последна редакция 05 Юни 2007 г.

д-р Б. Панайотов

МЕТОДИ ЗА ТРАНСЛАЦИЯ

учебно пособие

СЪДЪРЖАНИЕ

1. Класификация на транслаторите. Етапи на транслация. 2
1.1. Основни понятия 2
1.2. Класификация на транслаторите 2
1.3. Етапи на транслация 4
2. Лексически анализ 8
2.1. Основни определения 8
2.2. Азбука, дума. Свойства 8
2.3. Формални езици. 10
2.4. Пораждащи граматики. Йерархия на Чомски. 11
2.5. Регулярни изрази и регулярни множества 14
2.6. Крайни автомати 17
2.7. Програмиране на лексически анализатори 20
2.8. Генератор на лексически анализатори Lex 23
3. Синтактически анализ 26
3.1. Основни понятия и определения. 26
3.2. Привеждане (преработване) на граматиките 30
3.2.1. Отстраняване на неизводимите символи 30
3.2.2. Отстраняване на недостижимите символи 30
3.2.3. Отстраняване на безполезните символи 31
3.2.4. Отстраняване на ε-правила 31
3.2.5. Отстраняване на верижните правила 32
3.3. Неоднозначност на граматика. Отстраняване на неоднозначност 32
3.3.1. Основни определения. 32
3.3.2. Отстраняване на лява рекурсия 34
3.3.3. Лява факторизация 36
3.4. Стекови автомати 36
3.4.1. Основни определения. 36
3.4.2. Работа на стековия автомат. 37
3.4.3. Език разпознаван от стеков автомат. 37
3.4.4. Построяване на стеков автомат 38
3.4.5. Детерминирани и недетерминирани стекови автомати 38
3.5. Низходящ анализ 38
3.5.1. Основни понятия. 38
3.5.2. Таблично-управляем предсказващ анализ. 39
3.5.3. Множества First и Follow 41
3.5.4. Построяване на таблица за предсказващ анализ 42
3.5.5. LL(1) граматики 43
3.5.6. Рекурсивно спускане 44
3.5.7. Възстановяване след синтактически грешки 45
3.6. Възходящ анализ 45
3.6.1. Основни понятия. 45
3.6.2. LR(k) анализатори. 47

3.6.3. Конструиране на LR(1)-таблици.....50
3.6.4. Възстановяване след синтактични грешки. 54
3.6.5. SLR(1)-анализатори. 54
3.6.6. LALR-анализатори. 55
3.7. Генератор на синтактически анализатори YACC 55
4. Семантически анализ. + atributnite.....60
4.1. Въведение.....60
4.2. Атрибутни граматики.....60
4.2.1. Определение за атрибутна граматика 60
4.2.2. Атрибутно дърво на разбора 61
4.2.3. Коректност на семантичните правила.....62
4.2.4. Класове атрибутни граматики и тяхната реализация 63
4.2.5. Език за описание на атрибутни граматики.....64
4.3. Синтактично-управляема транслация 66
Литература 67

Предлаганият материал застъпва основните методи за разработка на транслатори и съдържа сведения, необходими за изучаване логиката на тяхното функциониране, използвания математически апарат (теорията на формалните езици и формалните граматики, метаезиците).

1. Класификация на транслаторите. Етапи на транслация.

1.1. Основни понятия

Компютър се нарича управляемата, отворена, динамична система с автоматично, дискретно действие и крайна памет. Системата е отворена, т.е. поддържа връзка с външния свят чрез канали. Компютърът се управлява с управляваща информация.

Програма се нарича система от предписания определящи поведението на компютъра за някакъв интервал от време.

Транслаторът е обслужваща програма преобразуваща (превеждаща) (изцяло или на отделни части) входна програма (входен код) (source code), представена на входен (формален) език (source language) за програмиране, в работна (изходна) програма (или съответни части), представена на изходен (формален) език.

Текст на един формален език се нарича линейна последователност от знаци.

Определението се отнася за всички разновидности на транслиращите програми. Всяка от тях има свои особености по организацията на процеса на транслация.

1.2. Класификация на транслаторите

Транслаторите се делят на три основни групи: асемблери (assembler), компилатори (compiler) и интерпретатори (interpreter), в зависимост от изпълняваните задачи.

Асемблерът е системна, обслужваща програма, която преобразува символични (мнемонични) конструкции в команди на машинен език. Специфична черта на асемблерите е точната транслация на една мнемонична команда в машинна команда. Асемблерният език се нарича още и автокод. Предназначен е за облекчаване на

възприемането на системните компютърни команди и ускоряване на програмирането с тези команди.

Асемблерните езици използват множество допълнителни директиви облекчаващи управлението на компютърните ресурси, написването на повтарящи се фрагменти, построяване на многомодулни програми.

Компилаторът е обслужваща програма, изпълняваща транслация на програма (данни и операции над тях) написана на входен език в програма на машинен език, с последващо изпълнение на получената програма във вид на отделни стъпки.

При компилацията се осъществява пълен превод на изходната програма на машинен език преди нейното изпълнение, получената програма се редактира и се свързва с всички необходими за изпълнението модули. Така се получава т.н. изпълнима програма, подходяща за многократно изпълнение без повторна транслация.

Интерпретатор се нарича програма или устройство осъществяващо анализ на отделните обекти на входния език с едновременното им (непосредствено) изпълнение (интерпретация).

Интерпретирането се извършва последователно, покомандно. Интерпретаторът позволява да се започне обработка на входната програма при наличие дори на една команда. При него отсъстват допълнителни файлове на машинен език. Той може лесно да адаптира програмата за различни компютърни архитектури, чрез използване на някой разпространен език за програмиране. Затова интерпретаторните програмни езици, като Java Script, VB Script, Perl, PHP имат широко разпространение. Недостатъкът е ниската скорост на изпълнение на програмата. Обикновено програмите, които се интерпретират се изпълняват 50-100 пъти по-бавно от тези, които се компилират.

Пример. Език Basic има транслатори от различни типове. Например, в средата TurboBasic – транслатор-интерпретатор, а средата QuickBasic – транслатор-компилатор.

Интерпретаторът може да работи по два основни начина:

Да обработва и изпълнява направо (директно) входния текст на програмата. В такъв случай периодът на транслация и периодът на изпълнение не са разграничени – те протичат едновременно.

Да изпълнява програма на междинен език, получена в резултат на транслация (компилация). В този случай периодът на транслация и периодът на изпълнение са разграничени. Този подход към реализация на програмния език се нарича още полуинтерпретация.

Един компилатор се нарича **крос-компилатор**, ако изходния му език е машинен език за компютър с друга архитектура, различна от тази, в която той работи.

JAVA е един различен интерпретатор, който интерпретира машинен код за виртуалната машина на Java (JVM). Java машинния код може да се компилира до машинен код за определен хардуер с помощта на JIT компилатор.

Конвертор се нарича транслатор, който превежда програми, написани на един език за програмиране от високо ниво на друг език за програмиране от високо ниво.

Емулатор се нарича програма или програмно-техническо средство позволяващо без допълнително препрограмиране да се изпълняват програми на една компютърна архитектура, написани за съвсем различна компютърна архитектура.

Емулаторите се използват за разработка на нови компютърни архитектури и за изпълнение на стари програми на нови компютърни системи.

Емулаторите на междинен код, както интерпретаторите, позволяват създаването на мобилни програмни системи. Това свойство преопределя и успеха на езика за програмиране Java, с който програмата се транслира в междинен код (байт-код). JVM, изпълняваща този код, се явява емулатор, работещ под управлението на произволна съвременна операционна система.

Прекодировчик се нарича програма или програмно устройство, превеждаща компютърни програми, написани на машинен език за една компютърна архитектура на машинен език за друга компютърна архитектура. Прекодировчиците са полезни при пренос на програми между различни компютърни архитектури.

Препроцесор (макропроцесор) се нарича програма, която заменя една входна последователност от символи с друга. Препроцесорите са надстройки над езиците за програмиране като увеличават функционалните им възможности. В език С директивите на препроцесора са оформени като отделни низове от програмата и започват със символа “#”. Обезпечават замяната на срещащите се в текста идентификатори с текстови низове.

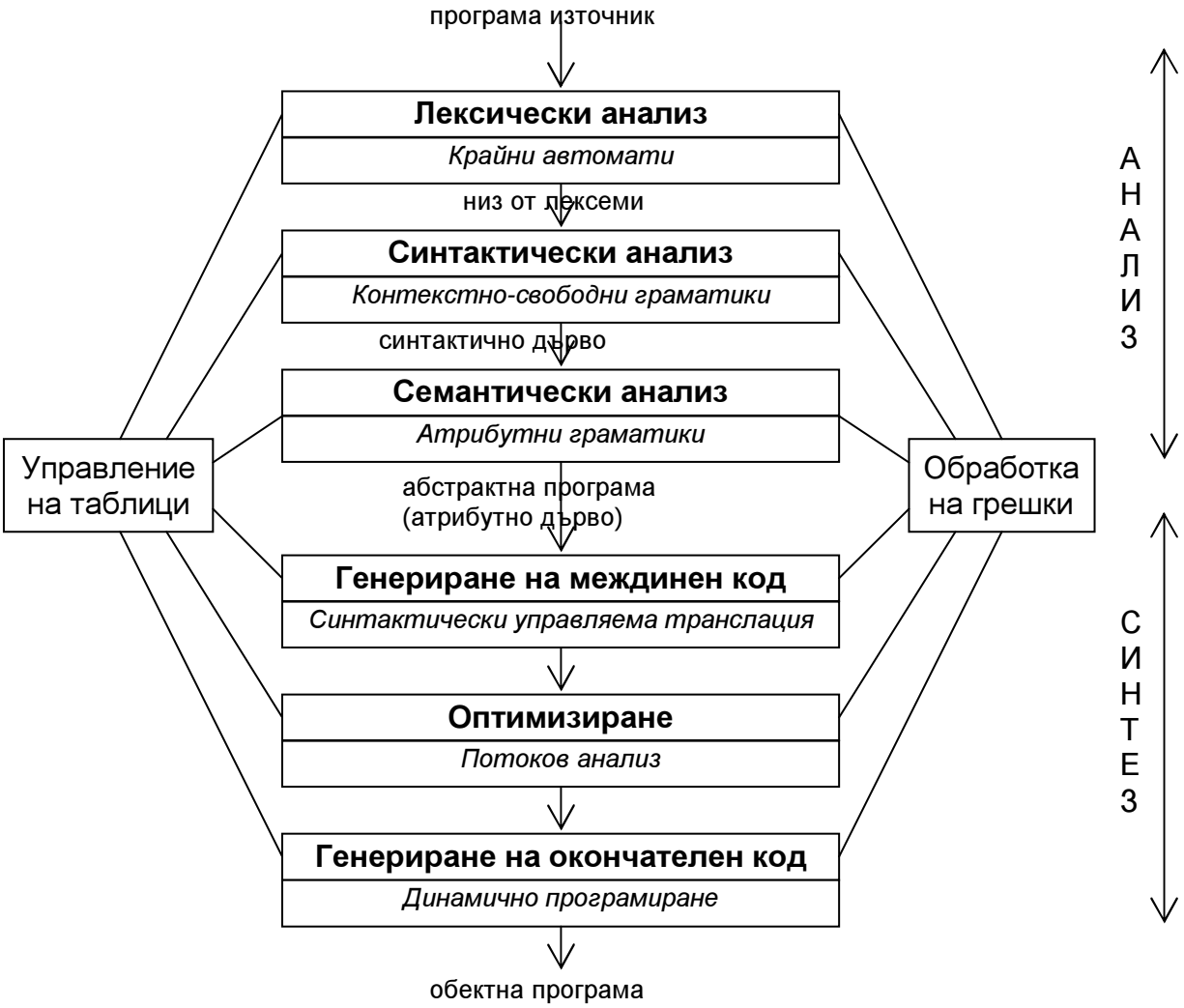
Език за програмиране се нарича съвкупност от семантични конструкции и синтактични правила наподобяваща човешкия език, която се използва за представяне на алгоритми във форма удобна за обработка от компютър.

Основните тенденции в развитието на езиците за програмиране са в посока те да се направят по-разбираеми за човек, да се ограничи възможността за неволни грешки, да се осигурят възможности за многократно използване на програмни фрагменти и за работа на някоколко програмиста в екип.

1.3. Етапи на транслация

Процесът на транслация се свързва с два основни етапа: анализ и синтез. В етапа на анализ се извършва лексически, синтактичен и семантичен анализ, а в етапа на синтез се генерира междинен код, извършва се оптимизация, генерира се крайния код.

Етапите на транслация са показани на следваща схема.



На фазата на лексическия анализ входната програма, представляваща поток от символи, се разбива на лексеми – думи в съответствие с определенията на езика. Основните формализми, лежащи в основата на реализацията на лексическите анализатори са крайните автомати и регулярните изрази. Лексическият анализатор може да работи в два основни режима: или като подпрограма, извиквана от синтактическия анализатор, или на пълен преход, резултата от който е файл с лексеми. Четенето на входния низ се извършва от ляво надясно.

Пример. Нека е даден следния оператор за присвояване:

map = start + displace*10

От този входен низ се изолират следните лексеми:

1. идентификатор **map**
2. знак-операция **=**
3. идентификатор **start**
4. знак **+**
5. идентификатор **displac**e
6. знак *****
7. число(константа) **10**

Обикновено интервалите се елиминират. Всяка лексема се състои от клас, който показва характера на информацията и от стойност, която служи за указател в таблица. Класовете

са краен брой. Лексическият анализ определя някои (прости) грешки като недопустими символи, неправилен запис на числа, идентификатори и др.

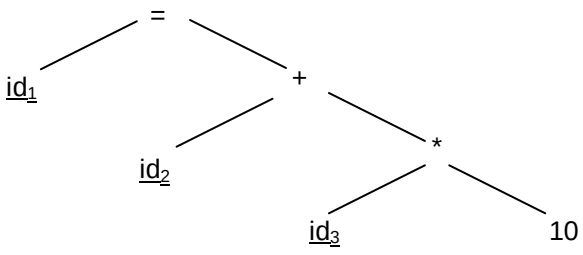
Основна задача на синтактическия анализ е групиране (разбор) на лексемите в йерахични структури. Под йерархични структури се разбират дърветата, съответстващи на разбора в безконтекстните граматики. Резултатът от синтактическия анализ е синтактично дърво с връзки към таблица с имена. В процеса на синтактически анализ се определят грешките свързани със структурата на програмата.

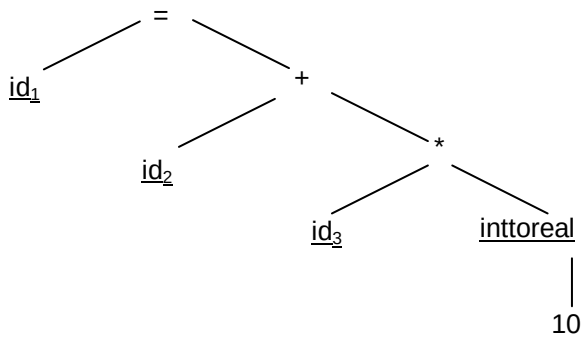
Пример. Лексемите от оператора за присвояване map = start + displace*10 се групират в следната йерархична структура:

След лексическия анализ на оператор завършва със следния израз: $id_1=id_2+id_3*10$. С id_j се подчертава, е съществува клас id и стойност на индекса.

Във фазата на семантическия анализ се проверява дали структурно правилните компоненти на програмата са съгласувани по смисъл според спецификациите на входния език. Извършва се съгласуваност на типовете. Основните формализми, които се използват при семантическия анализ са атрибутните граматики. Обектите от езика за програмиране се свързват допълнително между тях посредством други понятия, образуващи разнообразни асоциации. В процеса на семантическия анализ се строи таблица на символите/имената, в която се попълва информация за описанието (свойствата) на обектите.

Пример. В синтактическото дърво на оператора за присвояване map = start + displace*10 се вмъква нова операция: intoreal. Получава се следното атрибутно дърво:





Генерирането на междинен код се извършва се извършва за да подпомогне процеса на оптимизацията и процеса на генериране на окончателен код. За междинно представяне се използват префиксни и постфиксни записи, ориентирани графи и др.

Пример. Генерираният междинен код на оператора за присвояване map = start + displace*10 е следния:
term1 = inttoreal(10)
term2 = id3 * term1
term3 = id2 + term2
id1 = term3

Оптимизацията може да има няколко подетапа. Обикновено оптимизациите биват машинно-зависими и машинно-независими, локални и глобални. Част от машинно-зависимата се изпълнява на фазата на генерация на окончателния код. Глобалната оптимизация се основава на глобален потоков анализ върху графа на цялата програмата и неговото преобразуване. Локалната оптимизация взема под внимание само някои програмни фрагменти.

Пример. Оптимизираният междинен код на оператора за присвояване изглежда по следния начин:
term1 = id3 * 10.0
id1 = id2 + term1

Вместо да се извика процедура inttoreal цялото число 10 се замества с реалното число 10.0.

Генерацията на окончателен код е последната фаза от транслацията. В резултат от нея се генерира асемблерен модул, обектен модул или изпълним модул. По това време е възможно изпълнението и на локална оптимизация.

Пример. Генерираният окончателен код на оператора за присвояване е следния (F -> floating point):

```
MOVF   R2, id3
MULF   R2, #10.0
MOVF   R1, id2
ADDF   R1, R2
MOVF  id1, R1
```

В процеса на транслация информацията за обектите на програмата се организира по такъв начин, че да има възможност за бързо търсене при използване на по-малко памет.

Обикновено е необходимо поддържането на таблица за идентификатори и таблица за символи.

Обработката на грешките се извършва по време на транслацията. По време на лексическия анализ се установяват грешки като недопустими символи, неправилен запис на лексеми и др. Синтактическият анализ се обработват греши свързани с неправилни програмни структури. Семантическият анализ установява грешки свързани с неправилното използване на програмните обекти, несъгласуваност на типовете и др.

Едни или други етапи на транслацията може да отсъстват или да са обединени.

2. Лексически анализ

В тази глава се въвеждат основните понятия свързани с лексическия анализ, теорията на формалните езици и граматики и крайните автомати. Посочени са основните характеристики на лексическия анализатор LEX.

2.1. Основни определения

Лексема (lexeme) се нарича абстрактния, единен езиков обект, състоящ се от група символи. Отделните символи престават да съществуват като обекти, губят значението си (семантиката си); семантиката се приписва на лексемата.

Основна задача на лексическия анализ е разбиването на входния текст, състоящ се от последователност от единични символи, на последователност от думи (лексеми). Входната последователност се състои от символи, принадлежащи към думи и от разделящи лексеми (разделители). В някои случаи лексеми не са разделени.

Лексемите се разделят на класове (лексически класове), които определят общо название на категорията елементи, притежаващи общи свойства. Примери за такива класове са ключовите думи, числовите и символните константи, идентификаторите, символните низове (стрингове), операторите и пунтуационните знаци (ограничители). Ключовите думи са резервираните думи в езика и са подмножество на идентификаторите.

Значението на лексемите се определя от разпознатия клас. В зависимост от класа, значението на лексемите може да бъде преобразувано във вътрешно представяне още на етапа на лексическия анализ. Постъпващите числа може да бъдат преобразувани в еквивалентното им двоично представяне, което обезпечава по-компактно съхранение.

Размерите на лексическите класове са различни. Например, класът на идентификаторите е безкраен. Това налага за всяка лексема да се поддържа атрибут на лексемата (указател в таблица), който да сочи допълнителна информация за нея.

В таблица на представянето се запазват по един екземпляр от всички външни представяния на идентификаторите. Обикновено хеш-таблиците се използват за организация на лексемите. Хеш-функцията се избира такава, че разпределя равномерно идентификаторите в таблицата.Желателно е тя да зависи от всички символи на идентификатора.

2.2. Азбука, дума. Свойства

Азбуката Σ (сигма) е крайно множество от (абстрактни) букви (писмени знаци, символи). Азбуките се означават с главни латински или гръцки букви с индекси или без индекси, например Σ, T, V, V_N, W, ...

Примери за азбуки: латинска {a, b, c, ..., z}; гръцка {α, β, γ, ..., ω}; двоична {0, 1}, шестнадесетична {1, 2, 3, ..., F}. Азбуката Σ = {a, b, c, +, !} съдържа 5 букви, азбуката B = {00, 01, 10, 11} съдържа 4 букви, всяка от които се състои от два символа. Използват се и специални знаци от типа на # и $. Буквите от азбуката се обозначават с букви от латинската азбука: a, b, c, ...

Дума (символен (буквен) низ) за/към азбука може да бъде произволно наредени краен брой букви от азбуката. Думите се обозначават с малки букви от гръцката азбука: α, β, ... Броят на буквите влизащи в думата се нарича дължина. Дължината на думата α се обозначава с |α| или l(α).

Например, в азбуката $Σ_1$ думата α = abc++b е с дължина |α| = 6, а в азбука $Σ_2$ думата β = 011011 има дължина |β| = 3.

За естественото число n > 0 с редицата $<a_1a_2...a_n>$, съставена от букви от азбука Σ, се обозначава думата $α = a_1a_2...a_n$ с дължина n за азбуката Σ.

Празна дума е дума несъдържаща букви. Означава се с e, ε (епсилон) или Λ. Очевидно е, че |ε| = 0.

Ако е зададена азбуката Σ, то с $Σ^*$ се означава множеството от всички думи (с крайна дължина), които могат да бъдат построени с буквите от азбука Σ. За произволна азбука Σ е изпълнено, че $ε \in Σ^*$. В частност, когато Σ = ∅, то от определението следва, че Σ = { }. Очевидно е, че множеството $Σ^*$ е изброимо и, ако $Σ \neq ∅$, то $Σ^*$ е безкрайно.

Посредством $Σ^+$ се обозначава множеството $Σ^*/\{ε\}$, т.е. $Σ^+ = Σ^*/\{ε\}$.

Например, ако Σ = {0, 1}, то $Σ^*$ = {ε, 0, 1, 00, 01, 11, 001, ...}, а $Σ^+$ = {0, 1, 00, 01, 11, 001, ...}.

Индуктивно определяне на дума и нейната дължина се извършва по следния начин:

1. ε е дума с дължина 0 над азбука Σ.
2. Ако α е дума с дължина n над азбука Σ и a е буква от Σ, тогава αa и aα са думи с дължина n + 1 над азбука Σ.
3. α е дума над азбука Σ тогава и само тогава, когато може да се получи чрез прилагане на правилата 1 и 2 краен брой пъти.

Две думи $α = a_1a_2...a_n$ и $β = b_1b_2...b_m$ съвпадат, когато |α| = |β|, т.е. n = m и $a_1 = b_1$, $a_2 = b_2$, , $a_n = b_m$.

Конкатенация (съединение) на думи α и β е непосредственото записване на думата β след думата α, означавано с αβ. Ако $α = a_1a_2...a_n$ и $β = b_1b_2...b_m$, то $αβ = a_1a_2...a_nb_1b_2...b_m$. За произволна дума α е очевидно, че εα = αε = α.

Дължината на конкатенацията на две произволни думи се получава по следната формула |αβ| = |α| + |β|.

Нека α, β и γ са думи от произволна азбука. При конкатенацията на α с βγ се получава същата дума, както при конкатенацията на αβ с γ. Това означава, че конкатенацията е

асоциативна операция. Тя обаче не е комутативна операция, тъй като лесно могат да се намерят думи α и β, за които αβ не съвпада с βα.

Степените на произволна дума α се определят индуктивно чрез операцията конкатенация по следния начин: $\alpha^0 = \varepsilon$, $\alpha^1 = \alpha$, а за n = 2, 3, ... $\alpha^n = \alpha^{n-1}\alpha$.
Например, $a^3b^2 = aaabb$ и $(ab)^3 = ababab$.

Обръщане или огледален образ (reversal) на дума α (означава се с α^R) дума съставена от буквите на α записани в обратен ред, т.е. ако $\alpha = a_1...a_n$, то $\alpha^R = a_n...a_1$. Освен това то $\varepsilon^R = \varepsilon$.

Думата α е префикс (начало) на думата β, ако съществува такава дума γ, че $\beta = \alpha\gamma$.
Например, очевидно е, че ε е префикс на abb, a е префикс на abb, ab е префикс на abb и abb е префикс на abb.

Думата α е суфикс (край) на думата β, ако съществува дума такава дума γ, че $\beta = \gamma\alpha$.

Думата α се нарича поддума (substring) на думата β, ако съществуват думи υ и ν, за които $\beta = \upsilon\alpha\nu$.

2.3. Формални езици.

Формален език L над азбука Σ се нарича всяко множество от думи над тази азбука, т.е. $L \subseteq \Sigma^*$.

Пример: Ето някои примерни азбуки и възможни формални езици над тях:

a. Σ = {a, b}; $L_1 = \varnothing$ - празен език съставен от празното множество ∅;

b. Σ = {a, b}; $L_2 = \{\varepsilon\}$ – език, съдържащ само празната дума (L_1 и L_2 са различни езици;

c. Σ = {a, b}; L_3 – език, съдържащ всевъзможни думи от буквите a и b, които имат четен брой букви a и нечетен брой букви b; L_3 съдържа безкраен брой думи;

d. Σ = {a, b}; $L_4 = \{\varepsilon, a, b, aa, ab, ba, bb\}$ – език, съдържащ 7 думи, съставени от букви a и b, чиято дължина непревишава 2. Всички останали думи от множеството Σ^* непринадлежат на езика L_4.

e. Σ = {a, b, c}; $L_7 = \{a^nbc^m \mid n > 0, m > 0\}$. Очевидно е, че $aaabcc \in L_7$, $aabca \notin L_7$.

f. Σ = {a, b}; $L_5 = \{a^n \mid n > 0\}$ - език, съдържащ думи, съставени от буква a, с дължини равни на естествно число; L_3 съдържа безкраен брой думи.

g. Σ = {a, b}; $L_6 = \{a, aab\}$.

h. Σ = {(,)}; L_8 = {множество от всички правилни скобни изрази}. Очевидно е, че $(()(())) \in L_8$, $))(() \notin L_8$.

Формалните езици са множества и за тях е възможно въвеждането на операциите обединенеие, сечение, разлика:

$L_1 \cup L_2 = \{\alpha \mid \alpha \in L_1 \text{ или } \alpha \in L_2\}$,
$L_1 \cap L_2 = \{\alpha \mid \alpha \in L_1 \text{ и } \alpha \in L_2\}$,
$L_1 - L_2 = \{\alpha \mid \alpha \in L_1 \text{ и } \alpha \notin L_2\}$.

Нека $L \subseteq \Sigma^*$. Тогава език $\Sigma^* - L$ се нарича допълнение (complement) на език L над азбука Σ.

Нека са дадени езиците $L_1, L_2 \subseteq \Sigma^*$. Конкатенация (произведение) $L_1.L_2$ на езиците L_1 и L_2 се нарича езика $L_1.L_2 = \{\alpha\beta \mid \alpha \in L_1, \beta \in L_2\}$.
Конкатенацията на езици може да се означава и без използване на знак „.".

Пример. Нека $L_1 = \{a, abb\}$, $L_2 = \{bbc, c\}$. Тогава $L_1.L_2 = \{ac, abbc, abbbbc\}$.

Декартово произведение на крайна редица от множества $\Sigma_1, ..., \Sigma_n$ (означава се $\Sigma_1 \times ... \times \Sigma_n$) се нарича множеството от всички n-орки от вида $(a_1, ..., a_n)$, където $a_i \in \Sigma_i$.

Степените на произволен формален език L се определят индуктивно чрез операцията конкатенация по следния начин: $L^0 = \{\varepsilon\}$, $L^1 = L$, а за $n = 2, 3, ...$ $L^n = L^{n-1}\alpha$.

Итерация (Kleene closure) L^* на произволен език L се нарича обединението на всички степени на L: $L^* = \cup_{n \in N} L^n$. Тази операция се нарича още звездата на Клини (Kleene star, star operation).

Пример. Ето някои примери за итерации на езици:

a. $L = \{a, ab\}$. Тогава $L^2 = \{aa, aab, aba, abab\}$ и $L^* = \{\varepsilon, ,a, b, aa, aa, ab, ba, bb, aaa, aab, aba, ...\}$.

b. $L = \{aa\}$. Тогава $L^* = \{(aa)^n \mid n \geq 0\}$.

Нека $L \subseteq \Sigma^*$. Тогава обръщането L^R на език L е съставено от обръщанията на всички думи на L, т.е. $L^R = \{\alpha^R \mid \alpha \in L\}$.

Нека Σ_1 и Σ_2 са две произволни азбуки. Ако изображението $h: \Sigma_1^* \to \Sigma_2^*$ удовлетворява условието $h(\alpha\beta) = h(\alpha)h(\beta)$ за всички думи $\alpha \in \Sigma_1^*$ и $\beta \in \Sigma_2^*$, то изображението h се нарича хомоморфизъм. Възможно е да се докаже, че $h(\varepsilon) = \varepsilon$.

Ако $h: \Sigma_1^* \to \Sigma_2^*$ Прилагайки хомоморфизъм към език L се получава друг език h(L), който представлява множеството от думи $\{h(\alpha) \mid \alpha \in L\}$.

Пример. Нека са дадени азбука $\Sigma = \{0, 1\}$ и хомоморфизъм $h: \Sigma^* \to \Sigma^*$ зададен с равенствата $h(0) = 0110$ и $h(1) = \varepsilon$. Тогава $h(\{100, 11\}) = \{0110011001, \varepsilon\}$.

2.4. Пораждащи граматики. Йерархия на Чомски.

Един от способите за представяне на език се състои в това, да се даде алгоритъм, който да определи дали дадена дума е от езика или не. По-общия метод е да се използва процедура, която прекратява работа с отговор „да" за думи от езика и или не завършва или завършва с отговор „не" за думи, които не са от езика. Такива алгоритми и процедури разпознават език. Съществуват езици, които може да се разпознаят с помощта на процедури, но не и с помощта на алгоритми.

Друг способ за представяне на език е да се даде процедура, която систематически да поражда думи от езика последователно в някакъв ред.
Ако съществува алгоритъм или процедура, които разпознават думи от езика над азбука Σ, то на тяхна основа може да се построи пораждащ механизъм – алгоритъм или процедура, пораждащи този език.

Рекурсивно изчислим език се нарича език думите на който може да се пораждат с процедура.

Един език е рекурсивен, ако съществува алгоритъм за неговото разпознаване.

Пораждаща граматика (граматика от тип 0, generative grammar, rewrite grammar) се нарича четворката $\Gamma = \langle \Sigma, N, S, P \rangle$, където

a. Σ е крайна азбука на терминалните символи (терминали, terminal) (терминална азбука). Σ е азбуката на пораждания език.

b. N е крайна азбука на нетерминалните символи (нетерминали, променливи) (nonterminal, variable), $N \cap \Sigma = \varnothing$. N има помощна роля в процеса на пораждане и неучаства в думите на пораждания език; нетерминалите играят ролята на синтактични категории в езика.

c. P е крайно множество от правила (продукции) (rewriting rule, production) на пораждащата граматика, за което е изпълнено, че $P \subset (\Sigma \cup N)^+ \times (\Sigma \cup N)^*$. Правилата P са наредени двойки от думи (α, β) над азбуката $\Sigma \cup N$ и се записват във вида $\alpha \to \beta$, което се чете „думата α се замества с думата β. Думите α и β са съответно лява и дясна страна на правилото.

d. S е фиксиран начален символ (аксиома) (start symbol), $S \in N$.

Пример. Нека са дадени множествата $\Sigma = \{a, b, c\}$, $N = \{S\}$, $P = \{S \to acSbcS,\ cS \to \varepsilon\}$. Тогава $\langle \Sigma, N, S, P \rangle$ е пораждаща граматика.

За означаване на n правила с еднакви леви части $\alpha \to \beta_1, ..., \alpha \to \beta_n$ често се използва съкратения запис $\alpha \to \beta_1|...|\beta_n$.

Думата β се извежда непосредствено от думата α в граматиката $\Gamma = \langle \Sigma, N, S, P \rangle$ (означава се с $\alpha \Rightarrow_\Gamma \beta$ или само с $\alpha \Rightarrow \beta$), ако $\alpha = z_1uz_2$, $\beta = z_1vz_2$ и $u \to v \in P$, където z_1, z_2, u и v са думи от $\Sigma \cup N$.

Пример. Нека $\Gamma = \langle \{0, 1, 2\}, \{S\}, \{S \to 02S12S, 2S \to \varepsilon\} \rangle$. Тогава $2S02S \Rightarrow_\Gamma 2S0$.

Редицата от думи $\alpha_1, \alpha_2, ..., \alpha_n$, за която $\alpha_1 \Rightarrow_\Gamma \alpha_2 \Rightarrow_\Gamma ... \Rightarrow_\Gamma \alpha_n$, където $n \geq 0$, се нарича пълен извод (derivation) на α_n от α_1 в граматиката Γ. Означава се с $\alpha_1 \Rightarrow_\Gamma^* \alpha_n$ или само $\alpha_1 \Rightarrow^* \alpha_n$. Думата α_n е пълно изводима. Числото n се нарича дължина (брой стъпки, тактове) на извода. С $\alpha_1 \Rightarrow^* \alpha_n$ се означава пълен извод с произволно число стъпки. Изводът за точно n стъпки се означава с $\alpha \Rightarrow^n \beta$, когато $\alpha \Rightarrow \varphi$, $\varphi \Rightarrow^{n-1} \beta$. Очевидно е, че $\alpha \Rightarrow^0 \alpha$. Изводът за ненулево число стъпки се означава $\alpha \Rightarrow^+ \beta$, когато $\exists n \geq 0 : \alpha \Rightarrow^n \beta$.

В частност, за всяка дума $\alpha \in (\Sigma \cup N)^*$ е възможен следния извод $\alpha \Rightarrow^* \alpha$ с дължина 0.

Пример. Нека $\Gamma = \langle \{0, 1\}, \{S\}, S, \{S \to 0S0, S \to 1\} \rangle$. Тогава $0S0 \Rightarrow^* 0000S0000$ е извод с дължина 3.

Сентенциална форма се нарича всяка дума $\alpha \in (\Sigma \cup N)^*$, която може да се изведе от аксиомата (началния символ) S на граматиката, т.е. $S \Rightarrow^* \alpha$.

Дума от език – сентенциална форма, съдържаща само терминални символи.

Думата α се поражда от граматиката $\Gamma = \langle \Sigma, N, S, P \rangle$, ако $\alpha \in \Sigma$ и съществува извод на α от S : $S \Rightarrow^* \alpha$.

Език, породен от граматиката $\Gamma = \langle \Sigma, N, S, P \rangle$ (означава се с $L(\Gamma)$, се нарича множеството от всички думи, които граматиката Γ може да породи, т.е. $L(\Gamma) = \{\alpha \mid \alpha \in \Sigma^* \text{ и } S \Rightarrow^* \alpha\}$. Казва се още, че граматиката Γ поражда (generates) език $L(\Gamma)$.(Език – множеството от всички синтактично правилни думи, които могат да бъдат породени от граматиката Γ).

Две граматики Γ_1 и Γ_2 се наричат еквивалентни, ако $L(\Gamma_1) = L(\Gamma_2)$, т.е. ако пораждат един и същ език.

Пример. Нека е дадена граматиката $\Gamma = \langle \Sigma, N, S, P \rangle$, където $\Sigma = \{a, b\}$, $N = \{S\}$, $P = \{S \to aSb, S \to ab\}$. Тук S е единствен нетерминален символ и начален символ. Прилагайки първо правило n – 1 брой пъти и след това второ правило се получава: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow a^3Sb^3 \Rightarrow ... \Rightarrow a^{n-1}Sb^{n-1} \Rightarrow$. Граматиката Γ поражда език $L(\Gamma) = \{a^n b^n \mid n > 0\}$.

Контекстна граматика (контекстно-зависима граматика, граматика от тип 1) (context-sensitive grammar, phrase-structure grammar) се нарича пораждащата граматика $\Gamma = \langle \Sigma, N, S, P \rangle$, ако за всяко нейно правило $\alpha \to \beta \in P$ е изпълнено условието $|\alpha| \le |\beta|$. В този случай, когато $S \to \varepsilon \in P$, символ S не се среща в десните части на правилата.

Бележка. Правилата на контекстно-зависимата граматика имат вида: $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, където $\alpha_1, \alpha_2 \in V^*$, $\beta \neq \varepsilon$, $A \in N$.

Безконтекстна граматика (контекстно-свободна граматика, тип 2) (contex-free grammar) се нарича пораждащата граматика $\Gamma = \langle \Sigma, N, S, P \rangle$, ако всяко нейно правило има вида $A \to \alpha \in P$, където $A \in N$, $\alpha \in (\Sigma \cup N)^*$. В този случай, когато $S \to \varepsilon \in P$, символ S не се среща в десните части на правилата. Терминът „безконтекстна" в името на граматиката отразява факта, че замяната на нетерминален символ в дума независи от контекста.

Пример. Нека $\Sigma = \{0, 1\}$, $N = \{S\}$, $P = \{S \to a, S \to aSaSb\}$. Тогава $\Gamma = \langle \Sigma, N, S, P \rangle$ е безконтекстна граматика.

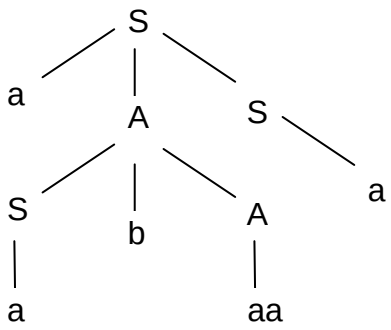
Автоматна граматика (регулярна граматика, рационална граматика, линейна граматика, P-граматика, граматика от тип 3) (linear grammar, regular grammar) се нарича пораждащата граматика $= \langle \Sigma, N, S, P \rangle$, ако всяко нейно правило има вида $A \to a$ или $A \to aB$, където $A, B \in N$, $a \in \Sigma^*$. Такива граматики се наричат още дяснолинейни граматики (right-linear grammar, right-regular grammar).

Автоматните граматики имащи правила $A \to a$ или $A \to Ba$, където $A, B \in N$, $a \in \Sigma^*$ се наричат ляволинейни граматики (left-linear grammar, left-regular grammar).

Модифицирани автоматни граматики се наричат автоматните граматики с правила $A \to aB$ и $A \to \varepsilon$, където $A, B \in N$, $a \in \Sigma^*$. В този случай, когато $S \to \varepsilon \in P$, символ S не се среща в десните части на правилата.

Представянето на изводите се извършва чрез дърво на разбора (извода) (parse (derivation) tree). Корен (root) на дървото е началният символ S. Връх (vertex) на дървото се означава с етикет (label) и това е лявата страна н а продукцията. Като наследници на върховете се изписват десните страни на приложеното правило от P.

Пример. Нека е дадена контекстно свободната граматика $\Gamma = \langle \Sigma, N, S, P \rangle$, където $\Sigma = \{a, b\}$, $N = \{S, A, B\}$, $P = \{S \to a, S \to aAS, A \to aa, A \to SS, A \to SbA\}$. За думата aabaaa се построява следното дърво на разбора:



Пример. Нека $\Sigma = \{a, b\}$, $N = \{S, A, B\}$, $P = \{S \to A, B \to abba\}$. Тогава $\Gamma = \langle \Sigma, N, S, P \rangle$ е автоматна граматика.
Нека е даден

Един формален език се нарича от общ вид (от тип 0), контекстен (от тип 1), безконтекстен (от тип 2) или автоматен (от тип 3), когато има пораждаща го граматика, която е съответно от общ вид (от тип 0), контекстна (от тип 1), безконтекстна (от тип 2) или автоматна (от тип 3).

От общ вид, контекстните, безконтекстните и автоматните граматики образуват йерархията на Чомски (Chomsky hierarchy) за пораждащите граматики. От общ вид, контекстните, безконтекстните и автоматните езици образуват йерархията на Чомски за формалните езици.

За задаването на синтаксиса на езиците за програмиране от високо ниво често се използват безконтекстните граматики.

Езиците от тип 0 се наричат рекурсивно изчислими.

Граматиката $\Gamma = \langle \Sigma, N, S, P \rangle$ се нарича рекурсивна, ако съществува алгоритъм, който определя (за крайно време) дали произволна дума $\alpha \in \Sigma^*$ се поражда от Г.

Лема. [Карпов] Ако граматиката $\Gamma = \langle \Sigma, N, S, P \rangle$ е контекстна, то тя е рекурсивна.

2.5. Регулярни изрази и регулярни множества

Нека Σ е крайна азбука. Регулярен израз и регулярно множество над азбука Σ се определят рекурсивно по следния начин:

(1) $\varnothing$ е регулярен израз над Σ и представя празното $\varnothing$ множество.

(2) ε е регулярен израз над Σ и представя множеството {ε} в азбука Σ.

(3) a е регулярен израз над Σ за всяка буква $a \in \Sigma$ и представя множеството {a}.

(4) ако x и y са регулярни изрази над Σ, представящи съответно множествата X и Y, то (x + y) (записва още (x | y)), (xy) и (x^*) са също регулярни изрази над Σ и представят множествата: $X \cup Y$ (обединение), XY (конкатенация) и X^* (итерация).

(5) регулярни са само изразите, които могат да се получат чрез прилагане на правила от (1) до (4) краен брой пъти.

За съкратено записване на израз xx^* се използва записа x^+. Излишните скоби в регулярните изрази, които не довеждат до недоразумение, се отстраняват. В регулярните изрази най-голям приоритет има операция * (итерация), след това конкатенация и най-малък $\cup$ (обединение). Със записа L(r) се означава регулярно множество, обозначаващо регулярния израз r. Използват се записи от вида:

$d_1 = r_1$
$d_2 = r_2$
.....
$d_n = r_n$

където d_i са различни имена, а всяко r_i е регулярен израз над символите $\Sigma \cup \{d_1, d_2, ..., d_{i-1}\}$, т.е. символи от основната азбука и по-рано определени символи. По този начин за произволно r_i е възможно да се построи регулярен израз над Σ, чрез повторно заменяне имената на регулярните изрази.

Пример. Ето някои примери за използване на имена за регулярни изрази:

a. Регулярен израз за множество от идентификатори.
   буква = a|b|c|...|x|y|z
   цифра = 0|1|...|9
   идентификатор = буква(буква|цифра)*
b. Регулярен израз за множества от числа с дробна част:
   цяло_число = цифра$^+$
   дробна_част = . цяло_число|ε
   експонента = (E(+|-|ε)цяло_число)|ε
   число = цяло_число дробна_част експонента

Два регулярни израза са равни (еквивалентни), ако представят едно и също множество. Очевидно е, че за всяко регулярно множество може да се намери един регулярен израз, който обозначава това множество. И обратно, за всеки регулярен израз може да се построи регулярно множество, обозначаващо този израз.

Пример. Ето някои примери за регулярни изрази и обозначаваните от тях регулярни множества:

a. ab – обозначава множеството {ab}.
b. a^* - обозначава $\{a\}^*$.
c. a(ε + a) + b – обозначава {a, b, aa}.
d. $(a + b)^*$ - обозначава {a, b}*.
e. (a + b)(a + b + 0 + 1) – обозначава всички думи от $\{0, 1, a, b\}^*$, започващи с a или b.

Лема [Ахо, Улман]. нека α, β и γ са регулярни множества. Тогава са валидни следните равенства:

(1)	$\alpha + \beta = \beta + \alpha$	(2)	$\varnothing^* = \varepsilon$
(3)	$\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$	(4)	$\alpha(\beta\gamma) = (\alpha\beta)\gamma$
(5)	$\alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma$	(6)	$(\alpha + \beta)\gamma = \alpha\gamma + \alpha\beta$
(7)	$\alpha\varepsilon = \varepsilon\alpha = \alpha$	(8)	$\varnothing\alpha = \alpha\varnothing = \varnothing$
(9)	$\alpha^* = \alpha + \alpha^*$	(10)	$(\alpha^*)^* = \alpha^*$
(11)	$\alpha + \alpha = \alpha$	(12)	$\alpha + \varnothing = \alpha$

Теорема. (Теорема на Клини). Един език $L \subseteq \Sigma^*$ е автоматен тогава и само тогава, когато се представя от регулярен израз над Σ.

За формализиране описанието на граматиките се използват т.н. синтактични метаезици.

Определение. Изкуствените езици, които се използват за описание на други езици (изкуствени или естествени) се наричат метаезици. Съществуват метаезици за описание на синтаксиса и метаезици за описание на семантиката.

Исторически първият метаезик е формата (нотацията) на Бекус-Наур (Backus-Naur Form, BNF) или т.н. БНФ, която е наречена така в чест на Бекус (главен разработчик на компилатора Фортран) и Наур (участник в разработката на АЛГОЛ-60). Първоначално БНФ е използвана за строго определяне на синтаксиса на езика Алгол.

Граматиката на един език за програмиране, описан чрез БНФ, се състои от краен брой граматични правила, всяко с лява и дясна част, отделени със символ “::=”, вместо със стрелка “$\rightarrow$”. Символ “::=” се чете „по дефиниция е”.

Лявата част на граматично правило се състои от металингвистична променлива – нетерминален символ, който представлява текст, заграден с металингвистични символи “$\langle$” и “$\rangle$”. Текстът в скобите е име на синтактична категория от езика, чиято роля в граматикат се изпълнява от нетерминалния символ.

Дясната част на граматичното правило се състои от една или повече думи от нетерминални и терминални символи – варианти (алтернативни дефиниции, алтернативни възможности). Те се отделят един от друг с металингвистичен символ вертикална черта “|”, чете се „или”.

Пример. Запис в БНФ на цяло число:
⟨цяло число⟩::=
<цяло число без знак>|+<цяло число без знак>|-<цяло число без знак>
<цяло число без знак>::=<цифра>|<цяло число без знак> <цифра>
<цифра>::= 0|1|2|3|4|5|6|7|8|9

БНФ не позволява задаването на контекстни условия. При използване на БНФ те се задават в словесна форма. Като цяло БНФ е неудобна при практическото описание на езици, затова в нея са внесени допълнения. Версията на БНФ, в която допълненията са най-много, се нарича разширена БНФ (РНБФ) (Extended BNF, EBNF). РБНФ неувеличава описателната сила на БНФ, тя увеличава удобството при четене и писане. Включват се следните допълнения:

- нетерминалните символи се записват като отделни думи.
- терминалните символи се записват в кавички, напр. “begin”.
- незадължителните елементи се поставят в квадратни скоби: “[” и “]”.
- повторенията (нула или повече) на група от символи се загражда във фигурни скоби “{” и “}”.
- символ равенство “=” се използва вместо символ “::=”.
- скобите “(” и “)” се използват за групиране.
- символ точка “.” се използва за край на правило.
- коментарите се поставят между символите “(*” и “*)”.

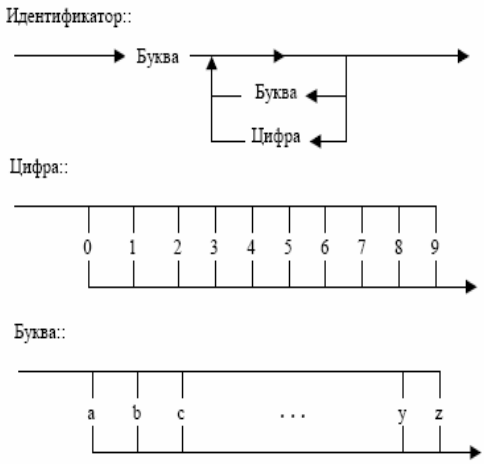
Пример. Запис в РБНФ на цяло число:
```
Integer = Sign UnsignedInteger.
UnsignedInteger = digit {digit}.
Sign =  ["+"|"-"].
digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
```

Пример. Запис в РБНФ и стандартен запис в граматика на Чомски:

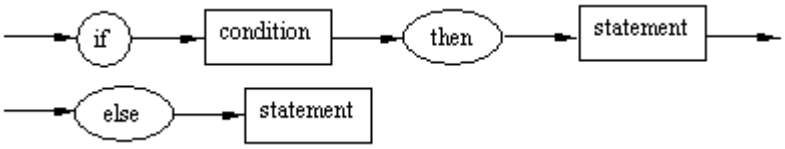
A = “α”{“β”}“γ”.(*РБНФ*)
$A \to \alpha B\gamma$, $B \to B\beta$ и $B \to \varepsilon$ - Чомски

Синтактическите диаграми служат за определяне на езици с помощта на графично представяне. Те са използвани за задаване синтаксиса на езиците Pascal, Modula-2 и Fortran-77. Синтактическите диаграми са пораждащ механизъм. Правилата за пораждане на думи от езика се представят във вид на няколко графови потока от данни с един вход и един изход. Произволен път в диаграма от входа към изхода задава възможна последователност от символи в пораждана дума от езика. На всеки нетерминален символ съответства отделна диаграма. Срещащите се нетерминални символи трябва да се заменят със съответна дума, получаваща се от синтактическата диаграма за съответния нетерминален символ.

Пример. Определяне на идентификатор чрез синтактична диаграма.



Често вместо горните синтактически диаграми се използват други, в които терминалните символи се записват в кръг, а нетерминалните в правоъгълници.
Пример. Описание на условен оператор в Pascal.



2.6. Крайни автомати

Регулярните изрази се използват за описание на регулярни множества. За разпознаване на регулярни множества служат т.н. крайните автомати.

Недетерминиран краен автомат M (НДКА M) над азбуката Σ се нарича петорката: $M = \langle Q, \Sigma, \delta, q_0, F\rangle$, в която :

Q - непразно крайно множество от вътрешни състояния, наречено азбука на вътрешните състояния (азбука на автомата).

Σ - крайно множество от допустими входни символи, наречено входна азбука.

δ - функция на преходите, която е изображение на множеството $Q \times \Sigma$ в множеството от подмножества P(Q) на Q, т.е. $\delta : Q \times \Sigma \to P(Q)$. δ определя поведението на управляващото устройство.

$q_0 \in Q$ – начално състояние.

$F \subseteq Q$ – множество от заключителните състояния.

Частен случай на недетерминираните крайни автомати са детерминираните крайни автомати.

Детерминиран краен автомат M (ДКА M) над азбуката Σ се нарича петорката: $M = \langle Q, \Sigma, \delta, q_0, F \rangle$, в която :

Q - непразно крайно множество от вътрешни състояния, наречено азбука на вътрешните състояния (азбука на автомата).

Σ - крайно множество от допустими входни символи, наречено входна азбука.

δ - функция на преходите, която е изображение на множеството $Q \times \Sigma$ в Q, т.е. $\delta : Q \times \Sigma \to Q$. δ определя поведението на управляващото устройство.

$q_0 \in Q$ – начално състояние.

$F \subseteq Q$ – множество от заключителните състояния.

При ДКА множеството $\delta(q, a)$ съдържа неповече от едно състояние за произволни q и a и освен това $\delta(q, \varepsilon) = \varnothing$.

Детерминираният краен автомат M се нарича напълно определен, ако функцията на преходите δ е дефинирана за всички наредени двойки $Q \times \Sigma$, т.е. няма неопределени преходи.

Думата $\alpha = a_1...a_k$ над азбука Σ се разпознава от краен автомат $M = \langle Q, \Sigma, \delta, q_0, F \rangle$, ако съществува редица от състояния $q_1, q_2,..., q_n$ такава, че $q_0 = q_1$, $q_n \in F$ и за $\forall i \forall j : 1 \le i < n, 1 \le j < k, \delta(q_i, a_j) = q_{i+1}$.

Казва се, че език L се разпознава от краен автомат M (записва се L(M), ако всяка дума от езика L се разпознава от крайния автомат M.

Всеки краен автомат може да се изобрази графично с диаграма на преходите (представляваща ориентиран граф), като за всяко вътрешно състояние се поставя по един връх. Ребрата се отбелязват със символ $a \in \Sigma \cup \{\varepsilon\}$ и свързват два върха p и q, ако $p \in \delta(q, a)$. На диаграмата началното и заключителното състояние се отбелязват съответно с входна стрелка и двоен кръг.

Теорема. Нека L е език, който се разпознава от недетерминиран краен автомат M_1, т.е. $L = L(M_1)$. Тогава съществува детерминиран краен автомат M_2, който също разпознава L, т.е. $L = L(M_2)$.

Възможно е да се извърши конструктивно доказателство на теоремата, т.е. по пътя на указване на общ алгоритъм за построяване на детерминиран краен автомат M_1, разпознаващ същия език какъвто се разпознава от M_2. Нека $M_1 = \langle Q_1, \Sigma, \delta_1, q_{01}, F_1 \rangle$, тогава нека M_2 се определи по следния начин $M_2 = \langle Q_2, \Sigma, \delta_2, q_{02}, F_2 \rangle$:

- Q_2 съвпада с множеството от състояния на автомат M.
- $q_{02} = q_{01}$.
- $F_2 = \{S_1 \in Q_1 : S_1 \cap F_1 \neq 0\}$
- $\delta_2(S_1, a_1) = S_2$ за $\forall S_1 \subseteq Q_1$, където
  $S_2 = \{p_1 : \delta_1(q_{01}, a_1)$ съдържа p1 за някое $q_{01} \in S_1\}$

Теорема. За произволен недетерминиран краен автомат може да се построи еквивалентен детерминиран краен автомат.

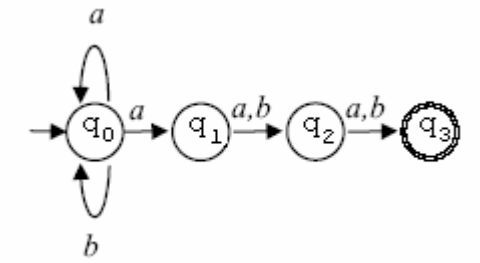
Теорема. Детерминираните и недетерминираните крайни автомати разпознават езиците, пораждани от автоматните граматики, и само тях.

Пример. Нека е дадено регулярното множество L(r), обозначаващо регулярния израз $r = (a+b)^*a(a+b)(a+b)$.

a. Недетерминирания краен автомат M, разпознаващ език L, се задава по следния начин: $M = \langle\{q_0, q_1, q_2, q_3\}, \{a, b\}, \delta, q_0, \{q_3\}\rangle$, където функцията на преходите δ се определя така:

$\delta(q_0, a) = \{q_0, q_1\}$, $\delta(q_1, b) = \{q_2\}$, $\delta(q_2, b) = \{q_3\}$.
$\delta(q_1, a) = \{q_2\}$, $\delta(q_2, a) = \{q_3\}$,

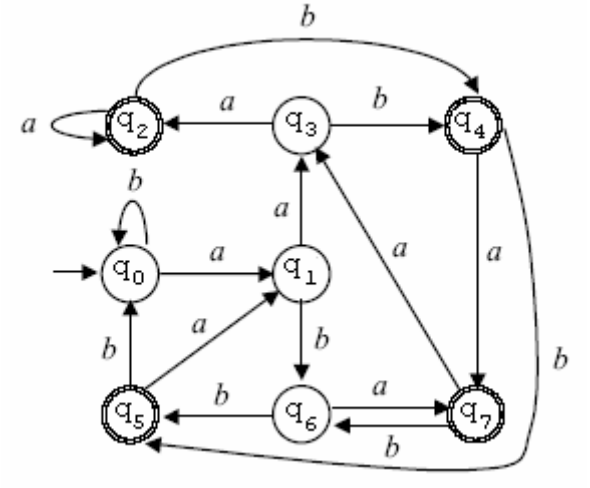
Диаграмата на преходите на автомата е следната:



b. Детерминирания краен автомат M, разпознаващ език L, се задава по следния начин: $M = \langle Q, \{a, b\}, \delta, q_0, F\rangle$, където $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$, $F = \{q_2, q_4, q_5, q_7\}$ и функцията на преходите δ е следната:

$\delta(q_0, a) = \{q_1\}$, $\delta(q_3, a) = \{q_2\}$, $\delta(q_6, a) = \{q_7\}$,
$\delta(q_0, b) = \{q_0\}$, $\delta(q_3, b) = \{q_4\}$, $\delta(q_6, b) = \{q_5\}$,
$\delta(q_1, a) = \{q_3\}$, $\delta(q_4, a) = \{q_7\}$, $\delta(q_7, b) = \{q_3\}$,
$\delta(q_1, b) = \{q_6\}$, $\delta(q_4, b) = \{q_5\}$, $\delta(q_7, b) = \{q_6\}$.
$\delta(q_2, a) = \{q_2\}$, $\delta(q_5, a) = \{q_1\}$,
$\delta(q_2, b) = \{q_4\}$, $\delta(q_5, b) = \{q_0\}$,

Диаграмата на преходите на автомата е следната:



Автоматите $M_1 = \langle Q_1, \Sigma, \delta_1, q_{01}, F_1\rangle$ и $M_2 = \langle Q_2, \Sigma, \delta_2, q_{02}, F_2\rangle$ се наричат еквивалентни, ако разпознават един и същ език над азбука Σ.

Две състояния s_i и s_j се наричат еквивалентни, ако за $\forall x \in \Sigma^*$ е изпълнено, че $\delta(q_i, x) \in F \Leftrightarrow \delta(q_j, x) \in F$. Очевидно е, че ако две състояния s_i и s_j са еквивалентни, то за $\forall x \in \Sigma$ състоянията $\delta(s_i, a)$ и $\delta(s_j, a)$ са също еквивалентни.

Освен това, както в ДКА преход δ(q, ε) може да възникне само за крайно състояние q, то не е възможно заключително състояние да е еквивалентно на незаключително състояние.

Между два ДКА, разпознаващи един и същи език,по-малко памет (при реализация) заема този, който има по-малко състояния. Броят състояния може да се намали чрез премахване на състоянията, които не се използват при анализ на разпознаваните думи.
Теорема. Нека $L \subseteq \Sigma$ е произволен автоматен език. Тогава сред детерминираните крайни автомати, разпознаващи L, съществува единствен (с точност до преименуване на вътрешните състояния) автомат с минимален брой вътрешни състояния.

2.7. Програмиране на лексически анализатори

Лексическият анализатор се извиква като подпрограма. При обръщение към него се връщат два резултата: тип на избраната лексема и значение (или указател към значението) за класовете лексеми (идентификатори, числа, низове и др.). При формиране на таблици с имена, то се връща указател към името. Тялото на лексическия анализатор е диаграма на преходите съответстваща на краен автомат.

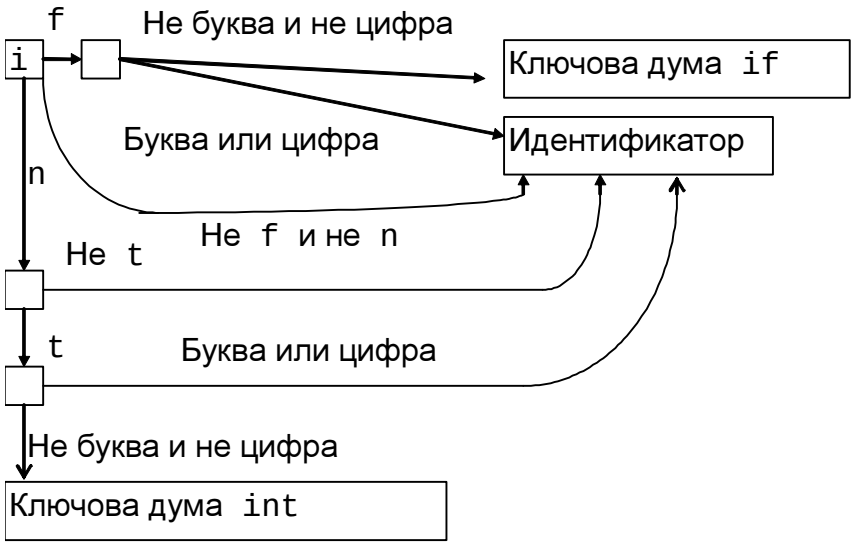
Проблем е отделянето на ключовите думи, тъй като те са подмножество на идентификаторите. Възможно е диагностициране на поредната лексема за съвпадение с ключова дума и в случай на неуспех да се направи опит за отделяне на лексемата в някой клас или след отделяне на лексема идентификатор да се погледне в таблицата от ключови думи за сравнение.

Основната операция в лексическия анализатор, която отнема най-много време при неговата работа е приемането на поредния символ и проверка на принадлежността му към някакъв диапазон.

Пример. Установяване на принадлежност на поредния символ, към групата цифри,букви или знаци, може да се осъществи с предварително запълнен с тях масив и непосредствена проверка в него.

Отделянето на ключовите думи може да се извърши и преди отделянето на идентификаторите. Лексическият анализатор работи бързо, ако ключовите думи се отделят непосредствено. За целта е необходимо да се построи краен автомат.

Пример. Показан е фрагмент от краен автомат, разпознаващ ключовите думи if и int, последван от генерираният код на език C:




```
case 'i':
   if (cp[0]=='f' && !(map[cp[1]] & (digit | letter)))
      {cp++; return IF;}
   if (cp[0]=='n' && cp[1]=='t'&&!(map[cp[2]] & (digit | letter)))
      {cp+=2; return INT;}
```

С `cp` е означен указател към текущия символ. В масив `map` класовете символи се кодират побитово.

Пример. Построяване на лексически анализатор различаващ коментари, идентификатори и осмични, десетични, шестнадесетични и низови константи.
Необходимо е множеството символи да се групират в следните класове:

- цифра 0
- цифри 1..7
- цифри 8..9
- букви A..F, a..f
- буква “x”
- всички други букви
- символ “/”
- символ “*”
- символ “””
- всички останали символи.

Групирането на символите в отделни класове се извършва с функция `sclass`:

```
int sclass(char c){
switch (c)
     {
case '*': return(7);
case '"': return(8);
case '/': return(6);
case '0': return(0);
case '8': return(2);
case '9': return(2);
case 'x': return(3);
default:
     if (isdigit(c)) return(1);
     if (isalpha(c)){
```

```
          if ((toupper(c) >= 'A') && (toupper(c) <='F'))
               return(4);
          return(5);
          }
     return(9);
     }
}
```

Необходимо е да се изгради матрица на преходите. Тя представлява двумерен масив, който за всяка двойка състояние (ред) и клас символ (стълб) определя новото състояние, в което се преминава. Номерът на това състояние се намира в матрицата. Принцип на запълване на матрицата: ако в състояние $\{q_i\}$ и входен символ a на диаграмата има дъга (преход) в състояние $\{q_j\}$, т.е. $\delta(q_i, a) = \{q_j\}$, то елемент от масива $D[\{q_i\}][cl]$ се инициализира със стойност $\{q_j\}$, където cl = sclass(a). В разглеждания пример матрицата изглежда така:

```
int D[11][10]= {
     { 2, 5, 5, 1, 1, 1, 6,-8, 9,-9 },
     { 1, 1, 1, 1, 1, 1,-1,-1,-1,-1 },
     { 5, 4, 5, 3,-4,-4,-4,-4,-4,-4 },
     { 3, 3, 3,-2, 3,-2,-2,-2,-2,-2 },
     { 4, 4,-3,-3,-3,-3,-3,-3,-3,-3 },
     { 5, 5, 5,-4,-4,-4,-4,-4,-4,-4 },
     {-5,-5,-5,-5,-5,-5,-5, 7,-5,-5 },
     { 7, 7, 7, 7, 7, 7, 7, 8, 7, 7 },
     { 7, 7, 7, 7, 7, 7,-6, 7, 7, 7 },
     { 9, 9, 9, 9, 9, 9, 9, 9,10, 9 },
     {-7,-7,-7,-7,-7,-7,-7,-7, 9,-7 }   };
```

Целта на лексическият анализатор е да се разпознават и отделят лексемите, затова той преминава в множество заключителни състояния, завършващи с разпознаване. Действията по разпознаване са еднакви за всички класове лексеми, затова лексическият анализатор има по едно заключително състояние за всеки клас. Тези състояния имат отрицателна стойност. Програмата изпълнява следните стъпки:

- формира се низ от натрупани символи, образуващи лексема;
- установява се типа на разпознатата лексема в съответствие с номера от крайното състояние на крайния автомат;
- крайният автомат се връща в начално състояние;
- при попадане в състояние, когато се обработва символ непринадлежащ на лексема, то програмата връща входната последователност на зададено число символи за повторно разпознаване. Например, десетична константа е разпозната, когато се прочете следващия символ – различен от цифра. Необходимо е връщане назад. По тази причина в програмата се определя масив, в който за всяко заключително състояние се указва количеството символи за връщане.

```
int  W[]={ 1,1,1,1,1,0,1,0,0 };
if (ST  < 0){
      int j; ST = -ST-1;        // тип лексема
      i=i-W[ST];                  // върнат символ
      printf("%s ",out[ST]);      // показване на символ
for (j=FIX; j<i; j++) putchar(S[j]);
puts("");
```

```
ST=0; FIX=i; }
```

За фиксиране на началния символ, образуващ лексема, в програмата се добавя променлива `FIX`, която при преход в начално състояние пази разположението на текущия символ. Целият изходен код на програмата се намира във файл `lexan.cpp`.

Лексическият анализатор се проектира така, че да се справя с лексическите грешки. Към тях се отнасят следните:

(1) замяна в лексема на правилен символ с неправилен,
(2) поставяне в лексема на излишен символ,
(3) пропуск на символ в лексема,
(4) отделяне на два съседни символа от лексема.

2.8. Генератор на лексически анализатори Lex

Lex е програма за генериране на лексически анализатори. Lex чете входен файл yyin и възприема описанията в него на регулярни изрази, след това построява детерминиран краен автомат за тяхното разпознаване.

Записването на регулярните изрази в Lex-програма се извършва по следните правила:
Символ от азбуката и низ от символи, записан в кавички, описват регулярен израз, зададен от тях: “continue”. Специалните символи (в това число +-*?()[]{}|/\^$.<>) се записват след префикс \ или в кавички. Например, ето три начина за задаване на символ a: a, “a”, \a. Непечатните символи се задават както в езика C: \t – табулация, \n – нов ред (разделител на низове), \b – връщане знак назад и негово изтриване, \\ - обратна черта, \007 и др.
Регулярен израз, състоящ се от един символ, принадлежащ на определен клас, се описва по следния начин:

. – произволен символ освен \n
[A-Za-z] – голяма или малка латинска буква
[0-9] или [0123456789] – произволна цифра
[^0-7] – произволен символ, освен осмична цифра
[^0-9A-Fa-f] – произволен символ, освен шестнадесетична цифра
[abc] – буква a, b или c

Използва се следната граматиката за записване на регулярните изрази в Lex-програма:

r* - итерация 0 или повече пъти
r+ - итерация 1 или повече пъти
r? – незадължителен символ
rr – конкатенация
r{m, n} – повторение на r от m до n пъти
r{m} – повторение m пъти
r{m, } – повторение m и повече пъти
^r – избор на r, ако е в началото на низ
r$ - избор на r, ако е в края на низ
$r_1|r_2$ - r_1 или r_2
r_1/r_2 - r_1, ако след него следва r_2
(r) – групиране, за указване реда за пресмятане

Примери. Запис на изрази в Lex-програма:

[A-za-z][A-za-z0-9]* - идентификатор
^#" "*define –оператор в препроцесора на C

Lex-програмата се състои от три секции: описания, правила на транслацията и процедури, разделени една от друга с низ %%.

описания
%%
правила
%%
процедури

Секцията за описания включва описания на променливи, константи и регулярни определения. Съдържат се определения на макросимволи (метасимволи, нетерминални символи от регулярни изрази) във вида:

име_макросимвол израз

Ако по-нататък в Lex-програмата се срещне {име_макросимвол}, то се заменя с израз. Ако низ от секцията описание започва с интервал или е ограден в скоби %{...}, то той така се копира в изходната С-програма. Тези низове могат да съдържат описание на променливи, реализация на функции и др.

Пример. Ето някои полезни макросимволи:

```
letter   [a-zA-Z_#]
digit    [0-9]
ident    {letter}({letter}{digit})*
iconst (\+|\-)?{iuconst}
```

Втората секция съдържа правила във вида

регулярен_израз {действие}

Действието представлява последователност от оператори на език С, изпълнявани при успешно разпознаване на низ регулярен_израз. Действие, указано в началото на секцията, без регулярен_израз се изпълнява до началото на разбора.
Lex прави опит да отдели най-дългия низ от входния поток. Ако няколко правила връщат низове с еднаква дължина, то се изпълнява първото правило.

Пример. Нека са дадени следните правила:

```
[0-9]+
(\+|\-)?[0-9]+
(\+|\-)?[0-9]+"."[0-9]+
```

При разбор по тези правила за низ “123” ще се приложи първо правило, за низ “123.” ще се приложи трето правило.

При условие, че неможе да се приложи нито едно правило, то входния символ се копира в yyout. Ако това не е желателно, то в края на правилото се добавя низ:

```
. {/* Не прави нищо */}
\n { }
```

Секцията процедури от Lex-програмата съдържа произволен текст на език С, които се копира изцяло в изходния файл. Обикновено, тук се записва функция yywrap(), която определя какво се прави при достигане от автомата на края на входния файл. Ако върнатият резултат има ненулево стойност, то разбора с завършва, при нулева стойност се продължава анализа.

Интерпретаторът на таблици на крайни автомати се нарича yylex(). Автоматът завършва работа (връщане на стойност от функция yylex()), ако в някое от нейните действия се

изпълни оператор return или се достигне края на файла и стойността на yywrap() е различна от нула (резултата от yylex() е равен на нула).

Файлът, получен в резултат от работата на Lex, представлява C-програма, която съдържа таблици, описващи построения краен автомат, функции, реализиращи интерпретатор на автомата, описания на структурите от данни, използвани от интерпретатора, и потребителски програми.

Пример. Следващата Lex-програма преброява редовете, думите и символите във входната програма:

```
/***************** Секция за определения *********************/
   /* NODELIM бележи произволен символ, освен разделител */

NODELIM     [^" "\t\n]
            int l, w, c;    /* Число редове, думи, символи */

%% /******************** Секция за правила ***********************/

                { l=w=c=0;          /* Инициализация */      }
{NODELIM}+      { w++; c+=yyleng;   /* Дума*/              }
\n              { l++;              /* Редове */     }
.               { c++;              /* Символи */  }

%% /******************* Раздел за програмата *********************/

main() { yylex(); }

yywrap() {
  printf( " Lines - %d  Words - %d Chars - %d\n", l, w, c );
  return( 1 );
}
```

В секцията правила се използват следните специални конструкции и функции на Lex:

yytext	указател към низ от символи
yyleng	дължина на този низ
yyless(n)	връщане на последните n символа обратно във входния поток
yymore()	преброяване на символите в буфера yytext след текущия низ
input(c)	четене на връща символ
unput(c)	връща обратно символ във входния поток за повторно четене
output(c)	извеждане на символ c в изходния поток
ECHO	копира текущия символ в yyout

Пример. Следващата Lex-програма разпечатва всички думи от входния поток, които са пренесени:

```
NODELIM [^, :\.;\n\-\-]
HYPHEN  [\-\-]
%%
{NODELIM}+{HYPHEN}\n{NODELIM}+ { printf ("%s\n",yytext); }
{NODELIM}*                     { /* Незадължително */ }
. | \n                         { }
%%
```

```
yywrap() { return(1); }
main() { yylex(); }
```

В някои случаи е удобно описанието на необходимите действия да става чрез няколко различни състояния (различни крайни автомати) с явно преминаване от един на друг. В този случай имената на състоянията се поставят в специален ред `%Start`, а пред изразите се записва името на състоянието в ъглови скоби < и >. Преминаването в ново състояие се извършва с оператор `BEGIN`.

Пример. Lex-програма, която премахва коментарите от C-програма (`out` – извън коментар, `in` – вътре в коментар):

```
%Start out in
%%
                { BEGIN out; }
<out>"/*"       { BEGIN in;  }
<out>.|\n       { printf("%s",yytext); }
<in> "*/"       { BEGIN out; }
<in> .|\n       { }
%%
yywrap() { return(1); }
main() { yylex(); }
```

Извикването на Lex се извършва със следния ред:

```
lex <име_на_входен_файл>
```

Обикновено, изходният файл е с име `lexyy.c`.

3. Синтактически анализ.

3.1. Основни понятия и определения.

Лексическите елементи са линейни последователности, анализа на които е извършва от крайни автомати. Синтаксисът на лексическите единици не е линейна структура. Елементите от синтаксиса на езика (изрази, оператори) са единици, от които се построява програма, то взаимоотношенията между тези единици може да бъдат представени като дърво. С дървото са тясно свързани понятията рекурсия и стек. По този начин за определяне и анализ на синтаксиса на език е необходим математически апарат, който допуска рекурсивни определения и пораждания на своите елементи, а при анализа, използва дървета, рекурсивни функции и работа със стек. Формалните граматики са именно този математически апарат, който изследва свойствата на низовете от символи, породени от зададения набор правила.

Синтактическият анализ (разбор, разпознаване) е важна и необходима фаза от работата на всеки транслатор, а също и на много други системни и програмни продукти (например, някои системи с изкуствен интелект, програми за символично преобразуване на математически изрази и др.). Съответното програмно средство, което извършва синтактически анализ, се нарича синтактически анализатор.

Определение. Синтаксис, се нарича системата от правила за построяване на правилни езикови форми (конструкции) в един език.

Синтактическият анализ преследва две основни цели. Първата от тях е проверка на синтактичната коректност и откриване на синтактични грешки. Фазата на разпознаване се извършва за доказване, че анализираната входна лексема принадлежи или непринадлежи към множеството низове, пораждани от граматиката на езика. Изпълнението на разпознаване се осъществява с разпознаватели – автомати. Отговор „ДА" се дава, ако е установена такава принадлежност. В противен случай се дава отговор „НЕ". Полученият отговор „НЕ" е свързан с понятието отказ. При наличието на грешки синтактичният анализатор трябва да ги индицира, като е желателно да посочи точната позиция и брой.

Втората цел на синтактичния анализ, която е и по-важната, се състои в разкриване структурата на входния низ, т.е. построяване на синтактичното дърво. Дървото разкрива структурата на фразата и позволява, доколкото граматиката отразява това, да се разкрие и семантиката на фразата.

Определение. Нека е дадена граматиката $Г = \langle \Sigma, N, S, P \rangle$ и един низ $\alpha \in \Sigma^*$, за който се предполага, че е изречение в езика L(Г). Задачата на синтактичният анализ е да се построи синтактично дърво на пораждането $S \Rightarrow \alpha$ с корона, равна на низа α. Ако дървото несъществува се дава съобщение за грешка (случай $\alpha \notin L(Г)$.

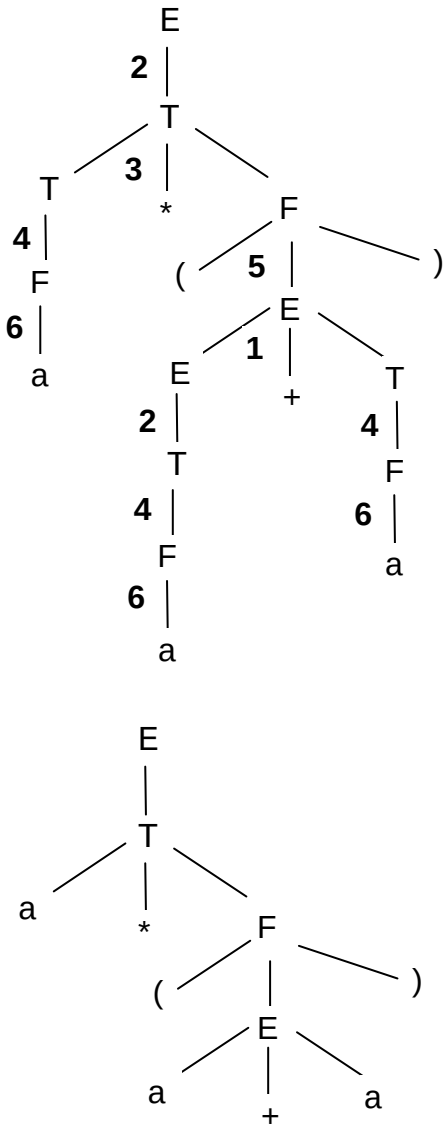
Неформалното обяснение на синтактическия анализ може да се формулира така: да се построи една дървовидна, знакова структура от входен низ, който първоначално е закодиран (изобразен) в линейна едномерна наредба от знакове. Дървовидната структура е пряко свързана със семантиката на езика и поради това не всяка граматика е подходяща за построяване на синтактично дърво. Например, при аритметичните изрази синтактичното дърво отразява реда на изчисление на изразите, съответно на изпълнение на действията.

Методите и алгоритмите за синтактичен анализ се отнасят до контекстно-свободните езици, тъй като езиците за програмиране са от този клас.

Пример. Нека е дадена граматика Г със следните номерирани правила:

1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to a$

Синтактическият разбор на израз a*(a+a) е следния:



Формата на задаване на синтактичното дърво може да бъде, например:

● последователност на правилата при ляв извод, т.н. ляв разбор. Левият разбор може да се получи при обхождане на синтактичното дърво отгоре надолу и отляво надясно.

● обратна последователност на правилата при десен извод, т.н. десен разбор.

Десният разбор може да се получи при обхождане на синтактичното дърво отдолу нагоре и отляво надясно.

Ляв разбор е прилагане на правилата по следната последователност: ЛР = 23465124646, а при десен разбор ДР = 64642641532.

Определение. Основа се нарича короната на най-лявото подддърво с дълбочина 1, която е равна на връх и преки потомци.

Определение. Отсичане на основите се нарича премахването на всички основи на дървото. Извършва се отдолу нагоре и отляво надясно.

На горната фигура е показано отсичането на основите на първоначалното синтактично дърво.

Два са основните методи за синтактичен анализ с използване на контекстно-свободните граматики: низходящи (отгоре надолу) и възходящи (отдолу нагоре).

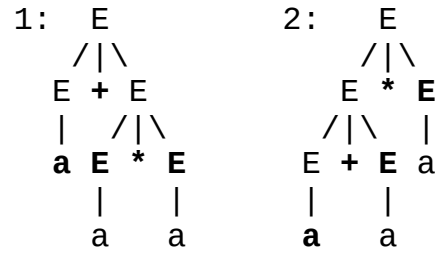
Низходящ разбор на дума α е прецесът на построяване на дърво на разбора за α, което започва с корен и на всяка стъпка в дървото се добавят нови потомци (наследници) на някои нетерминален възел.

Възходящ разбор на дума α е процесът на построяване на дърво на разбора за α, което започва с листата и на всяка крачка в дървото се добавя възел, наследниците на който са върхове на вече построените поддървета.

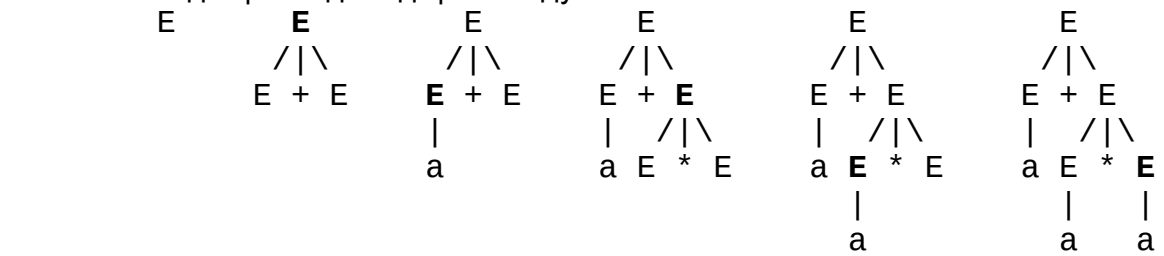
Пример. Нека е дадена КСГ Г = ⟨Σ, N, S, P⟩, която описва език на аритметическите изрази над променливата a:
Σ = {a, +, -, *, /, (,)},
N = {E, T, F},
S = E,
P = {E → E+E | E-E | E*E | E/E | (E) | a}
За думата a+a*a изводите могат да бъдат следните:

- произволен: $E \Rightarrow E*E \Rightarrow E*a \Rightarrow E+E*a \Rightarrow a+E*a \Rightarrow \mathbf{a+a*a}$
- ляв: $E \Rightarrow_l E+E \Rightarrow_l a+E \Rightarrow_l a+E*E \Rightarrow_l a+a*E \Rightarrow_l \mathbf{a+a*a}$
- десен: $E \Rightarrow_r E+E \Rightarrow_r E+E*E \Rightarrow_r E+E*a \Rightarrow_r E+a*a \Rightarrow_r \mathbf{a+a*a}$

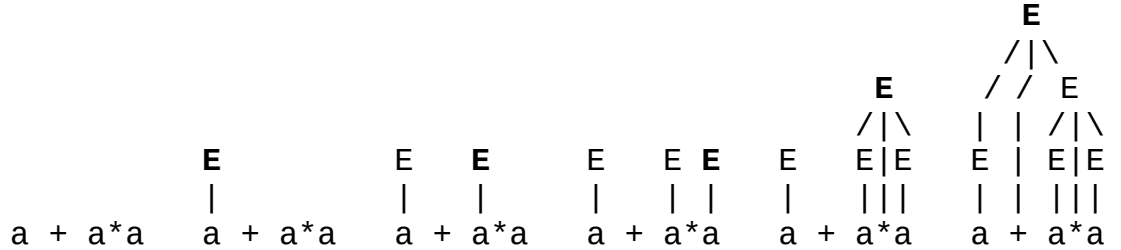
Две различни дървета на извод илюстрират нееднозначността на Г:



Низходящ извод на дърво за думата a+a*a:



Възходящ извод на дърво за думата a+a*a:



3.2. Привеждане (преработване) на граматиките

С помощта на контекстно-свободните граматики може да се определи по-голямата част от езиците за програмиране. При построяване на граматика, задаваща конструкции на език за програмиране, често се прибягва до нейното преобразуване, по такъв начин, че породеният език да придобие нужната структура.

3.2.1. Отстраняване на неизводимите символи

Символ $A \in \Sigma$ се нарича неизводим, ако от него неможе да бъде изведен краен терминален низ.

Разглеждайки правилата на дадена граматика, може да се направи извод, ако всички символи отдясно са изводими. Тогава изводими са и символите отляво на правилата. Последното позволява да се напише следната процедура за отстраняване на неизводимите символи:

Алгоритъм за отстраняване на неизводимите символи
Вход. Контекстно-свободна граматика $Г = \langle \Sigma, N, S, P \rangle$
Изход. Еквивалентна на Г контекстно-свободна граматика $Г' = \langle \Sigma, N', S, P' \rangle$, където N' е множество от изводими символи
Метод. Рекурсивно построяване на множеството $N_0, N_1, \ldots$.
(1) Положи $N_0 = \varnothing$, $i = 1$
(2) Положи $N_i = \{A \mid A \to \alpha \in P \text{ и } \alpha \in (N_{i-1} \cup T)^*\} \cup N_{i-1}$, $i = 1,\ldots$
(3) Ако $N_i \neq N_{i-1}$, то $i = i+1$ и преминаване към стъпка (2). Иначе, $N' = N_i$. Полагане на $Г' = \langle \Sigma, N', S, P' \rangle$, където P' се състои от правилата на множеството P, съдържащо само символи $N' \cup \Sigma$.

Пример. Нека е дадена граматика Г със следните правила: $P = \{S \to aSa, S \to bAd, S \to c, A \to cBd, A \to aAd, B \to dAf\}$

Неизводими са нетерминалите A и B. След отстраняване на правилата, съдържащи неизводими нетерминали, се получават следните правила: $P = \{S \to aSa, S \to c\}$.

3.2.2. Отстраняване на недостижимите символи

Символът $x \in (\Sigma \cup N)$ се нарича недостижим в граматиката $Г = \langle \Sigma, N, S, P \rangle$, ако той не се появява в нито един изводим низ.

Алгоритъм за отстраняване на недостижимите символи
Вход. Контекстно-свободна граматика $Г = \langle \Sigma, N, S, P \rangle$
Изход. Еквивалентна на Г контекстно-свободна граматика $Г' = \langle \Sigma', N', S, P' \rangle$ без недостижими символи
Метод. Рекурсивно построяване на множеството $V_0, V_1, \ldots$
(1) Положи $V_0 = \{S\}$, $i = 1$. (Образуване на едноелементен списък, състоящ се от началния символ.)
(2) Положи $V_i = \{x \mid x \in (\Sigma \cup N)$, в P има $A \to \alpha x \beta$ и $A \in V_{i-1}'$, $\alpha, \beta \in (\Sigma \cup N)^*\} \cup V_{i-1}$. (Ако е намерено правило, чиято лява част я има вече в списъка, то в списъка се включват всички символи, съдържащи се в неговате дясна част.)
(3) Ако $V_i \neq V_{i-1}$, то $i = i+1$ и преминаване към стъпка (2). Иначе, нека $N' = V_i \cap N$, $\Sigma' = V_i \cap \Sigma$. P' се състои от правилата на множеството P, съдържащи само символите от V_i. (Ако на

стъпка (2) не се добавят нови нетерминали в списъка, то е получен списък от всички достижими нетерминали, а нетерминалите, които не са попаднали в списъка, са недостижими.)

Пример. В граматиката с правила $P = \{S \to aSb,\ S \to c,\ A \to bS,\ A \to a\}$ A е недостижим символ.

3.2.3. Отстраняване на безполезните символи

Символът $x \in (\Sigma \cup N)$ се нарича безполезен в граматиката $\Gamma = \langle \Sigma, N, S, P \rangle$, ако той е недостижим и неизводим.

Алгоритъм за отстраняване на безполезните символи
Вход. Контекстно-свободна граматика $\Gamma = \langle \Sigma, N, S, P \rangle$
Изход. Еквивалентна на Γ контекстно-свободна граматика $\Gamma' = \langle \Sigma', N', S, P' \rangle$, която няма безполезни символи
Метод.
(1) Прилагане към Γ на алгоритъма за отстраняване на неизводимите символи и получаване на еквивалентна граматика $\Gamma_1 = \langle \Sigma, N_1, S, P_1 \rangle$.
(2) Прилагане към Γ_1 на алгоритъма за отстраняване на недостижимите символи и получаване на еквивалентна граматика $\Gamma' = \langle \Sigma', N', S, P' \rangle$.

Пример. Нека е дадена граматика Γ със следните правила: $P = \{S \to ac,\ A \to cBC,\ B \to aSA,\ C \to bc,\ C \to d\}$. Неизводими са нетерминалите A и B, затова се изключват правилата, които ги съдържат. Получава се граматика Γ_1 със следните правила: $P_1 = \{S \to ac,\ C \to bc,\ C \to d\}$.
В получените правила C е недостижим символ. След изключване на съответните правила, съдържащи този символ се получава граматика Γ' със следното правило: $P' = \{S \to ac\}$.

3.2.4. Отстраняване на ε-правила

Правилата от вида $\alpha \to \varepsilon$ се наричат ε-правила (епсилон правила, анулиращи правила).

Граматиката Γ е граматика без анулиращи правила, ако или несъдържа анулиращи правила или съдържа само едно правило от вида $S \to \varepsilon$, където S е началният символ на граматиката и символ S не се среща в десните части на останалите правила от граматиката.

Теорема. Нека L е контекстно-свободен език. Тогава езикът $L - \{\varepsilon\}$ се поражда от контекстно-свободната граматика L без ε-правила.

Доказателство. Нека е дадена контекстно-свободната граматика $\Gamma = \langle \Sigma, N, S, P \rangle$, пораждаща език L. Ако за всяко $A \in N$, $B \in N$, $\alpha \in (\Sigma \cup N)^*$ и $\beta \in (\Sigma \cup N)^*$ множеството P съдържа правила $B \to \alpha A \beta$ и $A \to \varepsilon$, но несъдържа правила $B \to \alpha\beta$, то такова правило се добавя в P. Тази процедура се добавя, докато е възможно. След това от множеството P се изключват всички правила от вида $A \to \varepsilon$. Получената граматика поражда език $L - \{\varepsilon\}$.

Пример. Нека е даден език L, пораждащ граматиката $S \to \varepsilon$, $S \to aSbS$. Езикът $L - \{\varepsilon\}$ се поражда от граматиката $S \to aSbS$, $S \to abS$, $S \to aSb$, $S \to ab$.

Теорема. За всяка контекстно-свободна граматика Г, съдържаща анулиращи правила, може да се построи еквивалентна контекстно-свободна граматика Г, такава че L(Г) = L(Г').

Алгоритъм за отстраняване на ε-правила
Вход. Контекстно-свободна граматика $Г = \langle \Sigma, N, S, P \rangle$,
Изход. Еквивалентна на Г контекстно-свободна граматика $Г' = \langle \Sigma, N', S', P' \rangle$, която няма ε-правила
(1) Построяване на множеството $N_\varepsilon = \{A \mid A \in N \text{ и } A \Rightarrow_Г^+ \varepsilon\}$ по аналогия с алгоритмите за отстраняване на неизводимите и недостижимите символи.
(2) Построяване на P' така, че

a. ако $A \to \alpha_0 B_1 \alpha_1 B_2 \alpha_2 B_3 ... B_k \alpha_k \in P$, $k \geq 0$ и $B_i \in N_\varepsilon$ за $1 \leq i \leq k$, но нито един символ в низовете α_j $(0 \leq j \leq k)$ непринадлежи на N_ε ($\alpha_j \notin N_\varepsilon$), то в P' се включват всички правила от вида $A \to \alpha_0 X_1 \alpha_1 X_2 \alpha_2 B_3 ... X_k \alpha_k$, където X_i е или B_i или ε, но не се включва правилото $A \to \varepsilon$ (това може да се случи, ако всички α_i са равни на ε. В множеството P' се добавят правилата, чиито десни части не съдържат символи от множеството N_ε.

b. ако $S \in N_\varepsilon$, то в P' се включват правилата $S' \to \varepsilon \mid S$, където S' е нов символ $S' \notin N$, и се полага $N' = N \cup \{S'\}$. В противен случай, се полага N' = N и S' = S.

(3) Полага се $Г' = \langle \Sigma, N', S', P' \rangle$.

3.2.5. Отстраняване на верижните правила

Правила от граматика $Г = \langle \Sigma, N, S, P \rangle$, във вида $A \to B$, където $A, B \in \Sigma$, се наричат верижни правила.

Алгоритъм за отстраняване на верижните правила
Вход. Контекстно-свободна граматика $Г = \langle \Sigma, N, S, P \rangle$ без ε-правила
Изход. Еквивалентна контекстно-свободна граматика $Г' = \langle \Sigma, N, S, P' \rangle$ без ε-правила и без верижни правила.
(1) За всяко $A \in N$ се построява $N_A = \{B \mid A \Rightarrow_Г^* B\}$ по следния начи:

a. полага се $N_0 = \{A\}$ и i = 1

b. полага се $N_i = \{C \mid B \to C \in P \text{ и } B \in N_{i-1}, C \in N\} \cup N_{i-1}$, където i = 1, 2,... .

c. ако $N_i \neq N_{i-1}$, полага се i = i + 1 и се повтаря стъпка (1.b). Иначе, се полага $N_A = N_i$.

(2) Построява се P' така че, ако $B \to \alpha \in P$ и не е верижно правило, то в P' се включва правилото $A \to \alpha$, за всички A, такива че $B \in N_A$.

3.3. Неоднозначност на граматика. Отстраняване на неоднозначност

3.3.1. Основни определения.

Контекстно-свободната граматиката се нарича неоднозначна, ако съществува дума, генерирана от граматиката, за която може да се построи повече от едно синтактическо дърво. В противен случай контекстно-свободната граматиката е еднозначна.

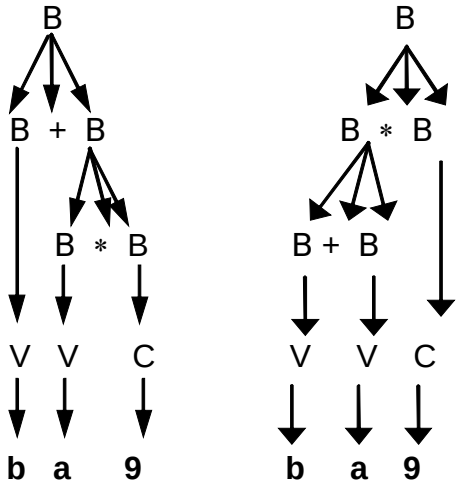
Пример. Нека е дадена контекстно-свободна граматика със следните правила за извод:

$B \to B + B \mid B * B \mid V \mid C$
$V \to a \mid b \mid c \mid ... \mid x \mid y \mid z$
$C \to 0 \mid 1 \mid 2 \mid ... \mid 8 \mid 9$

и дума b+a*9, генерирана от граматиката. За нея може да се напишат следните два различни извода:

$B \Rightarrow B+B \Rightarrow B+B*B \Rightarrow V+B*B \Rightarrow V+V*B \Rightarrow V+V*C \Rightarrow b+V*C \Rightarrow b+a*C \Rightarrow$ **b+a*9**
$B \Rightarrow B+B \Rightarrow B+B*B \Rightarrow B+B*C \Rightarrow V+V*9 \Rightarrow B+V*9 \Rightarrow B+a*C \Rightarrow V+a*C \Rightarrow$ **b+a*9**,

по които да се построят две различни синтактични дървета:

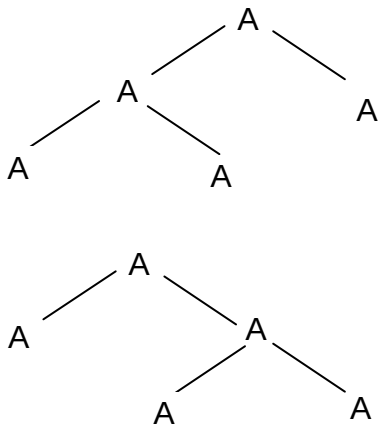


Неоднозначността е свойство на граматиката, а не на езика, т.е. за някои неоднозначни граматики съществува еквивалентна еднозначна граматика. Ако граматиката се използва за определяне на език за програмиране, то я трябва да е еднозначна.

Една контекстно-свободна граматика се нарича съществено неоднозначна, ако несъществува друга еднозначна контекстно-свободна граматика, която поражда същия език.

Теорема. Проблемът за определяне дали дадена контекстно-свободна граматика е еднозначна или не е неразрешим, т.е. не съществува алгоритъм, който да дава отговор ДА/НЕ за произволна граматика.
Неоднозначността може да се премахне чрез въвеждане на допълнително правило.

Пример. Ако граматика съдържаща правила $A \rightarrow AA \mid a$ е неоднозначна, тъй като подниз AAA допуска два различни разбора:



Тази неоднозначност може да се премахне, ако вместо $A \rightarrow AA \mid \alpha$ се въведат правилата
$A \rightarrow AB \mid B$
$B \rightarrow \alpha$
или правилата
$A \rightarrow BA \mid B$
$B \rightarrow \alpha$

Друг пример за неоднозначност е правилото $A \rightarrow AαA$. Двойката правила $A \rightarrow αA \mid Aβ$ също създават неоднозначност, така както дума $αAβ$ има два различни леви извода: $A \Rightarrow αA \Rightarrow αAβ$ и $A \Rightarrow Aβ \Rightarrow αAβ$.

Проблемът за неразрешимостта на неоднозначността е само за контекстно-свободните граматики. Автоматните граматики са еднозначни.

Съществуват някои признаци, по които може да се разпознае неоднозначната граматика, а именно:

a. двойната рекурсия по терминален символ. Например, $A \rightarrow AαA$, където $A \in N$, $α \in (Σ \cup N)^*$.

b. наличието в граматиката на ляволинейни и дяснолинейни правила за извод едновременно. Например, $A \rightarrow αA \mid Aβ$, където $A \in N$, $α, β \in (Σ \cup N)^+$.

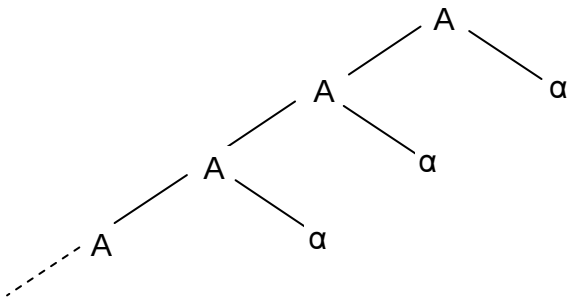
c. наличие в граматиката на ляволинейни (дяснолинейни) правила за извод заедно с правила, имащи централна рекурсия по терминален символ. Например, $A \rightarrow αA \mid αAβ$ или $A \rightarrow Aβ \mid αAβ$, където $A \in N$, $α, β \in (Σ \cup N)^+$.

3.3.2. Отстраняване на лява рекурсия

Г е ляворекурсивна граматика, ако съдържа нетерминал А позволяващ извод $A \Rightarrow^+ Aα$.

Г е непосредствена ляворекурсивна граматика, ако съдържа нетерминал А позволяващ правилото $A \rightarrow Aα$

Анализите отгоре надолу не са в състояние да работят с ляворекурсивни граматики, затова е необходимо преобразуване на граматиката, което да отстрани лявата рекурсия. Тъй като дървото се строи отляво надясно и отгоре надолу е възможен случай когато правилото $A \rightarrow Aα$ се повтаря до безкрайност без да се извежда терминален низ:



Елиминирането на непосредствена лява рекурсия на правила $A \rightarrow Aα \mid β$ се извършва чрез заместването им с неляворекурсивните правила:

$A \rightarrow βA'$

$A' \rightarrow αA' \mid ε$, където α и β са низове от терминални и нетерминални символи, незапочващи с А.

Пример. Нека е дадена граматика за аритметически изрази със следните правила:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Отстранявайки непосредствената лява рекурсия при Е и Т се получава следното:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

Отстраняването на лявата рекурсия се извършва по следната схема: Нека нетерминалният символ A има m ляворекурсивни правила $A \rightarrow A\alpha_i$, за $1 \leq i \leq m$ и n правила $A \rightarrow \beta_j$, за $1 \leq j \leq n$, които не са ляворекурсивни. Тази граматика генерира думи от вида $\beta_j\alpha_{i1}\alpha_{i2}...\alpha_{ir}$, където j = 1,...,n; i1, i2,...,in = 1,...,m. Правилата се заместват със следните нови правила:
$A \rightarrow \beta_1A' \mid \beta_2A' \mid...\mid\beta_nA'$
$A' \rightarrow \alpha_1A' \mid \alpha_2A' \mid...\mid\alpha_mA' \mid \varepsilon$

Граматиката поражда същите думи, но без лява рекурсия. Тази процедура отстранява всички непосредствени леви рекурсии (при условие, че нито един низ α_i не е ε), но не отстранява лява рекурсия пораждана от две и повече стъпки. Следващият алгоритъм елиминира лявата рекурсия. Той работи с граматики, които нямат цикли (пораждани от $A \Rightarrow^+ A$) и ε-правила ($A \rightarrow \varepsilon$).

Алгоритъм за отстраняване на лява рекурсия
Вход. Граматика без цикли и ε-правила.
Изход. Еквивалентна граматика без лява рекурсия.
(1.) Подреждане на нетерминалните символи: $A_1, A_2, ..., A_n$
(2.) for (i = 1; i <= n; i++){
    for (j = 1; j <= i-1; j++){
        Замести всяка продукция от вида $A_i \rightarrow A_i\gamma$
        с продукциите от вида $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid...\mid\delta_k\gamma$,
        където $A_j \rightarrow \delta_1 \mid \delta_2 \mid...\mid\delta_k$ са всички правила A_j
    }
    Елиминирай непосредствените леви рекурсии
    в правилата за A_i
}

Работоспособността на алгоритъма се основава на това, че след (i-1)-та итерация на вътрешния for произволна продукция от вида $A_k \rightarrow A_l\alpha$, k < l, трябва да бъде l > k. В резултат от това при следващата итерация на вътрешния цикъл (по j) ще нараства долната граница m за всички продукции от вида $A_i \rightarrow A_m\alpha$ до тогава докато не бъде достигнато m ≥ i. Затова от всички A_i продукции се отстраняват непосредствените леви рекурсии, вследствие на което се достига m > i.

Пример. Нека е дадена граматика със следните правила:
$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \varepsilon$
i = 1 Няма лява рекурсия
i = 2 $A \rightarrow Sd$ се замества чрез
    $A \rightarrow Aad \mid bd$.
Откъдето следва, че се получава следната A-продукция: $A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$.
Елиминирането на непосредствената лява рекурсия:
$A \rightarrow bdA' \mid A'$
$A' \rightarrow adA' \mid cA' \mid \varepsilon$

Правилата получената граматика след отстраняване на лявата рекурсия са следните:
$S \rightarrow Aa \mid b$
$A \rightarrow bdA' \mid A'$
$A' \rightarrow adA' \mid cA' \mid \varepsilon$
В този случай ε-правилото не усложнява отстраняването на лявата рекурсия.

3.3.3. Лява факторизация

Основната идея на лявата факторизация е в това, че когато не е ясно, коя от няколко алтернативи да се използва за развиване на нетерминал A, е нужно да се преобразуват A-правилата така, че да се отложи решението до тогава, докато не не се набере достатъчно информация, че да се вземе правилното решение. Лявата факторизация се нарича още извеждане на ляв множител.

Алгоритъм за лява факторизация
Вход. Граматика Г.
Изход. Еквивалентна ляво факторизирана граматика.
(1) За всеки нетерминал A се намира най-дългия префикс α, общ за две или повече алтернативи. Ако $\alpha \neq \varepsilon$, т.е. съществува нетривиален общ префикс, замени всички правила $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n \mid \gamma$, където γ е алтернатива незапочваща с α, с правилата: $A \rightarrow \alpha A' \mid \gamma$ и $A' \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_n$, където A' е новият нетерминал.
(2) Повтаряй (1) докато се отстранят алтернативите с еднакъв префикс.

Пример. Нека е дадена граматика Г на условен оператор със следните правила:
Statm $\rightarrow$ if Exprs then Statm
Statm $\rightarrow$ if Exprs then Statm else Statm
Statm $\rightarrow$ a
Exprs $\rightarrow$ b

След прилагане на лява факторизация се получава следната еквивалентна граматиката Г':
Statm $\rightarrow$ if Exprs then Statm Statm'
Statm $\rightarrow$ a
Statm' $\rightarrow$ else Statm
Statm' $\rightarrow$ ε
Exprs $\rightarrow$ b
Г и Г' са неоднозначни граматики.

3.4. Стекови автомати

3.4.1. Основни определения.

Определение. Стеков автомат или стек, се нарича седморката $M = \langle Q, \Sigma, Z, \delta, q_0, z_0, F \rangle$, където

Q – крайно множество символи на състоянията, представляващи всевъзможните състояния на управляващото устройство;

Σ – крайно множество от допустими входни символи, наречено входна азбука;

Z – крайно множество от стекови символи;

δ – функция на преходите, която е изображение на множеството $Q \times (\Sigma \cup \{\varepsilon\}) \times Z$ в множеството от крайни подмножества $Q \times Z^*$, т.е. $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times Z \rightarrow \{Q \times Z^*\}$;

$q_0 \in Q$ – начално състояние на управляващото устройство;

$z_0 \in Z$ – начален символ, намиращ се в стека в началния момент;

$F \subseteq Q$ – множество от заключителните състояния.

Стековият автомат представлява недетерминиран краен автомат, към който е добавена потенциално безкрайна външна памет, чиито елементи на запомнена информация са подредени последователно и за автомата е достъпна само информацията от първия (или най-горния) елемент. Този елемент може да се изтрива от паметта и на негово място да се добавят нови елементи на запомнена информация, като отново достъпен ще бъде само първият от новите елементи.

Определение. Конфигурация на стековия автомат се нарича тройката $(q, \alpha, \omega) \in Q \times \Sigma^* \times Z^*$, където

q – текущо състояние на управляващото устройство;

α – неизползвана част от входния низ; първият символ a (текущ символ) на низа се намира под входната глава; ако $\alpha = \varepsilon$, то входният низ е прочетен;

ω – наличното в стека; най-левият символ от низ ω се счита за връх на стека; ако $\omega = \varepsilon$, то стека се счита за празен (ω – омега).

3.4.2. Работа на стековия автомат.

Тактът на работа на стековия автомат се представя във вид на бинарно отношение, определено от конфигурацията. Записва се

$(q, a\alpha, z\omega) \rightarrow (q', \gamma\alpha, \omega)$

ако множеството $\delta(q, a, z)$ съдържа (q', γ), където $q, q' \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $\alpha \in \Sigma^*$, $z \in Z$ и $\omega, \gamma \in Z^*$.

Начална конфигурация на стековия автомат се нарича конфигурацията от вида (q_0, α, z_0), където $\alpha \in \Sigma^*$, т.е. управляващото устройство се намира в начално състояние, входната лента съдържа низ, който трябва да се разпознае, а в стека има само начален символ z_0.

Заключителна конфигурация – това е конфигурация от вида (q, ε, ω), където $q \in F$, $\omega \in Z^*$. Стековият автомат разпознава дума α чрез заключително състояние, ако $(q_0, \alpha, z_0) \rightarrow^* (q, \varepsilon, \omega)$, за $q \in F$, $\omega \in Z^*$.

Ако завършващата конфигурация е $(q, \varepsilon, \varepsilon)$, то стековият автомат разпознава входната дума α чрез празен стек.

3.4.3. Език разпознаван от стеков автомат.

Множеството L(M) от всички входни думи, които се разпознават от стеков автомат M чрез заключително състояние, се нарича език, разпознаван от M чрез заключително състояние.

Множеството L(M) от всички входни думи, които се разпознават от стеков автомат M чрез празен стек, се нарича език, разпознаван от M чрез празен стек.

Функцията на преходите δ се определя по следния начин:

a. $\delta(q, a, z) = \{(q_1, \omega_1), \ldots, (q_m, \omega_m\}$, където $q, q_1, \ldots, q_m \in Q$, $a \in \Sigma$, $z \in Z$, $\omega_1, \ldots, \omega_m \in Z^*$ означава, че стековият автомат в състояние q и с първи стеков символ z **прочита входния символ a**, чрез функцията на преходите δ определя възможните варианти

$q_1,..., q_m$ за ново състояние и възможните варианти $\omega_1,...,\omega_m$ за стекова дума, с която да замени z и след това преминава към следващия отдясно входен символ.

b. $\delta(q, \varepsilon, z) = \{(q_1, \omega_1),...,(q_m, \omega_m\}$, където $q, q_1,..., q_m \in Q$, $z \in Z$, $\omega_1,...,\omega_m \in Z^*$, ε е празната дума, означава, че стековият автомат в зависимост от вътрешното си състояние q и първия стеков символ z, **без да прочита входен символ**, определя чрез функцията на преходите δ възможните варианти $q_1,..., q_m$ за ново състояние и възможните варианти $\omega_1,...,\omega_m$ за стекова дума, с която да замени z и след това не се придвижва надясно, а остава на същото място. Това действие се нарича ε-действие.

Пример. Стековият автомат разпознаващ език $L = \{c^n b^n \mid n = 1, 2,...\}$ има следните компоненти: $\Sigma = \{c, b\}$, $Q = \{q_0, q_1, q_2, q_3\} \cup \{error\}$,….

1.rtf
Langs51

3.4.4. Построяване на стеков автомат.

Теорема. Ако $\Gamma = \langle \Sigma, N, S, P \rangle$ е контекстно-свободна граматика, то по нея може да се построи такъв стеков автомат M, че $L(M) = L(\Gamma)$.
Доказателство. tfyav11.pdf
Пример.

3.4.5. Детерминирани и недетерминирани стекови автомати

Определение. Стековият автомат M се нарича детерминиран, ако са изпълнени следните условия:

a. за всяка тройка (q,a,z), $a \in \Sigma$ има неповече от едно правило,
b. за всяка тройка (q,ε,z) има неповече от едно правило,
c. ако има правило за (q,a,z), то няма правила за (q,ε,z),

иначе стековият автомат се нарича недетерминиран.

Ако език L се разпознава от детерминиран стеков автомат, то L се нарича детерминиран език.

3.5. Низходящ анализ

3.5.1. Основни понятия.

Нека са дадени контекстно-свободната граматика $\Gamma = \langle \Sigma, N, S, P \rangle$ и подлежащ на анализ низ $\alpha \in \Sigma^*$. Казва се, че е извършен низходящ синтактичен анализ (отгоре-надолу, top-down), ако е построена редицата от непосредствни пораждания, в реда определен по следния начин (отляво-надясно): $S \Rightarrow \varphi_1 \Rightarrow \varphi_2 \Rightarrow ... \Rightarrow \varphi_k = \alpha$ и ако тя съществува, в противен случай анализът завършва със съобщение за грешка.

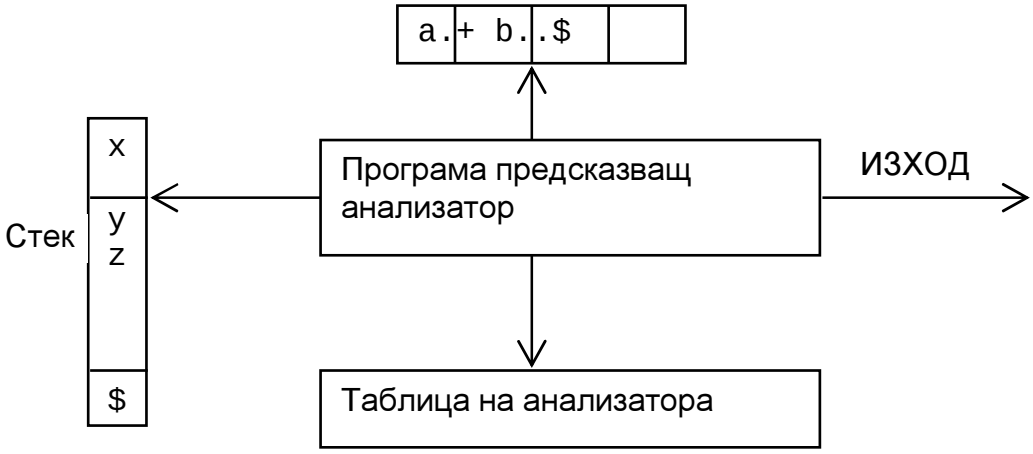
Сентенциалната форма φ_i трябва да бъде получена след φ_{i-1} в резултат на прилагане на продукция от P. Така определеният анализ е равносилен на построяването на синтактичното дърво, което се строи отгоре-надолу, като се започва от корена и на всяка стъпка се разширява с един нетерминален връх-листо A, като на този връх се построява фрагмент, съответен на α в продукцията $A \rightarrow \alpha$.

При анализа и строежа на дървото се използва ляв каноничен разбор. Това означава, че се взема винаги най-левия нетерминален символ и прилагайки правилата на граматиката се стреми да се получи част от входната редица.

3.5.2. Таблично-управляем предсказващ анализ.

Основният проблем на предсказващия анализ е определянето на правилата за извод, които трябва да се приложат към нетерминала.

Таблично-управляемият предсказващ анализатор има входен буфер, таблица на анализ и изход. Входният буфер съдържа низ за разпознаване, последван от $ - маркер за край на низа. Стеков автомат съдържа последователност от символи от граматиката с $ на дъното. Таблицата на анализа е двумерен масив M[A,a], където A е нетерминал, a е терминал или символ $.



Действието на предсказващият анализ се определя от x – символ на върха на стека и a – текущ входен символ. Съществуват три възможности:
(1) Ако x = a = $, анализаторът спира работа и съобщава за успешно завършване на разпознаването.
(2) Ако x = a ≠ $, анализаторът отстранява x от стека и придвижва указателя на входа на следващия входен символ.
(3) Ако x е нетерминал, се поглежда в таблицата за стойност M[x,a]. На този адрес се съхранява или правило за x или грешка. Например, ако M[x,a] = {x → uvw}, анализаторът заменя x от върха на стека с wvu (u е на върха). Като изход анализаторът може да разпечатва използваните правила за извод. Ако M[x,a] = error, анализаторът се обръща към подпрограма за обработка на грешката.

В термините на конфигурацията на автомата на разбор действието на анализатора се описва по следния начин:

Алгоритъм за нерекурсивен предсказващ анализ
Вход. Низ α и таблица на анализа M за граматика Г.
Изход. Ако w ∈ L(Г) – ляв извод на w, иначе – съобщение за грешка
Метод. В началото анализаторът се намира в конфигурация, в която стекът съдържа S$ (S – фиксиран начален символ от граматиката Г), във входния буфер се намира α$ (α – входен низ).

InSymbol = първи символ от w$
do

```
{x=символ на върха на стека;
 if (x ∈ Σ || x == '$')
        if (X == InSymbol)
                {отстраняване x от стека;
                 InSymbol = пореден символ;
                }
        else error();
else /*x - нетерминал*/
        if (M[x,InSymbol]== "x → y1y2...yk")
                {отстраняване x от стека;
                 поставяне ykyk-1...y1 в стека (y1 е на върха);
                 разпечатва се правилото x → y1y2...yk;
                }
        else error(); /*няма вход от таблицата*/
}
while (x!=$) /*празен стек*/
```

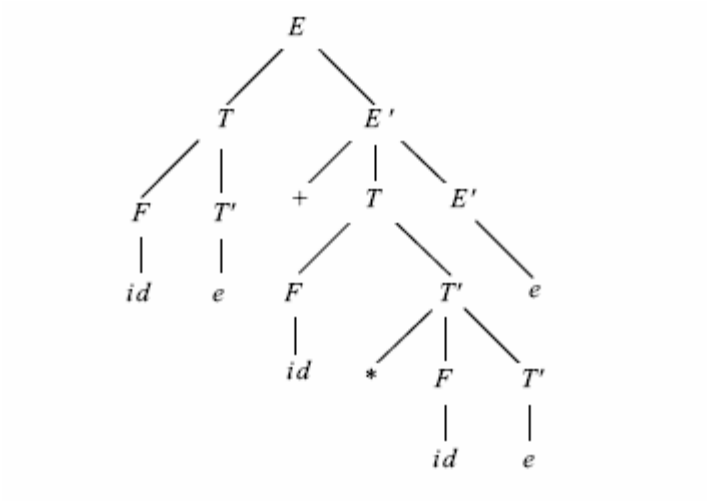
Пример. Нека е дадена граматика за аритметически изрази със следните правила:
P = {E → TE', E' → +TE' | ε, T → FT', T' → *FT' | ε, F → (E) | id}.
Таблицата на предсказващия анализатор е следната:

Нетер-	Входен символ					
минал	id	+	*	(	)	$
E	E->TE'			E->TE'		
E'		E'->+TE'			E'->e	E'->e
T	T->FT'			T->FT'		
T'		T'->e	T'->*FT'		T'->e	T'->e
F	F->id			F->(E)		

Четейки входен низ id+id*id$, с помощта на стек, предсказващият анализатор извършва следния ляв извод:

Стек	Вход	Изход
$E	id+id*id$	
$E'T	id+id*id$	E->TE'
$E'T'F	id+id*id$	T->FT'
$E'T'id	id+id*id$	F->id
$E'T'	+id*id$	
$E'	+id*id$	T'->e
$E'T+	+id*id$	E'->+TE`
$E'T	id*id$	
$E'T'F	id*id$	T->FT'
$E'T'id	id*id$	F->id
$E'T'	*id$	
$E'T'F*	*id$	T'->*FT'
$E'T'F	id$	
$E'T'id	id$	F->id
$E'T'	$	
$E'	$	T'->e
$	$	E'->e

На левия извод съответства следното дървото на разбора.



3.5.3. Множества First и Follow

Две функции, свързани с граматиката Г, се оказват полезни при построяване на предсказващ анализатор. Това са функциите First и Follow, които позволяват построяване на предсказваща разбора таблица за Г, ако това е възможно. Същите множества може да се използват при възстановяване след грешка.

Ако α е произволен низ от символи на граматиката, множеството First(α) включва всички терминални символи, с които може да започват всички низове, породени от α. Ако $\alpha \Rightarrow^* \varepsilon$, то $\varepsilon \in$ First(α).

За нетерминален символ A, множеството Follow(A) включва всички терминални символи a, такива че съществуват изводи от вида $S \Rightarrow^* \alpha A a \beta$, където $\alpha,\beta \in (\Sigma \cup N)^*$. Ако A е най-крайният десен символ в някоя сентенциална форма, то \$ ∈ Follow(A).

Построяване на множеството First(x) за всички символи от граматиката:
(1) Ако x – терминал, то First(x) = {x}.
(2) Ако има продукция $x \rightarrow \varepsilon$, то ε се добавя към First(x).
(3) Ако x – нетерминал и има продукция $x \rightarrow y_1 y_2 ... y_k$, то a се добавя към First(x), ако за някое i | $a \in$ First(y_i) и ε се добавя към всички множества First(y_1),...,First(y_{i-1}), т.е. $y_1...y_{i-1} \Rightarrow^* \varepsilon$. Ако ε се съдържа във всички First(y_i), i = 1,...,k, то се добавя ε към First(x).

Например, всички символи, които принадлежат на First(y_1) принадлежат и на First(x). Ако от y_1 не се извежда ε, то нищо не се добавя към First(x), но ако $y_1 \Rightarrow^* \varepsilon$, то към First(x). се добавя First(y_2).

Построяване на множеството First($x_1x_2...x_n$) за произволен низ $x_1x_2...x_n$:
(1) First($x_1x_2...x_n$) = {}.
(2) Към First($x_1x_2...x_n$) се добавя всички не-ε символи от First(x_1). Добавят се и не-ε символите от First(x_2), ако $\varepsilon \in$ First(x_1), добавят всички не-ε символи от First(x_3), ако $\varepsilon \in$ First(x_1) и $\varepsilon \in$ First(x_2) и т.н. Накрая, се добавя ε към First($x_1x_2...x_n$), ако всяко i | $\varepsilon \in$ First(x_i).

За изчисляване на Follow(A) за всички нетерминали A се прилагат следните стъпки:
(1) Follow(A) = {}.
(2) Добавяне на \$ в множеството Follow(S), където S – начален символ и \$ - десен маркер за край.

(3) Ако има правило за извод $A \to \alpha B \beta$, то цялото множество First(β), с изключение на ε, се добавя към Follow(B).
(4) Докато нищо не е добавено към множествата Follow: ако има продукция $A \to \alpha B$ или $A \to \alpha B \beta$, където First(β) съдържа ε (т.е. $\beta \Rightarrow^* \varepsilon$), то всички елементи от множеството Follow(A) се добавят към Follow(B).

Пример. За разгледаната по-горе граматика с правила:
P = {E → TE', E' → +TE' | ε, T → FT', T' → *FT' | ε, F → (E) | id}
множествата First и Follow са следните
First(E) = First(T)=First(F)={(,id}
First(E')={+,e}
First(T')={*,e}
Follow(E)=Follow(E')={),$}
Follow(T)=Follow(T')={+,),$}
Follow(F)={+,*,),$}

id и лява скоба се добавят към First(F) на стъпка (3) при i=1, докато First(id)={id} и First('(')={'('} в съответствие със стъпка (1). На стъпка (3) при i=1, в съответствие с правило T → FT' към First(T) се добавя също id и лява скоба. На стъпка (2) към First(E') се включва ε.

На стъпка (1) за изчисляване на множествата Follow в множество Follow(E) се включва $. На стъпка (2), използвайки правилото за извод F → (E), към Follow(E) се добавя дясна скоба. На стъпка 3, прилагайки правилото E → TE', в множеството Follow(E') се включва $ и дясна скоба. Тъй като $E' \Rightarrow^* \varepsilon$, тези символи се добавят и към Follow(T). От продукцията E → TE', съгласно правило (2), всички елементи от множество First(E') (с изключение на ε) трябва присъстват и в множеството Follow(T). В това множество влиза и $.

3.5.4. Построяване на таблица за предсказващ анализ

За построяване на таблица за предсказващ анализ се използва следващата идея. Нека се предположи, че $A \to \alpha$ е такава продукция, че $a \in$ First(α). Тогава синтактическият анализатор заменя символ A с низ α при текущ вход на символ a. Сложността възниква при $\alpha = \varepsilon$ или $\alpha \Rightarrow^* \varepsilon$. В този случай A се заменя с α, ако текущия входен символ се съдържа в Follow(A) или от входния поток е получен $, който се включва в Follow(A).

Алгоритъм за построяване на таблица на предсказващия анализ
Вход. Граматика Г.
Изход. Таблица на анализ M.
(1) За всяка продукция $A \to \alpha$ от граматиката се изпълняват стъпки (2) и (3).
(2). За всеки терминал $a \in$ First(α) се добавя $A \to \alpha$ на адрес M[A,a].
(3). Ако в множеството First(α) е добавен ε, за всеки терминален символ $b \in$ Follow(A) се добавя $A \to \alpha$ на адрес M[A,b]. Ако ε е добавен в множеството First(α), а $ в множеството Follow(A), то $A \to \alpha$ се поставя на адрес M[A,$].
(4) На всеки неопределен елемент от таблица M се указва адрес за обработка на грешки.

Пример. Нека алгоритъмът се приложи към граматиката от горния пример. Тъй като First(TE')=First(T)={(,id}, продукцията $E \to TE'$ се записва в M[E,(] и M[E,id]. Продукцията $E' \to +TE'$ се записва в M[E',+]. Тъй като продукцията Follow(E')={),$}, то $E' \to \varepsilon$ се записва в M[E',)] и M[E',$].

3.5.5. LL(1) граматики

Когато използваната граматика е ляворекурсивна или неоднозначна, то в един елемент от таблицата на предсказващия анализ M трябва да има по няколко записа – множествени записи.

LL(1) граматика, се нарича тази граматика, таблицата на предсказващия анализ на която няма множествени записи. Първото L (left), означава сканиране отляво надясно, а второто L, че се строи ляв извод, а 1, че на всяка стъпка за вземане на решение се разглежда само един символ от входния поток.

LL(1) имат няколко отличителни свойства. Неоднозначна или ляворекурсивна граматика не може да бъде LL(1). Възможно е да се покаже, че граматика Г е LL(1) тогава и само тогава, когато за две правила от вида $A \to u \mid v$ е изпълнено следното:

1. За нито един терминален символ a едновременно от u и v неможе да се изведат низове, започващи с a;
2. Само от един от низовете u и v може да се изведе празен низ;
3. Ако $v \Rightarrow^* \varepsilon$, то от u не се извежда никакъв низ, започващ с терминален символ от Follow(A).

На горното е еквивалентно следното определение:

Една контекстно-свободна граматика се нарича LL(1)-граматика, ако от съществуването на два леви извода:

(1) $S \Rightarrow^* w\,A\,u \Rightarrow w\,v\,u \Rightarrow^* wx$

(2) $S \Rightarrow^* w\,A\,u \Rightarrow w\,z\,u \Rightarrow^* wy$

за които First(x) = First(y), следва, че v = z. Това означава, че за даден низ wAu и първи символ, изводим от Au (или $), съществува неповече от едно правило, което може да бъде приложено към A, така че да се получи извод на терминален низ, започващ с w и продължаващ с този първи символ.

Следващият критерий се използва за определяне на LL(1) граматика. Граматиката $Г = \langle \Sigma, N, S, P \rangle$ е LL(1) граматика, тогава и само тогава, когато за всяка двойка правила от вида $A \to \alpha \mid \beta$ от P се изпълняват следващите две условия:

(1) $First(\alpha) \cap First(\beta) = \varnothing$;

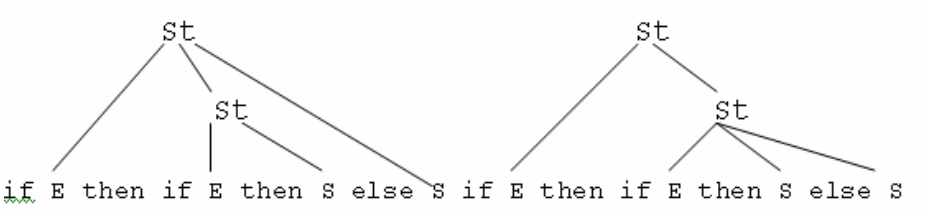
(2) Ако $\varepsilon \in First(\alpha)$, то $First(\beta) \cap Follow(A) = \varnothing$.

Език, за който може да се построи LL(1)-граматика, се нарича LL(1)-език.

Ако таблицата на анализа има неоднозначно определен вход, то граматиката не е LL(1).
Пример. Нека е дадена граматика със следните правила:

```
St → if Ex then St
   | if Ex then St else St
   | Cont
Ex → ...
```

Тази граматика е неоднозначна, което се илюстрира от следващата фигура.



Тъй като граматиката е неоднозначна, то тя не е LL(1)-граматика. Проблемът, поражда ли граматиката LL-език е алгоритмически неразрешим.

3.5.6. Рекурсивно спускане

Един от методите за синтез на програма-разпознавател за даден език е т.н. метод на рекурсивното спускане (resursive-descent parsing), спадащ към низходящите методи за синтактичен анализ. Програмирането на рекурсивното спускане се извършва на език от високо ниво, който допуска определянето на рекурсивни процедури (функции), които могат да се самоизвикват. При този метод на всеки нетерминален символ от граматиката се съпоставя отделна функция, която реализира разбор за всички правила, които имат този нетерминал в лявата си част. По този начин се извършва неявно използване на стек.

Нека е даден нетерминален символ от който се извеждат думи от езика – низове от терминални символи. При низходящ разбор този нетерминал на първата стъпка се заменя с дясната част на едно от правилата, в което той присъства в лявата част.

Процедурите при рекурсивното спускане може да изглеждат по следния начин:

```
void A()/*A → u1 | u2 | ... |uk*/
        {if (InSym ∈ First(ui))         //само един!
                if (parse(ui))
                        result("A → ui");
                else error();
         else
//Вариант 1:
        if (Има правило A → ui такова, че ui ⇒* ε)
//Вариант 1.1
                if (InSym ∈ Follow(A))
                        result("A → ui");
                else error();
 //Край на вариант 1.1

//Вариант 1.2:
                result("A → ui");
//Край на вариант 1.2
//Край на вариант 1
//Вариант 2:
        if (Няма правило A → ui такова, че ui ⇒* ε)
                error();
        }
//Край на вариант 2

boolean parse(u)
//от u не се извежда ε!
```

```
{       v = u;
        while (v != ε)
                {// v == Xz
                 if (X-терминал a)
                        if (InSym != a)
                                return(false);
                else InSym=getInsym();
                else //X-нетерминал B
                        B;
                v = z;
                }
        return(true);
}
```

В процедура A за случаите, когато има правило $A \rightarrow u_i$ такова, че $u_i \Rightarrow^* \varepsilon$, са приведени два варианта 1.1 и 1.2. Във вариант 1.1 се прави проверка принадлежи ли следващия входен символ на Follow(A). Ако не – издава се преминава в режим за обработка на грешката. Във вариант 1.2 това не се прави, така, че анализът на грешките се извършва от процедурата извикваща A.

3.5.7. Възстановяване след синтактически грешки

Съгласно техниката за реализация на LL(1) синтактичен анализ, на всяко правило на граматиката се съпоставя синтактична процедура, чието изпълнение е безусловно. В приведената по-горе програма за рекурсивно спускане се използва процедурата error() за реакция на синтактическите грешки. В най-простия случай тази процедура извършва диагностика и завършва работата на анализатора. Възстановяването след синтактична грешка изисква условност при влизане и излизане от синтактична процедура (входна и изходна проверка).

Ако в момента на регистриране на грешка на върха на стека се окаже нетерминалния символ A и за него няма правило, съответстващо на входния символ, то входа се сканира докато не се срещне символ или от First(A) или от Follow(A). В първия случай A се замества по съответстващите правила, а във втория – символ A се изважда от стека.

Ако на върха на стека се намира терминален символ, то е възможно изваждането на всички терминални символи от върха на стека, докато се достигне до първия нетерминален символ и да се продължи така, както по-горе.

3.6. Възходящ анализ

3.6.1. Основни понятия.

Поднизът от сентенциалната форма, който може да бъде съпоставен на дясната част на някое от правилата, се нарича **основа**. Ако граматиката е еднозначна, то тя има точно една основа. Замяната на основата в сентенциалната форма с нетерминал от лявата част се нарича **отсичане на основата**. Полученият подниз е отново в сентенциална форма.

При възходящия (bottom - up) синтактичен анализ се прави опит да се построи синтактично дърво, започвайки от листата (изходната програма), като се стреми да се достигне корена (аксиомата на граматиката). На всяка стъпка от анализа, основата на

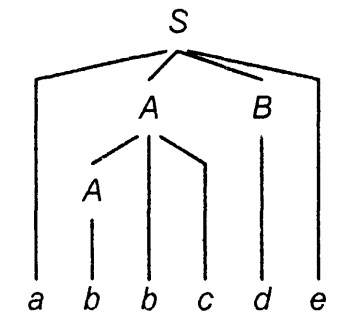
текущата сентенциална форма се заменя с нетерминалния символ, от който е изведена. На практика се извършва прилагане на граматичните правила в обратна посока, понеже всяка дясна част на правило е и основа на сентенциална форма.

При даден входен низ $\omega \in \Sigma^*$ трябва да се построи последователността от сентенциални форми $\omega \triangleright \varphi_1 \triangleright \varphi_2 \triangleright \ldots \triangleright S$, където S е стартовият символ на дадената граматика Г. По-горе знакът $\triangleright$ означава редукция. Понятието редукция се определя по следния начин: нека $X \to \alpha \in P$ и нека $\varphi_i = \beta\alpha\gamma$. Тогава $\varphi_{i+1} = \beta X\gamma$ и низовете φ_i и φ_{i+1} се намират в отношение **редукция**, т.е. $\varphi_i \triangleright \varphi_{i+1}$.

Например, ако в граматиката Г съществува сентенциална форма, получена чрез дясно канонично пораждане: $A \Rightarrow A + B \Rightarrow A + B * C$. Най-левият подниз на A + B *C, върху който може да се извърши редукция е $A + B * C \triangleright B * C$, която довежда до низ, който не е сентенциална форма в Г, т.е. такава редукция е неправилна. Правилната редукция е $A + B * C \triangleright A * B$.

Един от основните проблеми при възходящия синтактичен анализ е намирането и отсичането на основите.

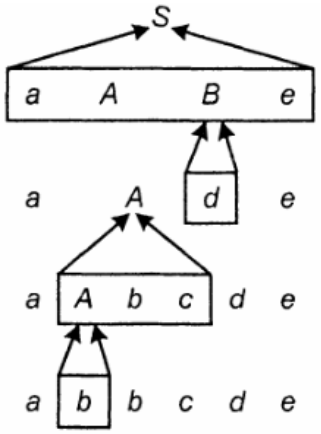
Нека е дадена граматиката Г: P = {S → aABe , A → Abc, A → b, B →d}. Една от изводимите сентенциалните форми е abbcde може да бъде породена съгласно следното синтактично дърво:



Възстановяването на дървото на извода за входния низ от езика на граматиката с помощта на възходящия метод на синтактическия анализ се извършва по следния начин. Входният низ се претърсва отляво надясно и в него се търси най-левия подниз – основата, която съвпада с дясната част от едно от правилата на граматиката. Намереният подниз се редуцира към нетерминал – лява част от съответното правило. В примера е видно, че основата в низ abbcde е първото срещане на символ b, като се изпълнява редукцията abbcde ▷ aAbcde, в съответствие с правилото A → b. Полученият редуциран низ aAbcde е в сентенциална форма.

На втора стъпка се анализира получената сентенциална форма aAbcde. Чрез отсичането на основата Abc се извършва редукцията aAbcde ▷ aAde, съобразно правилото A → Abc.

Възходящият синтактичен анализ, по изходния низ abbcde и граматиката Г, построява следната последователност от сентенциални форми: abbcd ▷ aAbcde ▷ aAde ▷ aABe ▷ S, завършваща с аксиомата на граматиката S:



Тази последователност е възстановения десен каноничен извод на низ abbcde в граматиката Г: $S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$.

Алгоритмите за възходящ синтактичен анализ се основават на търсенето на основата на текущата сентенциална форма и нейното редуциране; следва търсене на основата на новополучената сентенциална форма, отново редуциране и т.н., докато се получи аксиомата на граматиката. Ако на някоя от стъпките не може да се намери основа или ако крайната форма не е аксиомата на граматиката, това ще бъде индикация за грешка в дадения за разбор входен низ.

Чрез търсенето и редуцирането на основите се получава винаги дясно-каноничен разбор. Известна е следната лема:

Лема. Надясно от основата α на която и да е сентенциална форма има само терминални символи, т.е. $\varphi_i = \beta\alpha u$.

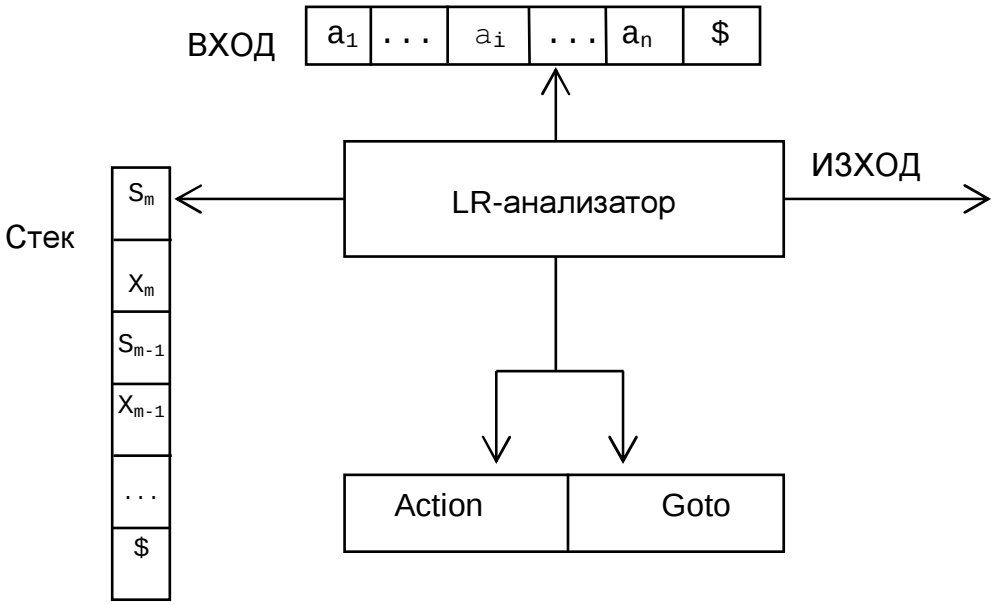
Следователно $\varphi_{i+1} = \beta X u$ съгласно продукция $X \rightarrow \alpha$. X е най-десният нетерминален символ във φ_{i+1}, т.е. при обратен ред на сентенциалните форми винаги се разширява най-десният нетерминален символ.

3.6.2. LR(k) анализатори.

LR(k)-граматиките са клас контекстно-свободни граматики, които допускат ефективен детерминиран възходящ синтактичен анализ с помощта на стеков автомат. В името LR(k) символ L означава, че входната редица се сканира отляво надясно, R, че се прави десен каноничен разбор в обратен ред, k – брой символи от входния низ, които започват от текущия сканиран символ, определящи вида на стъпката в анализиращия алгоритъм. Действието на анализатора на всяка стъпка от анализа се определя само от състоянието на върха на стека (един символ от азбуката на стека) и от не повече от k входни символа от входния низ, като се започне от текущия сканиран символ.

LR-анализаторът се състои от вход, изход, стек, анализиращ алгоритъм и управляваща таблица, която има две части – действия и преходи. Анализиращите алгоритми се различават по управляващите таблици. На всяка стъпка анализиращият алгоритъм чете символи от входния буфер. В процеса на анализ се използва стек, в който се съхраняват низове от вида $S_0X_1S_1X_2\ldots X_mS_m$ (S_m – връх на стека). X_i – терминален или нетерминален символ, а S_i – символ състояние. Всеки символ състояние изразява информация, съдържаща се в стека под него, а комбинацията между символа състояние на върха на

стека и текущия входен символ се използва за индексация в управляващата таблица и определя решението за пренос или редукция.



Управляващата таблица се състои от две части: действия (action) и преходи (goto). Началното състояние на този детерминиран краен автомат е състоянието, поместено на върха на стека на LR-анализатора в началото на неговата работа.

Конфигурация на LR-анализатора е двойката, първата компонента на която е съдържанието на стека, а втората – непрочетения вход:
(S_0 X_1 S_1 X_2 S_2… X_m S_m, a_i a_{i+1} ... a_n \$).

Тази конфигурация съответства на дясната сентенциална форма X_1 X_2 … X_m a_i a_{i+1} ... a_n. Префиксите на дясната сентенциална форма, които могат да се появят в стека на анализатор, се наричат **активни** префикси. Основата на сентенциалната форма се разполага винаги на върха на стека.

Поредната стъпка на анализатора се определя от текущия входен символ a_i и символа на състоянието S_m на върха на стека. Елементът от таблицата на действието Action[Sm, ai] може да има едно от следните четири значения:

1) Shift S, пренос, където S е състояние,
2) Reduce A $\rightarrow$ w, редукция по правилото A $\rightarrow$ w,
3) Accept, допускане,
4) Error, грешка.

Конфигурациите, получаващи се при всяка от тези четири стъпки, са следните:

1. Случай Action[S_m, a_i]=shift S. Анализаторът изпълнява стъпка на пренос, преминавайки в конфигурация: (S_0 X_1 S_1 X_2 S_2 … X_m S_m a_i S, a_{i+1} ... a_n \$). В стека се премества входният символ a_i и следващото състояние S, определено чрез action[S_m, a_i]. Текущ входен символ става a_{i+1}.

2. Случай Action[S_m, a_i]=A $\rightarrow$ w. Анализаторът изпълнява редукция и преминава в следната конфигурация: (S_0 X_1 S_1 X_2 S_2 … X_{m-r} S_{m-r} A S, a_i a_{i+1} ... a_n \$), където S = Goto[S_{m-r}, A] и r – дължината на w, дясната част на правилото за извод. Функцията goto е функцията на преходите на детерминирания краен автомат, разпознаващ активните префикси на Г. Анализаторът първо изважда от стека 2r символа (r символа на

състоянието и r символа на граматиката), така на върха на стека се оказва състояние S_{m-r}. След това анализаторът пренася в стека лявата част от правилото за извод – A, а също и S – стойността от таблицата Goto[S_{m-r}, A]. В случая на редукция текущия входен символ не се променя. За един LR-анализатор последователността от символи на граматиката X_{m-r+1} … X_m е, които се изваждат от стека, винаги съответстват на w – дясната част от правилото за извод, по което се извършва редукция.

След извършване на редукция LR-анализаторът генерира изход, т.е. изпълняват се семантични действия, свързани с правило, по което се прави редукция. Например, изпечатва се номер на правилото, по което се прави редукция.

3. Случай Action[S_m, a_i]=accept. Разборът е завършен.
4. Случай Action[S_m, a_i]=error. Анализаторът намира грешка и изпълнява действия по диагностика и възстановяване.

Следва алгоритъм на LR-анализатор.

```
while (true)
        {Нека S - състояние на върха на стека;
         if (action[S,InSym]==“shift S'“)
                {премести InSym и след това S' на върха
                на стека;
                прочитане в InSym на следващия входен символ;
                }
        else if (action[S,InSym]==“reduce N->w”)
                        {изваждане от стека на 2*|w| символа;
                        на върха на стека е S';
                        преместване на N на върха на стека, а
                        след това и на състояние goto[S',N];
                        вывести правило N->w;
                        }
                else if (action[S,InSym]==“accept”)
                                {result(“success”);
                                 break;
                                }
                         else {result(error());
                                        break;
                                  }
        }
```

В началото в стека се намира началното състояние S_0, а в буфера w$, InSym съдържа първия символ на w$; Анализаторът изпълнява горния алгоритъм, докато не достигне състояние accept или error.

Пример. Нека е дадена граматика Г = ⟨{id, +, *}, {E, T, F}, E, P⟩ аритметични изрази с бинарни операции + и * със следните правила:

(1) $E \to E + T$,
(2) $E \to T$,
(3) $T \to T * F$,
(4) $T \to F$,
(5) $F \to id$.

Следващата таблица изобразява функциите action и goto, образуващи LR(1)-таблицата за тази граматика.

Състояния	*Action*				*Goto*		
	id	+	*	$	E	T	F
0	S6				1	2	3
1		S4		acc			
2		R2	S7	R2			
3		R4	R4	R4			
4	S6					5	3
5		R1	S7	R1			
6		R5	R5	R5			
7	S6						8
8		R3	R3	R3			

За елемент S_i функция Action означава пренос и поместване в стека на състоянието с номер i, R_j – редукция по правило номер j, acc – допускане (край), празната клетка – грешка. За функция Goto символ i означава поместване в стека на състояние с номер i, празната клетка – грешка.

За входната последователността id+id*id в следващата таблица са показани състоянието на стека, входа и извършваните действия.

Активен префикс	Стек	Вход	Действие
	0	$id+id*id\$$	**пренос**
id	$0\ id\ 6$	$+id*id\$$	$F \to id$
F	$0\ F\ 3$	$+id*id\$$	$T \to F$
T	$0\ T\ 2$	$+id*id\$$	$E \to T$
E	$0\ E\ 1$	$+id*id\$$	**пренос**
$E+$	$0\ E\ 1+4$	$id*id\$$	**пренос**
$E+id$	$0\ E\ 1+4\ id\ 6$	$*id\$$	$F \to id$
$E+F$	$0\ E\ 1+4\ F\ 3$	$*id\$$	$T \to F$
$E+T$	$0\ E\ 1+4\ T\ 5$	$id\$$	**пренос**
$E+T*$	$0\ E\ 1+4\ T\ 5*7$	$id\$$	**пренос**
$E+T*id$	$0\ E\ 1+4\ T\ 5*7\ id\ 6$	$\$$	$F \to id$
$E+T*F$	$0\ E\ 1+4\ T\ 5*7\ F\ 8$	$\$$	$T \to T*F$
$E+T$	$0\ E\ 1+4\ T\ 5$	$\$$	$E \to E+T$
E	$0\ E\ 1$		**край**

В първия ред на LR-анализатора се намира нулевото състояние и се чете първия входен символ id. Действието S_6 в нулевия ред и стълба id в поле Action означава пренос и поставяне на символа на състоянието S_6 на върха на стека. Това е изобразено във втория ред: първия символ id и символа на състоянието S_6 са поместени в стека, а id се премахва от входния низ. Текущ входен символ става +, и действието в състояние S_6 за входен символ + е редукция съгласно $F \to id$. От стека се изваждат два символа (един символ на състоянието и един символ на граматиката). След това се анализира нулевото състояние. Тъй като Goto в нулевото състояние е 3, то F и 3 се поставят в стека. Така се преминава към конфигурацията, съответстваща на третия ред. Останалите стъпки се определят аналогично.

3.6.3. Конструиране на LR(1)-таблици

Граматиките, за които може да се построи таблица на LR-разбора, се наричат LR-граматики. За да е възможно граматиката да е LR, то анализаторът трябва да може да разпознава основата на върха на стека. Разпознаването на основите се извършва с крайни автомати, които сканират стека от дъното към върха. Състоянието на автомата след прочитане на елементите от стека и текущия входен символ определя действието му. Функциите за преход на крайния автомат се явяват таблица на преходите за LR-анализатора. За да не се преглежда стека на всяка стъпка от анализа, на върха на стека винаги се пази състоянието, в което трябва да се окаже крайният автомат, след като е прочел символите от граматиката от дъното към върха на стека.

Вземането на решение за пренос или редукция на анализатора се извършва чрез прочитане на поредните k входни символа. Практически интерес представляват случаите при $k = 0$ и $k = 1$. Граматика, която може да бъде анализирана с LR-анализатор, който прочита k входни символа на всяка стъпка, се нарича **LR(k)-граматика**.

Нека $\Gamma = \langle \Sigma, N, S, P \rangle$ е контекстно-свободна граматика. **Попълнена** граматика за дадена граматика Γ се нарича контекстно-свободната граматика $\Gamma' = \langle \Sigma\ N \cup \{S'\}, S', P \cup \{S' \to S\} \rangle$, т.е. еквивалентната граматика, в която е въведен нов начален символ S' и ново правило за извод $S' \to S$. Това допълнително правило спомага за определянето кога анализаторът трябва да прекрати разбора и да допусне входния низ. Разборът е завършен когато анализаторът е готов да извърши редукция по правилото $S' \to S$.

LR(1)-ситуация (LR(1)-точка) се нарича двойката $[A \to \alpha.\beta, a]$, където $A \to \alpha\beta$ – правило от граматиката, a – терминален символ или маркер за край \$. Единицата „1" указва дължината на втората компонента.

LR(1)-ситуацията $[A \to \alpha.\beta, a]$ е **допустима за активния префикс** δ, ако съществува извод $S \Rightarrow_r^* \gamma A\omega \Rightarrow_r \gamma\alpha\beta\omega$, където $\delta = \gamma\alpha$ и или a е първия символ на ω или $\omega = \varepsilon$ и $a = \$$. LR(1)-ситуацията е **допустима**, ако е допустима за произволен активен префикс.

Пример. Нека е дадена граматиката $\Gamma = \langle \{a, b\}, \{S, B\}, S, P \rangle$ с правила $S \to BB$, $B \to aB \mid b$. Съществува десен извод $S \Rightarrow_r^* aaBab \Rightarrow_r aaaBab$. Вижда се, че ситуацията $[B \to a.B, a]$ е допустима за активния префикс $\delta = aaa$, ако в горното определение се положи $\delta = aa$, $A = B$, $\omega = ab$, $\alpha = a$, $\beta = B$. Съществува десен извод $S \Rightarrow_r^* BaB \Rightarrow_r BaaB$. По този начин за активния префикс Baa допустима ситуация е $[B \to a.B, \$]$.

Идеята на метода е, че по граматиката се строи детерминиран краен автомат, разпознаващ активните префикси. Необходимо е ситуациите да се групират в множества, които образуват състояния на автомата. Ситуациите може да се разглеждат като състояния на детерминирания краен автомат, разпознаващ активните префикси, а групирането им е процес на построяване на детерминиран краен автомат от недетерминиран.

Разглеждайки ситуация от вида $[A \to \alpha.B\beta, a]$, от цялото множество ситуации, допустима за активен префикс z, то съществува десен извод $S \Rightarrow_r^* \gamma Aax \Rightarrow_r \gamma\alpha B\beta ax$, където $z = \gamma\alpha$. Да предположим, че от βax се извежда терминалния низ $b\omega$. Тогава за някое правило от вида $B \to q$ има извод $S \Rightarrow_r^* zBb\omega \Rightarrow_r zqb\omega$. По този начин $[B \to .q, b]$ е също допустима за z и ситуацията $[A \to \alpha.B\beta, a]$ е допустима за активния префикс z. Получава се, че или b може да бъде първи терминален символ, изводим от β, или от β се извежда ε от извода $\beta ax \Rightarrow_r^* b\omega$ и тогава b е равно на a, т.е. b принадлежи на First(βax). Построяването на всички тези ситуации за дадено множество ситуации, се извършва с функцията Closure.

Системата от множеството допустими LR(1)-ситуации за всички възможни активни префикси на попълнена граматика се нарича каноническа система на множеството от допустими LR(1)-ситуации.

Алгоритъм за конструиране на каноническа система на множеството от допустими LR(1)-ситуации.
Вход. Контекстно-свободна граматика Г = ⟨Σ, N, S, P⟩
Изход. Канонически система C на множеството от допустими LR(1)-ситуации за граматиката Г.
Метод. За попълнената граматика Г' се изпълнява процедурата Items, която използва функциите Closure и Goto.

```
function closure(I)/*I - множество от ситуации*/
{do
        {for (всяка ситуация [A->α.Bβ,a] от I,
                  всяко правило за извод B → q от Г',
                  всеки терминал b от First(βa),
                        такъв, че [B → .q, b] не е в I
                 )
                добави [B → .q, b] към I;
        }
        while (към I може да се добавят нови ситуации);
    return I;
}
```

В анализаторите от тип LR(0) при построяване на Closure не се отчитат терминалните символи от First(βa).

Ако I е множеството от допустими ситуации за произволен активен префикс z, то Goto(I, X) е множеството от допустими ситуации за активния префикс zX.

```
function Goto(I, X)/*I - множество от ситуации;
                        X - символ грамматики*/
{       Нека J = [A → αX.β, a] | [A → αX.β, a] ∈ I;
        return Closure(J);
}
```

Работата на алгоритъма за конструиране на каноническа система C на множеството от допустими LR(1)-ситуации започва с поместване в C на началното множество от ситуации I_0 = Closure({[S' → .S, $]}). След това с помощта на функция Goto се изчисляват новите множества ситуации и се включват в C. Goto(I, X) е преход на крайния автомат от състояние I по символ X.

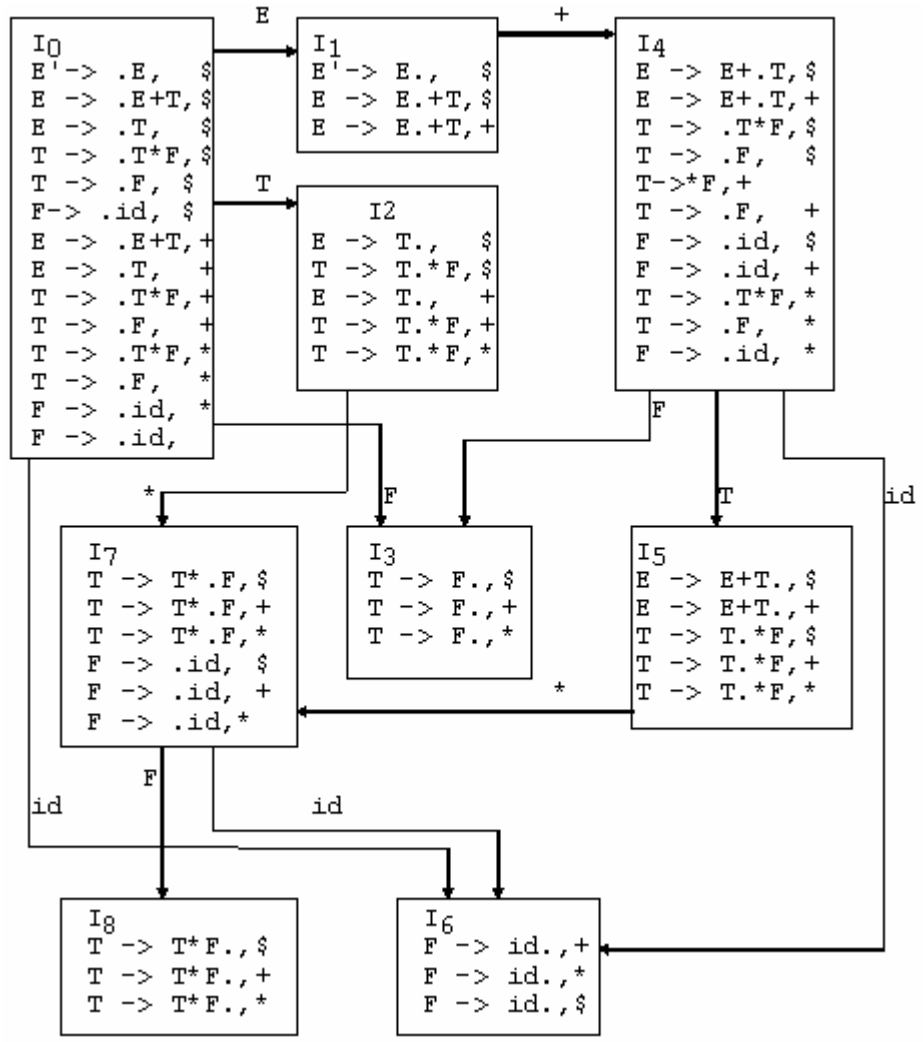
```
void Items(Grammar Г')
{       do
        {C={Closure({[S' → .S, $]})};
         for (всяко множество от ситуации I от C,
                  всеки символ X от граматиката такъв,
                 че Goto(I,X) не е празно и не принадлежи на C
                 )
                 добави Goto(I, X) към C;
```

```
        }
        while (към C може да се добави ново
                множество от ситуации);
}
```

Пример. Дадена е попълнената граматика със следните правила:

(0) E' → E
(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → id

Множеството от ситуации и допустими правила са приведени на следващата фигура.



Във всеки момент от анализа в стека се намира активният префикс, който съответства на последователностите от преходи от началното състояние I_0 в текущото. Редукцията, това е замяната на суфикса с префикса и преход към новото състояние, което се определя от таблицата Goto по символа в лявата част от редукционното правило и предходния символ на стека, който съответства на състоянието след разбора. За определяне на това новото състояние, на върха на стека се пази състоянието, съответстващо на активния префикс без този суфикс. Новото състояние се определя като Goto[предно състояние, нов нетерминал].

Построяването на LR(1)-таблицата (функциите на действието и преходите) на LR(1)-анализатора по множеството от LR(1)-ситуациите се извършва по следните стъпки:

(1) Построява се набор множество LR(1)-ситуации C = {I_0, I_1, …, I_n} за Г'

(2) Състояние i на анализатора се строи от I_i. Стойността на функцията на действието (Action) на анализатора за състояние i се определят по следния начин:

a) Ако [A $\rightarrow$ α.aβ, b] $\in$ I_i и Goto(I_i, a) = I_j, то се полага Action[i, a] = Shift j. Тук a е терминален символ;

b) Ако [A $\rightarrow$ α., a] $\in$ I_i такова, че A ≠ S', то се полага Action[i, a] = Reduce A $\rightarrow$ α;

c) Ако [S' $\rightarrow$ S., $] $\in$ I_i, то се полага Action[i, $] = Accept.

(3) Преходите за състояние i се определят по следния начин: ако Goto(I_i, A) = I_i, то Goto[i, A] = j. Тук A е нетерминал.

(4) Всички входове в Action и Goto, които не са определени по стъпки (2) и (3), се полагат равни на Error.

(5) Началното състояние на анализатора се строи от множеството съдържащо ситуацията [S' $\rightarrow$ .S, $].

Ако при прилагане на тези правила възникне конфликт, т.е. в едно и също множество се намерят повече от един вариант на действие (или пренос/редукция, или редукция/редукция), то тогава граматиката не е LR(1) и алгоритъмът завършва неуспешно.

LR(1)-таблицата, получаваща се от функциите на анализатора Action и Goto, се нарича **каноническа LR(1)-таблица**. LR(1)-анализаторът, работещ с тази таблица, се нарича канонически LR(1)-анализатор.

3.6.4. Възстановяване след синтактични грешки.

При синтактични грешки се чете стека от върха, докато не се намери състояние s с преход към отделения нетерминален символ A. След това се сканират входните символи, докато не бъде намерен такъв, който да е допустим след A. В този случай на върха на стека се поставя състоянието Goto[s, A] и разборът продължава. За нетерминалния символ A може да има няколко такива варианта. Обикновено A е този нетерминален символ, който представя една от основните конструкции на езика, например оператор. Тогава s е например, символа точка и запетая или end.

Възможно е поставянето на обръщение към подпрограма за обработка на грешките във всички празни клетки на анализатора. Такава подпрограма може да поставя или изтрива входни символи и символи от стека, да изменя наредбата на входните символи.

3.6.5. SLR(1)-анализатори.

Един вариант на LR-анализаторите са SLR(1)-анализаторите (опростен Simple LR(1)). Те се конструират по следния начин. Нека C = {I_0, I_1, …, I_n} е набор множество от допустими **LR(0)-ситуации**, където I_i са състоянията на анализатора. Функциите на действието и прехода се определят така:

(1) Ако [A $\rightarrow$ α.aβ] $\in$ I_i и Goto(I_i, a) = I_j, то се полага Action[i, a] = Shift j. Тук a е терминален символ;

(2) Ако [A $\rightarrow$ α.] $\in$ I_i, то се полага Action[i, a] = Reduce A $\rightarrow$ α за всички a $\in$ Follow(A), такова, че A ≠ S';

(3) Ако [S' $\rightarrow$ S] $\in$ I_i, то се полага Action[i, $] = Accept.

(4) Ako Goto(I_i, A) = I_i, то Goto[i, A] = j. Тук A е нетерминал.

(5) Всички останали входове за функции Action и Goto се полагат равни на Error.

(6) Началното състояние на анализатора съответства на множеството ситуации, съдържащо ситуацията [S' $\rightarrow$ .S, $].

Таблицата на синтактическия анализ, състояща се от функциите Action и Goto, се нарича SLR(1)-таблица на граматиката Г. LR-анализатор, използващ SLR(1)-таблица за граматиката Г, се нарича SLR(1)-анализатор за Г, а съответстващата граматика – SLR(1)-граматика.

3.6.6. LALR-анализатори.

Методът LALR (lookahead LR) също се използва за построяване на синтактически анализатор. Този метод често се използва на практика, тъй като получаваните с негова помощ таблици за значително по-малки отколкото каноническите LR-таблици; освен това, с помощта на LALR-граматиката се изразяват лесно голяма част от стандартните синтактични конструкции на езиците за програмиране.

Първата компонента на една LR-ситуация, се нарича **ядро на ситуацията**.

Основната идея при построяване на LALR-таблицата е в създаването на множество от LR(1)-ситуации и, ако това не доведе до конфликт, обединение на множествата с еднакви ядра. След това на основата на системата от множества ситуации се построява таблицата на синтактичния анализ.

Алгоритъм. Пряко построяване на LALR-таблица
Вход. Попълнена граматика Г'
Изход. Таблица с функциите Action и Goto на LALR-анализа за граматиката Г'.

(1) Построява се C = {I_0, I_1, …, I_n}, система от множествата LR(1)-ситуации за Г'.

(2) За всяко ядро, сред множествата от LR(1)-ситуации, се намират всички множества, имащи същото ядро, и тези множества се заменят с техните обединения.

(3) Нека C' = {J_0, J_1, …, J_m} са получените множества от LR(1)-ситуации. Функция Action за състояние i се строи от J_i, така както в алгоритъма за построяване на LR(1)-таблицата.

(4) Таблица Goto се построява по следния начин: Ако J е обединение на едно или няколко множества LR(1)-ситуации, т.е. J = $I_1 \cup I_2 \cup \ldots I_к$, то ядрата на множествата Goto(I_1, X), Goto(I_2, X),…, Goto(I_k, X) са едни и същи, тъй като I_0, I_1, …, $I_к$ имат едни и също като Goto(I_1, X). Тогава Goto(J, X) = K.

Построената по горния алгоритъм таблица, се нарича таблица на LALR-анализа за граматиката Г. Ако конфликтните действия в синтактическия анализ отстъстват, граматиката се нарича LALR(1)-граматика; системата множества от ситуации C' = {J_0, J_1, …, J_m}, построена на стъпка (3) от алгоритъма, се нарича LALR(1)-система.

3.7. Генератор на синтактически анализатори YACC

Програмата YACC (Yet Another Compiler Compiler, още един компилатор на компилатори) е предназначена за построяване на синтактически анализатори на контекстно-свободни езици. Анализируемият език се описва с помощта на граматика, която е в близка форма до Бекус-Науровата форма. Резултатът от работата на YACC е С-програма, реализираща LALR(1)-анализатор.

YACC-програмата се състои от три секции, разделени с %% - секция описания, секция правила, в която се описва граматиката, и секция подпрограми, която се копира в изходния файл.

Построяването на калкулатор може да бъде извършено със следващата граматика за аритметични действия:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid digit$

Токенът digit представлява отделна цифра от 0 до 9. YACC-калкулаторът на основата на тази граматика има следната структура:

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line    :       \ '\n'    {printf("%d\n", $1);   }
        ;
expr    :       expr    '+'     term    {$$ = $1 + $3;}
        |       term
        ;
term    :       term    '*'     factor  {$$ = $1 * $3;}
        |       factor
        ;
factor  :       '(' expr ')'    {$$ = $2;}
        |       DIGIT
        ;
%%
yylex() {
        int c;
        c = getchar();
        if (isdigit(c)) {
                yylval = c – '0';
                return DIGIT;
        }
        return c;
}
```

В секцията описания има два незадължителни раздела описания. В първия се поместват обикновените C описания, ограничени с %{ и %}. В горната програма този раздел съдържа само една директива #include <ctype.h>, което заставя препроцесора на C да я включи стандартния заглавен файл <ctype.h>, съдържащ описание на isdigit.

В частта за описания се специфицират и терминалите на граматиката. В случая това е DIGIT.

В секцията за правила след двойката %% се изписват правилата на транслацията. Всяко правило се състои от <лява част> → <Alt 1> | <Alt 2 | … | <Alt n> и в YACC се записва така:

```
<лява част>     :     <Alt 1>     {Семантично действие 1}
                |     <Alt 2>     {Семантично действие 2}
            ……
                |     <Alt n>     {Семантично действие n}
                ;
```

В продукциите на YACC отделните символи в единични кавички ‘c’ представляват терминален символ c, а низ от букви и цифри, без да е обявен с кавички или като терминален символ, се счита за нетерминален символ. Алтернативните десни части от продукцията се разделят с вертикална черта, а след всяка продукция с нейните алтернативи и семантични действия се поставя символ точка и запетая. Първата лява част се счита за стартов символ.

Семантичните действия на YACC са последователности от инструкции на C. В тях могат да се използват специални символи: $$ - стойност на атрибут, свързан с нетерминал в лявата част, $i – стойност, свързана с i-тия граматичен символ (терминале или нетерминален) отдясно. Семантическите действия се изпълняват при редукция със свързаната с тях продукция, така че обикновено се изчислява $$ по $1.

Двете продукции E → E + T | T имат следните YACC спецификации и семантически действия:

```
expr  :     expr  ‘+’     term   {$$ = $1 + $3;}
      |     term
      ;
```

Нетерминалният символ term е трети граматически символ. Семантичното действие за втората продукция се изпуска. В общия случай { $$ = $1; } е действие по подразбиране.

Към правилата на граматиката е добавено ново стартово правило:

```
line  :     \ ‘\n’   {printf(“%d\n”, $1);   }
```

То е свързано с това, че входният поток на калкулатора е израз, последван от символ за нов ред. Семантичното действие е извеждането на десетичната стойност на израза, последвана от символ за нов ред.

Третата част от спецификациите съдържа подпрограми на езика C. Сред тях задължително трябва да се намира функцията лексическия анализатор yylex(). Други функции, такива като подпрограми за обработка на грешки, може да бъдат добавени при необходимост.

Лексическият анализатор yylex() произвежда двойки лексема-стойност. Ако функцията върне лексема от тип DIGIT, то лексемата трябва да е обявена в първата част от спецификациите на YACC. Стойността на лексемата се предава на синтактичния анализатор чрез променлива yylval.

Ако дадена граматика е неоднозначна, то LALR-алгоритъмът ще генерира конфликт при действието на синтактичния анализ. Описанието породените конфликтни точки може да

се получи чрез стартиране на YACC с параметър –v. Тогава се създава допълнителен файл, съдържащ местата на конфликтите и начините за тяхното разрешаване.

При условие, че YACC не получи указание в своите спецификации как да разрешава конфликтите, то всички конфликти се разрешават при следните правила:

(1) Конфликт редукция/редукция се разрешава с избора на продукция, намираща се в първата спецификация на YACC.

(2) Конфликт пренос/редукция се разрешава в полза на преноса.

Друг механизъм за разрешаване на конфликт пренос/редукция е назначаването на приоритет и асоциативност. Описанието

```
%left  '+' '-'
```

задава операторите + и + като лявоасоциативни, с еднакъв приоритет. Обявяването на оператор като дясноасоциативен се извършва по следния начин:

```
%right      '^'
```

Освен това обезпечаването на неасоциативност на два оператора се извършва по следния начин:

```
%noassoc '<'
```

Лексемите получават възходящ приоритет в този ред, в който се срещат в описанията. Задаването на приоритет на редукцията се извършва чрез израза:

```
%prec <лексема>
```

вдясно от правилото.

Задаването в граматика на аритметичен израз на най-висок приоритет на операцията на унарния минус, отколкото бинарните операции +, -, * и /, се извършва така:

```
%token      DIGIT
%left       '+' '-'
%left       '*' '/'
%left       UMIN
%%
e     :     e '+' e | e '-' e | e '*' e | e '/' e
      |     '(' e ')' | DIGIT;
e     :     '-' e %prec UMIN;
%%
```

Фиктивната лексема UMIN се използва само за задаване приоритет на редукцията по правилото e : '-' e;

При условие, че анализираните низове не съответстват на езика, то в някой момент аналилзаторът ще се окаже в състояние, в което не е предвидено нито пренос, нито редукция за получаване на грешната лексема. Обикновено парсерът извиква функцията void yyerror(const char*) с аргумент "Syntax error" и завършва работа – възврат от функцията yyparse със стойност 1. Реализацията на функция yyerror се възлага на ползвателя и той може да се опита да организира в нея по-разумна диагностика.

В много случаи е желателно разборът да продължи. За възстановяване след грешки YACC съдържа специална лексема с името error, която може да се употребява в граматиката. При възникване на грешка се установява флаг за грешки, извиква се функцията yyerror и след това от стека се изваждат състояния, докато не се срещне

състояние, допускащо лексемата error. Тогава се осъществява пренос, съответстващ на лексемата error в това състояние и разборът продължава. Ако при установяване на флага за грешка отново възникне грешка, то за избягване на многократното извикване на съобщенията yyerror не се извиква, а грешната лексема се игнорира.

Използват се следните специални функции:

yyerrok – изчиства флага за грешка;
yyclearin() – изтрива прочетената отпред грешна лексема;
Макрос YYERROR явно предизвиква ситуация на грешка.

Например:

```
statement     : …
              | error ‘;’
```

При възникване на грешка вътре в statement продължението на разбора е възможно само чрез започване на ‘;’ – в резултат ще бъдат пропуснати всички лексеми до символа точка и запетая, който ще бъде редуциран до нетерминал statement.

Следва примерна програма на YACC за интерпретатор на формули:

```
%token ICONST
%left '+' '-'
%left '*' '/'

%%

p : /* empty */
  | p s
  ;

s : e '\n' { printf( "%d\n", $1 ); }
  | error '\n'
  ;

e : e '+' e  { $$ = $1 + $3; }
  | e '-' e  { $$ = $1 - $3; }
  | e '*' e  { $$ = $1 * $3; }
  | e '/' e  { $$ = $1 / $3; }
  | '(' e ')'{ $$ = $2; }
  | ICONST;
  ;
%%

#include <stdio.h>
main() { yyparse(); }
yyerror( mes ) char *mes; {  printf( "%s\n", mes ); }

yylex() {
  int c, d;
  while((c=getchar())==' '); /* Skip spaces */
  if( c>='0' && c<='9' ) {   /* Integer constant */
    for( d=c-'0'; (c=getchar()) >='0' && c<='9'; ) d=d*10+c-'0';
    ungetc( c, stdin );
```

```
    yylval = d;
    return ICONST;
  }
  return c;              /* Others */
}
```

4. Семантически анализ. + atributnite

4.1. Въведение

Обикновено в транслаторите на всяко правило от граматиката, на всяка алтернатива на произволен нетерминален символ се съпоставя семантична подпрограма. Тези подпрограми се изпълняват при редукция по зададените правила на граматиката.

В случаите, когато програмният език е сложен или към транслатора са предявени повишени изисквания (например, необходима е машинно-независима оптимизация на програмата с цел получаване на по-ефективен обектен код), първоначалната изходна програма се превежда в някаква вътрешна, междинна форма, по-удобна за проста машинна обработка. В повечето вътрешни представяния операторите се разполагат в този порядък, в който те са длъжни да се изпълнят, като по този начин съществено облекчават последващия анализ, интерпретация или генерация на обектен код.

4.2. Атрибутни граматики

Атрибутните граматики са един от най-често използваните формални методи за описания на езиците за програмиране. В тях променливите от периода на транслацията се свързват с върховете на дървото на синтактичния разбор. Тези променливи се наричат атрибути. Атрибутите носят информация за смисъла (семантиката) на съответните върхове. Всеки връх (терминал или нетерминал) може да има един или повече атрибути.

4.2.1. Определение за атрибутна граматика

Нека $\Gamma = \langle \Sigma, N, S, P \rangle$ е контекстно-свободна граматика, където Σ, N, S, P са съответно множество от терминални символи, нетерминални символи, множество правила за извод и аксиомата на граматиката.

На всеки символ X от $\Sigma \cup N$ се съпоставя множеството $A_S(X)$ от синтезирани атрибути и множеството $A_I(X)$ от наследяеми атрибути. Множеството от всички синтезирани атрибути за всички символи от $\Sigma \cup N$ се означава с A_S, наследяеми – A_I, като $A_S \cap A_I = \varnothing$. Атрибутите на различните символи са различни атрибути. Атрибут a на символ X се означава с $a(X)$. Стойностите на атрибутите може да бъдат произволни типове, например да представляват числа, низове, адреси от паметта и др.

Нека правилото $p \in P$ има вида $X_0 \to X_1X_2...X_n$. На всяко правило $p \in P$ се съпоставя крайното множество R(p) от семантични правила от вида $a(X_i) = f(b(X_j), c(X_k), \ldots, d(X_m))$, където $0 \le j, k, ..., m \le n$, като $1 \le i \le n$, ако $a(X_i) \in A_I(X_i)$, т.е. $a(X_i)$ – наследяем атрибут (дясна част на правило), и $i = 0$, ако $a(X_i) \in A_S(X_i)$, т.е. $a(X_i)$ – синтезиран атрибут (лява част на правило). По този начин се вижда, че семантичното правило определя стойността на атрибут a за символ X_i на основата на стойността на атрибутите b, c, ..., d, съответно за символите X_j, X_k, ..., X_m.

Частен случай е, когато дължината *n*, на дясната страна на правилото, е равна на нула, тогава се казва, че атрибут a на символ X_i получава стойност константа.

Граматика, която изпълнява горните условия, се нарича атрибутна граматика.

По-нататък ще се счита, че атрибутната граматика не съдържа семантични правила за изчисляване на атрибутите на терминалните символи. Предполага се, че атрибутите на терминалните символи или са предопределени константи или са достъпни като резултат от работата на лексическия анализатор.

Пример. Дадена е атрибутната граматика $\Gamma = \langle \Sigma, N, S, P \rangle$, позволяваща изчисляването на естествено число, записано в десетична форма, където $N = \{Num, Int, Frac\}$, $\Sigma = \{digit, .\}$, $S = Num$, а правилата за извод и семантическите правила се определят по следния начин (горните индекси се използват за връзки към различни използвания на един и същ нетерминален символ):

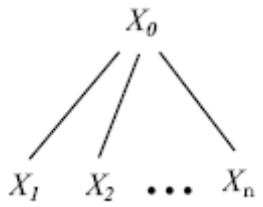
$$
\begin{array}{ll}
Num \rightarrow Int\ .\ Frac & v(Num) = v(Int) + v(Frac) \\
 & p(Frac) = 1 \\
 & \\
Int \rightarrow e & v(Int) = 0 \\
 & p(Int) = 0 \\
 & \\
Int^1 \rightarrow digit\ Int^2 & v(Int^1) = v(digit) * 10^{p(Int^2)} + v(Int^2) \\
 & p(Int^1) = p(Int^2) + 1 \\
 & \\
Frac \rightarrow e & v(Frac) = 0 \\
 & \\
Frac^1 \rightarrow digit\ Frac^2 & v(Frac^1) = v(digit) * 10^{-p(Frac^1)} + v(Frac^2) \\
 & p(Frac^2) = p(Frac^1) + 1
\end{array}
$$

За тази атрибутна граматика

$$
\begin{array}{ll}
A_S(Num) = \{v\}, & A_I(Num) = \varnothing, \\
A_S(Int) = \{v, p\}, & A_I(Int) = \varnothing, \\
A_S(Frac) = \{v\}, & A_I(Frac) = \{p\}.
\end{array}
$$

4.2.2. Атрибутно дърво на разбора

Нека е дадена атрибутната граматика $\Gamma = \langle \Sigma, N, S, P \rangle$ и дума принадлежаща на езика, определен от тази граматика. За тази дума може да се построи дърво на разбора *T* за граматиката *Γ*. На всеки вътрешен връх в това дърво е поместен нетерминален символ X_0 от граматиката Γ, съответстващ на прилагането на *p*-тото правило от граматиката; по този начин този връх ще има *n* непосредствени потомци.

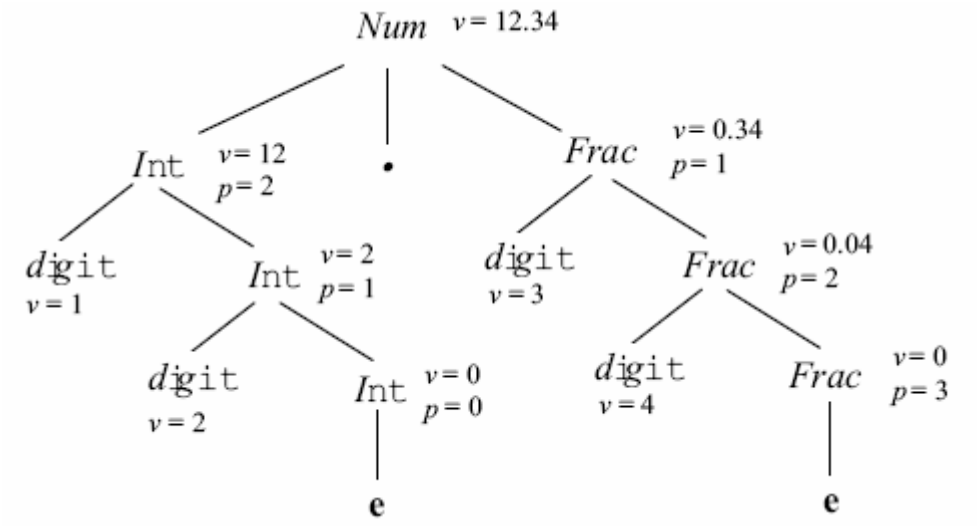


Нека сега X е етикет на възел от дървото и a е атрибут на символ X. Ако a е синтезиран атрибут, то $X = X_0$ за някое $p \in P$; ако a е наследяем атрибут, то $X = X_j$ за някое $p \in P$ и $1 \le j \le n$. По определение, атрибут a има стойност v в този възел, ако в съответното семантическо правило $a(X_i) = f(b(X_j), c(X_k), \ldots, d(X_m))$ всички атрибути $b, c, ..., d$ са вече определени и имат във възлите с етикети $X_j, X_k, ..., X_m$ съответни стойности $v_j, v_k, ..., v_m$, а $v = f(v_1, v_2, ..., v_m)$. Процесът на изчисление на атрибутите на дървото продължава до тогава, докато се изчислят всички атрибути.

Стойностите на синтезираните атрибути на символите във възлите на синтактичното дърво се изчисляват по атрибутите на символите на потомците на този възел; стойностите на наследствените атрибути се изчисляват по атрибутите на родителя и съседите.

Атрибутите, които се съпоставят на входния символ в дървото на разбора, се наричат входни атрибути в дървото на разбора, а дървото със съпоставени на всеки връх атрибути – атрибутно дърво на разбора.

Пример. Атрибутното дърво за граматиката от горния пример и дума w = 12.34 е показано на следващата фигура:

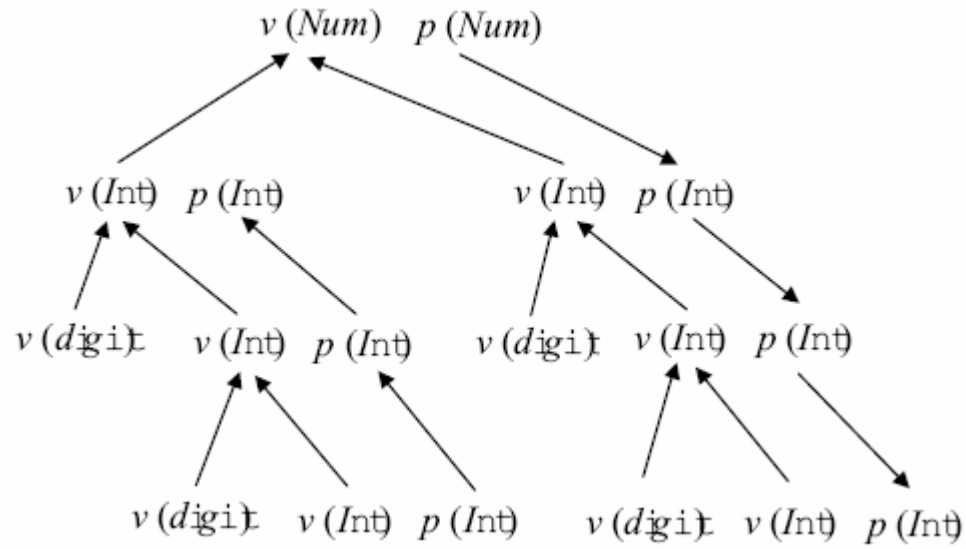


4.2.3. Коректност на семантичните правила

Семантичните правила са зададени коректно, ако позволяват изчисляването на всички атрибути на произволен възел в произволно дърво на извода.

Между атрибутите в дървото на разбора съществуват зависимости, определени от семантичните правила, съответстващи на прилагането на синтактичните правила. Зависимостите могат да бъдат представени във вид на ориентиран граф.

Нека T е дърво на разбора. На дървото се съпоставя ориентиран граф $D(T)$, чиито възлите са двойките (n, a), където n – възел на дървото, a – атрибут на символа, служещ за етикет на възел n. Графът съдържа дъга от (n_1, a_1) към (n_2, a_2) тогава и само тогава, когато семантичното правило, изчисляващо атрибут a_2, непосредствено използва стойността на атрибут a_1. По този начин възлите на граф $D(T)$ се явяват атрибути, които трябва да се изчислят, а дъгите определят зависимости, определящи кои атрибути се изчисляват по-рано и кои по-късно.



Семантичните правила са коректни тогава и само тогава, когато за произволно дърво на разбора *T* съответстващия граф *D*(*T*) не съдържа цикли (т.е. *D*(*T*) е ориентиран ацикличен граф).

4.2.4. Класове атрибутни граматики и тяхната реализация

Реализацията на атрибутните граматики е трудна задача поради множеството стойности на атрибути, свързани с данните от дървото, които трябва да се изчисляват в съответствие със зависимостите на атрибутите, които образуват ориентиран ацикличен граф. На практика това се извършва при обхождане на дървото по един или друг начин.

Два прости подкласа атрибутни граматики, изчисляващи всички атрибути, за които той може да бъде осъществен едновременно със синтактическия анализ, са S-атрибутните и L-атрибутните граматики.

Атрибутна граматика се нарича *S*-атрибутна, ако съдържа само синтезирани атрибути. За всяка *S*-атрибутна граматика може да бъдат изчислени всички атрибути за един обход на дървото отдолу нагоре. По този начин изчисляването на атрибутите може да се извърши паралелно с възходящия синтактичен анализ, например, *LR*(1)-анализ.

Атрибутна граматика се нарича *L*-атрибутна, ако произволен наследствен атрибут за произволен символ X_j от десните части на всяко правило $X_0 \rightarrow X_1X_2...X_n$ от граматиката зависи само от атрибутите на символите $X_1X_2...X_{j-1}$, намиращи се в правилата отляво на X_j и наследствените атрибути на символа X_0.

Всяка S-атрибутна граматика е L-атрибутна. Всички атрибути на произволно дърво за L-атрибутна граматика може да бъдат изчислени за един обход на дървото от горе надолу и отляво надясно. По този начин изчисляването на атрибутите може да се извърши паралелно с низходящия синтактичен анализ, например, LL(1)-анализ или рекурсивно спускане.

В случая на рекурсивно спускане във всяка функция, съответстваща на нетерминален символ, се определят формални параметри, предавани по стойност, за наследствените атрибути, и формални параметри, предавани по указател, за синтезираните атрибути.

Пример. Атрибутната граматика за изчисляване на число в десетична форма е *L*-атрибутна граматика и има следната реализация:

```
void int_part(float * V0, int * P0)
{if (Map[InSym]==Digit)
        {       int I=InSym;
                InSym=getInSym();
                float V2;
                int P2;
                int_part(&V2,&P2);
                *V0=I*exp(P2*ln(10))+V2;
                *P0=P2+1;
                else {  *V0=0;
                        *P0=0;
                  }
}
void fract_part(float * V0, int P0)
{if (Map[InSym]==Digit)
        {       int I=InSym;
                InSym=getInSym();
                float V2;
                int P2=P0+1;
                fract_part(&V2,P2);
                *V0=I*exp(-P0*ln(10))+V2;
                else {*V0=0;
           }
}
void number()
{       float V1,V3,V0;
        int P;
  int_part(&V1,&P);
  if (InSym!='.') error();
  fract_part(&V3,1);
  V0=V1+V3;
}
```

4.2.5. Език за описание на атрибутни граматики

Атрибутните граматики са удобно средство за описание семантиката на езиците за програмиране. На тяхна основа са създадени различни системи за автоматизация разработката на транслатори.

При записа на синтаксиса се използва разширената БНФ. Елементът вдясната част от синтактичното правило, затворен в скоби [], може да отсъства. Елементът вдясната част на синтактичното правило затворен в скоби [()], означава възможност за повторение на нула или повече брой пъти. В скобите [] или [()] може да се указват разделители на конструкции.

По-долу е показан синтаксиса на езика за описание на атрибутни граматики. Приведените конструкции описват атрубутните изчисления. Синтаксис на изрази и оператори не се привежда – той е основан на езика C.

```
Атрибутна грамматика ::= 'ALPHABET'
        ( ОписаниеНетерминал ) ( Правило )
ОписаниеНетерминал ::= ИмеНетерминал
              '::' [( ОписаниеАтрибут / ';')] '.'
ОписаниеАтрибут ::= Тип ( ИмеАтрибут / ',')
Правило ::= 'RULE' Синтаксис 'SEMANTICS' Семантика '.'
Синтаксис ::= ИмеНетерминал '::=' ДяснаЧаст
ДяснаЧаст ::= [( ЕлементДяснаЧаст )]
ЕлементДяснаЧаст ::= ИмеНетерминал
        | Терминал
        | '(' Нетерминал [ '/' Терминал ] ')'
        | '[' Нетерминал ']'
        | '[(' Нетерминал [ '/' Терминал ] ')]'
Семантика ::= [( ЛокалнаДефиниция ])
           [( СемантичноДействие ])
СемантичноДействие ::= Присвояване
         | [ Етикет ] Оператор
Присвояване ::= Променлива ':=' Израз
Променлива ::= ЛокалнаПроменлива
         |   Атрибут
Атрибут ::= ЛокаленАтрибут
       |   ГлобаленАтрибут

ЛокаленАтрибут ::= ИмеАтрибут '<' Номер '>'
ГлобаленАтрибут ::= ИмеАтрибут '<' Нетерминал '>'
Етикет ::= Цяло ':'
      |  Цяло 'E' ':'
      |  Цяло 'A' ':'
Оператор ::= Условен
        |   ОператорПроцедура
        |   ЦикълПоМножество
        |   ПростЦикъл
        |   ЦикълСУсловиеВКрая
```

Описанието на атрибутната граматика се състои от раздел за описание на атрибутите и раздел за описание на правилата. Разделът за описание на атрубутите определя състава на атрибутите за всеки символ от граматиката и типа на всеки атрибут. Правилата се състоят от синтактична и семантична част. В синтактичната част се използва разширената БНФ. Семантичната част от правилата се състои от локални дефиниции и семантични действия. В качеството на семантични действия се допускат както атрибутни присвоявания, така и съставни оператори.

Етикетът в семантичната част на правилата довежда до изпълнение на оператор за обхождане на дървото на разбора отгоре надолу и отляво надясно. Конструкция `i : оператор` означава, че операторът трябва да бъде изпълнен веднага след обхождане на i-тата компонента вдясната част. Конструкция `i E : оператор` означава, че операторът трябва да бъде изпълнен само, ако i-тата компонента вдясната част няма пораждане. Конструкция `i A : оператор` означава, че операторът трябва да бъде изпълнен след разбора на всяко повторение на i-тата компонента вдясната част.

Всяко правило може да има локални дефиниции (типове и променливи). Във формулите се използват както атрубути на символите от даденото правило (локални атрибути) и в

този случай съответстващите символи се указват с номера в правилото (0 – за символи вдясната част, 1 – за първи символ вдясната част, 2 – за втори символ вдясната част и т.н.), така и атрибути на символите предшествените левите части от правилата (глобални атрибути). В този случай съответния атрибут се указва с името на нетерминала. По този начин на дървото се образуват области на видимост на атрибутите.

Стойностите на терминалните символи за достъпни чрез атрибут VAL на съответния тип.

Пример. Атрибутната граматика от горния пример се записва в следната форма:

```
ALPHABET
Num :: float V.
Int :: float V;
int P.
Frac :: float V;
int P.
digit :: int VAL.

RULE
Num ::= Int '.' Frac
SEMANTICS
V<0>=V<1>+V<3>; P<3>=1.

RULE
Int ::= e
SEMANTICS
V<0>=0; P<0>=0.

RULE
Int ::= digit Int
SEMANTICS
V<0>=VAL<1>*10**P<2>+V<2>; P<0>=P<2>+1.

RULE
Frac ::= e
SEMANTICS
V<0>=0.

RULE
Frac ::= digit Frac
SEMANTICS
V<0>=VAL<1>*10**(P<0>)+V<2>; P<2>=P<0>+1.
```

4.3. Синтактично-управляема транслация

Литература

[1] Ахо, А., Р. Сети, Дж. Ульман, Компиляторы: принципы, технологии и инструменты, Пер. с англ.- М.: Издательский дом „Вильямс”, 2003, 768 с.

[2] Ахо, А., Дж. Ульман, Теория синтаксического анализа, перевода и компиляции, Том 1, Синтаксический анализ, перевод с английского В.Н.Агафонова, под редакцией В.М.Курочкина, Издательство Мир, Москва, 1978, 613 с.

[3] Брежнев, А., Трансляторы, Конспект лекций, Северодонецк, 1995, 43 с.

[4] Варсанофьев, Д., Дымченко, Основы компиляции, Материал к лекции, 1991, 42 с.

[5] Волкова, И., Т. Руденко, Формальные грамматики и языки. Элементы теории трансляции. (учебное пособие для студентов II курса), Москва, 1996, 61 с.

[6] Волосатова, Т.М., Построение компиляторов, Електронны учебны курс, 2004, http://www.techno.edu.ru/db/msg/16771/compile.htm.

[7] Гинзбург, С., Математическая теория контекстно-свободных языков, перевод с английского А. Я. Диковского и Л. С. Модиной, под редакцией А. В. Гладкого, издательство „МИР”, Москва, 1970, 327 с.

[8] Големанов, Ц., Езикови процесори, Ръководство за практически упражнения, част I: Компилатори, Русе, 1998, 82 с.

[9] Денев, Й., Р. Павлов, Я. Деметрович, Дискретна математика, ДИ „Наука и изкуство”, София, 1984, 256 с.

[10] Карпов, Ю., Теория и технология программирования. Основы построения трансляторов. – Спб.: БХВ-Петербург, 2005, 272 с.

[11] Крицкий, С., Трансляция языков программирования: синтаксис, семантика, перевод,

[12] Легалов, А., Основы разработки трансляторов, 2005.

[13] Мартыненко, Б., Языки и трансляции, Учеб. пособие-СПб.: Издательство С.-Петербургского университета, 2003, 235 с.

[14] Пентус, А., М. Пентус, Теория формальныь языков, Учебное пособие.-М.: Изд-во ЦПИ при механико-математическом ф-те МГУ, 2004, 80 с.

[15] Полетаева, И., Методы трансляции, Конспект лекции, Част 1, Новосибирск, 1997, 59 с.

[16] Романов, Е., Основы построения трансляторов. Конспект лекций, 2003.

[17] Русков, Т., Д. Станева, Компилатори и интерпретатори, Ръководство за лабораторни упражнения, Офсетно-печатна база при ТУ-Варна, Варна, 2001, 152 с.

[18] Серебряков, В., Лекции по конструированию компиляторов, 1993.

[19] Серебряков, В., Лекции по конструированию компиляторов, 1997.

[20] Серебряков, В., М. Галочкин, Основы конструирования компиляторов, 1999, 192 с.

[21] Станев, И., К. Крачанов, В. Вълева, Езикови процесори, Том 1, Авангард принт, Русе, 1998, 126 с.

[22] Станев, И., К. Крачанов, В. Вълева, Езикови процесори, Том 2, Авангард принт, Русе, 1998, 120 с.

[23] Фомичев, В., Формальные языки, грамматики и автоматы. Курс по структурному синтезу автоматов, 2003, http://www.etu.ru/misc/edu/Index.htm.

[24] Хантер, Р., Проектирование и конструирование компиляторов, перевод с английского С. М. Круговой, под редакцией В. М. Савинкова, Финансы и статистика, Москва, 1987, 234 с.

[25] Янков, Б., Транслатори и операционни системи, Техника, 1992, 232 с.