Министерство образования и науки РФ
Государственное образовательное учреждение
Высшего и профессионального образования
Иркутский Государственный университет
Институт математики, экономики и информатики

# Дополнительные главы теории вычислительных процессов и структур

учебное пособие

2014

Учебное пособие предназначено для студентов обучающихся по направлению подготовки бакалавров 0100400.62 «Прикладная математика и информатика». Может быть использовано в курсах «Основы информатики», «Языки и методы программирования» и «Теория вычислительных процессов и структур»

**Составитель:** Мезенцев А.В.

# Оглавление.

# Глава 1. Нисходящий разбор без возвратов. LL(1)-грамматики

LL($k$)-грамматики позволяют выполнить нисходящий синтаксический анализ, просматривая входную цепочку *слева* (первое L) при восстановлении *левого* канонического вывода (второе L) данной терминальной цепочки заглядывая вперед по входной цепочке на каждом шаге не более чем на $k$ символов при принятии решения о том, какой из альтернативных правых частей заменить текущий – самый левый – нетерминал очередной сентенциальной формы.

Формально, КС-грамматика G называется LL($k$)-грамматикой, если для любой сентенциальной формы $\omega A\alpha \in \{V_N \cup V_T\}^*$ и первых $k$ терминальных символов выводимых из $A\alpha$ существует не более одного правила, которое можно применить к нетерминалу $A$, чтобы получить левый вывод цепочки, начинающейся с $\omega$ и продолжающейся $k$ терминалами.

На практике обычно используют $k$=1, то есть LL(1)-грамматики.

Основная проблема при *нисходящем разборе без возвратов* – как осуществить альтернативный выбор правил. Пусть $A$ текущий нетерминал. И имеется n правил: $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$. Если хотя бы одна из правых частей $\alpha_i$ начинается с нетерминала, следует рассматривать множества $FIRST(\alpha_i)$.

$FIRST(\alpha)$ определяет множество тех терминальных символов, с которых могут начинаться цепочки, выводимые из $\alpha$. Если $\alpha \Rightarrow^* \varepsilon$, то $\varepsilon \in FIRST(\alpha)$. Формально, $FIRST(\alpha) = \{a \in V_T \mid \alpha \Rightarrow^* a\beta\} \cup \{\varepsilon \mid \alpha \Rightarrow^* \varepsilon\}$.

Эти множества должны быть непересекающимися. Если при этом среди альтернатив нетерминала $A$ есть пустая цепочка, то каждое из множеств $FIRST(\alpha_i)$ должно не пересекаться с множеством $FOLLOW(A)$.

$FOLLOW(A)$ для нетерминала $A$ определяется как множество таких терминальных символов, которые могут следовать за $A$ в какой-нибудь сентенциальной форме. Формально, $FOLLOW(A) = \{a \in V_T \mid S \Rightarrow^* aA\beta, a \in FIRST(\beta)\}$ для начального символа $S$ грамматики и $\alpha, \beta \in (V_T \cup V_N)^*$.

Пример 1.1. Дана грамматика:

$$S \to AbC \mid Baa$$

$$A \to CS \mid cBa$$

$$B \to ac$$

$$C \to b \mid dC$$

Пусть дан левый вывод цепочки:

$S \Rightarrow AbC \Rightarrow CSbC \Rightarrow dCSbC \Rightarrow dbSbC \Rightarrow dbBaabC \Rightarrow dbacaabC \Rightarrow dbacaabb$

Попробуем восстановить его, начиная с $S$.

Имеем $S \Rightarrow ? \Rightarrow dbacaabb$

Какой из альтернативных правых частей ($AbC$ или $Baa$) следует заменить текущий нетерминал ($S$) для восстановления вывода этой цепочки?

Глядя на *dbacaabb* выбрать непосредственно нельзя. Используем функцию $FIRST(\alpha)$ для каждой альтернативы. Построим функции *FIRST* для всех нетерминалов нашей грамматики. Легче всего начать с $B$:

$FIRST(B)=FIRST(ac)=\{a\}$

$FIRST(C) = \{b, d\}$

$FIRST(A)=FIRST(CS) \cup FIRST(cBa)=\{b, d\} \cup \{c\}$

$FIRST(S)=FIRST(AbC) \cup FIRST(Baa)=\{b, c, d\} \cup \{a\}=\{a, b, c, d\}$

Выбор $S \Rightarrow AbC \Rightarrow \ldots$ или $S \Rightarrow Baa \Rightarrow \ldots$ прост. Так как, первый терминал $d$, а $d \in FIRST(A)$ и $d \notin FIRST(B)$. Если бы первый терминал входной цепочки не принадлежал $FIRST(S)$, то можно было бы говорить об ошибке в первом же символе входной цепочки.

Алгоритм разбора LL(1)-грамматик работает с магазином (стеком). В магазин помещается начальный нетерминал $S$. По грамматике строится таблица решений, показывающая, какую цепочку следует записать в магазин вместо

его верхнего символа для каждой пары (верхний символ магазина, очередной терминал входной цепочки):

| | очередной терминал входной цепочки | | | | |
|---|---|---|---|---|---|
| верхний символ магазина | | *a* | *b* | *c* | *d* |
| | *S* | *Baa* | *AbC* | *AbC* | *AbC* |
| | *A* | ошибка | *CS* | *cBa* | *CS* |
| | *B* | *ac* | ошибка | ошибка | ошибка |
| | *C* | ошибка | *b* | ошибка | *dC* |
| | *a* | ε, сдвиг | ошибка | ошибка | ошибка |
| | *b* | ошибка | ε, сдвиг | ошибка | ошибка |
| | *c* | ошибка | ошибка | ε, сдвиг | ошибка |
| | *d* | ошибка | ошибка | ошибка | ε, сдвиг |

Как и любая другая грамматика, использующая нисходящий разбор, LL(1)-грамматика не должна содержать леворекурсивных правил. Если такие правила есть их надо модифицировать так, что бы получилась эквивалентная грамматика не содержащая леворекурсивных правил.

Пусть в исходной грамматике G=(N, T, P, S) для нетерминала $A$ есть леворекурсивные правила:

$A \to A\alpha_1|A\alpha_2|\ldots|A\alpha_m|\beta_1|\beta_2|\ldots|\beta_n$, где $\beta_i$ $(1 \le i \le n)$ не начинается с $A$.

Построим грамматику G′=(N′, T, P′, S) в которой заменим данные правила на следующие:

$A \to \beta_1 A'|\beta_2 A'|\ldots|\beta_n A'$

$A' \to \alpha_1 A'|\alpha_2 A'|\ldots|\alpha_m A'|\varepsilon$, где $A' \in$ N новый нетерминал.

Построенная таким образом грамматика G′ будет эквивалентна исходной, то есть будет порождать то же самое множество цепочек, что и грамматика G.

Пример 1.2. Пусть дана грамматика:

$$E \to E+T|T$$

$T \rightarrow T*F|F$

$F \rightarrow (E)|a$

Эта грамматика леворекурсивна. Преобразуем ее правила:

$E \rightarrow TE'$

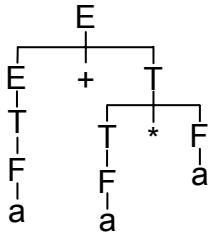
$E' \rightarrow +TE'|\varepsilon$

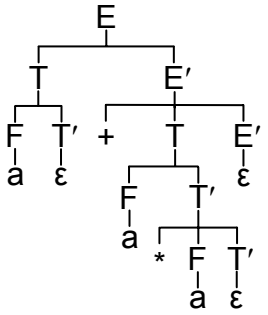
$T \rightarrow FT'$

$T' \rightarrow *FT'|\varepsilon$

$F \rightarrow (E)|a$

Деревья синтаксической структуры для одной и той же сентенциальной формы $a+a*a$ отличаются:




Однако, цепочки составленные из листьев этих деревьев совпадают.

Рассмотрим более формально, как вычислять функции *FIRST* и *FOLLOW*.

***Вычисление функции FIRST(X).***

Применять следующие правила до тех пор, пока ни к какому из множеств *FIRST* не смогут быть добавлены ни терминалы, ни ε.

1. Если $X$ – терминал, то положить $FIRST(X) = \{X\}$.
2. Если имеется продукция $X \rightarrow \varepsilon$, добавить ε к $FIRST(X)$.
3. Если $X$ – нетерминал и $X \rightarrow Y_1 Y_2 \ldots Y_k$, то добавить $a$ в $FIRST(X)$, если для некоторого $i$ $a \in FIRST(Y_i)$ и ε входит во все множества $FIRST(Y_1)$, …,

$FIRST(Y_{i-1})$, т. е. $Y_1 \ldots Y_{i-1} \Rightarrow^* \varepsilon$. Если ε имеется во всех $FIRST(Y_i)$ $1 \le i \le k$, то добавить ε к $FIRST(X)$.

**Вычисление $FIRST(X_1X_2 \ldots X_n)$.**

Добавить к $FIRST(X_1X_2 \ldots X_n)$ все не ε символы из $FIRST(X_1)$. Добавить также все не ε символы из $FIRST(X_2)$, если $\varepsilon \in FIRST(X_1)$, все не ε символы из $FIRST(X_3)$, если $\varepsilon \in FIRST(X_1)$ и $\varepsilon \in FIRST(X_2)$ и т. д. И, наконец, добавить ε к $FIRST(X_1X_2 \ldots X_n)$, если $\forall i \ \varepsilon \in FIRST(X_i)$.

**Вычисление функции $FOLLOW(A)$ для всех нетерминалов.**

Применять следующие правила до тех пор, пока ни к одному из множеств *FOLLOW* нельзя будет добавить ни одного символа.

1. Поместить маркер конца входной строки в $FOLLOW(S)$, где $S$ начальный символ грамматики.
2. Если имеется продукция $A \to \alpha B \beta$, то все элементы множества $FIRST(\beta)$, кроме ε, поместить в множество $FOLLOW(B)$.
3. Если имеется продукция $A \to \alpha B$, или $A \to \alpha B \beta$, где $\varepsilon \in FIRST(\beta)$ (т.е. $\beta \Rightarrow^* \varepsilon$), то все элементы из множества $FOLLOW(A)$ поместить в множество $FOLLOW(B)$.

Пример 1.3. Построим функции *FIRST* и *FOLLOW* для преобразованной грамматики из примера 1.2:

$FIRST(a) = \{a\}$

$FIRST(() = \{(\}$

$FIRST(*) = \{*\}$

$FIRST(+) = \{+\}$

$FIRST()) = \{)\}$ Здесь мы использовали первое правило для *FIRST*.

Второе правило дает нам $FIRST(E') = \{\varepsilon\}$ и $FIRST(T') = \{\varepsilon\}$.

Третье правило итерационное, мы должны применять еще раз, если на предыдущей итерации, хотя бы одно из множеств пополнилось.

Первая итерация: продукция E′→+TE′ добавляет к *FIRST*(E′) все что есть в *FIRST*(+), т.е. *FIRST*(E′) = {ε, +}; продукция T′→∗FT′ дает *FIRST*(T′) = {ε, ∗}; продукции F→(E) и F→*a* дают *FIRST*(F) = {(, *a*}.

Вторая итерация: продукция T→FT′ дает *FIRST*(T) = {(, *a*}.

Третья итерация: продукция E→TE′ дает *FIRST*(E) = {(, *a*}.

Четвертая итерация ничего не добавляет, построение *FIRST* окончено:

*FIRST*(*a*) = {*a*}

*FIRST*(() = {(}

*FIRST*(*) = {∗}

*FIRST*(+) = {+}

*FIRST*()) = {)}

*FIRST*(E) = {(, *a*}

*FIRST*(E′) = {ε, +}

*FIRST*(T) = {(, *a* }

*FIRST*(T′) = {ε, ∗}

*FIRST*(F) = {(, *a* }

Строим функцию *FOLLOW*:

Первое правило дает *FOLLOW*(E) = {$}, здесь $ - маркер конца входной строки (любая разбираемая строка должна заканчиваться этим символом).

Второе правило: продукция E→TE′ дает *FOLLOW*(T) = {+}; продукция T→FT′ дает *FOLLOW*(F) = {∗}; продукция F→(E) дает *FOLLOW*(E) = {$, )}.

Третье правило итерационное, мы должны применять еще раз, если на предыдущей итерации, хотя бы одно из множеств пополнилось.

Первая итерация: продукция E→TE′ дает *FOLLOW*(E′) = {$, )}; продукция T→FT′ дает *FOLLOW*(T′) = {+}; продукция E′→+TE′ дает *FOLLOW*(T) = {+, $, )}; продукция T′→∗FT′ дает *FOLLOW*(F) = {∗, +}.

Вторая итерация: продукция T→FT′ дает *FOLLOW*(T′) = {+, $, )}; продукция T′→∗FT′ дает *FOLLOW*(F) = {∗, +, $, )}.

Третья итерация ничего не добавляет, построение FOLLOW закончено:

*FOLLOW*(E) = {$, )}

*FOLLOW*(E′) = {$, )}

*FOLLOW*(T) = {+, $, )}

*FOLLOW*(T′) = {+, $, )}

*FOLLOW*(F) = {∗, +, $, )}

Проверяем, является ли грамматика LL(1)-грамматикой:

правила E′→+TE′|ε  *FIRST*(+TE′)∩*FOLLOW*(E′)={+}∩{$, )}=∅

правила T′→∗FT′|ε  *FIRST*(∗FT′)∩*FOLLOW*(T′)={∗}∩{+, $, )}=∅

правила F→(E)|*a*  *FIRST*((E))∩ *FIRST*(*a*)={(}∩{*a*}=∅.

Грамматика является LL(1)-грамматикой.

Определим функцию *CHOICE*(*A*→α) как *FIRST*(α), если ε∉*FIRST*(α) и *FOLLOW*(*A*)∪ *FIRST*(α), если ε∈*FIRST*(α).

***Построение управляющей таблицы.***

Правила построения управляющей таблицы (детерминированного автомата с магазинной памятью) по LL(1)-грамматике:

1. Имена столбцов – все терминальные символы плюс символ маркера конца цепочки.
2. Имена строк – все нетерминальные символы плюс те терминалы, которые не стоят первыми в правых частях правил плюс символ дна магазина.
3. Заполнение таблицы:
   - Если правило имеет вид $A \to t\alpha$, где $A \in N$ и $t \in T$, $\alpha \in \{N \cup T\}^*$, то этому правилу в строке A и столбце t будет содержаться ЗАМЕНИТЬ($\alpha^R$), СДВИГ (если $\alpha=\varepsilon$, вместо ЗАМЕНИТЬ(ε) можно использовать ВЫТОЛКНУТЬ)
   - Если правило имеет вид $A \to \alpha$, где $A \in N$, $\alpha = X\beta$, $X \in N$, $\beta \in \{N \cup T\}^*$, то этому правилу в строке A и во всех столбцах принадлежащих множеству *CHOICE*($A \to \alpha$) будут содержаться ЗАМЕНИТЬ($\alpha^R$), ДЕРЖАТЬ
   - Если магазинным символом является терминал *b*, то в строке *b* и столбце *b* будет содержаться ВЫТОЛКНУТЬ, СДВИГ
   - В строке соответствующей символу дна магазина и столбце соответствующему символу маркера конца цепочки будет содержаться ДОПУСТИТЬ
   - Все остальные элементы таблицы содержат ОТВЕРГНУТЬ (ошибка).

Построим управляющую таблицу для нашей грамматики. Будем использовать символы ⊢ для маркера конца цепочки и # для маркера дна магазина:

| | + | * | ( | ) | *a* | ⊢ |
|---|---|---|---|---|---|---|
| E | отвергнуть | отвергнуть | замен(E′T)<br>держать | отвергнуть | замен(E′T)<br>держать | отвергнуть |

| | | | | | | |
|---|---|---|---|---|---|---|
| E′ | замен(E′T) сдвиг | отвергнуть | отвергнуть | вытолкнуть держать | отвергнуть | вытолкнуть держать |
| T | отвергнуть | отвергнуть | замен(T′F) держать | отвергнуть | замен(T′F) держать | отвергнуть |
| T′ | вытолкнуть держать | замен(T′F) держать | отвергнуть | вытолкнуть держать | отвергнуть | вытолкнуть держать |
| F | отвергнуть | отвергнуть | замен()E) сдвиг | отвергнуть | вытолкнуть сдвиг | отвергнуть |
| ) | отвергнуть | отвергнуть | отвергнуть | вытолкнуть сдвиг | отвергнуть | отвергнуть |
| # | отвергнуть | отвергнуть | отвергнуть | отвергнуть | отвергнуть | допустить |

Работу этого автомата с магазинной памятью рассмотрим на примере разбора цепочки $a+a*a\vdash$ в виде следующей таблицы:

| | Содержимое магазина | Остаток входной цепочки | Команды |
|---|---|---|---|
| 1 | $\#E$ | $a+a*a\vdash$ | заменить(E′T), держать |
| 2 | $\#E'T$ | $a+a*a\vdash$ | заменить(T′F), держать |
| 3 | $\#E'T'F$ | $a+a*a\vdash$ | вытолкнуть, сдвиг |
| 4 | $\#E'T'$ | $+a*a\vdash$ | вытолкнуть, держать |
| 5 | $\#E'$ | $+a*a\vdash$ | заменить(E′T), сдвиг |
| 6 | $\#E'T$ | $a*a\vdash$ | заменить(T′F), держать |
| 7 | $\#E'T'F$ | $a*a\vdash$ | вытолкнуть, сдвиг |
| 8 | $\#E'T'$ | $*a\vdash$ | заменить(T′F), сдвиг |
| 9 | $\#E'T'F$ | $a\vdash$ | вытолкнуть, сдвиг |
| 10 | $\#E'T'$ | $\vdash$ | вытолкнуть, держать |
| 11 | $\#E'$ | $\vdash$ | вытолкнуть, держать |
| 12 | # | $\vdash$ | допустить |

# Глава 2. Теория перевода.

**Определение.** Пусть $\Sigma$ – входной алфавит и $\Delta$ – выходной алфавит. Переводом с языка $L_1 \subseteq \Sigma^*$ на язык $L_2 \subseteq \Delta^*$ назовем отношение T из $\Sigma^*$ в $\Delta^*$, для которого $L_1$ – область определения, а $L_2$ – множество значений.

Если $(x, y) \in T$, то цепочка y называется выходом для цепочки x.

В общем случае в переводе T для данной входной цепочки может быть более одной выходной цепочки. Однако перевод, предназначенный для описания языка программирования, должен быть функцией, т.е. для каждого входа должно быть не более одного выхода.

## 2.1. Схемы синтаксически управляемого перевода.

Проблема задания бесконечного перевода конечными средствами аналогична проблеме задания бесконечного языка. Известно несколько возможных подходов к определению переводов. Аналогично порождению языка с помощью грамматики можно использовать систему, порождающую пары цепочек, принадлежащие переводу. Можно воспользоваться распознавателем с двумя лентами, распознающим пары, принадлежащие переводу, или же определить автомат, который принимает в качестве входа цепочку x и выдает (недетерминировано, если нужно) все цепочки y, являющиеся переводом цепочки x. Этот список не исчерпывает всех возможностей, но охватывает наиболее распространенные модели.

Одним из формализмов, используемых для определения переводов, является схема синтаксически управляемого перевода (трансляции). Интуитивно такая схема представляет собой просто грамматику, в которой к каждому правилу присоединяется элемент перевода. Всякий раз, когда правило участвует в выводе входной цепочки, с помощью элемента перевода вычисляется часть выходной цепочки, соответствующая части входной цепочки, порожденной этим правилом.

**Пример 2.1.** Рассмотрим схему, определяющую перевод $\{(x, xR) \mid x \in \{0, 1\}^*\}$ (для каждого входа x выходом служит обращенная цепочка) по правилам заданным таблицей:

| Правило | Элемент перевода |
|---|---|
| (1) $S \to 0S$ | S = S0 |
| (2) $S \to 1S$ | S = S1 |
| (3) $S \to e$ | S = e |

В переводе, определяемом этой схемой, пару вход – выход можно получить, порождая последовательность выводимых пар цепочек ($\alpha$, $\beta$), где $\alpha$ - входная выводимая цепочка, а $\beta$ - выходная выводимая цепочка. Начинается последовательность парой (S, S). Затем к этой паре можно применить первое правило. Тогда первое S заменяется на 0S по правилу $S \to 0S$, а второе S заменяется на S0 в соответствии с элементом перевода S=S0. Пока можно рассматривать этот элемент перевода просто как правило $S \to S0$. Так получается выводимая пара (0S, S0). Снова применяя правило (1) к символу S в этой новой паре, получаем (00S, S00). Затем правило (2) дает (001S, S100). Если теперь применить правило (3), то будет (001, 100). К последней паре никакого правила применить нельзя, и поэтому она принадлежит переводу, определяемому этой схемой.

Схема трансляции T определяет некоторый перевод $\tau(T)$. По схеме T можно построить транслятор, реализующий перевод $\tau(T)$, который работает так. По данной входной цепочке x с помощью правил схемы перевода транслятор находит (если это возможно) некоторый вывод цепочки x из S. Допустим, что $S=\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n = x$ – такой вывод. Затем транслятор строит вывод

$$(\alpha_0, \beta_0) \Rightarrow (\alpha_1, \beta_1) \Rightarrow \ldots \Rightarrow (\alpha_n, \beta_n)$$

состоящий из выводимых пар цепочек, для которого $(\alpha_0, \beta_0) = (S, S)$, $(\alpha_n, \beta_n) = (x, y)$ и каждая цепочка $\beta_i$ получается из $\beta_{i-1}$ с помощью элемента перевода, соответствующего правилу, примененному в надлежащем месте при переходе от $\alpha_{i-1}$ к $\alpha_i$. Цепочка *y* служит выходом для цепочки *x*.

**Пример 2.2.** Рассмотрим схему перевода, отображающую арифметические выражения из языка $L(G_0)$ в соответствующие постфиксные польские записи. Она задается таблицей:

| Правило | Элемент перевода |
|---|---|
| $E \to E+T$ | $E = ET+$ |
| $E \to T$ | $E = T$ |
| $T \to T*F$ | $T = TF*$ |
| $T \to F$ | $T = F$ |
| $F \to (E)$ | $F = E$ |
| $F \to a$ | $F = a$ |
| $F \to b$ | $F = b$ |
| $F \to c$ | $F = c$ |

Правилу $E \to E+T$ соответствует элемент перевода $E = ET+$. Этот элемент говорит о том, что перевод, порождаемый символом $E$, стоящим в левой части правила, получается из перевода, порождаемого символом $E$, стоящим в правой части правила, за которым идут перевод, порождаемый символом $T$, и знак +.

Определим выход, соответствующий входу $(a+b)*c$. Для этого сначала по правилам схемы перевода найдем левый вывод цепочки $(a+b)*c$ из $S$:

$$
\begin{aligned}
E &\Rightarrow T \\
&\Rightarrow T*F \\
&\Rightarrow F*F \\
&\Rightarrow (E)*F \\
&\Rightarrow (E+T)*F \\
&\Rightarrow (T+T)*F \\
&\Rightarrow (F+T)*F \\
&\Rightarrow (a+T)*F \\
&\Rightarrow (a+F)*F \\
&\Rightarrow (a+b)*F \\
&\Rightarrow (a+b)*c
\end{aligned}
$$

Затем вычислим соответствующую последовательность выводимых пар цепочек:

$(E, E) \Rightarrow (T, T)$

$\Rightarrow (T*F, TF*)$

$\Rightarrow (F*F, FF*)$

$\Rightarrow ((E)*F, EF*)$

$\Rightarrow ((E+T)*F, ET+F*)$

$\Rightarrow ((T+T)*F, TT+F*)$

$\Rightarrow ((F+T)*F, FT+F*)$

$\Rightarrow ((a+T)*F, aT+F*)$

$\Rightarrow ((a+F)*F, aF+F*)$

$\Rightarrow ((a+b)*F, ab+F*)$

$\Rightarrow ((a+b)*c, ab+c*)$

Каждая выходная цепочка этой последовательности получается из предыдущей выходной цепочки заменой подходящего нетерминала правой частью элемента перевода, присоединенного к правилу, примененному при выводе соответствующей входной цепочки.

Схемы перевода в примерах 1 и 2 относятся к важному классу схем, называемых схемами синтаксически управляемого перевода.

**Определение.** Схемой синтаксически управляемого перевода (сокращенно СУ-схемой) называется пятерка $T = (N, \Sigma, \Delta, R, S)$, где

(1) N – конечное множество нетерминальных символов,

(2) $\Sigma$ – конечный входной алфавит,

(3) $\Delta$ – конечный выходной алфавит,

(4) $R$ – конечное множество правил вида $A \to \alpha, \beta$, где $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$ и вхождения нетерминалов в цепочку $\beta$ образуют перестановку вхождений нетерминалов в цепочку $\alpha$,

(5) $S$ – начальный символ, выделенный нетерминал из N.

Пусть $A \to \alpha, \beta$ - правило. Каждому вхождению нетерминала в цепочку $\alpha$ *соответствует* некоторое вхождение того же нетерминала в цепочку $\beta$.

Если нетерминал $B$ входит в цепочку $\alpha$ только один раз, то соответствие очевидно. Если $B$ входит более одного раза, то для указания соответствия мы будем пользоваться верхними целочисленными индексами. Это соответствие является (если оно явно не указано, то подразумеваемой) частью правила. Например, в правиле $A \to B^{(1)}CB^{(2)}$, $B^{(2)}B^{(1)}C$ 1-й, 2-й и 3-й позиции цепочки $B^{(1)}CB^{(2)}$ соответствует 2-я, 3-я и 1-я позиции цепочки $B^{(2)}B^{(1)}C$.

Определим *выводимую пару цепочек* схемы $T$:

(1) (S, S) – выводимая пара, в которой символы S соответствуют друг другу.
(2) если ($\alpha A\beta$, $\alpha' A\beta'$) – выводимая пара, в которой два выделенных вхождения нетерминала $A$ соответствуют друг другу, и $A \to \gamma$, $\gamma'$ правило из $R$, то ($\alpha\gamma\beta$, $\alpha'\gamma'\beta'$) – выводимая пара. Вхождения нетерминалов в $\gamma$ и $\gamma'$ соответствуют друг другу точно так же, как они соответствовали в правиле. Вхождения нетерминалов в $\alpha$ и $\beta$ соответствуют вхождениям нетерминалов в $\alpha'$ и $\beta'$в новой выводимой паре точно так же, как они соответствовали в старой выводимой паре. Когда надо, это соответствие будет указываться верхними индексами; оно составляет существенную часть выводимой пары.

Если между парами ($\alpha A\beta$, $\alpha' A\beta'$) и ($\alpha\gamma\beta$, $\alpha'\gamma'\beta'$) с учетом соответствия их нетерминалов установлена описанная выше связь, то мы будем писать ($\alpha A\beta$, $\alpha' A\beta'$) $\Rightarrow_T$ ($\alpha\gamma\beta$, $\alpha'\gamma'\beta'$). Транзитивное замыкание, рефлексивно-транзитивное замыкание и $k$-ю степень отношения $\Rightarrow_T$ будем обозначать $\Rightarrow_T^+$ $\Rightarrow_T^*$ и $\Rightarrow_T^k$ соответственно. Когда можно, будем опускать индекс $T$.
Переводом, определяемым схемой $T$ (обозначается $\tau(T)$), называют множество пар $\{(x, y) \mid (S, S) \Rightarrow^* (x, y), x \in \Sigma^* \text{ и } y \in \Delta^*\}$

**Определение.** Если $T = (N, \Sigma, \Delta, R, S)$ – СУ-схема, то $\tau(T)$ называется синтаксически управляемым переводом (СУ-переводом). Грамматика $G_i = (N, \Sigma, P, S)$, где $P = \{A \to \alpha \mid A \to \alpha, \beta$ принадлежит $R\}$, называется *входной грамматикой* СУ-схемы $T$. Грамматика $G_o = (N, \Delta, P', S)$, где $P' = \{A \to \beta \mid A \to \alpha, \beta$ принадлежит $R\}$, называется *выходной грамматикой* СУ-схемы $T$.

**Определение.** СУ-схема $T = (N, \Sigma, \Delta, R, S)$ называется *простой*, если для каждого правила $A \to \alpha, \beta$ из $R$ соответствующие друг другу вхождения нетерминалов встречаются в $\alpha$ и $\beta$ в одном и том же порядке. Перевод, определяемый простой СУ-схемой, называется *простым синтаксически управляемым переводом* (простым СУ-переводом).

**Пример 2.3.** Следующая простая СУ-схема отображает арифметические выражения из языка $L(G_0)$ в арифметические выражения, не содержащие избыточных скобок:

(1) $E \to (E)$, $E$

(2) $E \to E+E$, $E+E$

(3) $E \to T$, $T$

(4) $T \to (T)$, $T$

(5) $T \to A*A$, $A*A$

(6) $T \to a$, $a$

(7) $A \to (E+E)$, $(E+E)$

(8) $A \to T$, $T$

Например, для выражения $((a+(a*a))*a)$ эта СУ-схема дает перевод $(a+a*a)*a$.

Заметим, что эта входная грамматика неоднозначна, но каждой входной цепочке соответствует точно одна выходная.

## 2.2. Конечные преобразователи.

Преобразователь – это просто распознаватель, выдающий на каждом такте выходную цепочку (которая может быть и пустой).

Для большей общности рассмотрим в качестве основы конечного преобразователя недетерминированный конечный автомат, способный делать $e$-такты.

**Определение.** *Конечным преобразователем* называется шестерка

$M = (Q, \Sigma, \Delta, \delta, q_0, F)$, где

(1) $Q$ – конечное множество *состояний*,

(2) $\Sigma$ – конечный *входной алфавит*,

(3) $\Delta$ – конечный *выходной алфавит*,

(4) $\delta$ – отображение множества $Q\times(\Sigma\cup\{e\})$ в множество всех подмножеств множества $Q\times\Delta^*$,

(5) $q_0 \in Q$ – *начальное состояние*,

(6) $F \subseteq Q$ – множество *заключительных состояний*.

Определим конфигурацию преобразователя $M$ как тройку $(q, x, y)$, где

(1) $q\in Q$ – текущее состояние управляющего устройства,

(2) $x\in\Sigma^*$ – оставшаяся непрочитанной часть входной цепочки, причем самый левый символ цепочки x расположен под входной головкой,

(3) $y\in\Delta^*$ – часть выходной цепочки, выданная вплоть до текущего момента.

Определим бинарное отношение $\vdash_M$ (или $\vdash$, когда ясно, о каком $M$ идет речь) на конфигурациях, соответствующее одному такту работы преобразователя $M$: для всех $q\in Q$, $a\in\Sigma\cup\{e\}$, $x\in\Sigma^*$ и $y\in\Delta^*$, таких, что $\delta(q, a)$ содержит $(r, z)$, будем писать

$$(q, ax, y) \vdash (r, x, yz)$$

Обычным образом далее определяются $\vdash^i$, $\vdash^*$, $\vdash^+$.

Цепочку $y$ назовем *выходом* для цепочки $x$, если $(q_0, x, e) \vdash^* (q, e, y)$ для некоторого $q\in F$. *Переводом, определяемым преобразователем* $M$ (обозначается $\tau(M)$), назовем множество $\{(x, y)|\ ((q_0, x, e) \vdash^* (q, e, y)$ для некоторого $q\in F\}$. Перевод, определяемый конечным преобразователем, будем называть *регулярным переводом* или *конечным преобразованием*.

Заметим, что для того, чтобы выходную цепочку $y$ можно было считать переводом цепочки $x$, цепочка $x$ должна перевести преобразователь $M$ из начального состояния в заключительное.

**Пример 2.4.** Построим конечный преобразователь, который распознает арифметические выражения, порождаемые правилами

$$S \to a{+}S \mid a{-}S \mid {+}S \mid {-}S \mid a$$

и устраняет из этих выражений избыточные унарные операции.

Например, выражение $-a+-a-+-a$ он переведет в $-a-a+a$. Во входном языке символ $a$ представляет идентификатор и перед идентификатором допускается произвольная последовательность знаков унарных операций + и –. Заметим, что входной язык является регулярным множеством.

Пусть $M = (Q, \Sigma, \Delta, \delta, q_0, F)$, где

(1) $Q = \{q_0, q_1, q_2, q_3, q_4\}$,

(2) $\Sigma = \{a, +, -\}$,

(3) $\Delta = \Sigma$,

(4) $\delta$ задается следующей таблицей:

| δ | $a$ | + | – |
|---|---|---|---|
| $q_0$ | $q_1, a$ | $q_0, e$ | $q_4, e$ |
| $q_1$ | | $q_2, e$ | $q_3, e$ |
| $q_2$ | $q_1, +a$ | $q_2, e$ | $q_3, e$ |
| $q_3$ | $q_1, -a$ | $q_3, e$ | $q_2, e$ |
| $q_4$ | $q_1, -a$ | $q_4, e$ | $q_0, e$ |

(5) $F = \{q_1\}$.

Преобразователь $M$ начинает работу в состоянии $q_0$ и, чередуя состояния $q_0$ и $q_4$ на входном символе –, определяет, четное или нечетное число знаков – предшествует первому символу $a$. Когда появляется $a$, преобразователь $M$ переходит в состояние $q_1$, допуская вход, и выдает $a$ или $-a$ в зависимости от того, четно или нечетно число появившихся минусов. Для следующих символов $a$ он подсчитывает, четно или нечетно число предшествующих минусов, с помощью состояний $q_2$ и $q_3$. Единственное различие между парами $q_2$, $q_3$ и $q_0$, $q_4$ состоит в том, что если символу $a$ предшествует четное число минусов, то первая из них выдает $+a$, а не только $a$.

Для входа $-a+-a-+-a$ последовательность тактов преобразователя $M$ такова:

$(q_0, -a+-a-+-a, e) \vdash (q_4, a+-a-+-a, e)$

$\vdash (q_1, +–a–+–a, –a)$

$\vdash (q_2, –a–+–a, –a)$

$\vdash (q_3, a–+–a, –a)$

$\vdash (q_1, –+–a, –a–a)$

$\vdash (q_3, +–a, –a–a)$

$\vdash (q_3, –a, –a–a)$

$\vdash (q_2, a, –a–a)$

$\vdash (q_1, e, –a–a+a)$

Отсюда ясно, что $M$ переводит цепочку $–a+–a–+–a$ в , $–a–a+a$ поскольку $q_1$ - заключительное состояние.

Конечный преобразователь $M$ назовем *детерминированным*, если для всех $q \in Q$

(1) либо $\delta(q, a)$ содержит не более одного элемента для каждого $a \in \Sigma$ и $\delta(q, e)$ пусто, либо

(2) $\delta(q, e)$ содержит один элемент и $\delta(q, a)$ пусто для всех $a \in \Sigma$.

## 2.3. Преобразователи с магазинной памятью.

Эти преобразователи получаются из автоматов с магазинной памятью, если их снабдить выходом и разрешить на каждом такте выдавать выходную цепочку конечной длины.

**Определение.** Преобразователем с магазинной памятью (МП-преобразователем) называется восьмерка $P = (Q, \Sigma, \Delta, \Gamma, \delta, q_0, z_0, F)$, где все символы имеют тот же смысл, что и в определении МП-автомата, за исключением того, что $\Delta$ – конечный выходной алфавит, а $\delta$ – отображение

множества $Q\times(\Sigma\cup\{e\})\times\Gamma$ в множество всех подмножеств множества $Q\times\Gamma^*\times\Delta^*$.

Определим конфигурацию преобразователя $P$ как четверку $(q, x, \alpha, y)$, где $q$, $x$ и $\alpha$ те же, что и у МП-автомата, а $y$ - выходная цепочка, выданная вплоть до настоящего момента. Если $\delta(q, a, Z)$ содержит $(r, \alpha, z)$, то будем писать $(q, ax, Z\gamma, y) \vdash (r, x, \alpha\gamma, yz)$ для любых $x\in\Sigma^*$, $\gamma\in\Gamma^*$ и $y\in\Delta^*$.

Цепочку $y$ назовем *выходом* для $x$, если $(q_0, x, z_0, e) \vdash^* (q, e, \alpha, y)$ для некоторых $q\in F$ и $\alpha\in\Gamma^*$. *Переводом* (или *преобразованием*), *определяемым* МП-*преобразователем* $P$ (обозначается $\tau(P)$), назовем множество

$\{(x, y)|\ (q_0, x, z_0, e) \vdash^* (q, e, \alpha, y)$ для некоторых $q\in F$ и $\alpha\in\Gamma^*\}$

**Пример 2.5.** Рассмотрим МП-преобразователь
$P = (\{q\}, \{a, +,*\}, \{+, *, E, \{a, +, *\}, \delta, q, E, \{q\})$, где $\delta$ определяется равенствами

$$\delta(q, a, E) = \{(q, e, a)\}$$
$$\delta(q, +, E) = \{(q, EE+, e)\}$$
$$\delta(q, *, E) = \{(q, EE*, e)\}$$
$$\delta(q, e, +) = \{(q, e, +)\}$$
$$\delta(q, e, *) = \{(q, e, *)\}$$

Для входа $+*aaa$ МП-преобразователь $P$ сделает такую последовательность тактов:

$(q, +*aaa, E, e) \vdash (q, *aaa, EE+, e)$

$\vdash (q, aaa, EE*E+, e)$

$\vdash (q, aa, E*E+, a)$

$\vdash (q, a, *E+, aa)$

$\vdash (q, a, E+, aa*)$

$\vdash (q, e, +, aa*a)$

$\vdash (q, e, e, aa*a+)$

Таким образом, $P$ переводит цепочку $+*aaa$ в цепочку $aa*a+$, опустошая магазин. Можно проверить, что $\tau(P) = \{(x, y) | \ x$ – префиксное польское арифметическое выражение в алфавите $\{+, *, a\}$ и $y$ – соответствующая постфиксная польская запись$\}$

Будем говорить, что МП-преобразователь $P = (Q, \Sigma, \Delta, \Gamma, \delta, q_0, z_0, F)$ *детерминированный* (ДМП-преобразователь), если

(1) для всех $q \in Q$, $a \in \Sigma \cup \{e\}$ и $Z \in \Gamma$ множество $\delta(q, a, Z)$ содержит не более одного элемента,

(2) если $\delta(q, e, Z) \neq \varnothing$, то $\delta(q, a, Z) = \varnothing$ для всех $a \in \Sigma$.

# Глава 3. Обобщения грамматик Хомского.

Грамматики Хомского могут порождать цепочки символов, которые можно рассматривать как одномерные последовательности из конечного множества объектов. Очевидно, что символы (как терминальные, так и нетерминальные) можно рассматривать как объекты, имеющие ровно две точки связи – левую и правую, а строку символов – как такое их соединение, в котором правая точка связи одного символа связывается с левой точкой связи своего соседа. Простое обобщение порождающих грамматик Хомского можно построить, рассматривая в качестве символов произвольные объекты с несколькими точками соединения. Простейшей математической моделью такого рода является граф. Правила такой обобщенной грамматики, порождающей графы, должны определять, как более сложные графы (конструкции) строятся из менее сложных, соединяясь в допустимых точках связи.

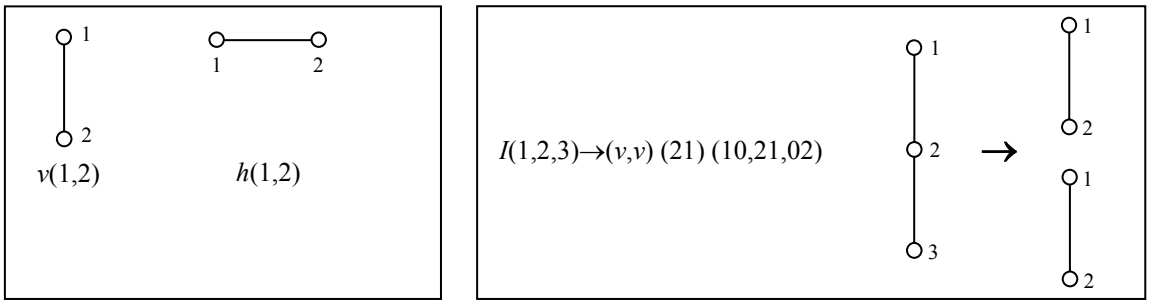
## 3.1. Плекс-грамматики.

Одним из первых расширений понятия порождающей грамматики Хомского для порождения структур, отличных от цепочек символов, явились так называемые плекс-грамматики (plex grammars). Слово plexus в английском языке означает переплетение. В соответствии с названием, такие грамматики порождают структуры произвольных объектов, соединенных в заранее определенных точках связи. Плекс-грамматики были введены впервые Feder J. в 1971 году.

Терминальные и нетерминальные символы плекс-грамматики называются нейпами. Нейп – это произвольный объект, имеющий произвольное конечное число точек связи (от англ. nape (n attaching point entity) – объект с n точками связи). Нейп представляется именем и точками связи. Аналогом нетерминала в грамматиках Хомского является нетерминальный нейп – конструкция или объект с несколькими точками связи. Правила плекс-грамматики

определяют, как сложные нейпы формируются из примитивных нейпов и других структур.

Рассмотрим простейший пример плекс-грамматики. Терминальные символы грамматики составляют вертикальное и горизонтальное ребра, каждое с двумя точками связи, обозначенными 1 и 2.




Это правило определяет нетерминальную структуру, поименованную I и имеющую три точки связи, пронумерованные по порядку 1, 2 и 3. Структура строится из двух терминальных объектов v (это показывает первый член правой части продукции) только одним соединением: второй точки связи первого с первой точкой связи второго. Именно это и показывает первые две группы в правой части правила. Третья группа характеризует точки связи (1, 2 и 3) новой нетерминальной структуры. Первая точка связи у I образуется только из первой точки связи первого связанного объекта (обозначено 10), вторая точка связи I – это объединение второй точки связи первого составляющего структуру объекта с первой точкой связи второго (обозначено 21). Третья точка связи I – это просто вторая точка связи второго объекта структуры (обозначено 02). Очевидно, что правила плекс-грамматики можно представлять либо в графической форме, либо в эквивалентной текстовой форме.

Формально, контекстно-свободная плекс-грамматика имеет продукции в форме:

$$A\ \Delta_A \rightarrow \beta\ \Gamma_\beta\ \Delta_\beta$$

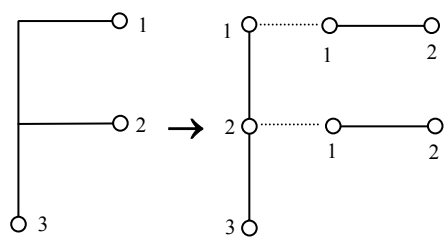
где $A$ – имя нетерминального объекта грамматики, $\Delta_A$ – это список внешних точек соединений нетерминальной конструкции $A$, $\beta$ - это список

компонентов структуры, образующей $A$, $\Gamma_\beta$ - список взаимосвязей подструктур правой части при образовании конструкции $A$. Наконец, $\Delta_\beta$ - это список соответствия, показывающий, как каждая внешняя точка связи у конструкции $A$ соотносится с точками связи составляющих $A$ подконструкций.

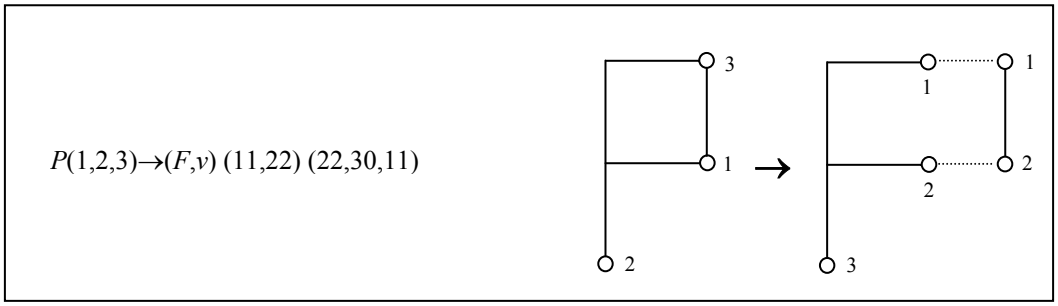
Следующее порождающее правило нашей плекс-грамматики связывает три объекта. Первый из них – нетерминальный объект $I$, два остальных – это горизонтальные терминальные объекты грамматики, каждый со своими точками связи.

Поясним правило. Конструкция $F$ с тремя внешними точками связи, обозначенными 1, 2 и 3, строится из трех подконструкций – конструкции $I$ и двух одинаковых конструкций $h$. Соединяются эти подконструкции для получения $F$ в двух точках. Первое соединение получено соединением точки связи 1 первого объекта и точки связи 1 второго объекта. Третий объект не участвует в этом соединении. Таким образом, эта точка соединения представлена в списке как 110. Вторая точка соединения (201) получена соединением второй точки связи первого объекта – $I$ и первой точки связи третьего объекта $h$, второй объект в этой связи не участвует. Список соответствия здесь построен для трех внешних точек конструкции $F$. Первая внешняя точка связи $F$ образована второй точкой связи второго объекта, первый и третий объекты в этой связи не участвуют. Вторая внешняя точка связи $F$ образована второй точкой связи третьего объекта, первый и второй в этой связи не участвуют. Наконец, третья внешняя точка $F$ образована третьей точкой связи первого из составляющих эту конструкцию объектов.

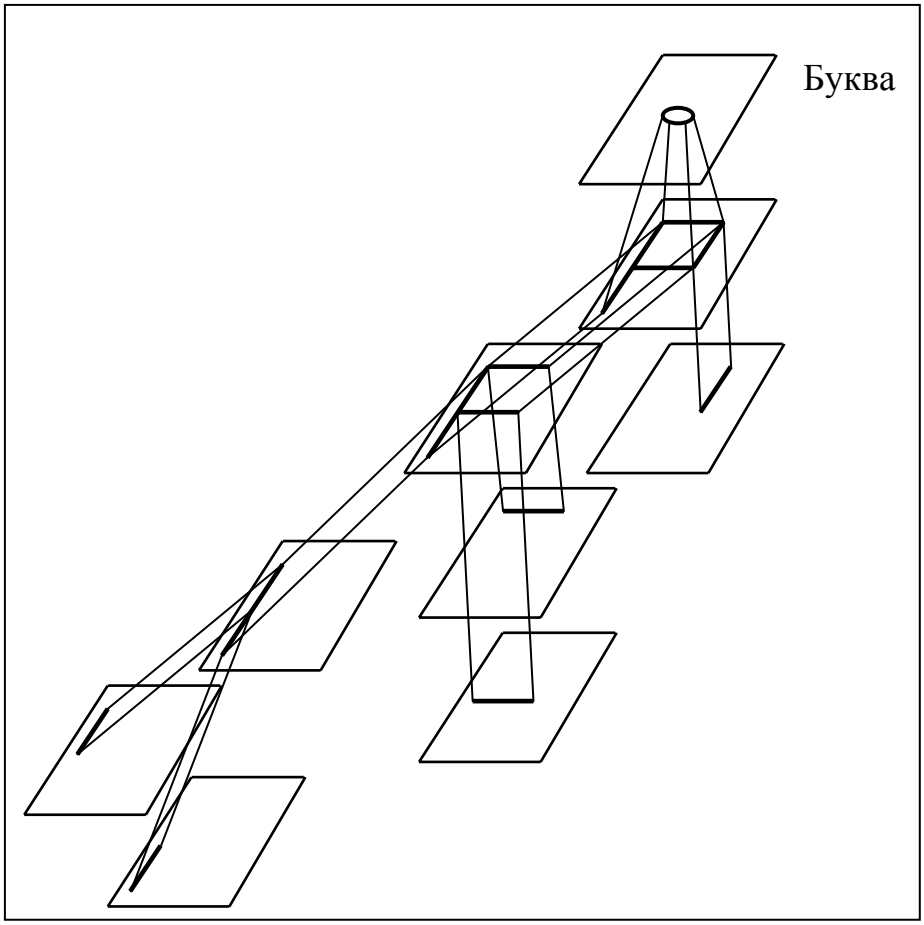
$F(1,2,3)\rightarrow(I,h,h)$ (110,201) (020,002,300)



Третье правило грамматики имеет очевидный смысл.




Порожденная конструкция – буква *P* – будет иметь в этой грамматике следующее дерево вывода.




Рассмотрим пример плекс-грамматики совсем из другой области – органической химии. Плекс-грамматика, порождающая сложные СН-структуры, может выглядеть следующим образом. Пусть терминальные нейпы этой грамматики – это атомы углерода и водорода с соответствующими валентностями, представленными точками связи.

Рассмотрим плекс-грамматику, порождающую цепочки соединений углерода и водорода. Грамматика содержит пять правил.

1. ⟨Цепь⟩(1, 2) → ⟨Секция⟩( )(1, 2) | ⟨Секция⟩⟨Цепь⟩(21)(10, 02)

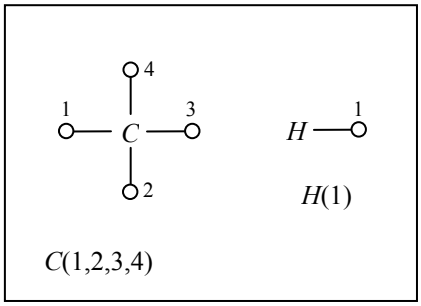
2. ⟨Секция⟩(1, 2) → ⟨$CH_2$⟩⟨C⟩⟨$CP_3$⟩⟨CH⟩⟨$CH_2$⟩

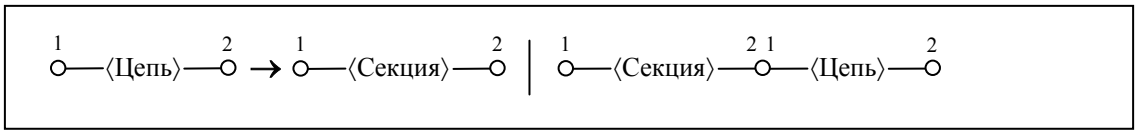
(21000, 02100, 03020, 04010, 00031)(10000, 00002)

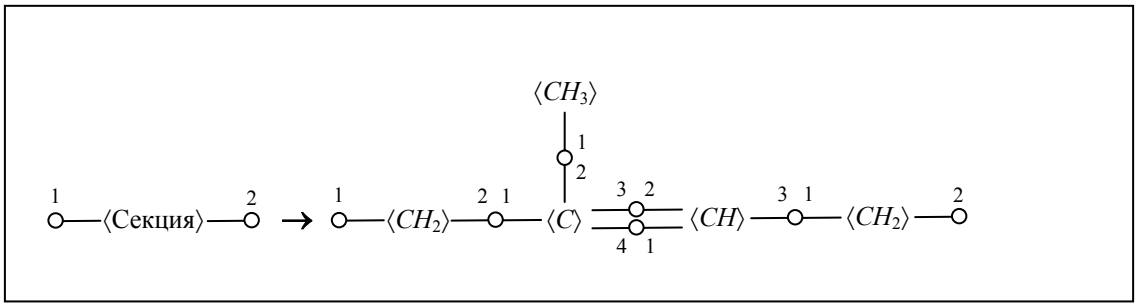
3. ⟨$CH_3$⟩(1) → ⟨$CH_2$⟩⟨H⟩(21)(10)

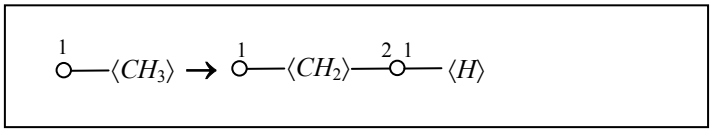
4. ⟨$CH_2$⟩(1, 2) → ⟨CH⟩⟨H⟩(21)(10, 30)

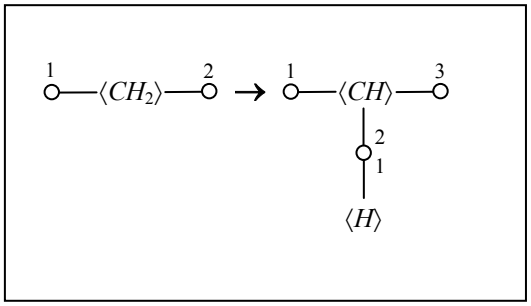
5. ⟨CH⟩(1, 2, 3) → ⟨C⟩⟨H⟩(41)(10, 20, 30)

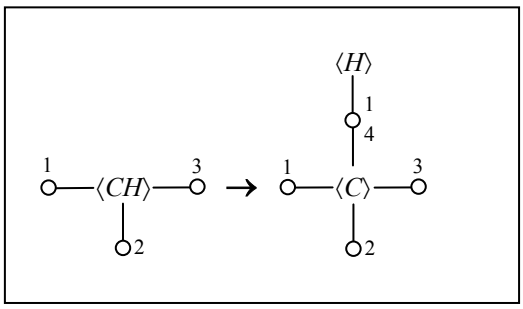


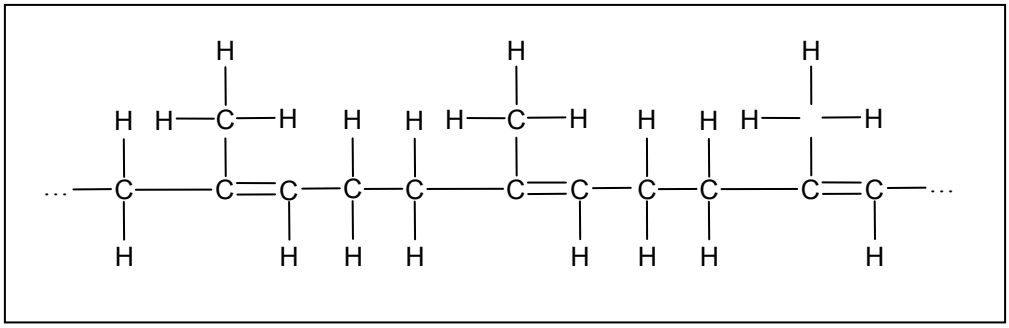






Эта грамматика порождает цепочки произвольной конечной длины, представляющие собой структуру молекулы резины.



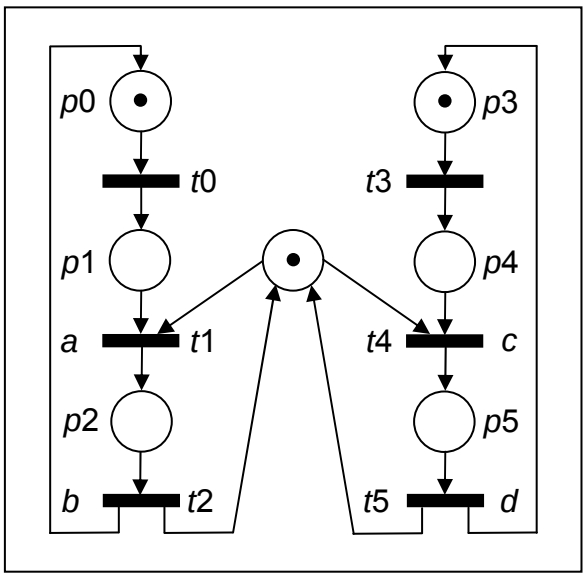
# Глава 4. Языки сетей Петри.

Грамматики Хомского не являются единственным возможным способом задания языков. Фактически порождающим механизмом при задании языка может служить любая формальная модель, каким-либо образом определяющая множество одномерных последовательностей объектов (символов) над конечным словарем. Рассмотрим одну из таких моделей порождения – сети Петри.

Сеть Петри – это ориентированный граф, состоящий из двух типов вершин, с ребрами, которые могут соединять только вершины разных типов. Вершины первого типа называются *позициями* и обозначаются кружками. Вершины второго типа называются *переходами* и обозначаются прямоугольниками. В каждой позиции может быть некоторое количество маркеров. Динамика сети Петри выражается в том, что у нее может сработать переход, если он активный. Активным переход будет в случае, если во всех позициях, из которых в переход ведут ребра, находится, по крайней мере, один маркер. При срабатывании перехода из каждой позиции, из которой в этот переход ведут ребра, изымается по одному маркеру и в каждую позицию, в которую направлены ребра из данного перехода, по одному маркеру добавляется.

Сети Петри используются для моделирования систем параллельных процессов. События, которые могут произойти в системе, представляются в сети Петри переходами. Множество возможных последовательностей срабатываний переходов характеризует систему, которая моделируется сетью Петри.

Пометим некоторые (существенные для конкретного приложения) переходы сети символами конечного алфавита. Множество всех префиксов последовательностей пометок срабатываний переходов сети Петри является, очевидно, языком, который называется *свободным префиксным языком сети Петри*. Таким образом, язык определяется динамикой, работой сети Петри.

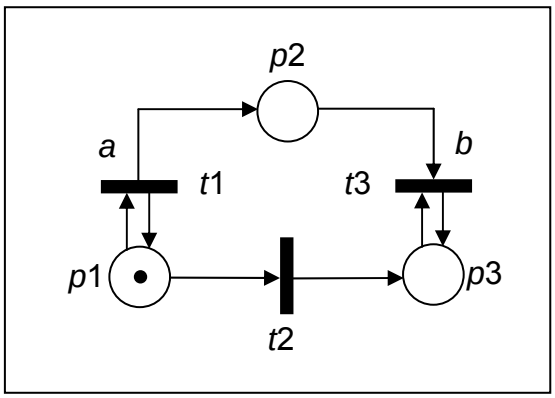



На рисунке представлена сеть Петри, моделирующая использование семафора для решения проблемы взаимного исключения двух параллельных процессов. Эта проблема заключается в том, чтобы синхронизировать два параллельных процесса (представленных здесь последовательностями позиций <*p*0, *p*1, *p*2> и <*p*3, *p*4, *p*5>) таким образом, чтобы в своих критических интервалах, связанных, например, с использованием общего ресурса, оба процесса не могли бы находиться одновременно. Критические интервалы в двух процессах нашего примера моделируются позициями *p*2 и *p*5. События входа в критический интервал и выхода из него моделируются для двух процессов переходами *t*1, *t*2 в одном процессе, и *t*4, *t*5 во втором. Именно эти интересующие нас события помечены (*t*1, *t*2 помечены *a* и *b*, а *t*4, *t*5 помечены *c* и *d*).

Легко видеть, что префиксный язык, порождаемый этой сетью Петри – это язык, определяемый регулярным выражением (*ab*+*cd*)*. Простейший анализ этого языка показывает, что проблема взаимного исключения здесь решена корректно. Действительно, все цепочки языка удовлетворяют требованию взаимного исключения: если один процесс вошел в свой критический интервал, то другой не может войти в свой критический интервал, пока первый не выйдет из своего. Что в терминах языка представлено так: цепочки языка не содержат фрагментов *ac* и *ca*. Таким образом, анализ правильности функционирования двух параллельных процессов (а именно корректность

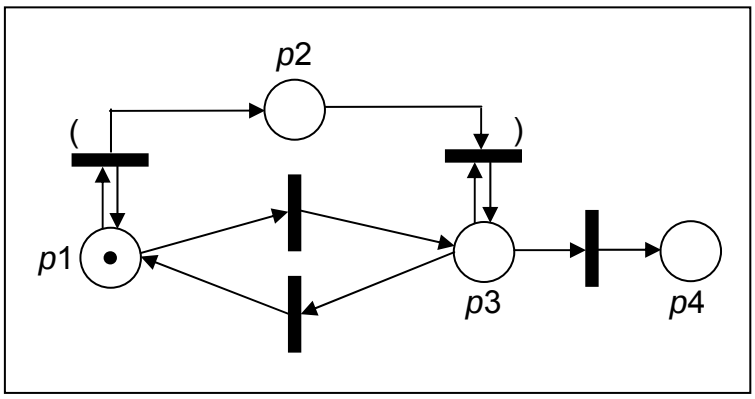
решения проблемы взаимного исключения) проведен здесь с помощью языка, порождаемого сетью Петри.

Кроме префиксных языков иногда удобно рассматривать так называемые *терминальные языки сетей Петри*. Терминальным языком сети Петри называется множество последовательностей пометок срабатывающих переходов сети, приводящих к некоторой заданной финальной маркировке сети.




Пусть в качестве финальной маркировки сети Петри, представленной на рисунке, выбрана маркировка (0, 0, 1), т. е. такое состояние сети, при котором единственный маркер находится в позиции $p3$, а две других позиции не содержат маркеров. При достижении финальной маркировки эта сеть порождает терминальный язык $L = \{a^n b^n \mid n \geq 0\}$, который, как известно, является КС-языком.

Действительно, стартуя с начальной маркировки (1, 0, 0), переход $t1$ может сработать многократно произвольное число раз, после чего сработает переход $t2$. Произвольное конечное число маркеров, равное числу срабатываний перехода $t1$, накопится в позиции $p2$. Для достижения финальной маркировки все маркеры, появившиеся в позиции $p2$, должны исчезнуть, что возможно только при соответствующем числе срабатываний перехода $t3$.

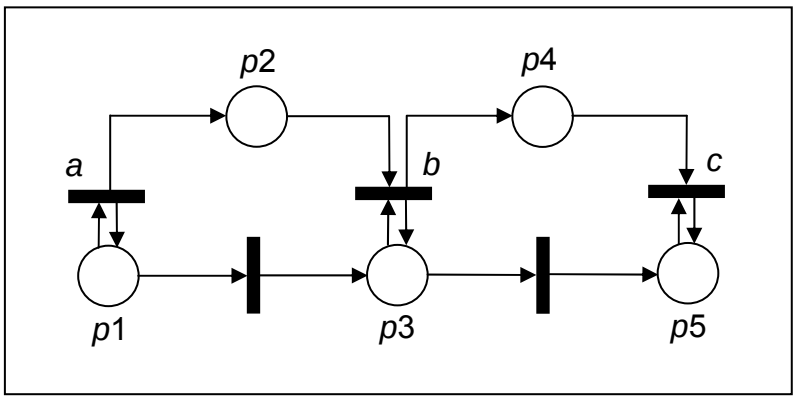



Сеть Петри, представленная на этом рисунке, при достижении финальной маркировки (0, 0, 0, 1) порождает язык согласованных скобок.

Возникает вопрос: какой порождающей мощностью обладают сети Петри, иными словами, какие языки могут ими порождаться? Оказывается, что языки, порождаемые сетями Петри, занимают промежуточное положение между автоматными языками и языками, порождаемыми неограниченными грамматиками Хомского (и, следовательно, распознаваемыми машинами Тьюринга).

Очевидно, что частным случаем сетей Петри являются конечные автоматы. Действительно, любая сеть Петри, в каждый переход которой входит ровно одна стрелка и из каждого перехода которой выходит ровно одна стрелка, имеющая в качестве начальной маркировки ровно один маркер только в одной какой-нибудь позиции, представляет конечный автомат. Следовательно, все автоматные языки порождаются сетями Петри. Поскольку сети Петри могут порождать и неавтоматные языки (например, $\{a^n b^n \mid n \geq 0\}$), то они строго мощнее конечных автоматов.

Легко построить сеть Петри, порождающую терминальный язык $\{a^n b^n c^n \mid n \geq 0\}$ для терминальной маркировки (0, 0, 0, 0, 1):




Этот язык не является КС-языком. Однако существуют КС-языки, которые не могут порождаться сетями Петри. Таким языком, например, является $\{wcw^R \mid w \in \{a, b\}^*, w^R$ – обращение $w\}$.

Таким образом, класс помеченных сетей Петри строго мощнее класса конечных автоматов, несравним с классом автоматов с магазинной памятью и строго менее мощен, чем машины Тьюринга.

# Глава 5. Системы Линденмайера.

Понятие систем Линденмайера (L-систем) появилось в 1968 году благодаря Аристиду Линденмайеру. Изначально L-системы были предназначены для биологических моделей селекции. С их помощью можно строить многие известные самоподобные фракталы, включая снежинку Коха и ковер Серпинского.

***Определение***. Детерминированной контекстно-независимой L-системой называется набор, состоящий из *алфавита*, *аксиомы*, и *множества правил вывода*.

*Алфавитом* называется конечное множество, а его элементы – *символами*. Природа символов не важна, их единственная функция – отличаться друг от друга. *Строкой над алфавитом* называется конечная последовательность символов алфавита. *Аксиома* – некоторая строка. *Правило* – это пара, в котором левая часть – символ алфавита, правая часть, строка над алфавитом. Левая часть должна быть уникальной.

Все очень похоже на порождающие грамматики Хомского. Отличия следующие:

- алфавит не делится на нетерминальные и терминальные символы;
- аксиома не символ, а строка;
- не может быть нескольких правил с одинаковой левой частью;
- к текущей строке правила применяются не по одному, а все вместе (смотри ниже).

Пример L-системы:

Алфавит: {A, B, F, H, J, +, –}

Аксиома: FB

Правила:

A→FBFA+HFA+FB–FA

B→FB+FA–FB–JFBFA

F→ε

H→–

J→+

В дальнейшем, описывая L-системы, мы не будем указывать алфавит, поскольку он всегда будет состоять из символов, упомянутых в аксиоме и в правилах.

## 5.1 Развитие

Как только L-система определена, она начинает развиваться в соответствии с ее правилами. Начальным состоянием L-системы является ее аксиома. При дальнейшем развитии эта строка, будет меняться. Развитие L-системы происходит циклически. В каждом цикле развития текущая строка просматривается слева направо, символ за символом. Для каждого символа ищется правило, для которого этот символ является левой частью. Если такого правила не нашлось, символ остается без изменений. Иными словами, для тех символов X, для которых нет явного правила, действует неявное правило: X→X. Если же соответствующее правило найдено, символ заменяется на правую часть из этого правила. Просмотр продолжается со следующего символа.

Пример. Рассмотрим следующую L-систему (она называется *Algae* – водоросль, поскольку ее развитие моделирует рост одного из видов водорослей):

Аксиома: A

Правила:

A→B

B→AB

В таблице приведены состояния этой L-системы, соответствующие первым десяти циклам развития системы.

| поколение | состояние |
|---|---|
| 0 | A |
| 1 | B |
| 2 | AB |
| 3 | BAB |
| 4 | ABBAB |
| 5 | BABABBAB |
| 6 | ABBABBABABBAB |
| 7 | BABABBABABBABBABABBAB |
| 8 | ABBABBABABBABBABABBABABBABBABABBAB |
| 9 | BABABBABABBABBABABBABABBABBABABBABABBABBABABBABABBABBABABBAB |

Длины строк, представляющих состояние этой L-системы, образуют последовательность чисел Фибоначчи. Последовательностями Фибоначчи будут также количества символов A и B в этих строках. Удивительным является тот факт, что в последовательности строк имеется та же закономерность, что и в последовательности чисел Фибоначчи: каждая строка является «суммой» (конкатенацией) двух предыдущих.

## 5.2 L-системы и моделирование процессов роста

L-системы находят применение при моделировании процессов роста как живых организмов, так и неживых объектов (например, кристаллов, раковин моллюсков или пчелиных сот). Для моделирования требуется ***проинтерпретировать*** символы алфавита L-системы.

Часто символы интерпретируются как программы на языке «черепашьей графики».

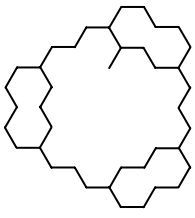
Например, если сопоставить в рассмотренной выше L-системе *Algae* символам A и B программы:

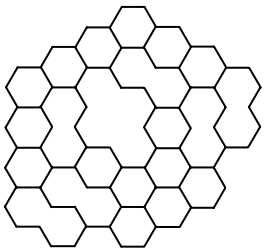
```
ROTATE 60°
FORWARD 1
```

и

```
ROTATE -60°
FORWARD 1
```

соответственно, мы получим такие изображения:

| поколение | изображение |
|---|---|
| 0 |  |
| 5 |  |
| 10 |  |

| | |
|---|---|
| 15 |  |

## 5.3 L-системы и фрактальные кривые

Рассмотрим использование L-систем в построении фрактальных кривых. Если говорить строго, то фрактальная кривая получится только при стремлении номера поколения к бесконечности.

Снежинка Коха задается следующей L-системой:

Аксиома:

`F++F++F`

Правила:

`F→F–F++F–F`

Интерпретация:

`F` `FORWARD 1`

`+` `ROTATE 60°`

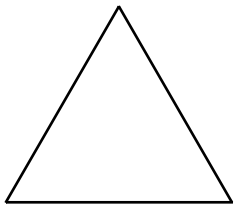
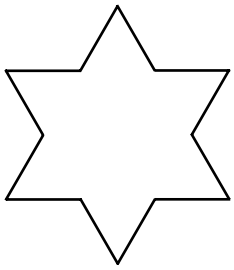
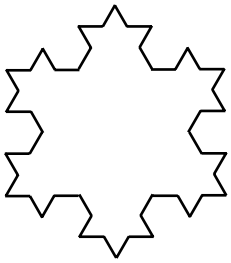
`–` `ROTATE -60°`

Чтобы оставлять размер ломанных постоянным и компенсировать их трехкратный рост на каждом шаге, при рисовании ломанной n-го поколения символу `F` следует ставить в соответствие команду `FORWARD` $3^{-n}$

| поколение | состояние |
|---|---|
| 0 | `F++F++F` |
| 1 | `F++F++F–F++F++F++F++F++F–F++F++F` |

| | |
|---|---|
| 2 | F–F++F–F++F–F++F–F++F–F++F–F–F–F++F–F++F–F++F–F++F–F+<br>+F–F++F–F++F–F++F–F++F–F++F–F++F–F++F–F–F–F++F–F++F–F++F–F<br>++F–F++F–F |

Длина строки растет весьма быстро.

| поколение | изображение |
|---|---|
| 0 |  |
| 1 |  |
| 2 |  |

При бесконечном повторении этой процедуры мы получим фрактальную кривую известную как «Снежинка Коха».

## 5.4 Скобочные L-системы и деревья

Рассмотрим следующую L-систему:

Аксиома:

X

Правила:

F→FF

X→F[+X]F[–X]+X

Интепретация:

F FORWARD 1

\+ ROTATE 20°

– ROTATE -20°

[ SAVE

] RESTORE

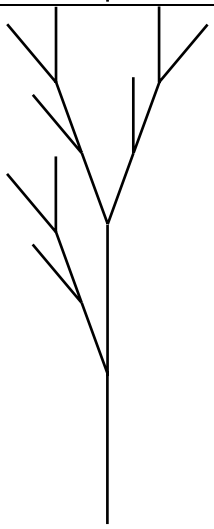
Здесь в отличии от всех предыдущих примеров начальное направление «черепашки» не вправо, а вверх.

При интерпретации символа [ «текущая» черепашка прерывает свою работу, создает новую черепашку, которая интерпретирует все вплоть до символа ], после чего самоуничтожается. Текущая черепашка продолжает свою работу с той точки в рисунке, на которой она прервалась.

| поколение | состояние |
|---|---|
| 0 | X |
| 1 | F[+X]F[–X]+X |
| 2 | FF[+F[+X]F[–X]+X]FF[–F[+X]F[–X]+X]+F[+X]F[–X]+X |
| 3 | FFFF[+FF[+F[+X]F[–X]+X]FF[–F[+X]F[–X]+X]+F[+X]F[–X]+X]FFFF[–FF[+F[+X]F[–X]+X]FF[–F[+X]F[–X]+X]+F[+X]F[–X]+X]+FF[+F[+X]F[–X]+X]FF[–F[+X]F[–X]+X]+F[+X]F[–X]+X |

Символ X не интерпретируется при рисовании, он лишь указывает «невидимую» почку, из которой в следующем поколении вырастет ветка.

| поколение | изображение |
|---|---|
| 0 | |

| | |
|---|---|
| 1 |  |
| 2 |  |
| 3 |  |

# Глава 6. Реализация с доказанной правильностью.

Формальная спецификация семантики будет полезной при реализации языка программирования. В частности становится возможным рассуждение о корректности реализации. Мы проиллюстрируем это на примере перевода языка WHILE в структурную форму ассемблерного кода абстрактной машины и докажем, что этот перевод корректен. Идея заключается в том, что мы сначала определяем смысл (значение) инструкций абстрактной машины как операционную семантику, затем определяем функции ПЕРЕВОДА, которые отображают выражения и операторы языка WHILE в последовательность инструкций абстрактной машины. Корректность реализации языка программирования будет состоять в том, что если мы

- переведем программу в код и
- выполним код на абстрактной машине,

то получим тот же самый результат, специфицируемый семантическими функциями $\mathcal{S}_{ns}$ и $\mathcal{S}_{sos}$ [1].

## 6.1 Абстрактная машина

Абстрактная машина AM имеет конфигурации вида $\langle c, e, s\rangle$ , где

- c – последовательность инструкций (кода) (еще не выполненных)
- e – стек (для вычислений)
- s – память.

Мы используем стек для вычисления арифметических и булевых выражений $e \in Stack$, $Stack=(Z \cup T)^*$.

По соображениям простоты мы примем, что память подобна состоянию [1], то есть $s \in State$ и используется для хранения значений переменных.

Синтаксис инструкций AM задается следующей грамматикой:

inst ::= PUSH-*n* | ADD | MULT | SUB | TRUE | FALSE | EQ | LE |
| AND | NEG | FETCH-*x* | STORE-*x* | NOOP| BRANCH(c,c) |

| LOOP(c,c)

c ::= ε | inst:c

Будем писать Code для обозначения синтаксической категории последовательности инструкций, то есть c – метапеременная категории Code (c∈Code).

Таким образом, мы имеем

⟨c, e, s⟩ ∈ Code×Stack×State

Конфигурация называется терминальной (конечной) если она имеет вид:

⟨ε, e, s⟩.

Семантику инструкций абстрактной машины зададим операционной семантикой. Отношение перехода имеют вид:

⟨c, e, s⟩ ▷ ⟨c', e', s'⟩

То есть один шаг выполнения преобразует конфигурацию ⟨c, e, s⟩ в конфигурацию ⟨c', e', s'⟩.

Отношение определяется аксиомами таблицы 6.1:

| | | |
|---|---|---|
| ⟨PUSH-$n$:c, e, s⟩ | ▷ | ⟨c, $\mathcal{N}$[$n$]:e, s⟩ |
| ⟨ADD:c, $z_1$:$z_2$:e, s⟩ | ▷ | ⟨c, ($z_1$+$z_2$):e, s⟩, если $z_1$:$z_2$∈**Z** |
| ⟨MULT:c, $z_1$:$z_2$:e, s⟩ | ▷ | ⟨c, ($z_1$*$z_2$):e, s⟩, если $z_1$:$z_2$∈**Z** |
| ⟨SUB:c, $z_1$:$z_2$:e, s⟩ | ▷ | ⟨c, ($z_1$−$z_2$):e, s⟩, если $z_1$:$z_2$∈**Z** |
| ⟨TRUE:c, e, s⟩ | ▷ | ⟨c, tt:e, s⟩ |
| ⟨FALSE:c, e, s⟩ | ▷ | ⟨c, ff:e, s⟩ |
| ⟨EQ:c, $z_1$:$z_2$:e, s⟩ | ▷ | ⟨c, ($z_1$=$z_2$):e, s⟩, если $z_1$:$z_2$∈**Z** |
| ⟨LE:c, $z_1$:$z_2$:e, s⟩ | ▷ | ⟨c, ($z_1$≤$z_2$):e, s⟩, если $z_1$:$z_2$∈**Z** |
| ⟨AND:c, $t_1$:$t_2$:e, s⟩ | ▷ | ⟨c, tt:e, s⟩, если $t_1$=tt и $t_2$=tt, $t_1$, $t_2$∈**T** |
| ⟨AND:c, $t_1$:$t_2$:e, s⟩ | ▷ | ⟨c, ff:e, s⟩, если $t_1$=ff или $t_2$=ff, $t_1$, $t_2$∈**T** |
| ⟨NEG:c, t:e, s⟩ | ▷ | ⟨c, ff:e, s⟩, если t=tt, t∈**T** |
| ⟨NEG:c, t:e, s⟩ | ▷ | ⟨c, tt:e, s⟩, если t=ff, t∈**T** |
| ⟨FETCH-$x$:c, e, s⟩ | ▷ | ⟨c, (s $x$):e, s⟩ |
| ⟨STORE-$x$:c, z:e, s⟩ | ▷ | ⟨c, e, s[$x$↦z]⟩, если z∈**Z** |
| ⟨NOOP:c, e, s⟩ | ▷ | ⟨c, e, s⟩ |
| ⟨BRANCH($c_1$,$c_2$):c, t:e, s⟩ | ▷ | ⟨$c_1$, e, s⟩, если t=tt, t∈**T** |
| ⟨BRANCH($c_1$,$c_2$):c, t:e, s⟩ | ▷ | ⟨$c_2$, e, s⟩, если t=ff, t∈**T** |
| ⟨LOOP($c_1$,$c_2$):c, e, s⟩ | ▷ | ⟨$c_1$:BRANCH($c_2$:LOOP($c_1$,$c_2$),NOOP):c, e, s⟩ |

В добавлении к обычным арифметическим и булевым операциям мы имеем 5 инструкций, которые модифицируют стек: PUSH-*n* заносит значение константы в стек, TRUE и FALSE значения констант tt и ff в стек, FETCH-*x* заносит значение связанное с переменной *x* в стек. STORE-*x* выталкивает верхний элемент из стека и делает его значением переменной *x*.

Для данных c – последовательности инструкций и s – памяти вычислительной последовательностью является или

- конечная последовательность $\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_k$ конфигураций $\gamma_0=\langle c, \varepsilon, s\rangle$ и $\gamma_i \triangleright \gamma_{i+1}$ $(0\le i<k, k\ge 0)$ и не существует $\gamma$ такая, что $\gamma_k \triangleright \gamma$, или
- бесконечная последовательность $\gamma_0, \gamma_1, \ldots,$ конфигурация $\gamma_0=\langle c, \varepsilon, s\rangle$ и $\gamma_i \triangleright \gamma_{i+1}$ $(0\le i)$

Заметим, что начальная конфигурация всегда имеет пустой стек.

Вычислительная последовательность *завершается* тогда и только тогда, когда она конечная.

Вычислительная последовательность *зацикливается* тогда и только тогда, когда она бесконечная.

Завершающаяся вычислительная последовательность может заканчиваться терминальной конфигурацией ($\langle \varepsilon,$ e, s$\rangle$) или *тупиковой* конфигурацией (например, $\langle$ADD, $\varepsilon$, s$\rangle$).

Пример 6.1

Построим завершающуюся вычислительную последовательность для программы PUSH-1:FETCH-*x*:ADD:STORE-*x* в состоянии в котором s *x* = 3:

$\langle$PUSH-1:FETCH-*x*:ADD:STORE-*x*, $\varepsilon$, s$\rangle$ $\triangleright$

$\langle$FETCH-*x*:ADD:STORE-*x*, 1, s$\rangle$ $\triangleright$

$\langle$ADD:STORE-*x*, 3:1, s$\rangle$ $\triangleright$

$\langle$STORE-*x*, 4, s$\rangle$ $\triangleright$

$\langle \varepsilon, \varepsilon, s[x\mapsto 4]\rangle$

Пример 6.2

Построим теперь зацикливающуюся вычислительную последовательность для программы LOOP(TRUE,NOOP):

⟨LOOP(TRUE,NOOP), ε, s⟩ ▷

⟨TRUE:BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP), ε, s⟩ ▷

⟨BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP), tt, s⟩ ▷

⟨NOOP:LOOP(TRUE,NOOP), ε, s⟩ ▷

⟨LOOP(TRUE,NOOP), ε, s⟩ ▷ … ▷

Упражнение 6.3

Построить вычислительную последовательность для программы:

PUSH-0:STORE-*z*:FETCH-*x*:STORE-*r*:
LOOP(FETCH-*r*:FETCH-*y*:LE,
FETCH-*y*:FETCH-*r*:SUB:STORE-*r*:
PUSH-1:FETCH-*z*:ADD:STORE-*z*)

Определить функцию вычисляемую этой программой.

Мы определим смысл (значение) последовательности инструкций как частичную функцию из State в State то есть

$\mathcal{M}$:Code→(State→State) или

$$\mathcal{M}\ [\![c]\!]\ s = \begin{cases} s', \textit{если } \langle c, \varepsilon, s\rangle \triangleright^* \langle \varepsilon, \varepsilon, s'\rangle \\ \textit{undef}, \text{ в противном случае} \end{cases}$$

Ниже, в параграфе 6.3 мы рассмотрим различные модификации абстрактной машины AM. Абстрактная машина $AM_1$ отличается тем, что ссылается на переменные не по их именам, а по их адресам. В абстрактной машине $AM_2$ убираются инструкции BRANCH и LOOP и, добавляются инструкции метки и перехода (условного и безусловного). В абстрактной машине $AM_3$ убирается инструкция метки, а параметрами инструкций переходов становится не метка, а абсолютный адрес инструкции в последовательности инструкций.

## 6.2 Определение перевода

Определим три функции:

$\mathcal{CA} : \mathcal{A}_{exp} \rightarrow \text{Code}$

$\mathcal{CB} : \mathcal{B}_{exp} \rightarrow \text{Code}$

$\mathcal{CS} : \text{Stm} \rightarrow \text{Code}$

Первые две функции задаются следующей таблицей 6.2:

| | | |
|---|---|---|
| $\mathcal{CA}[\![n]\!]$ | = | PUSH-$n$ |
| $\mathcal{CA}[\![x]\!]$ | = | FETCH-$x$ |
| $\mathcal{CA}[\![a_1+a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:ADD |
| $\mathcal{CA}[\![a_1*a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:MULT |
| $\mathcal{CA}[\![a_1-a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:SUB |
| $\mathcal{CA}[\![(a)]\!]$ | = | $\mathcal{CA}[\![a]\!]$ |
| $\mathcal{CB}[\![true]\!]$ | = | TRUE |
| $\mathcal{CB}[\![false]\!]$ | = | FALSE |
| $\mathcal{CB}[\![a_1=a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:EQ |
| $\mathcal{CB}[\![a_1\leq a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:LE |
| $\mathcal{CB}[\![\neg b]\!]$ | = | $\mathcal{CB}[\![b]\!]$:NEG |
| $\mathcal{CB}[\![b_1\wedge b_2]\!]$ | = | $\mathcal{CB}[\![b_2]\!]$:$\mathcal{CB}[\![b_1]\!]$:AND |
| $\mathcal{CB}[\![(b)]\!]$ | = | $\mathcal{CB}[\![b]\!]$ |

Пример 6.4

Для арифметического выражения $x$+1 языка WHILE мы получим код следующим образом:

$\mathcal{CA}[\![x+1]\!] = \mathcal{CA}[\![1]\!]:\mathcal{CA}[\![x]\!]$:ADD = PUSH-1:FETCH-$x$:ADD

Перевод операторов задается таблицей 6.3:

| | | |
|---|---|---|
| $\mathcal{CS}[\![x:=a]\!]$ | = | $\mathcal{CA}[\![(a)]\!]$:STORE-$x$ |
| $\mathcal{CS}[\![skip]\!]$ | = | NOOP |
| $\mathcal{CS}[\![S_1;S_2]\!]$ | = | $\mathcal{CS}[\![S_1]\!]:\mathcal{CS}[\![S_2]\!]$ |
| $\mathcal{CS}[\![\textit{if } b \textit{ then } S_1 \textit{ else } S_2]\!]$ | = | $\mathcal{CB}[\![b]\!]$:BRANCH($\mathcal{CS}[\![S_1]\!],\mathcal{CS}[\![S_2]\!]$) |
| $\mathcal{CS}[\![\textit{while } b \textit{ do } S]\!]$ | = | LOOP($\mathcal{CB}[\![b]\!],\mathcal{CS}[\![S]\!]$) |
| $\mathcal{CS}[\![(S)]\!]$ | = | $\mathcal{CS}[\![S]\!]$ |

Пример 6.5

Для программы:

$y$:=1; *while* ¬ ($x$=1) *do* ($y$:=$y$∗$x$; $x$:=$x$−1) получим следующий код:

$\mathcal{CS}$**[**$y$:=1; *while* ¬ ($x$=1) *do* ($y$:=$y$∗$x$; $x$:=$x$−1)**]** =

$\mathcal{CS}$**[**$y$:=1**]**:$\mathcal{CS}$**[***while* ¬ ($x$=1) *do* ($y$:=$y$∗$x$; $x$:=$x$−1)**]** =

$\mathcal{CA}$**[**1**]**:STORE-$y$:LOOP($\mathcal{CB}$**[**¬($x$=1)**]**,$\mathcal{CS}$**[**($y$:=$y$∗$x$; $x$:=$x$−1)**]**) =

PUSH-1: STORE-$y$:LOOP($\mathcal{CB}$**[**($x$=1)**]**:NEG,$\mathcal{CS}$**[**$y$:=$y$∗$x$; $x$:=$x$−1**]**) =

PUSH-1: STORE-$y$:LOOP($\mathcal{CB}$**[**$x$=1**]**:NEG,$\mathcal{CS}$**[**$y$:=$y$∗$x$**]**:$\mathcal{CS}$**[**$x$:=$x$−1**]**) =

PUSH-1: STORE-$y$:LOOP($\mathcal{CA}$**[**1**]**:$\mathcal{CA}$**[**$x$**]**:EQ:NEG,
$\mathcal{CA}$**[**$y$∗$x$**]**:STORE-$y$:$\mathcal{CA}$**[**$x$−1**]**:STORE-$x$) =

PUSH-1: STORE-$y$:LOOP($\mathcal{CA}$**[**1**]**:$\mathcal{CA}$**[**$x$**]**:EQ:NEG,
$\mathcal{CA}$**[**$x$**]**:$\mathcal{CA}$**[**$y$**]**:MULT:STORE-$y$:
$\mathcal{CA}$**[**1**]**:$\mathcal{CA}$**[**$x$**]**:SUB:STORE-$x$) =

PUSH-1: STORE-$y$:LOOP(PUSH-1:FETCH-$x$:EQ:NEG,
FETCH-$x$:FETCH-$y$:MULT:STORE-$y$:
PUSH-1:FETCH-$x$:SUB:STORE-$x$)

## 6.3 Модификации абстрактной машины

Абстрактная машина $AM_1$ имеет конфигурацию ⟨c, e, m⟩, где m – память. Ее можно интерпретировать как одномерный массив целых чисел (напомним, что переменные у нас только целые). При переводе с языка WHILE на язык абстрактной машины можно воспользоваться таблицей идентификаторов, которую построит компилятор. Тогда индекс массива можно рассматривать как адрес переменной.

Грамматика абстрактной машины $AM_1$:

inst ::= PUSH-$n$ | ADD | MULT | SUB | TRUE | FALSE | EQ | LE |
| AND | NEG | GET-$n$ | PUT-$n$ | NOOP| BRANCH(c,c) |
| LOOP(c,c)

c ::= ε | inst:c

Отношение определяется аксиомами таблицы 6.4:

| | | |
|---|---|---|
| $\langle$PUSH-$n$:c,e,m$\rangle$ | $\triangleright$ | $\langle$c,$\mathcal{N}$**[**$n$**]**:e,m$\rangle$ |
| $\langle$ADD:c,$z_1$:$z_2$:e,m$\rangle$ | $\triangleright$ | $\langle$c,($z_1$+$z_2$):e,m$\rangle$, если $z_1$:$z_2\in$**Z** |
| $\langle$MULT:c,$z_1$:$z_2$:e,m$\rangle$ | $\triangleright$ | $\langle$c,($z_1$*$z_2$):e,m$\rangle$, если $z_1$:$z_2\in$**Z** |
| $\langle$SUB:c,$z_1$:$z_2$:e,m$\rangle$ | $\triangleright$ | $\langle$c,($z_1-z_2$):e,m$\rangle$, если $z_1$:$z_2\in$**Z** |
| $\langle$TRUE:c,e,m$\rangle$ | $\triangleright$ | $\langle$c,tt:e,m$\rangle$ |
| $\langle$FALSE:c,e,m$\rangle$ | $\triangleright$ | $\langle$c,ff:e,m$\rangle$ |
| $\langle$EQ:c,$z_1$:$z_2$:e,m$\rangle$ | $\triangleright$ | $\langle$c,($z_1$=$z_2$):e,m$\rangle$, если $z_1$:$z_2\in$**Z** |
| $\langle$LE:c,$z_1$:$z_2$:e,m$\rangle$ | $\triangleright$ | $\langle$c,($z_1\le z_2$):e,m$\rangle$, если $z_1$:$z_2\in$**Z** |
| $\langle$AND:c,$t_1$:$t_2$:e,m$\rangle$ | $\triangleright$ | $\langle$c,tt:e,m$\rangle$, если $t_1$=tt и $t_2$=tt, $t_1$, $t_2\in$**T** |
| $\langle$AND:c,$t_1$:$t_2$:e,m$\rangle$ | $\triangleright$ | $\langle$c,ff:e,m$\rangle$, если $t_1$=ff или $t_2$=ff, $t_1$, $t_2\in$**T** |
| $\langle$NEG:c,t:e,m$\rangle$ | $\triangleright$ | $\langle$c,ff:e,m$\rangle$, если t=tt, если t$\in$**T** |
| $\langle$NEG:c,t:e,m$\rangle$ | $\triangleright$ | $\langle$c,tt:e,m$\rangle$, если t=ff, если t$\in$**T** |
| $\langle$GET-$n$:c,e,m$\rangle$ | $\triangleright$ | $\langle$c,m[$n$]:e,m$\rangle$ |
| $\langle$PUT-$n$:c,z:e,m$\rangle$ | $\triangleright$ | $\langle$c,e,m[$n$]$\mapsto$z$\rangle$, если z$\in$**Z** |
| $\langle$NOOP:c,e,m$\rangle$ | $\triangleright$ | $\langle$c,e,m$\rangle$ |
| $\langle$BRANCH($c_1$,$c_2$):c,t:e,m$\rangle$ | $\triangleright$ | $\langle c_1$,e,m$\rangle$, если t=tt, если t$\in$**T** |
| $\langle$BRANCH($c_1$,$c_2$):c,t:e,m$\rangle$ | $\triangleright$ | $\langle c_2$,e,m$\rangle$, если t=ff, если t$\in$**T** |
| $\langle$LOOP($c_1$,$c_2$):c,e,m$\rangle$ | $\triangleright$ | $\langle c_1$:BRANCH($c_2$:LOOP($c_1$,$c_2$),NOOP):c,e,m$\rangle$ |

Перевод задается таблицами 6.5 и 6.6:

Таблица 6.5

| | | |
|---|---|---|
| $\mathcal{CA}$**[**$n$**]** | = | PUSH-$n$ |
| $\mathcal{CA}$**[**$x$**]** | = | GET-$n$, где $n$ – номер строки в таблице идентификаторов для $x$ |
| $\mathcal{CA}$**[**$a_1$+$a_2$**]** | = | $\mathcal{CA}$**[**$a_2$**]**:$\mathcal{CA}$**[**$a_1$**]**:ADD |
| $\mathcal{CA}$**[**$a_1$*$a_2$**]** | = | $\mathcal{CA}$**[**$a_2$**]**:$\mathcal{CA}$**[**$a_1$**]**:MULT |
| $\mathcal{CA}$**[**$a_1-a_2$**]** | = | $\mathcal{CA}$**[**$a_2$**]**:$\mathcal{CA}$**[**$a_1$**]**:SUB |
| $\mathcal{CA}$**[**($a$)**]** | = | $\mathcal{CA}$**[**$a$**]** |
| $\mathcal{CB}$**[**$true$**]** | = | TRUE |
| $\mathcal{CB}$**[**$false$**]** | = | FALSE |
| $\mathcal{CB}$**[**$a_1$=$a_2$**]** | = | $\mathcal{CA}$**[**$a_2$**]**:$\mathcal{CA}$**[**$a_1$**]**:EQ |
| $\mathcal{CB}$**[**$a_1\le a_2$**]** | = | $\mathcal{CA}$**[**$a_2$**]**:$\mathcal{CA}$**[**$a_1$**]**:LE |
| $\mathcal{CB}$**[**$\neg b$**]** | = | $\mathcal{CB}$**[**$b$**]**:NEG |
| $\mathcal{CB}$**[**$b_1\wedge b_2$**]** | = | $\mathcal{CB}$**[**$b_2$**]**:$\mathcal{CB}$**[**$b_1$**]**:AND |
| $\mathcal{CB}$**[**($b$)**]** | = | $\mathcal{CB}$**[**$b$**]** |

Таблица 6.6

| | | |
|---|---|---|
| $\mathcal{CS}$**[**$x$:=$a$**]** | = | $\mathcal{CA}$**[**($a$)**]**:PUT-$n$, где $n$ – номер строки в таблице идентификаторов для $x$ |

| | | |
|---|---|---|
| $\mathcal{CS}$[*skip*] | = | NOOP |
| $\mathcal{CS}$[$S_1$;$S_2$] | = | $\mathcal{CS}$[$S_1$]:$\mathcal{CS}$[$S_2$] |
| $\mathcal{CS}$[*if b then* $S_1$ *else* $S_2$] | = | $\mathcal{CB}$[*b*]:BRANCH($\mathcal{CS}$[$S_1$],$\mathcal{CS}$[$S_2$]) |
| $\mathcal{CS}$[*while b do* S] | = | LOOP($\mathcal{CB}$[*b*],$\mathcal{CS}$[S]) |
| $\mathcal{CS}$[(S)] | = | $\mathcal{CS}$[S] |

Абстрактная машина $AM_2$ имеет конфигурацию ⟨pc, c, e, m⟩, где pc – счетчик команд – целое число большее нуля – номер следующей выполняемой инструкции из последовательности инструкций. Для $AM_2$ выполнение каждой инструкции не приводит к исключению ее из последовательности, так как есть вероятность повторного ее выполнения (если будет выполнен переход).

Грамматика абстрактной машины $AM_2$:

inst ::= PUSH-*n* | ADD | MULT | SUB | TRUE | FALSE | EQ | LE |
| AND | NEG | GET-*n* | PUT-*n* | NOOP | LABEL-*n* | JUMP-*n* |
| JUMPFALSE-*n*

c ::= ε | inst:c

Семантика инструкций $AM_2$ задается операционной семантикой. Отношение перехода имеют вид:

⟨pc, c, e, m⟩ ▷ ⟨pc', c, e', m'⟩

То есть один шаг выполнения преобразует конфигурацию ⟨pc, c, e, m⟩ в конфигурацию ⟨pc', c, e', m'⟩.

Отношение определяется аксиомами таблицы 6.7:

| | | |
|---|---|---|
| ⟨pc,c[pc]=PUSH-*n*,e,m⟩ | ▷ | ⟨pc+1,c,$\mathcal{N}$[*n*]:e,m⟩ |
| ⟨pc,c[pc]=ADD,$z_1$:$z_2$:e,m⟩ | ▷ | ⟨pc+1,c,($z_1$+$z_2$):e,m⟩, если $z_1$:$z_2$∈**Z** |
| ⟨pc,c[pc]=MULT,$z_1$:$z_2$:e,m⟩ | ▷ | ⟨pc+1,c,($z_1$*$z_2$):e,m⟩, если $z_1$:$z_2$∈**Z** |
| ⟨pc,c[pc]=SUB,$z_1$:$z_2$:e,m⟩ | ▷ | ⟨pc+1,c,($z_1$−$z_2$):e,m⟩, если $z_1$:$z_2$∈**Z** |
| ⟨pc,c[pc]=TRUE,e,m⟩ | ▷ | ⟨pc+1,c,tt:e,m⟩ |
| ⟨pc,c[pc]=FALSE,e,m⟩ | ▷ | ⟨pc+1,c,ff:e,m⟩ |
| ⟨pc,c[pc]=EQ,$z_1$:$z_2$:e,m⟩ | ▷ | ⟨pc+1,c,($z_1$=$z_2$):e,m⟩, если $z_1$:$z_2$∈**Z** |
| ⟨pc,c[pc]=LE,$z_1$:$z_2$:e,m⟩ | ▷ | ⟨pc+1,c,($z_1$≤$z_2$):e,m⟩, если $z_1$:$z_2$∈**Z** |
| ⟨pc,c[pc]=AND,$t_1$:$t_2$:e,m⟩ | ▷ | ⟨pc+1,c,tt:e,m⟩, если $t_1$=tt и $t_2$=tt, $t_1$, $t_2$∈**T** |
| ⟨pc,c[pc]=AND,$t_1$:$t_2$:e,m⟩ | ▷ | ⟨pc+1,c,ff:e,m⟩, если $t_1$=ff или $t_2$=ff, $t_1$, $t_2$∈**T** |

| | |
|---|---|
| $\langle pc,c[pc]=NEG,t:e,m\rangle$ | $\triangleright \langle pc+1,c,ff:e,m\rangle$, если $t=tt$, $t\in\mathbf{T}$ |
| $\langle pc,c[pc]=NEG,t:e,m\rangle$ | $\triangleright \langle pc+1,c,tt:e,m\rangle$, если $t=ff$, $t\in\mathbf{T}$ |
| $\langle pc,c[pc]=GET\text{-}n,e,m\rangle$ | $\triangleright \langle pc+1,c,m[n]:e,m\rangle$ |
| $\langle pc,c[pc]=PUT\text{-}n,z:e,m\rangle$ | $\triangleright \langle pc+1,c,e,m[n]\mapsto z\rangle$, если $z\in\mathbf{Z}$ |
| $\langle pc,c[pc]=NOOP,e,m\rangle$ | $\triangleright \langle pc+1,c,e,m\rangle$ |
| $\langle pc,c[pc]=LABEL\text{-}n,e,m\rangle$ | $\triangleright \langle pc+1,c,e,m\rangle$ |
| $\langle pc,c[pc]=JUMP\text{-}n,e,m\rangle$ | $\triangleright \langle pc',c,e,m\rangle$, $c[pc']=LABEL\text{-}n$ |
| $\langle pc,c[pc]=JUMPFALSE\text{-}n,t:e,m\rangle$ | $\triangleright \langle pc+1,c,e,m\rangle$, если $t=tt$, $t\in\mathbf{T}$ |
| $\langle pc,c[pc]=JUMPFALSE\text{-}n,t:e,m\rangle$ | $\triangleright \langle pc',c,e,m\rangle$, $c[pc']=LABEL\text{-}n$, если $t=ff$, $t\in\mathbf{T}$ |

Перевод задается таблицами 6.8 и 6.9:

Таблица 6.8

| | | |
|---|---|---|
| $\mathcal{CA}[\![n]\!]$ | = | PUSH-$n$ |
| $\mathcal{CA}[\![x]\!]$ | = | GET-$n$, где $n$ – номер строки в тблице идентификаторов для $x$ |
| $\mathcal{CA}[\![a_1+a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]:\mathcal{CA}[\![a_1]\!]$:ADD |
| $\mathcal{CA}[\![a_1*a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]:\mathcal{CA}[\![a_1]\!]$:MULT |
| $\mathcal{CA}[\![a_1-a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]:\mathcal{CA}[\![a_1]\!]$:SUB |
| $\mathcal{CA}[\![(a)]\!]$ | = | $\mathcal{CA}[\![a]\!]$ |
| $\mathcal{CB}[\![true]\!]$ | = | TRUE |
| $\mathcal{CB}[\![false]\!]$ | = | FALSE |
| $\mathcal{CB}[\![a_1=a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]:\mathcal{CA}[\![a_1]\!]$:EQ |
| $\mathcal{CB}[\![a_1\le a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]:\mathcal{CA}[\![a_1]\!]$:LE |
| $\mathcal{CB}[\![\neg b]\!]$ | = | $\mathcal{CB}[\![b]\!]$:NEG |
| $\mathcal{CB}[\![b_1\wedge b_2]\!]$ | = | $\mathcal{CB}[\![b_2]\!]:\mathcal{CB}[\![b_1]\!]$:AND |
| $\mathcal{CB}[\![(b)]\!]$ | = | $\mathcal{CB}[\![b]\!]$ |

Таблица 6.9

| | | |
|---|---|---|
| $\mathcal{CS}[\![x:=a]\!]$ | = | $\mathcal{CA}[\![(a)]\!]$:PUT-$n$, где $n$ – номер строки в таблице идентификаторов для $x$ |
| $\mathcal{CS}[\![skip]\!]$ | = | NOOP |
| $\mathcal{CS}[\![S_1;S_2]\!]$ | = | $\mathcal{CS}[\![S_1]\!]:\mathcal{CS}[\![S_2]\!]$ |
| $\mathcal{CS}[\![\textit{if } b \textit{ then } S_1 \textit{ else } S_2]\!]$ | = | $\mathcal{CB}[\![b]\!]$:JUMPFALSE-$n_1$:$\mathcal{CS}[\![S_1]\!]$:JUMP-$n_2$: LABEL-$n_1$:$\mathcal{CS}[\![S_2]\!]$:LABEL-$n_2$ |
| $\mathcal{CS}[\![\textit{while } b \textit{ do } S]\!]$ | = | LABEL-$n_1$:$\mathcal{CB}[\![b]\!]$:JUMPFALSE-$n_2$:$\mathcal{CS}[\![S]\!]$:JUMP-$n_1$: LABEL-$n_2$ |
| $\mathcal{CS}[\![(S)]\!]$ | = | $\mathcal{CS}[\![S]\!]$ |

Абстрактная машина $AM_3$ отличается от $AM_2$ тем, что убирается инструкция LABEL-$n$, и аргументы в инструкциях JUMP-$n$ и JUMPFALSE-$n$ интерпретируются как адреса (номера инструкций от начала последовательности).

Отношение определяется аксиомами таблицы 6.10:

| | |
|---|---|
| $\langle pc,c[pc]=PUSH\text{-}n,e,m\rangle$ | $\triangleright \langle pc+1,c,\mathcal{N}[\![n]\!]:e,m\rangle$ |
| $\langle pc,c[pc]=ADD,z_1:z_2:e,m\rangle$ | $\triangleright \langle pc+1,c,(z_1+z_2):e,m\rangle$, если $z_1:z_2\in\mathbf{Z}$ |
| $\langle pc,c[pc]=MULT,z_1:z_2:e,m\rangle$ | $\triangleright \langle pc+1,c,(z_1*z_2):e,m\rangle$, если $z_1:z_2\in\mathbf{Z}$ |
| $\langle pc,c[pc]=SUB,z_1:z_2:e,m\rangle$ | $\triangleright \langle pc+1,c,(z_1-z_2):e,m\rangle$, если $z_1:z_2\in\mathbf{Z}$ |
| $\langle pc,c[pc]=TRUE,e,m\rangle$ | $\triangleright \langle pc+1,c,tt:e,m\rangle$ |
| $\langle pc,c[pc]=FALSE,e,m\rangle$ | $\triangleright \langle pc+1,c,ff:e,m\rangle$ |
| $\langle pc,c[pc]=EQ,z_1:z_2:e,m\rangle$ | $\triangleright \langle pc+1,c,(z_1=z_2):e,m\rangle$, если $z_1:z_2\in\mathbf{Z}$ |
| $\langle pc,c[pc]=LE,z_1:z_2:e,m\rangle$ | $\triangleright \langle pc+1,c,(z_1\le z_2):e,m\rangle$, если $z_1:z_2\in\mathbf{Z}$ |
| $\langle pc,c[pc]=AND,t_1:t_2:e,m\rangle$ | $\triangleright \langle pc+1,c,tt:e,m\rangle$, если $t_1=tt$ и $t_2=tt$, $t_1, t_2\in\mathbf{T}$ |
| $\langle pc,c[pc]=AND,t_1:t_2:e,m\rangle$ | $\triangleright \langle pc+1,c,ff:e,m\rangle$, если $t_1=ff$ или $t_2=ff$, $t_1, t_2\in\mathbf{T}$ |
| $\langle pc,c[pc]=NEG,t:e,m\rangle$ | $\triangleright \langle pc+1,c,ff:e,m\rangle$, если $t=tt$, $t\in\mathbf{T}$ |
| $\langle pc,c[pc]=NEG,t:e,m\rangle$ | $\triangleright \langle pc+1,c,tt:e,m\rangle$, если $t=ff$, $t\in\mathbf{T}$ |
| $\langle pc,c[pc]=GET\text{-}n,e,m\rangle$ | $\triangleright \langle pc+1,c,m[n]:e,m\rangle$ |
| $\langle pc,c[pc]=PUT\text{-}n,z:e,m\rangle$ | $\triangleright \langle pc+1,c,e,m[n]\mapsto z\rangle$, если $z\in\mathbf{Z}$ |
| $\langle pc,c[pc]=NOOP,e,m\rangle$ | $\triangleright \langle pc+1,c,e,m\rangle$ |
| $\langle pc,c[pc]=JUMP\text{-}n,e,m\rangle$ | $\triangleright \langle n,c,e,m\rangle$ |
| $\langle pc,c[pc]=JUMPFALSE\text{-}n,t:e,m\rangle$ | $\triangleright \langle pc+1,c,e,m\rangle$, если $t=tt$, $t\in\mathbf{T}$ |
| $\langle pc,c[pc]=JUMPFALSE\text{-}n,t:e,m\rangle$ | $\triangleright \langle n,c,e,m\rangle$, если $t=ff$, $t\in\mathbf{T}$ |

Перевод осуществляется сначала с использованием таблиц 6.8 и 6.9 абстрактной машины $AM_2$. Затем, для каждой инструкции LABEL-*n* вычисляем номер инструкции следующей за ней в последовательности. Сами инструкции LABEL не считаем! После этого заменяем в инструкциях JUMP и JUMPFALSE их аргументы на вычисленные значения. Перевод окончен.

Пример 6.6

Выполним перевод программы из примера 6.5 сначала для $AM_2$, а затем и для $AM_3$. Будем считать, что в таблице идентификаторов 2 строчки: 1 соответствует переменной *x*, 2 – переменной *y*.

$\mathcal{CS}[\![y:=1;\ while\ \neg\ (x=1)\ do\ (y:=y*x;\ x:=x-1)]\!] =$

$\mathcal{CS}[\![y:=1]\!]:\mathcal{CS}[\![while\ \neg\ (x=1)\ do\ (y:=y*x;\ x:=x-1)]\!] =$

$\mathcal{CA}[\![1]\!]$:PUT-2:LABEL-1:$\mathcal{CB}[\![\neg(x=1)]\!]$:JUMPFALSE-2:$\mathcal{CS}[\![(y:=y*x;$

$x:=x-1)]\!]$:JUMP-1:LABEL-2 =

PUSH-1:PUT-2:LABEL-1:$\mathcal{CB}$[($x$=1)]:NEG:JUMPFALSE-2:$\mathcal{CS}$[$y$:=$y$∗$x$;

$x$:=$x$−1]:JUMP-1:LABEL-2 =

PUSH-1:PUT-2:LABEL-1:$\mathcal{CB}$[$x$=1]:NEG:JUMPFALSE-2:$\mathcal{CS}$[$y$:=$y$∗$x$]:

$\mathcal{CS}$[$x$:=$x$−1]:JUMP-1:LABEL-2 =

PUSH-1:PUT-2:LABEL-1:$\mathcal{CA}$[1]:$\mathcal{CA}$[$x$]:EQ:NEG:JUMPFALSE-2:

$\mathcal{CA}$[$y$∗$x$]:PUT-2:$\mathcal{CA}$[$x$−1]:PUT-1:JUMP-1:LABEL-2 =

PUSH-1:PUT-2:LABEL-1:PUSH-1:GET-1:EQ:NEG:JUMPFALSE-2:

$\mathcal{CA}$[$x$]:$\mathcal{CA}$[$y$]:MULT:PUT-2:$\mathcal{CA}$[1]:$\mathcal{CA}$[$x$]:SUB:PUT-1:JUMP-1:LABEL-2 =

PUSH-1:PUT-2:LABEL-1:PUSH-1:GET-1:EQ:NEG:JUMPFALSE-2:

GET-1:GET-2:MULT:PUT-2:PUSH-1:GET-1:SUB:PUT-1:JUMP-1:LABEL-2

Это конечный результат перевода в код абстрактной машины $АМ_2$. Для перевода этой же программы в код $АМ_3$ вычислим значения инструкций LABEL: LABEL-1 = 3, а LABEL-2 = 17. Заменим JUMP-1 на JUMP-3, а JUMPFALSE-2 на JUMPFALSE-17. Нам придется добавить в конец кода инструкцию NOOP под номером 17, так как LABEL-2 стояла в конце кода.

PUSH-1:PUT-2:PUSH-1:GET-1:EQ:NEG:JUMPFALSE-17:

GET-1:GET-2:MULT:PUT-2:PUSH-1:GET-1:SUB:PUT-1:JUMP-3:NOOP

Перевод окончен.

# Список использованных источников.

1. Мезенцев А.В. Теория вычислительных процессов и структур. Учебное пособие. – Иркутск, 2010. – 100 с. // math.isu.ru/ru/chairs/it/files/theory.pdf

2. Карпов Ю.Г. Теория и технология программирования. Основы построения трансляторов.-СПб.: БХВ-Петербург, 2005. -272 с.

3. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. (в 2-х томах) том 1. Синтаксический анализ. –М.: Мир, 1978. – 612 с.

4. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. (в 2-х томах) том 2. Компиляция. –М.: Мир, 1978. – 487 с.

5. Мозговой М.В. Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход. – СПб.: Наука и техника, 2006, - 320 с.

6. Nielson H.R., Nielson F. Semantics with Applications. A Formal Introduction. 1999, John Wiley & Sons. – 240 pp.