатів, а термінальними вершинами є вузли імен або констант у виразах.

a := b * c - e / b ОНБ



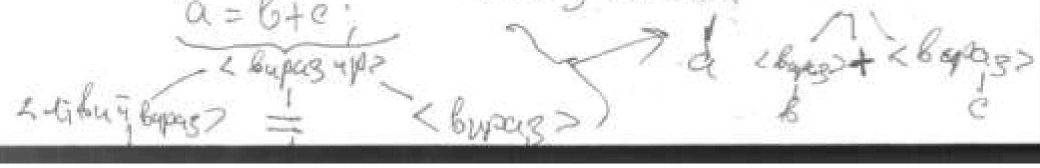
Такі дерева і графи зручні для всіх видів семантичної або змістовної обробки, також семантичного синтезу інтерпретації понят. кодів, оптимізації і генерації об'єктних кодів.

Особливості побудови дерев для обробки складних операторів процедурних мов програмування

Звич. при вик. синтаксичного аналізу уявл. дерева розбору. Вузли яких предст. собою нетермінальні позн., за винятком термінальних вузлів, які є термін. по значенню. Правило визн. нетермін. означень беруться з форм Бекуса, із них формують дерева при деталізації або розборі відповідних нетерміналів. Тобто якщо якесь речення класифікувалося як нетерм. то в проц. доведення його коректності ми повинні вивести ієрархію підлеглих нетерміналів, що в кінці будуть завершуватись терм. вузлами з терм. позн. мови.

<вир. присв> ::= <l value> = <вираз>

ліве значення

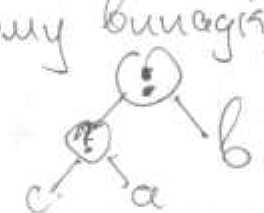


Частіше за все в літературі вигляд синт. аналізатора описується так, що має вигляд дерева розбору. Один з методів обробки такого дерева — вик. правил форми Бекуса. Однак такі дерева незручні для змістовної обробки. Тому прийнято перед семантичною обробкою створювати syntax tree ⟶ Directed Acyclic Graph (DAG). Однак це дозволяє довільно розширювати вузлами, що відп. ключовим словам складних операторів мов програмування. Заг. правило для таких вузлів, що ближче до кореня лежать вузли, що вик. останніми.

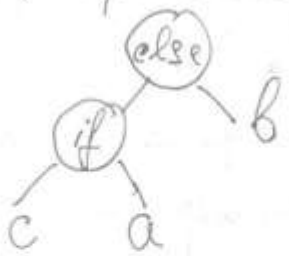
Для умовних операторів

$c ? a : b$

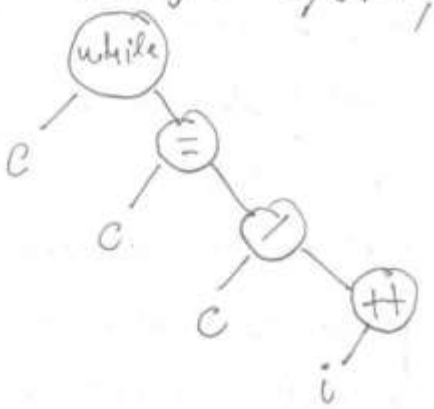
Якщо вираз b не дор. нулю, то як результат повертається вираз a, в іншому випадку повертається c.



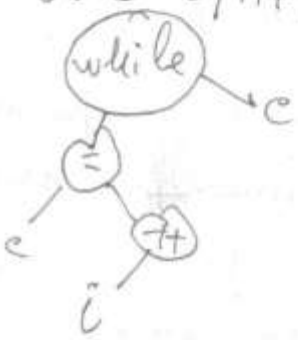
За аналогією оператор if else:
if (c) a; else b;



Для циклів передумова
while(c) c = c/c++;



постумова
do c = c/i++; while(c);



Цикли з вик. індексів
for

В аргументах оператора for , в виразі и ее зау - початкові установки (в врисвоюва допускається операція ,, , "). В е2 зап. умова продовження циклу. В е3 - зміна параметрів цикл. Оператор switch (але case) вибор більш складними структура тим щоб зберегти зв'язки для швидк. обробки визч. операцій. В циклах крім звичайних операторів можуть вик. операто-ри break і continue. Оператор break закінчує ти збиває цикл (виходить з циклу, тому на графі можна показа-ти зв'язок оператора break з наступним вузлом оператора, а continue - перехо-дить на наступний цикл. Однак при цьому граф перестає бути ациклічним

Побудова графів для правил підстановки в різних формах Бекуса

Мова правил Бекуса, які є інші комп'ю-терні мови, теж розбивається на різні

нози або лексеми. Для відображення правил на дії мові головними є термінальні та нетермінальні нози. а також операції "як" вич. при заданні правил.

::= операція визначення, задає в заголовку і відділяє ліву часть від правої:

| - операція альтернативи.

[ ] - факультативність, необов'язкова конструкція.

{ } - показ повторювання конструкції або показ проміжного списку альтернатив.

В нормальній формі Бекуса повторювані елементи також. замінюються рекурсивними правилами. Якщо ліва частина нови. зліва списку альтернатив правила, що маєть рекурсію нск. лівосторонньою, а якщо в правій альтернативі — то правосторонньою. В більшості стандартних описів, зрозуміння в нормальній формі Бекуса вич. лівостороння рекурсія, оск при перетв. цих правил на машинну

форму, цю рекурсію намагаються замінити правосторонньою. Більшість сучасних стандартів мов запис. У формі Бекуса-Наура, в якій, щоб уникнути рекурсій вик. фігурні дужки з крапкою. Всі операції, перелічені при визн. форми Бекуса при переводі форми Бекуса у внутр. форму відобр. нетермінальними вузлами, але через можливість рекурсій відповідні графи не завжди будуть ациклічні.

Основи теорії граматик

Теорії граматик взяли свій початок з вивч. природніх мов. Граматикою наз. четвірку математичних об'єктів, які виконують наступне $G[Z]=(D_T, D_{NT}, R, Z)$

Для того щоб орієнтуватись які граматики засп. для опису мов, з вич. вик. т.з. класифікацію граматик за Хомським. Виділяють наступні

поділ:
1) Граматики без обмежень.
2) Мови з фразовою структурою (мови самодовлі).
3) Контекстно-залежні граматики, де всім. правим зачепили від контекстів, в яких ці правила вик.
4) Контекстно-незалежні граматики.
Але дає вик. деяких видів обробки більш звужено вик. окремий випадок Кон.-нез. грам. які називаються детермінованими або однозначними граматиками (здаються через автомати, але можна задати і через форми Бекуса).
D/N). Контекстнонезалежні граматики звичайно вик. для синтаксичного аналізу, а для лексичного аналізу найчастіше вик. регулярні або автоматні граматики.
Регулярні гр. можуть бути праволінійні

за допомогою скінченних автоматів, тобто
що в цьому випадку правила граматики
треба перетворити на визначення
станів і послідовності переходу по них.
Для вик. лексичного аналізу вхідні дані
подаються як послідовність літер у
файлі, представленій або в однобайтовому
коді ASCII, або в двобайтовому
коді Unicode. Вирішуючі частину
літеру можна розглядати як вхідний
символ автомату. Оскільки в 1 байті
можна записати один з 256 варіантів,
то в цьому випадку треба визн. 256
варіантів станів, однак більшість
літер, цифр і роздільників обробляються
автоматами однаково, тобто з одного
і того ж попереднього стану пере-
ходять в ті самі наступні
стани. Тому як вхідні символи до-
цільно вик. не самі літери, а класи

літер, які можна одержати з наб. класів виконати пошуку за прямого доступу. В лексичному аналізі мов програмування вик. зазвичай 20-30 типів лексем. Таким чином макрочук переходів при кількості класів літер 25 буде складати до 500 елементів.

Основні підходи до синтаксичного аналізу

Вирішувані можна побудувати єдину граматику для лексичного і синтаксичного аналізу, але також побудувати окрему граматику, що вона буде вкл. надто багато правил, тому в практиці вдаються до третього розширювання граматик. В першій парадигмі лексичний аналіз а синтаксичний аналіз вик. в окремому парі. При реалізації синтаксичного аналізу вик. 2 підходи:

висхідний чи низхідний.

Висхідний підхід спирається на елементарні конструкт. чи конструкції низького рівня і вик. рух до конструк. більш високого рівня, або в термінах правил підстановки форм Бекуса вони організують рух від термінальних позначень через проміжні нетермінальні позначення до кінцевого позначення граматики.

Низхідний розбір - аналіз пов. зі зіставлення кінцевого поз. з поміж вик. рух через проміжні нетермінали до термінальних позначень.

Результати синтаксичного аналізу - незалежно від підходу найчастіше відображаються у синтаксичному дереві (рідше інше). Вважається, що для коректної граматики дерево аналізу повинно бути однозначним незалежно від підходу (висхідного, низхідного),

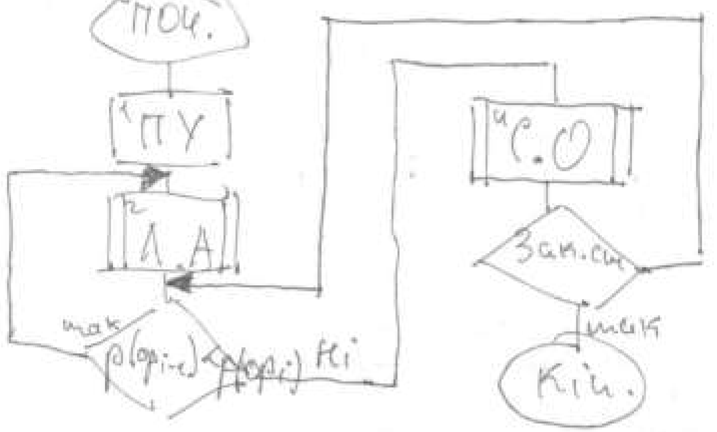
саме при висхідному підході основним є поняття переважання або пріоритету операцій. Кажуть, що операція R переважає наступній операції S, якщо вона обробляється або вик. раніше наступної операції S. Якщо у контексті виразу за операцією з меншим пріоритетом іде операція з більшим пріор., або така, що переважає операції, то вона повинна обробитись в першу чергу. Пріоритети і місцезнаходження кодуються в різних напрямках. Однак, коли корист. формулою Бекуса, то ми не маємо змоги явно вказувати пріоритети або переважання операцій.

Стековий алгоритм висхідного розбору для мат. виразів

В цьому алгоритмі визн. пріоритети арифметичних і логічних операцій,

мат, як вони пріоритетні в мат. означеннях, тобто відношення в системі має найвищий пріоритет, множення, ділення, взяття модуля в вигляді дужки, уже нижче операції додавання, відніснання, порівняння і логічні операції. Для спец. операцій — згуби, інкр., декремент, визн. спец. значення пріоритетів (в заг. логічних операцій). Базовий алгоритм орієнтований на обробку виразів без дужок, включаючи оператори присвоювання. Звичайно операція привласнювання має найнижчий пріоритет і вик. останньою. Наперед підготовл. заміна на лексеми значення константи можна вважати такими лексемами, що мають найбільший пріоритет. При одержанні чергової операції в цьому алгоритмі порівнюють її пріоритет з попередньою операцією, яка зберіга-

ється у верхівці стеку. Якщо попередня операція має пріоритет, що ≥ наступній або одержаний стоп-символ, то алгоритм звертається до якогось з видів семантичної обробки. Це може бути побудова дерева (синтаксичного), інтерпретація конст. виразів і генерація об'єктних кодів. Після цього, якщо стек звільнився, можна закінчувати обробку виразів. У іншому випадку, якщо наступна операція має більш високий пріоритет, то це операція такої вноситься до стеку і відбувається перехід на введення наступної лексеми. Блок-схема такої прогр.



Два цикли - завантаження and розвантаження стеку. Такий алгоритм добре працює для виразів без дужок. Для роботи з дужками і деякими видами операторів замість простих пріоритетів вик. функції передування. Одна з цих функцій дає значення передування при одержанні операції від лекс. аналізу, а друга — бере інше значення при вибірці цієї операції зі стеку. Ці ф. перевіряють наз. f і g. Для відкритої дужки пріоритет визн. функц. f повинен бути високим (операції в дужках вик. раніше) тобто вище за найвищого ступеню операцій, а ф. g - низький пріоритет, принаймі такий, як пріоритет закритої дужки, який повинен бути нижчим ніж найменший пріоритет операцій. Тоді при порівнянні пріоритетів за шляхом нового зн. вик. ф. f, а в

ітному – Ф. Алгоритм записує свою структуру позитивно. Така граматика буде мінімальною, але не універсальна і однозначна. Але щоб побудувати найбільш загальну граматику виду. узагальнене відношення передування. Це відношення може записуватись 3 варіантами: якщо R і S – попередні і наступні слова то:

R •> S – S передує R

R <• S – R передує S

R •= S – однакові передування

(R, S) – комбінація є неприпустимою.

Але мало, щоб вик. такі відношення, будують матриці передування. В цих матрицях кількість колонок і стовпців виду. суміжно визнач. терм. і нетерм. позначено. За допомогою матриць передування точна виду. граматику для синтаксичного аналізу будь-якої мови.

Однак, якщо граматика первинно визначається в правилах підстановки, що її породжують праві види у відповідних значеннях так, щоб вони відповідали правилам підстановки. Тра вик. граматик породжування без додаткової інф. легше одержати дерево підлеглості операцій ніж дерево розбору, завдяки тому, що у граматиках спираються на сосолюші або відносні підпорядкування. Дерева підлеглості більше зручні на послідовних етапах, тому за часткових методів розбору легше будувати синтаксичні аналізатори.

Організація низхідного синтаксичного розбору

Загальним методом низх. синт. розбору найлегше вважати метод рекурсивного спуску, в якому вхідний текст програми

ми приймаємо за кінцеве нетерміналь-
не позначення розбору. Після цього поч.
розгляд правої частини правила, що
визн. відповідний нетермінал. Для того,
щоб перевірити підлеглі вершини, нам
треба послідовно звертатись до всіх аль-
тернативних правил з ним, щоб
виконати розбір, вважаючи ці нетер-
мінали кінцевим символом граматик. К/р

Низхідний розбір починається з аналізу
конструкцій або нетермінальних пози-
високого рівня (поч. з кін. симв. граматики)
і ведеться через симв до термінальних
позначень. Для виконання алгоритмів
рекурсивного спуску можна всяк безпосе-
редньо правила у різних формах Бекуса.
Такі правила підстановки вимагають
для доведення кожного з нерівних
правил підставляти замість нетер-
мінального пози- з лівої частини правила

почергово підст. різні варіанти чи альтернативи з правого варіанту правила, але в зв'язку з тим, що ці правила можуть бути нові нетермінальні позн. що доводиться рекурсивно звертатись до визн. таких позн. Такі рекурсивні звертання повинні повт. доти, доки не з'являться відповідні терм. позначень. Такий метод доведення синтакс. коректності програм отримав назву „метод рекурсивного спуску". Головна проблема методу — в більшості випадків доводиться задавати правила, що задають з лівосторонньою рекурсією, але це призв. до того, що відбуваються рекурсивні звертання до лапугого пра- вил без просування за вхідним тек- стом програми, що практично веде до зациклювання. Таким чином, при вик. цього методу рекоменду-

етьке перетворити ліворектурсивні правила на праворекурсивні. У зв'язку з тим, що формальна методика перетв. таких правил відсутня, то фактично доводиться переписувати всі лівореєдрсивні правила вручну. Результатом буде виявлення синтакс. помилок, або відсутності від. правил, і побудова синтакс. дерева або дерева розбору. В дереві розбору повторяю відосн. всі нетермінали, які зустр. у конструкціях. Крім смешн. особливі дерева розбору перетв. на дерева підлеглості операторів та операцій або спрямовані ациклічні графи. В цих деревах чи графах нетермінальні вузли вже відсутні.

Більшість авт. систем побудови компіляторів спираються саме на такий підхід до низхідної обробки (синт. аналізу), тобто для настроювання синт. аналі-

затора коли-годуються у машинній формі правила підстанов з правосторонньою рекурсією, з яких, фактично, генерується версія синтаксичного аналізатора.

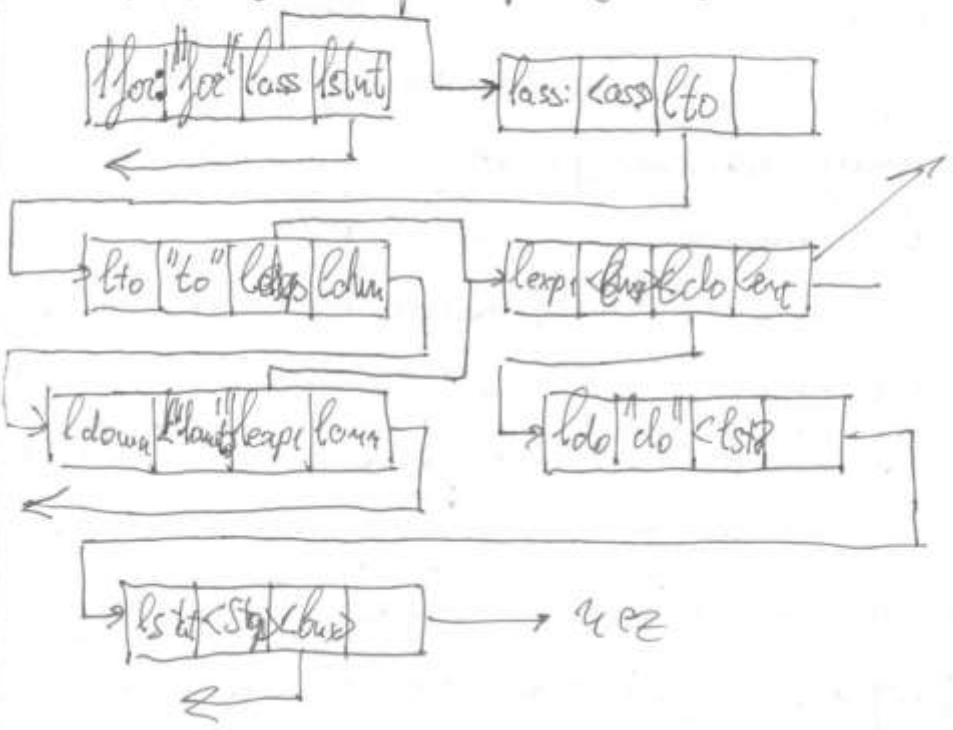
Деякі автори виділяють як самостійний метод так званий „метод побудови синтаксичного управляючого графа". Кожен вузол такого графа вкл. 4 елементи:

1) мітка вузла (передача управління)
2) ознака базового синтакс. елемента (терміналу чи нетерміналу)
3) показник на вузол графа, який буде вик. при успішній роботі аналізатора
4) показник переходу на альтернативний варіант (у випадку невідповідності констр. аналізатору у заданому вузлі).

Такий підхід дозв. будувати відносно прості графи для будь-яких констр.

мов, однак вони, як правило, не дуже
прості оскільки рекурсивних нетермінальних,
які в цьому випадку досить ускладнюють
алгоритмічний висхідного розбору. Тому
також може бути побудована одна з
найпоширеніших версій аналізаторів, що
вм. в себе елементи висхідного та
низхідного розборів.

Граф для оператора for з мови Pascal



Правила визн. пріоритетів для складних операторів мов програмування.

При побудові скл. операторів бажано, щоб ключові слова, що визн. сутність оператора, зосереджені кореневими для підсилення підсумків з вик. операторів, а всі проміжні оператори повинні утворювати ланцюжки підлеглих операторів. Дужки в виразах і операторні дужки можуть не відповідати в тому випадку, якщо підлегні операції мають більший пріоритет, ніж наступні. Тому при вик. функції перевірці обо програмістів бажано, що відкриті дужки та початкові слова операторів мали високий пріоритет у ф.f і низький у ф. g, а закриті дужки мали низький пріоритет як у f так і у ф. g.

Роздільники списків повинні мати пріоритет (як { так і ;) проти інших цих внутрішні операції, і проти відкритих закриті дужки. При пошуку ніужуї проблеми, що потребує додаткових дій, — перевірка заданої послідовності дій.

Видu семантичної обробки

Попередні види - перетворення графу синтаксичного аналізу на спрямований ациклічний граф. Можна розгл. з синтакс. аналізу, а можна вик. як окремий перегляд графу.

Загальні методи семантичної обробки:

1) Семантичний аналіз. При семант. аналізі перевіряються відповідність на допустимість підвиразів операндів для всіх операторів програми. Для мат. операцій визн. тип результату операцій. Алгоритми семант. аналізу вик.

шляхом обходу дерева. Дерево (або граф) звич. обходиться за привилеями вик. окремих операцій. Видається діагностика про помилки у визн. типів операндів. Для вик. семант. аналізу будується таблиця сумісності операндів та операцій, в якій, як ключові поля, задаються поєд. операцій та припустимі типи аргументів. У функц. полях визн. типи результатів. Кодки визн. тільки для припустимих сполучень операцій та їх вхідних даних. Відсутність відповідного рядка буде свідчити про те, що таке сполучення несумісне.