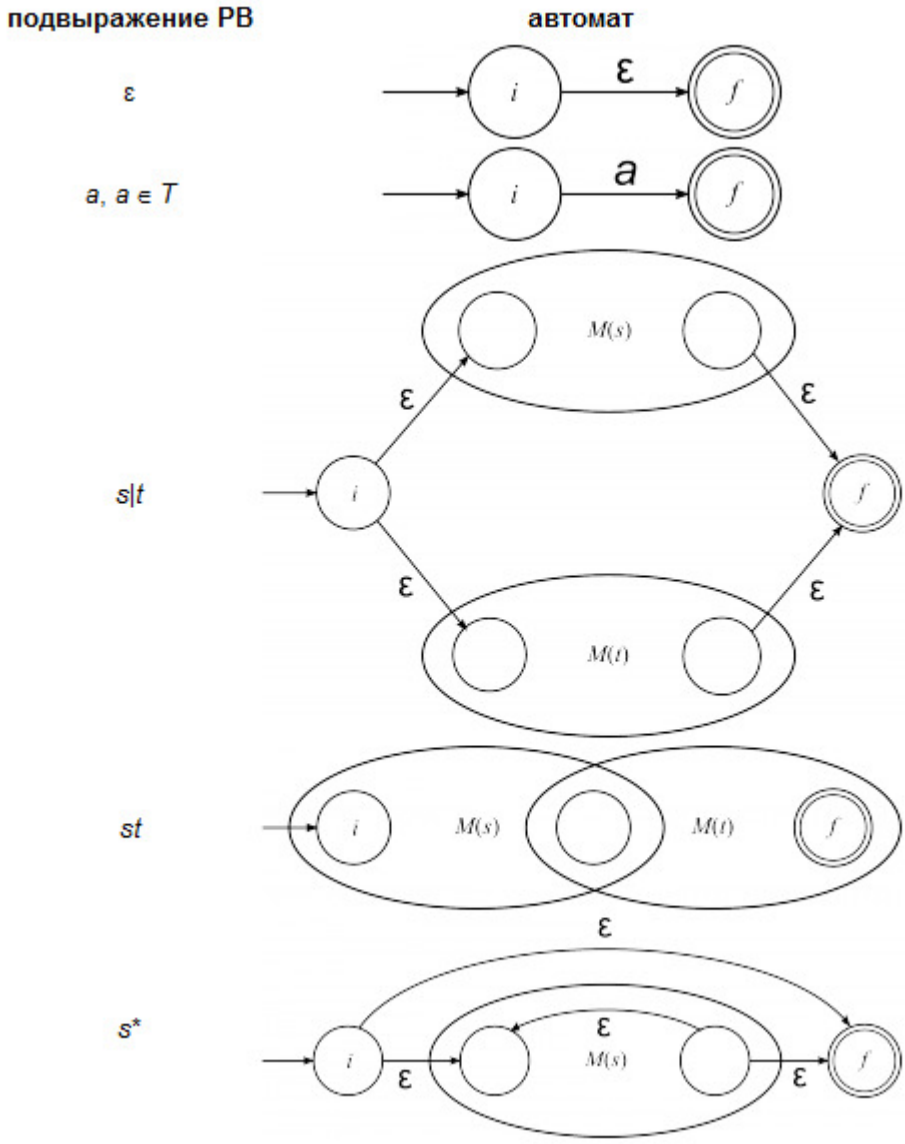
Конструирование Компиляторов, Алгоритмы решения задач

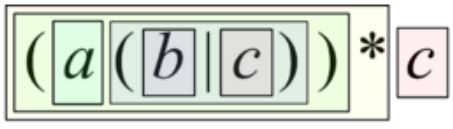
Построение НКА по РВ

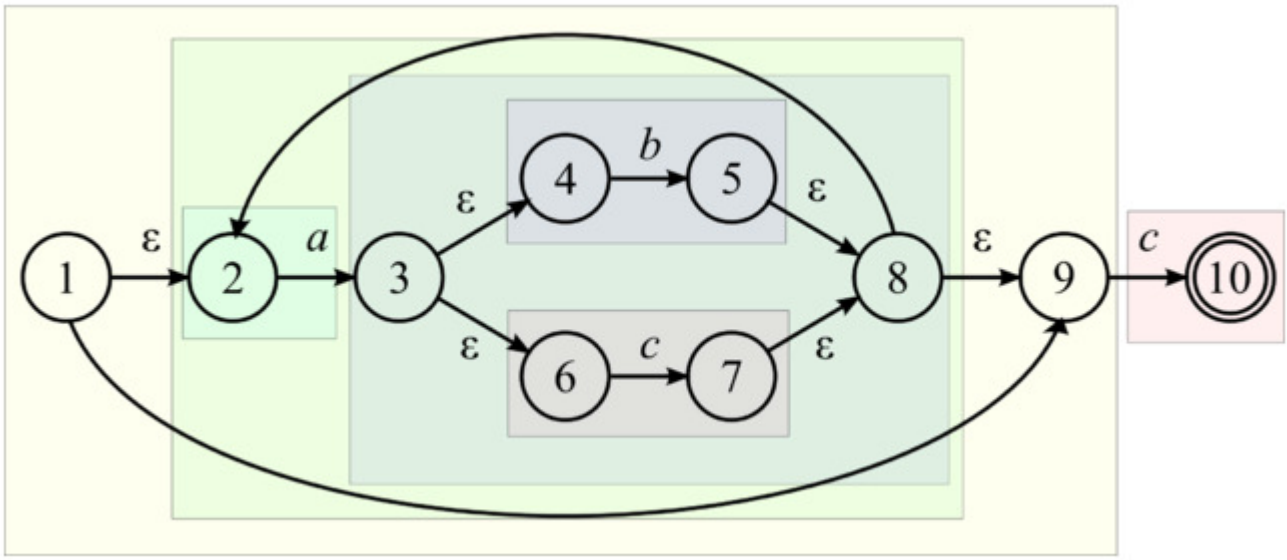
Автомат для выражения строится композицией автоматов, соответствующих подвыражениям. На каждом этапе ∃! заключительное состояние, и нет переходов из заключительного состояния и в начальное. Для построения НКА используются следующие преобразования (*M*(*s*) и *M*(*t*) ниже обозначают соответственно автоматы, соответствующие регулярным выражениям *s* и *t*; *i* и *f* — некоторые номера состояний НКА):



Пример

Обычно конечный автомат строится из регулярного выражения, начиная с внутренних символов. То есть, сначала строятся переходы по *b* и *c*, потом образуется конструкция *b*|*c*, добавляется *a*, строится автомат для итерации (*a*(*b*|*c*))* и в конце добавляется *c*.



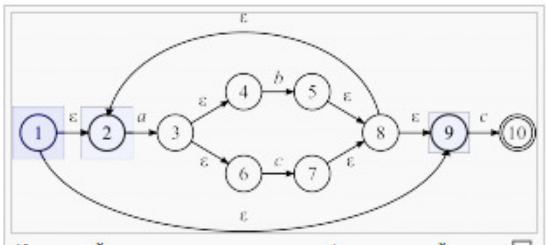


Построение ДКА по НКА

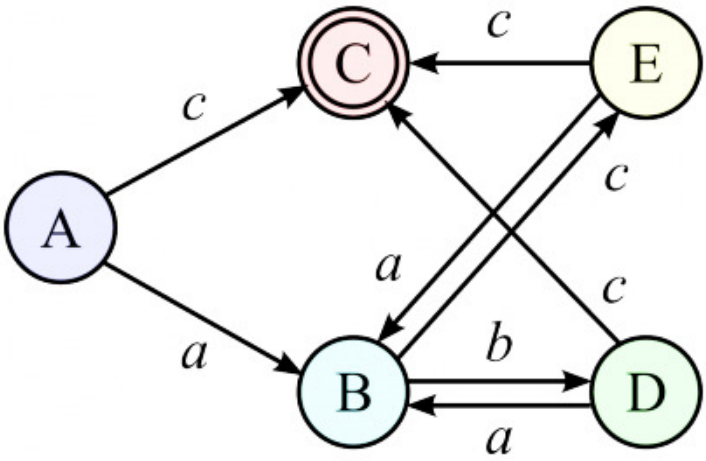
Необходимо по недетерминированному конечному автомату $M = (Q, T, D, q_0, F)$ построить детерминированный конечный автомат $M = (Q', T, D', q'_0, F')$. Начальным состоянием для строящегося автомата является ε-замыкание начального состояния автомата исходного. ε-замыкание — множество правил, которые достижимы из данного путём переходов по ε. Далее, пока есть состояния, для которых не построены переходы (переходы делаются по символам, переходы по которым есть в исходном автомате), для каждого символа вычисляется ε-замыкание множества состояний, которые достижимы из рассматриваемого состояния путём перехода по рассматриваемому символу. Если состояние, которое соответствует найденному множеству, уже есть, то добавляется переход туда. Если нет, то добавляется новое полученное состояние.

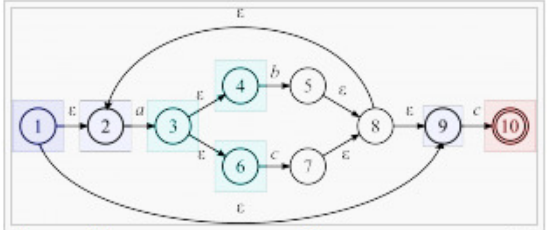
Пример

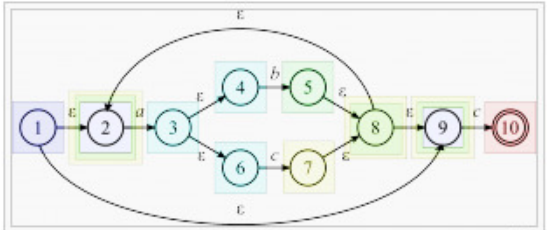
Состояние ДКА	Множество состояний НКА	Символы, по которым осуществляется переход		
		a	***b***	***c***
A	{1, 2, 9}	B	-	C
B	{3, 4, 6}	-	D	E
C	{10}	-	-	-
D	{2, 5, 8, 9}	B	-	C
E	{2, 7, 8, 9}	B	-	C


Конечный автомат после пометки tg состояний, соответствующих ε-замыканию начального

Результат:




Конечный автомат после первой итерации

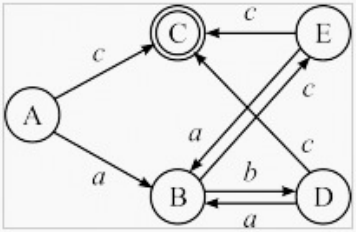

Конечный автомат после второй итерации

Построение праволинейной грамматики по конечному автомату

Каждому состоянию ставим в соответствие нетерминал. Если есть переход из состояния X в состояние Y по a, добавляем правило $X \to aY$. Для конечных состояний добавляем правила $X \to \varepsilon$. Для ε-переходов — $X \to Y$.

Пример 1 (детерминированный конечный автомат)

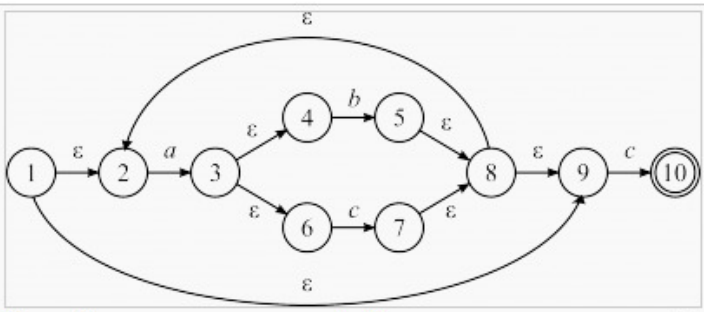
- A → *a*B | *c*C
- B → *b*D | *c*E
- C → ε
- D → *a*B | *c*C
- E → *a*B | *c*C



Пример 1 для построения праволинейной грамматики по конечному автомату

Пример 2 (недетерминированный конечный автомат)

- 1 → 2 | 9
- 2 → *a*3
- 3 → 4 | 6
- 4 → *b*5
- 5 → 8
- 6 → *c*7
- 7 → 8
- 8 → 2 | 9
- 9 → *c*10
- 10 → ε



Пример 2 для построения праволинейной грамматики по конечному автомату

Построение ДКА по РВ

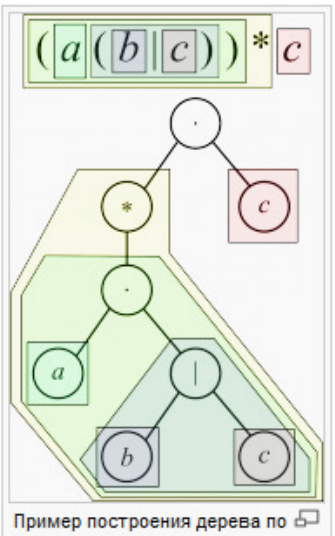
Пусть есть регулярное выражение r. По данному регулярному выражению необходимо построить детерминированный конечный автомат D такой, что $L(D) = L(r)$.

Модификация регулярного выражения

Добавим к нему символ, означающий конец РВ — «#». В результате получим регулярное выражение $(r)\#$.

Построение дерева

Представим регулярное выражение в виде дерева, листья которого — терминальные символы, а внутренние вершины — операции конкатенации «.», объединения «∪» и итерации «*». Каждому листу дерева (кроме ε-листьев) припишем уникальный номер и ссылаться на него будем, с одной стороны, как на позицию в дереве и, с другой стороны, как на позицию символа, соответствующего листу.

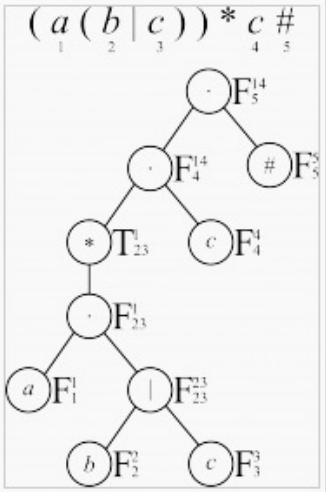


Пример построения дерева по регулярному выражению

Вычисление функций nullable, firstpos, lastpos

Теперь, обходя дерево T снизу вверх слева-направо, вычислим три функции: *nullable*, *firstpos*, и *lastpos*. Функции *nullable*, *firstpos* и *lastpos* определены на узлах дерева. Значением всех функций, кроме *nullable*, является множество позиций. Функция *firstpos*(*n*) для каждого узла n синтаксического дерева регулярного выражения дает множество позиций, которые соответствуют первым символам в подцепочках, генерируемых подвыражением с вершиной в *n*. Аналогично, *lastpos*(*n*) дает множество позиций, которым соответствуют последние символы в подцепочках, генерируемых подвыражениями с вершиной *n*. Для узлов *n*, поддеревья которых (т. е. дерево, у которого узел *n* является корнем) могут породить пустое слово, определим *nullable*(*n*) = *true*, а для остальных узлов *false*. Таблица для вычисления *nullable*, *firstpos*, *lastpos*:

узел *n*	*nullable*(*n*)	*firstpos*(*n*)	*lastpos*(*n*)
ε	*true*	∅	∅
$i \neq \varepsilon$	*false*	{*i*}	{*i*}
u ∪ v	*nullable*(*u*) or *nullable*(*v*)	*firstpos*(*u*) ∪ *firstpos*(*v*)	*lastpos*(*u*) ∪ *lastpos*(*v*)
u . v	*nullable*(*u*) and *nullable*(*v*)	if *nullable*(*u*) then *firstpos* (*u*) ∪ *firstpos*(*v*) else *firstpos*(*u*)	if *nullable*(*v*) then *lastpos* (*u*) ∪ *lastpos*(*v*) else *lastpos*(*v*)
v*	*true*	*firstpos*(*v*)	*lastpos*(*v*)



Разметка дерева

Построение followpos

Функция *followpos* вычисляется через *nullable*, *firstpos* и *lastpos*. Функция *followpos* определена на множестве позиций. Значением *followpos* является множество позиций. Если *i* — позиция, то *followpos*(*i*) есть множество позиций *j* таких, что существует некоторая строка *...cd...*, входящая в язык, описываемый РВ, такая, что *i* соответствует этому вхождению *c*, а *j* — вхождению *d*. Функция *followpos* может быть вычислена также за один обход дерева по следующим двум правилам

1. Пусть *n* — внутренний узел с операцией «.» (конкатенация); *a*, *b* — его потомки. Тогда для каждой позиции *i*, входящей в *lastpos*(*a*), добавляем к множеству значений *followpos*(*i*) множество *firstpos*(*b*).
2. Пусть *n* — внутренний узел с операцией «*» (итерация), *a* — его потомок. Тогда для каждой позиции *i*, входящей в *lastpos* (*a*), добавляем к множеству значений *followpos*(*i*) множество *firstpos*(*a*).

Пример

Вычислить значение функции *followpos* для регулярного выражения (*a*(*b*|*c*))**c*.

Позиция	Значение *followpos*
1:	(***a***(*b*|*c*))**c* {2, 3}
2:	(*a*(***b***|*c*))**c* {1, 4}
3:	(*a*(*b*|***c***))**c* {1, 4}
4:	(*a*(*b*|*c*))*****c*** {5}

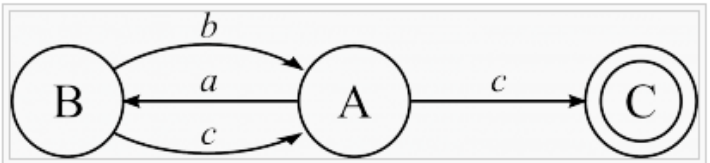
Построение ДКА

ДКа представляет собой множество состояний и множество переходов между ними. Состояние ДКА представляет собой множество позиций. Построение ДКА заключается в постепенном добавлении к нему необходимых состояний и построении переходов для них. Изначально имеется одно состояние, *firstpos*(*root*) (*root* — корень дерева), у которого не построены переходы. Переход осуществляется по символам из регулярного выражения. Каждому символу соответствует множество позиций $\{p_j\}$. Объединение *followpos* позиций всех символов, входящих в данное состояние и есть состояние в которое необходимо перейти. Если такого состояния нет, то его необходимо добавить. Процесс необходимо повторять, пока не будут построены все переходы для всех состояний.

Пример

Построить ДКА по регулярному выражению (*a*(*b*|*c*))**c*.

Состояние ДКА	Символ		
	a {1}	**b {2}**	**c {3, 4}**
A {1, 4}	B {2, 3}	—	C {5}
B {2, 3}	—	A {1, 4}	A {1, 4}
C {5}	—	—	—



ДКА, полученный из РВ (*a*(*b*|*c*))**c*

Построение ДКА с минимальным количеством состояний

Инициализация

Разобъём множество состояний на две группы: заключительные состояния ($q \in F$) и остальные ($q \in S \setminus F$).

Построение разбиения

Каждую группу G из текущего разбиения разбиваем на подгруппы так, чтобы состояния s и t из G оказались в одной группе тогда и только тогда, когда для каждого входного символа a состояния s и t имеют переходы по a в состояния из одной и той же группы в исходном разбиении. Полученные подгруппы добавляем в новое разбиение. Повторяем эту операцию для разбиения, заменяя текущее новым, пока разбиение не перестанет меняться.

Построение приведённого автомата

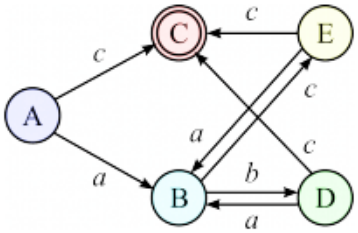
Выберем по одному состоянию из каждой группы в полученном разбиении в качестве представителя для этой группы. Представители будут состояниями приведенного ДКА M. Пусть s — представитель. Предположим, что на входе a в M существует переход из t. Пусть r — представитель группы t. Тогда M имеет переход из s в r по a. Пусть начальное состояние M — представитель группы, содержащей начальное состояние s_0 исходного автомата, и пусть заключительные состояния M — представители в F. Отметим, что каждая группа полученного разбиения либо состоит только из состояний из F, либо не имеет состояний из F.

Удаление лишних состояний

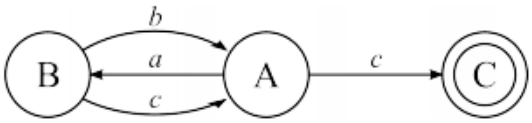
Если M имеет мертвое состояние, т. е. состояние d, которое не является допускающим и которое имеет переходы в себя по любому символу, удалим его из M. Удалим также все состояния, не достижимые из начального.

Пример

Для построим ДКА с минимальным числом состояния для ДКА следующего вида:



- Инициализация: {C} *конечное состояние* {A,B,D,E} *все остальные состояния*
- {C} *без изменений* {A,D,E}, {B}, *так как из A,D,E по a,c переходим в B и C соответственно*
- больше никаких разбиений сделать не можем.
- Пусть группе {C} соответствует состояние C, группе {A,D,E} - состояние A, а группе {B} - состояние B. Тогда получаем ДКА с минимальным числом состояний:



Построение LL(k) анализатора

Преобразование грамматики

Не всякая грамматика является LL(k)-анализируемой. Грамматика принадлежит классу LL(1), если в ней нет левых рекурсий и проведена левая факторизация. Иногда удаётся преобразовать не LL(1)-грамматики так, чтобы они стали LL(1). Некоторые (точнее, те, которые рассматривались в курсе) преобразования приведены ниже.

Удаление левой рекурсии

Пусть у нас имеется правило вида (здесь и далее в этом разделе, заглавные буквы — *нетерминальные символы*, строчные — *цепочки любых символов*):

- A → A*a* | A*b* | ... | A*k* | *m* | *n* | ... | *z*

Оно не поддается однозначному анализу, поэтому его следует преобразовать.

Легко показать, что это правило эквивалентно следующей паре правил:

- A → *m*B | *n*B | ... | *z*B
- B → *a*B | *b*B | ... | *k*B | ε

Левая факторизация

Суть данной процедуры — устранение неоднозначности в выборе правил по левому символу. Для этого находится общий левый префикс и то, что за ним может следует выносится в новое правило (строчные буквы — *цепочки любых символов*)

Пример

- A → ***ac*** | ***adf*** | ***adg*** | *b*

Преобразуется в

- A → *a*B | *b*
- B → *c* | ***df*** | ***dg***

Что в свою очередь превратится в

- A → *a*B | *b*
- B → *c* | *d*C
- C → *f* | *g*

Пример преобразования грамматики

G = {{S, A, B}, {a, b, c}, P, S}

P:

- S → SAbB | a
- A → ab | aa | ε
- B → c | ε

Удаление левой рекурсии для S:

- $S \rightarrow aS_1$
- $S_1 \rightarrow AbBS_1 \mid \varepsilon$

- $S \to aS_1$
- $S_1 \to AbBS_1 \mid \varepsilon$

Левая факторизация для A:

- $A \to aA_1 \mid \varepsilon$
- $A_1 \to b \mid a$

Итоговая грамматика:

- $S \to aS_1$
- $S_1 \to AbBS_1 \mid \varepsilon$
- $A \to aA_1 \mid \varepsilon$
- $A_1 \to b \mid a$
- $B \to c \mid \varepsilon$

Построение FIRST и FOLLOW

FIRST(α), где $\alpha \in (N \cup T)^*$ — множество терминалов, с которых может начинаться α. Если $\alpha \Rightarrow \varepsilon$, то $\varepsilon \in$ FIRST(α). Соответственно, значение функции FOLLOW(*A*) для нетерминала *A* — множество терминалов, которые могут появиться непосредственно после *A* в какой-либо сентенциальной форме. Если *A* может являться самым правым символом в некоторой сентенциальной форме, то заключительный маркер $ также принадлежит FOLLOW(*A*)

Вычисление FIRST

Для терминалов

- Для любого терминала *x*, $x \in T$, FIRST(*x*) = {*x*}

Для нетерминалов

- Если *X* — нетерминал, то положим FIRST(*X*) = {∅}
- Если в грамматике есть правило $X \to \varepsilon$, то добавим ε к FIRST(*X*)
- Для каждого нетерминала *X* и для каждого правила вывода $X \to Y_1 \dots Y_k$ добавим в FIRST(*X*) множества FIRST всех символов в правой части правила до первого, из которого не выводится ε, включая его

Для цепочек

- Для цепочки символов $X_1 \dots X_k$ FIRST есть объединение FIRST входящих в цепочку символов до первого, у которого $\varepsilon \notin$ FIRST, включая его.

Пример

Посчитать FIRST для всех нетерминалов и правил вывода грамматики:

- $S \to aS_1$
- $S_1 \to AbBS_1 \mid \varepsilon$
- $A \to aA_1 \mid \varepsilon$
- $A_1 \to b \mid a$
- $B \to c \mid \varepsilon$

FIRST нетерминалов в порядке разрешения зависимостей:

- FIRST(S) = {a}
- FIRST(A) = {a, ε}
- FIRST(A_1) = {b, a}
- FIRST(B) = {c, ε}
- FIRST(S_1) = {a, b, ε}

FIRST для правил вывода:

- FIRST(aS_1) = {a}
- FIRST($AbBS_1$) = {a, b}
- FIRST(ε) = {ε}
- FIRST(aA_1) = {a}
- FIRST(a) = {a}
- FIRST(b) = {b}
- FIRST(c) = {c}

Вычисление FOLLOW

Вычисление функции FOLLOW для символа X:

- Пусть FOLLOW(X) = {∅}
- Если X — аксиома грамматики, то добавить в FOLLOW маркер $
- Для всех правил вида A → αXβ добавить FIRST(β)\{ε} к FOLLOW(X) (за X могут следовать те символы, с которых начинается β)
- Для всех правил вида A → αX и A → αXβ, ε ∈ FIRST(β) добавить FOLLOW(A) к FOLLOW(X) (то есть, за X могут следовать все символы, которые могут следовать за A, в случае, если в правиле вывода символ X может оказаться крайним правым)
- Повторять предыдущие два пункта, пока возможно добавление символов в множество

Пример

Посчитать FOLLOW для всех нетерминалов грамматики:

- S → aS_1
- S_1 → $AbBS_1$ | ε
- A → aA_1 | ε
- A_1 → b | a
- B → c | ε

Результат:

- FOLLOW(S) = {$}
- FOLLOW(S_1) = {$} ($S_1$ — крайний правый символ в правиле S → aS_1)
- FOLLOW(A) = {b} (после A в правиле S_1 → $AbBS_1$ следует b)
- FOLLOW(A_1) = {b} (A_1 — крайний правый символ в правиле A → aA_1, следовательно, добавляем FOLLOW(A) к FOLLOW(A_1))
- FOLLOW(B) = {a, b, $} (добавляем FIRST($S_1$)\{ε} (следует из правила S_1 → $AbBS_1$), FOLLOW(S_1) (так как есть S_1 → ε))

Составление таблицы

В таблице *M* для пары нетерминал-терминал (в ячейке *M*[*A*, *a*]) указывается правило, по которому необходимо выполнять свёртку входного слова. Заполняется таблица следующим образом: для каждого правила вывода заданной грамматики A → α (где под α понимается цепочка в правой части правила) выполняются следующие действия:

1. Для каждого терминала *a* ∈ FIRST(α) добавить правило *A* → *α* к *M*[*A*, *a*]
2. Если ε ∈ FIRST(α), то для каждого *b* ∈ FOLLOW(*A*) добавить *A* → *α* к *M*[*A*, *b*]
3. ε ∈ FIRST(α) и $ ∈ FOLLOW(*A*), добавить *A* → *α* к *M*[*A*, $]
4. Все пустые ячейки — ошибка во входном слове

Пример

Построить таблицу для грамматики

- $S \to aS_1$
- $S_1 \to AbBS_1 \mid \varepsilon$
- $A \to aA_1 \mid \varepsilon$
- $A_1 \to b \mid a$
- $B \to c \mid \varepsilon$

Результат:

	a	b	c	$
S	**$S \to aS_1$** (Первое правило, вывод $S \rightharpoonup aS_1$, a ∈ FIRST(aS_1))	**Error** (Четвёртое правило)	**Error** (Четвёртое правило)	**Error** (Четвёртое правило)
S_1	**$S_1 \to AbBS_1$** (Первое правило, вывод $S_1 \rightharpoonup AbBS_1$, a ∈ FIRST($AbBS_1$))	**$S_1 \to AbBS_1$** (Первое правило, вывод $S_1 \rightharpoonup AbBS_1$, b ∈ FIRST($AbBS_1$))	**Error** (Четвёртое правило)	**$S_1 \to \varepsilon$** (Третье правило, вывод $S_1 \rightharpoonup \varepsilon$, ε ∈ FIRST(ε), $ ∈ FOLLOW(S_1))
A	**$A \to aA_1$** (Первое правило, вывод $A \rightharpoonup aA_1$, a ∈ FIRST(aA_1))	**$A \to \varepsilon$** (Второе правило, вывод $A_1 \rightharpoonup \varepsilon$, b ∈ FOLLOW($A_1$))	**Error** (Четвёртое правило)	**Error** (Четвёртое правило)
A_1	**$A_1 \to a$** (Первое правило, вывод $A_1 \rightharpoonup a$, a ∈ FIRST(a))	**$A_1 \to b$** (Первое правило, вывод $A_1 \rightharpoonup b$, b ∈ FIRST(b))	**Error** (Четвёртое правило)	**Error** (Четвёртое правило)
B	**$B \to \varepsilon$** (Второе правило, вывод $B \rightharpoonup \varepsilon$, a ∈ FOLLOW(B))	**$B \to \varepsilon$** (Второе правило, вывод $B \rightharpoonup \varepsilon$, a ∈ FOLLOW(B))	**$B \to c$** (Первое правило, вывод $B \rightharpoonup c$, c ∈ FIRST(c))	**$B \to \varepsilon$** (Третье правило, вывод $B \rightharpoonup \varepsilon$, $ ∈ FOLLOW(B))

Разбор строки

Процесс разбора строки довольно прост. Его суть в следующем: на каждом шаге считывается верхний символ *v* из стека анализатора и берется крайний символ *c* входной цепочки.

- Если *v* - терминальный символ
  - Если *v* совпадает с *c*, то они оба уничтожаются, происходит сдвиг
  - Если *v* не совпадает с *c*, то сигнализируется ошибка разбора
- Если *v* - нетерминальный символ, *c* возвращается в начало строки, вместо *v* в стек возвращается правая часть правила, которое берется из ячейки таблицы *M*[*v*, *c*]

Процесс заканчивается, когда и строка и стек дошли до концевого маркера (#).

Пример

разберем строку «aabbaabcb»:

стек	строка	действие
S#	aabbaabcb$	$S \to aS_1$
aS_1#	aabbaabcb$	сдвиг
S_1#	abbaabcb$	$S_1 \to AbBS_1$
$AbBS_1$#	abbaabcb$	$A \to aA_1$
aA_1bBS_1#	abbaabcb$	сдвиг
A_1bBS_1#	bbaabcb$	$A_1 \to b$
$bbBS_1$#	bbaabcb$	сдвиг
bBS_1#	baabcb$	сдвиг
BS_1#	aabcb$	$B \to \varepsilon$
S_1#	aabcb$	$S_1 \to AbBS_1$
$AbBS_1$#	aabcb$	$A \to aA_1$
$AbBS_1$#	aabcb$	$A \to aA_1$
aA_1bBS_1#	aabcb$	сдвиг
A_1bBS_1#	abcb$	$A_1 \to a$
$abBS_1$#	abcb$	сдвиг
bBS_1#	bcb$	сдвиг
BS_1#	cb$	$B \to c$
cS_1#	cb$	сдвиг
S_1#	b$	$S_1 \to AbBS_1$
$AbBS_1$#	b$	$A \to \varepsilon$
bBS_1#	b$	сдвиг
BS_1#	$	$B \to \varepsilon$
S_1#	$	$S_1 \to \varepsilon$
#	$	готово

Построение LR(k) анализатора

Вычисление k в LR(k)

Не существует алгоритма, который позволял бы в общем случае для произвольной грамматики вычислить k. Обычно, стоит попробовать построить LR(1)-анализатор. Если у него на каждое множество приходится не более одной операции (Shift, Reduce или Accept), то грамматика LR(0). Если же при построении LR(1)-анализатора возникает конфликт и коллизия, то данная грамматика не является LR(1) и стоит попробовать построить LR(2). Если не удаётся построить и её, то LR(3) и так далее.

Пополнение грамматики

Добавим новое правило S' → S, и сделаем S' аксиомой грамматики. Это дополнительное правило требуется для определения момента завершения работы анализатора и допуска входной цепочки. Допуск имеет место тогда и только тогда, когда возможно осуществить свёртку по правилу S → S'.

Построение канонической системы множеств допустимых LR(1)-ситуаций

В начале имеется множество I_0 с конфигурацией анализатора S' → .S, $. Далее к этой конфигурации применяется операция закрытия до тех пор, пока в результате её применения не перестанут добавляться новые конфигурации. Далее, строятся переходы в новые множества путём сдвига точки на один символ вправо (переходы делаются по тому символу, который стоял после точки до перехода и перед ней после перехода), и в эти множества добавляются те конфигурации, которые были получены из имеющихся данным образом. Для них также применяется операция закрытия, и весь процесс повторяется, пока не перестанут появляться новые множества.

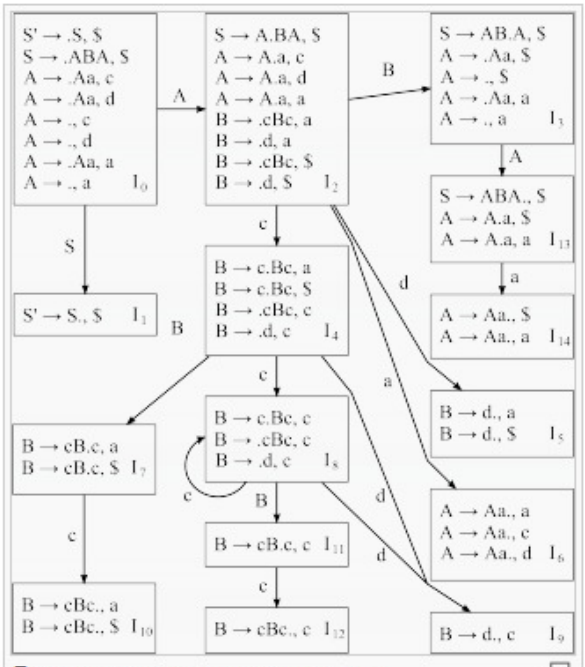
Пример

Построить каноническую систему множеств допустимых LR(1)-ситуаций для указанной грамматики:

- S' → S
- S → ABA
- A → Aa | ε
- B → cBc | d

Решение:

- Строим замыкание для конфигурации S' → .S, $:
  - S → .ABA, $
- Для полученных конфигураций (S → .ABA, $) также строим замыкание:
  - A → .Aa, c
  - A → .Aa, d
  - A → ., c
  - A → ., d
- Для полученных конфигураций (A → .Aa, c; A → .Aa, d) также строим замыкание:
  - A → .Aa, a
  - A → ., a
- Больше конфигураций в состоянии I_0 построить нельзя — замыкание построено
- Из I_0 можно сделать переходы по S и A и получить множество конфигураций I_1 и I_2, состоящее из следующих элементов:



То, что должно получиться в результате

- I_1 = {S' → S., $}
- I_2 = {S → A.BA, $; A → A.a, c; A → A.a, d; A → A.a, a}

- I_1 не требует замыкания
- Построим замыкание I_2:
  - B → .cBc, a
  - B → .cBc, $
  - B → .d, a
  - B → .d, $
- Аналогично строятся все остальные множества.

Построение таблицы анализатора

Заключительным этапом построения LR(1)-анализатора является построение таблиц *Action* и *Goto*. Таблица *Action* строится для символов входной строки, то есть для терминалов и маркера конца строки $, таблица *Goto* строится для символов грамматики, то есть для терминалов и нетерминалов.

Построение таблицы Goto

Таблица Goto показывает, в какое состояние надо перейти при встрече очередного символа грамматики. Поэтому, если в канонической системе множеств есть переход из I_i в I_j по символу A, то в Goto(I_i, A) мы ставим состояние I_j. После заполнения таблицы полагаем, что во всех пустых ячейках Goto(I_i, A) = Error

Построение таблицы Actions

- Если есть переход по терминалу a из состояния I_i в состояние I_j, то Action(I_i, a) = Shift(I_j)
- Если A ≠ S' и есть конфигруация A → α., a, то Action(I_i, a) = Reduce(A → α)
- Для состояния I_i, в котором есть конфигурация S' → S., $, Action($I_i$, $) = Accept
- Для всех пустых ячеек Action(I_i, a) = Error

Пример

Построить таблицы Action и Goto для грамматики

- S' → S
- S → ABA
- A → Aa | ε
- B → cBc | d

Решение:

	Action				Goto						
	a	**c**	**d**	**$**	**S**	**S'**	**A**	**B**	**a**	**c**	**d**
I_0	Reduce(A → ε)	Reduce(A → ε)	Reduce(A → ε)		I_1		I_2				
I_1				Accept							
I_2	Shift(I_6)	Shift(I_4)	Shift(I_5)					I_3	I_6	I_4	I_5
I_3	Reduce(A → ε)			Reduce(A → ε)			I_{13}				
I_4		Shift(I_8)	Shift(I_9)					I_7		I_8	I_9
I_5	Reduce(B → d)			Reduce(B → d)							
I_6	Reduce(A → Aa)	Reduce(A → Aa)	Reduce(A → Aa)								
I_7		Shift(I_{10})								I_{10}	
I_8		Shift(I_8)	Shift(I_9)					I_{11}		I_8	I_9
I_9		Reduce(B → d)									
I_{10}	Reduce(B → cBc)			Reduce(B → cBc)							
I_{11}		Shift(I_{12})								I_{12}	
I_{12}		Reduce(B → cBc)									
I_{13}	Shift(I_{14})			Reduce(S → ABA)					I_{14}		
I_{14}	Reduce(A → Aa)			Reduce(A → Aa)							

Разбор цепочки

На каждом шаге считывается верхний символ **v** из стека анализатора и берется крайний символ **c** входной цепочки.

Если в таблице действий на пересечении **v** и **c** находится:

- Shift($\mathbf{I_k}$), то в стек кладется **c** и затем $\mathbf{I_k}$. При этом **c** удаляется из строки.
- Reduce(**A** → **u**), то из верхушки стека вычищаются все терминальные и нетерминальные символы, составляющие цепочку **u**, после чего смотрится состояние $\mathbf{I_m}$, оставшееся на верхушке. По таблице переходов на пересечении $\mathbf{I_m}$ и **A** находится следующее состояние $\mathbf{I_s}$. После чего в стек кладется A, а затем $\mathbf{I_s}$. Строка остается без измененей.
- Accept, то разбор закончен
- пустота - ошибка

Пример

Построим разбор строки aaaccdcc:

Стек	Строка	Действие
I_0	**a**aaccdcc$	Reduce(A → ε), goto I_2
I_0 A **I_2**	**a**aaccdcc$	Shift(I_6)
I_0 A I_2 a **I_6**	**a**accdcc$	Reduce(A → Aa), goto I_2
I_0 A **I_2**	**a**accdcc$	Shift(I_6)
I_0 A I_2 a **I_6**	**a**ccdcc$	Reduce(A → Aa), goto I_2
I_0 A **I_2**	**a**ccdcc$	Shift(I_6)
I_0 A I_2 a **I_6**	**c**cdcc$	Reduce(A → Aa), goto I_2
I_0 A **I_2**	**c**cdcc$	Shift(I_4)
I_0 A I_2 c **I_4**	**c**dcc$	Shift(I_8)
I_0 A I_2 c I_4 c **I_8**	**d**cc$	Shift(I_9)
I_0 A I_2 c I_4 c I_8 d **I_9**	**c**c$	Reduce(B → d), goto I_{11}
I_0 A I_2 c I_4 c I_8 B **I_{11}**	**c**c$	Shift(I_{12})
I_0 A I_2 c I_4 c I_8 B I_{11} c **I_{12}**	**c**$	Reduce(B → cBc), goto I_7
I_0 A I_2 c I_4 B **I_7**	**c**$	Shift(I_{10})
I_0 A I_2 c I_4 B I_7 c **I_{10}**	**$**$	Reduce(B → cBc), goto I_3
I_0 A I_2 B **I_3**	**$**$	Reduce(A → ε), goto I_{13}
I_0 A I_2 B I_3 A **I_{13}**	**$**$	Reduce(S → ABA), goto I_1
I_0 S **I_1**	**$**$	Accept

Трансляция арифметических выражений (алгоритм Сети-Ульмана)

Примечание. Код генерируется ~~doggy style~~ Motorola-like, то есть

```
Op Arg1, Arg2
```

обозначает

```
Arg2 = Arg1 Op Arg2
```

Построение дерева

Дерево строится как обычно для арифметического выражения: В корне операция с наименьшим приоритетом, далее следуют операции с приоритетом чуть выше и так далее. Скобки имеют наивысший приоритет. Если имеется несколько операций с одинаковым приоритетом — a op b op c, то дерево строится как для выражения (a op b) op c.

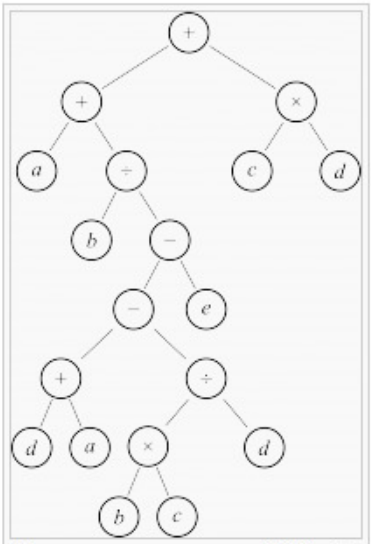
Пример

Построить дерево для выражения a + b / (d + a − b × c / d − e) + c × d

Решение: Запишем выражение в виде

((a) + ((b) / ((((d) + (a)) − ((b) × (c)) / (d)) − (e)))) + ((c) × (d))

Тогда в корне каждого поддерева будет операция, а выражения в скобках слева и справа от неё — её поддеревьями. Например, для подвыражения ((b) × (c)) / (d) в корне соответствующего поддерева будет операция «/», а её поддеревьями будут являться подвыражения ((b) × (c)) и (d).

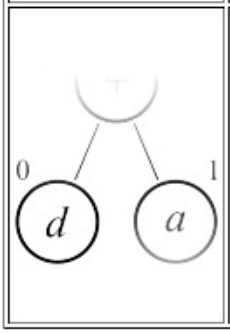
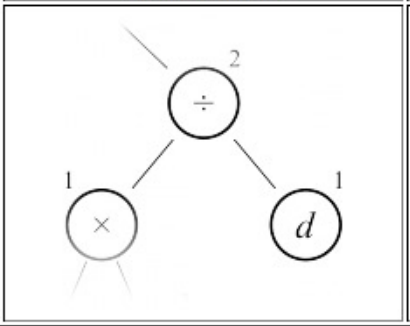
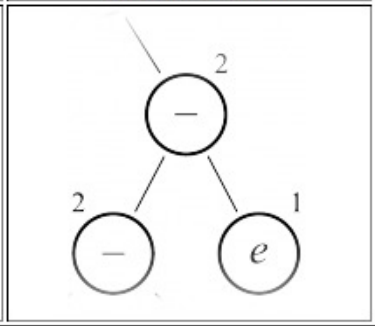
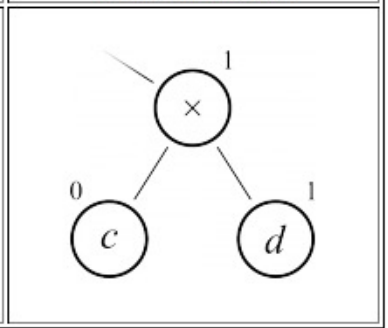


Дерево для выражения a + b / (d + a − b × c / d − e) + c × d

Разметка дерева (вычисление количества регистров)

Далее необходимо вычислить для каждого поддерева, сколько регистров потребуется для его вычисления. Делается это разметкой дерева снизу вверх по следующим правилам:

- Если вершина — левый лист (то есть, переменная), то помечаем её нулём.
- Если вершина — правый лист, то помечаем её единицей
- Если мы разметили для некоторй вершины оба её поддерева, то её размечаем следующим образом:
  - Если левое и правое поддеревья помечены разными числами, то выбираем наибольшее из них
  - Если левое и правое поддеревья помечены одинаковыми числами, то данному поддереву сопоставляем число, на единицу большее того, которым помечены поддеревья

Разметка листьев	Разметка дерева с одинаковыми поддеревьями	Левое поддерево помечено большим числом	Правое поддерево помечено большим числом
			

Пример

Разметить дерево, построенное для выражения a + b / (d + a − b × c / d − e) + c × d

Распределение регистров и генерация кода

Распределение регистров происходит следующим образом:

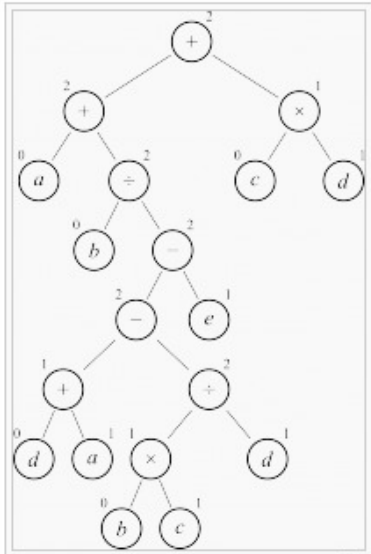
- Корню назначается R0
- Если метка **левого** потомка **меньше** или **равна** метке **правого**, то **левому** потомку назначается регистр **на единицу больший**, чем предку, а **правому — такой же**, как и предку.
- Если метка **левого** потомка **больше** метки **правого**, то **правому** потомку назначается регистр **на единицу больший**, чем предку, а **левому — такой же**, как и предку.

Формируется код путём обхода дерева снизу вверх следующим образом:

1. Для вершины с меткой 0 код не генерируется

2. Если вершина — лист X с меткой 1 и регистром R*i*, то ему сопоставляется код

```
MOVE X, Ri
```



Размеченное дерево для выражения a + b / (d + a − b × c / d − e) + c × d

3. Если вершина внутренняя с регистром R*i* и её левый потомок — лист X с меткой 0, то ей соответствует код

```
<Код правого поддерева>
Op X, Ri
```

4. Если поддеревья вершины с регистром R*i* — не листья и метка правой вершины больше или равна метке левой (у которой регистр Rj, j = i + 1), то вершине соответствует код

```
<Код правого поддерева>
<Код левого поддерева>
Op Rj, Ri
```

5. Если поддеревья вершины с регистром R*i* — не листья и метка правой вершины (у которой регистр Rj, j = i + 1) меньше метки левой, то вершине соответствует код

```
<Код левого поддерева>
<Код правого поддерева>
Op Ri, Rj
MOVE Rj, Ri
```

При этом **нельзя** сразу сделать Op Rj, Ri так как Op в общем случае некоммутативна. В случае, если Op коммутативна, делать Op Rj, Ri вместо Op Ri, Rj; MOVE Rj, Ri ~~можно~~ **всё равно нельзя**.

Общее правило: выписывать код снизу вверх сначала для вершины с большей меткой, потом с меньшей (если метки равны, то сначала для правой).

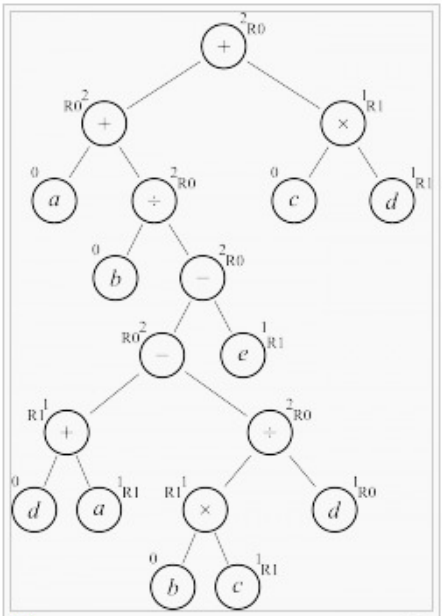
Пример

Распределить регистры и сгенерировать код для выражения a + b / (d + a − b × c / d − e) + c × d

Решение:

Распределение регистров показано на графике справа. Сгенерированный код:

```
MOVE d, R0 ;R0 = d
MOVE c, R1 ;R1 = c
MUL b, R1  ;R1 = (b × c)
DIV R1, R0 ;R0 = (b × c) / d
MOVE a, R1 ;R1 = a
ADD d, R1  ;R1 = a + d
SUB R1, R0 ;R0 = (a + d) − ((b × c) / d)
MOVE e, R1 ;R1 = e
SUB R0, R1 ;R1 = ((a + d) − ((b × c) / d)) − e
MOVE R1, R0;R0 = ((a + d) − ((b × c) / d)) − e
DIV b, R0  ;R0 = b / (((a + d) − ((b × c) / d)) − e)
ADD a, R0  ;R0 = a + (b / (((a + d) − ((b × c) / d)) − e))
MOVE d, R1 ;R1 = d
MUL c, R1  ;R1 = c × d
ADD R0, R1 ;R1 = (a + (b / (((a + d) − ((b × c) / d)) − e))) + (c × d)
MOVE R1, R0;R0 = (a + (b / (((a + d) − ((b × c) / d)) − e))) + (c × d)
```



Распределение регистров для выражения a + b / (d + a − b × c / d − e) + c × d

Трансляция логических выражений

В данном разделе показан способ генерации кода для ленивого вычисления логических выражений. В результате работы алгоритма получается кусок кода, который с помощью операций TST, BNE, BEQ вычисляет логическое выражение путём перехода на одну из меток: TRUELAB или FALSELAB.

Построение дерева

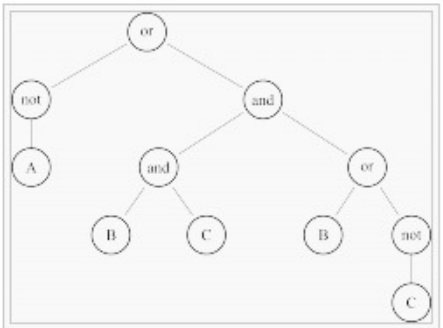
Дерево логического выражения отражает порядок его вычисления в соответствии с приоритетом операций, то есть, для вычисления значения некоего узла дерева (который есть операция от двух операндов, являющимися поддеревьями узла) мы должны сначала вычислить значения его поддеревьев.

Приоритет операций: наивысший приоритет у операции NOT, далее идёт AND, а затем OR. Если в выражении используются другие логические операции то они должны быть выражены через эти три определённым образом (обычно, других операций нет и преобразования выражения не требуется). Ассоциативность у операций одного приоритета — слева направо, то есть A and B and C рассматривается как (A and B) and C

Пример

Построить дерево для логического выражения not A or B and C and (B or not C).

Решение: см. схему справа.



дерево для логического выражения not A or B and C and (B or not C)

Разметка дерева

Для каждой вершины дерева вычисляются 4 атрибута:

- Номер узла
- Метка, куда необходимо перейти, если выражение в узле — истина (true label, tl)
- Метка, куда необходимо перейти, если выражение в узле — ложь (false label, fl)
- Метка-знак (sign) (подробнее см. далее)

Нумерация вершин выполняется в произвольном порядке, единственным условием является уникальность номеров узлов.

Разметка дерева производится следующим образом:

- В tl указывается номер вершины, в который делается переход или truelab, если данная вершина true
- В fl указывается номер вершины, в который делается переход или falselab, если данная вершина false

Sign указывает то, в каком случае можно прекратить вычисления текущего поддерева.

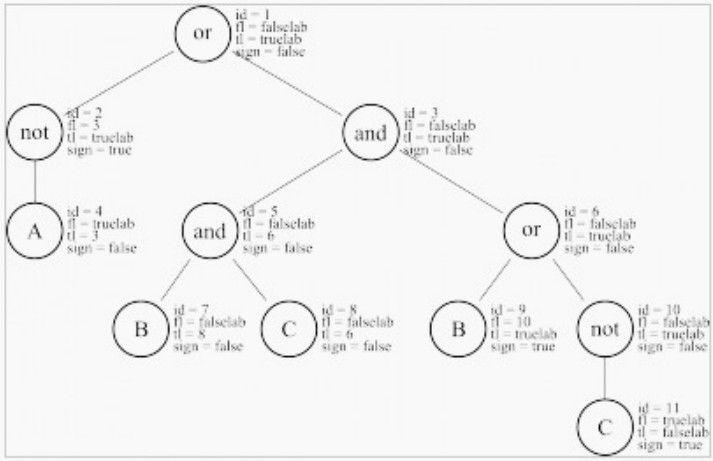
Для корня дерева tl=truelab, fl=falselab, sign=false.

Таким образом:

Операнд	fl	tl	sign
Левый операнд OR'a	Номер правого операнда	tl родительского узла	true
Правый операнд OR'a	fl родительского узла	tl родительского узла	sign родительского узла
Левый операнд AND'a	fl родительского узла	Номер правого операнда	false
Правый операнд AND'a	fl родительского узла	tl родительского узла	sign родительского узла
Операнд NOT'a	tl родительского узла	fl родительского узла	отрицание sign родительского узла

Пример

Разметить дерево, построенное для логического выражения not A or B and C and (B or not C).



Размеченное дерево для логического выражения not A or B and C and (B or not C)

Генерация кода

Машинные команды, используемы в сгенерированном коде:

- **TST <boolean value>** — проверка аргумента на истинность и установка флага, если аргумент ложен
- **BNE <label>** — переход по метке в случае, если флаг не установлен, то есть проверяенное при помощи TST условие **истинно**
- **BEQ <label>** — переход по метке в случае, если флаг установлен, то есть проверяенное при помощи TST условие **ложно**

Построение кода производится следующим образом:

- дерево обходится от корня, для AND и OR сначала обходится левое поддерево затем правое
- для каждой пройденной вершины печатается ее номер (метка)
- для листа A(номер, tl, fl, sign) печатается TST A
  - если sign == true, печатается BNE tl
  - если sign == false, печатается BEQ fl

Пример

Для рассмотренного выше выражения сгенерится следующий код:

```
1:2:4:    TST A
          BEQ TRUELAB
3:5:7:    TST B
          BEQ FALSELAB
8:        TST C
          BEQ FALSELAB
6:9:      TST B
          BNE TRUELAB
10:11:    TST C
          BNE FALSELAB
TRUELAB:
FALSELAB:
```

Метод сопоставления образцов

Идея метода заключается в том, что для одного и того же участка программы может быть код сгенерирован различными способами, и, как следствие, можно добиться оптимизации по тому или иному параметру.

Постановка задачи

Имеется множество образцов, для каждого из которых определён кусок промежуточного представления, для которого он применим, вес и генерируемый код. Есть дерево промежуточного представления, представляющее собой фрагмент программы, для которой необходимо код сгенерировать. Целью является построение такого покрытия дерева промежуточного представления образцами, чтобы суммарный вес образцов был минимален.

Образцы - это ассемблерные команды и деревья разбора, которые им соответствуют. Про каждый образец известно время его выполнения (в тактах). С их помощью и будем генерировать оптимальный (по времени выполнения) код.

Примеры образцов

Построение промежуточного представления

Сначала строим дерево разбора для всего выражения.

Построение покрытия

Теперь для каждой вершины (перебираем их в порядке от листьев к корню) будем генерировать оптимальный код для ее поддерева. Для этого просто переберем все образцы, применимые в данной вершине. Время выполнения при использовании конкретного образца будет складываться из времени вычисления его аргументов (а оптимальный код для их вычисления мы уже знаем благодаря порядку обхода дерева) и времени выполнения самого образца. Из всех полученных вариантов выбираем лучший - он и будет оптимальным кодом для поддерева данной вершины. В корне дерева получим оптимальный код для всего выражения.

Генерация кода

Код для всех вершин выписывать не обязательно - достаточно записать минимальное необходимое время и образец, которым нужно воспользоваться. Все остальное из этого легко восстанавливается.

Регистров у нас в этих задачах бесконечное количестов, так что каждый раз можно использовать новый.

Построение РВ по ДКА

Построение НКА по праволинейной грамматике

Приведение грамматики

Для того чтобы преобразовать произвольную КС-грамматику к приведенному виду, необходимо выполнить следующие действия:

- удалить все бесплодные символы;
- удалить все недостижимые символы;

Удаление бесполезных символов

Вход: КС-грамматика G = (T, N, P, S).

Выход: КС-грамматика G' = (T, N', P', S), не содержащая бесплодных символов, для которой L(G) = L(G').

Метод:

Рекурсивно строим множества N_0, N_1, ...

1. $N_0 = \varnothing$, i = 1.
2. $N_i = \{A \mid (A \to \alpha) \in P \text{ и } \alpha \in (N_{i-1} \cup T)^*\} \cup N_{i-1}$.
3. Если $N_i \neq N_{i-1}$, то i = i + 1 и переходим к шагу 2, иначе N' = N_i; P' состоит из правил множества P, содержащих только символы из N' ∪ T; G' = (T, N', P', S).

Определение: символ $x \in (T \cup N)$ называется недостижимым в грамматике G = (T, N, P, S), если он не появляется ни в одной сентенциальной форме этой грамматики.

Пример

Удалить бесполезные символы у грамматики G({A, B, C, D, E, F, S}, {a, b, c, d, e, f, g}, P, S)

- S → AcDe | CaDbCe | SaCa | aCb | dFg
- A → SeAd | cSA
- B → CaBd | aDBc | BSCf | bfg
- C → Ebd | Seb | aAc | cfF
- D → fCE | ac | dEdAS | ε
- E → ESacD | aec | eFf

Решение

- $N_0 = \varnothing$
- N_1 = {B (B → bfg), D (D → ac), E (E → aec)}
- N_2 = {B, D, E, C (C → Ebd)}
- N_3 = {B, D, E, C, S (S → aCb)}
- $N_4 = \{B, D, E, C, S\} = N_3$

G'({B, C, D, E, S}, {a, b, c, d, e, f, g}, P', S)

- S → CaDbCe | SaCa | aCb
- B → CaBd | aDBc | BSCf | bfg
- C → Ebd | Seb
- D → fCE | ac | ε
- E → ESacD | aec

Удаление недостижимых символов

Вход: КС-грамматика G = (T, N, P, S)

Выход: КС-грамматика G' = (T', N', P', S), не содержащая недостижимых символов, для которой L(G) = L(G').

Метод:

1. $V_0 = \{S\}$; i = 1.
2. $V_i = \{x \mid x \in (T \cup N), (A \to \alpha x \beta) \in P \text{ и } A \in V_{i-1}\} \cup V_{i-1}$.
3. Если $V_i \neq V_{i-1}$, то i = i + 1 и переходим к шагу 2, иначе $N' = V_i \cap N$; $T' = V_i \cap T$; P' состоит из правил множества P, содержащих только символы из V_i; G' = (T', N', P', S).

Определение: КС-грамматика G называется приведенной, если в ней нет недостижимых и бесплодных символов.

Пример

Удалить недостижимые символы у грамматики G'({B, C, D, E, S}, {a, b, c, d, e, f, g}, P', S)

- S → CaDbCe | SaCa | aCb
- B → CaBd | aDBc | BSCf | bfg
- C → Ebd | Seb
- D → fCE | ac | ε
- E → ESacD | aec

Решение

- $V_0 = \{S\}$
- V_1 = {S, C (S → CaDbCe), D (S → CaDbCe), a (S → CaDbCe), b (S → CaDbCe), e (S → CaDbCe)}
- V_2 = {S, C, D, a, b, e, E (C → Ebd), d (C → Ebd), f (D → fCE)}
- V_3 = {S, C, D, E, a, b, d, e, f, c (E → ESacD)}
- $V_4 = \{S, C, D, E, a, b, d, e, f, c\} = V_3$

G''({C, D, E, S}, {a, b, c, d, e, f}, P'', S)

- S → CaDbCe | SaCa | aCb
- C → Ebd | Seb
- D → fCE | ac | ε
- E → ESacD | aec