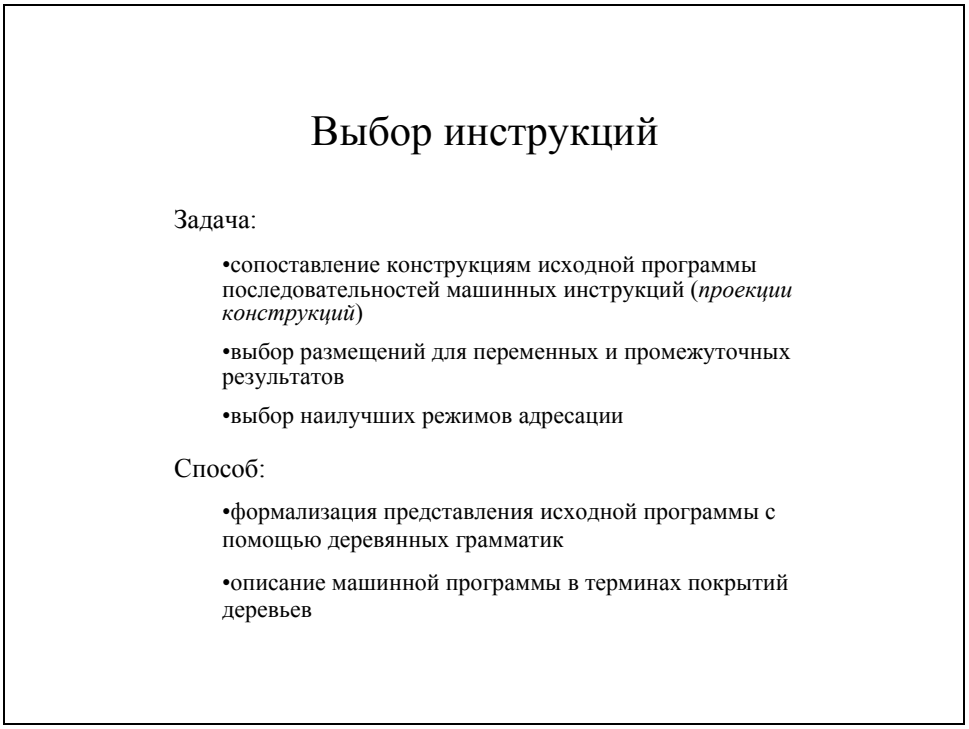
15. Выбор инструкций при генерации кода

- Постановка задачи
- Деревянные языки
- Деревянные грамматики
- BURS и ее приложения

Лекция 15. Выбор инструкций при генерации кода

В этой лекции рассматриваются следующие вопросы:

- Постановка задачи выбора оптимальных инструкций
- Деревянные языки
- Деревянные грамматики
- BURS и ее приложения



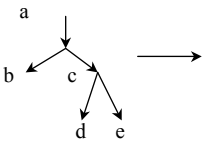
Выбор инструкций

Выбор инструкций – это одна из стадий генерации машинного кода. В ее задачу входит сопоставление высокоуровневым конструкциям исходного языка последовательностей реализующих их машинных команд (проекция конструкций), выбор размещений для переменных и временных значений и выбор подходящих режимов адресации для доступа к данным.

Практически все системы команд предоставляют разнообразные режимы адресации. Выбор способов представления операндов и конкретных инструкций для их обработки представляет нетривиальную проблему. К счастью, существует простой способ описания этого процесса, который при определенных допущениях дает оптимальный результат. Кроме того, описываемый подход дает возможность порождать генераторы кода по формальному описанию.

Для решения задачи построения генератора кода применяется теория деревянных языков и грамматик. Именно, входная программа разбивается на фрагменты (выражения), каждый из которых трактуется как предложение языка специального вида. Вывод этого предложения в грамматике, определяющей такой язык, затем интерпретируется как последовательность машинных инструкций.

Деревянные языки

Скобочная запись дерева: 

a (b, c (d, e))

Алфавит:

$A=\{l_1,l_2,...l_k\}$

Функция арности:

$\#: A \rightarrow \{0,1,2,...\}$

Множество всех деревьев в алфавите A:

$T^*_A=\{\varepsilon\} \cup \{a : a \in A, \#(a)=0\} \cup \{a\,(t_1,...,t_{\#(a)}) : a \in A, \#(a)>0, t_i \in T^*_A, t_i \neq \varepsilon\}$

Деревянный язык в алфавите A:

$L \subseteq T^*_A$

Деревянные языки

Деревянные языки позволяют описать множества деревьев, обладающих определенными свойствами (везде далее будут иметься в виду деревья, содержащие конечное число вершин). Мы будем использовать скобочную запись дерева, которая может быть получена выписыванием пометок вершин при простом обходе дерева слева-направо и сверху-вниз. При этом поддеревья одной вершины отделяются от нее скобками, а между собой – запятыми.

Для определения понятия деревянного языка введем в расмотрение алфавит A, снабженный взаимно-однозначной *функцией арности* #, которая приписывает буквам алфавита неотрицательные целые числа.

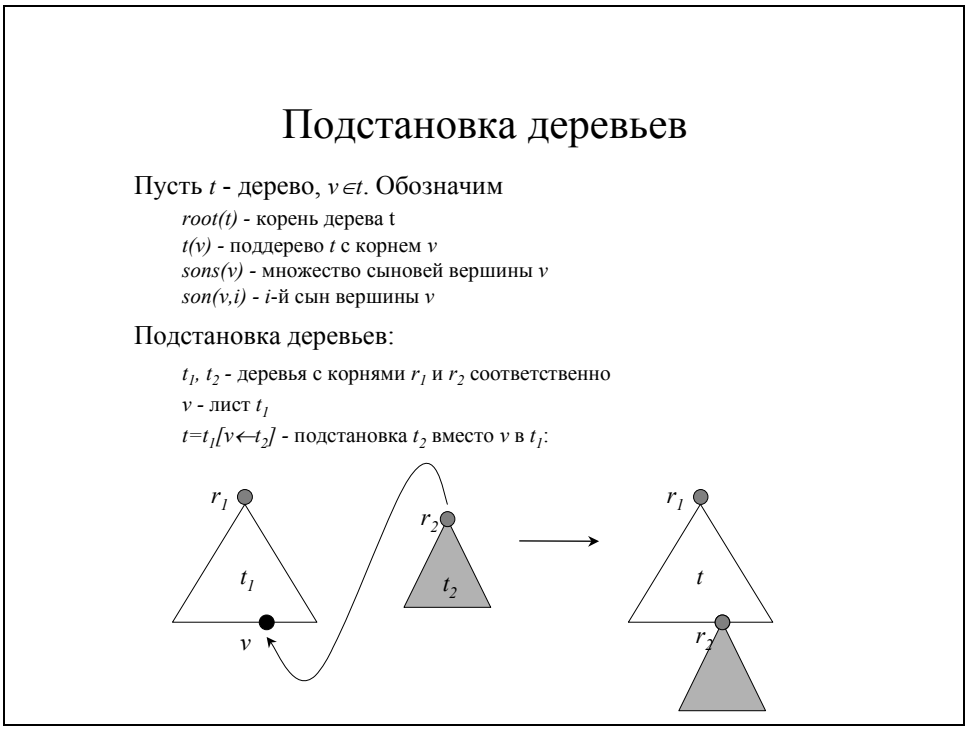
Тогда можно определить множество всех деревьев в этом алфавите как совокупность всех деревьев, у которых каждая вершина

- помечена символом a из алфавита A
- имеет ровно $\#(a)$ поддеревьев
- каждое ее поддерево в свою очередь является непустым деревом в алфавите A

В множество всех деревьев в алфавите A мы искусственно включаем также пустое дерево ε, которое не содержит вершин.

Наконец, деревянным языком L в алфавите A мы назовем произвольное (возможно, пустое) подмножество множества всех деревьев в алфавите A.

Поскольку при функции арности, не превосходящей единицы, все деревья в данных определениях превращаются в линейные списки, деревянные языки можно рассматривать как обобщение обычных «линейных» языков.



Подстановка деревьев

Для дальнейших рассуждений введем следующие обозначения:

- через *root(t)* обозначим корень дерева *t*
- через *t(v)* обозначим поддерево дерева *t* с корнем *v* (таким образом, *t(root(t))=t*)
- через *sons(v)* обозначим множество сыновей вершины *v*
- через *son(v,i)* обозачим *i*-го сына вершины *v*

Для пары деревьев в одном и том же алфавите можно определить операцию подстановки. Для этого в одном из них выбирается лист, который затем заменяется на корень другого дерева. Если t_2 подставляется вместо вершины v в дерево t_1, то результат подстановки будем обозначать $t_1[v \leftarrow t_2]$.

Регулярные деревянные грамматики

Четверка

$G=(A, N, S, R)$, где

$A=\{l_1, \ldots, l_m\}$ - входной алфавит с функцией арности #

$N=\{N_1, N_2, \ldots, N_k\}$ - алфавит нетерминалов

$S \in N$ - стартовый нетерминал

$R=\{r_1,\ldots,r_n\}$ - множество правил вида

- $N_1:N_2$, где $N_1, N_2 \in N$, или
- $N_1:a(K_1,\ldots,K_n)$, где $N_1,K_1,\ldots,K_n \in N, a \in A, \#(a)=n$, или
- $N_1:b$, где $N_1 \in N, b \in A$

- *деревянная грамматика в нормальной форме*

Регулярные деревянные грамматики

Одним из способов задания деревянных языков являются автоматные *деревянные грамматики*.

Деревянная грамматика в нормальной форме – это четверка, содержащая входной алфавит (снабженный функцией арности), алфавит нетерминалов, стартовый нетерминал и множество правил. В отличие от обычных контекстно-свободных грамматик, у которых в правой части правила стоит слово из объединенного алфавита терминалов и нетерминалов, в деревянной грамматике в правой части правила находится так называемый *деревянный образец*, который в общем случае является деревом, у которого нетерминалами могут быть помечены только (некоторые) листья. В приведенном выше определении требуется, чтобы деревянный образец имел высоту не более единицы и чтобы все его листья были помечены нетерминалами (именно поэтому данные грамматики называются грамматиками в нормальной форме).

Очевидно, что деревянные грамматики в данном виде есть обобщение обычных контекстно-свободных автоматных грамматик. Легко видеть, что возможен и контекстно-зависимый вариант – для этого достаточно разрешить присутствие деревянного образца в левой части правила. Заметим, наконец, что неавтоматный вариант контекстно-свободных грамматик не обобщается на деревянный случай – непонятно, как можно обобщить правило

$$N=K_1K_2,\ N,K_1,K_2 \in N$$

Выводимость образцов

Образец в грамматике $G=(A, N, S, R)$:

K, где $K \in N$, или
b, где $b \in A$, или
$a(w_1,...,w_k)$, где $a \in A$, $w_i \in N \cup A$, или
$a(p_1,...,p_k)$, где $a \in A$, p_i-образцы

Пусть p-образец. Тогда

leaves(p) - множество листьев p, помеченных нетерминалами

Образец p_2 выводится из образца p_1 по правилу R_i в G, примененному к вершине $v \in leaves(p_1) \Leftrightarrow$

v помечена нетерминалом K,
$R_i = K:p' \in R$ и
$p_2 = p_1[v \leftarrow p']$

Обозначение: $t_1 \Longrightarrow_G t_2$

Выводимость образцов

Для понимания того, как грамматики определяют язык, нам потребуется определить понятие выводимости образцов.

Как уже упоминалось выше, под образцом мы понимаем произвольное дерево, у которого некоторые листья помечены нетерминалами. В частности, тривиальное дерево, состоящее из одной вершины, является образцом (при этом единственная вершина может быть помечена как терминалом, так и нетерминалом).

Правила грамматики в нормальной форме содержат в правой части образцы специального вида, однако в выводе будут встречаться образцы произвольной формы. Далее через *leaves(p)* мы будем обозначать множество листьев образца p, помеченных нетерминалами.

Пусть есть два образца p_1 и p_2, вершина $v \in leaves(p_1)$ и правило грамматики $R=N:p'$. Будем говорить, что образец p_2 выводится из образца p_1 по правилу R, если v помечена нетерминалом N из левой части правила и p_2 может быть получен подстановкой образца p' из правой части правила в образец p_1 вместо вершины v.

Вывод в деревянной грамматике

Последовательность

$(p_1,v_1,R_1), (p_2,v_2,R_2),\ldots, (p_k,v_k,R_k),$

такая, что

$\forall i, 1 \le i \le k$:

- p_i-образец
- $v_i \in Leaves(p_i)$
- $R_i \in R$ - правило грамматики
- $i<k \Rightarrow p_i \text{—} (v_i:R_i) \rightarrow_G p_{i+1}$

называется *выводом* p_k из p_1 в грамматике G (обозначение $p_1 \rightarrow_G p_k$).

$D_G(p_1,p_k)$ - множество всех выводов p_k из p_1 в G.

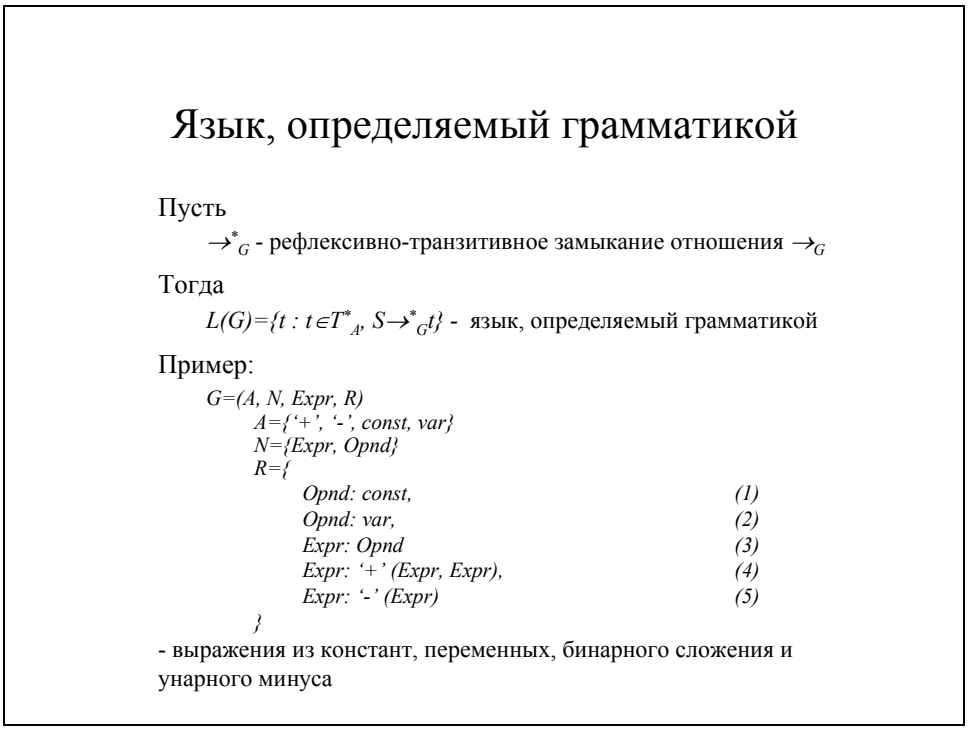
Вывод в деревянной грамматике

Выводом в грамматике назовем последовательность троек

$$(p_1,v_1,R_1),(p_2,v_2,R_2),\ldots,(p_k,v_k,R_k)$$

каждая из которых состоит из образца p_i, вершины $v \in leaves(p_i)$ и правила R_i, таких, что образец в каждой тройке, исключая первую, выведен применением правила из предыдущей тройки к образцу из предыдущей тройки относительно вершины из предыдущей тройки.

Для произвольных образцов p_1 и p_k определим $D_G(p_1,p_k)$ как множество всех выводов p_k из p_1.



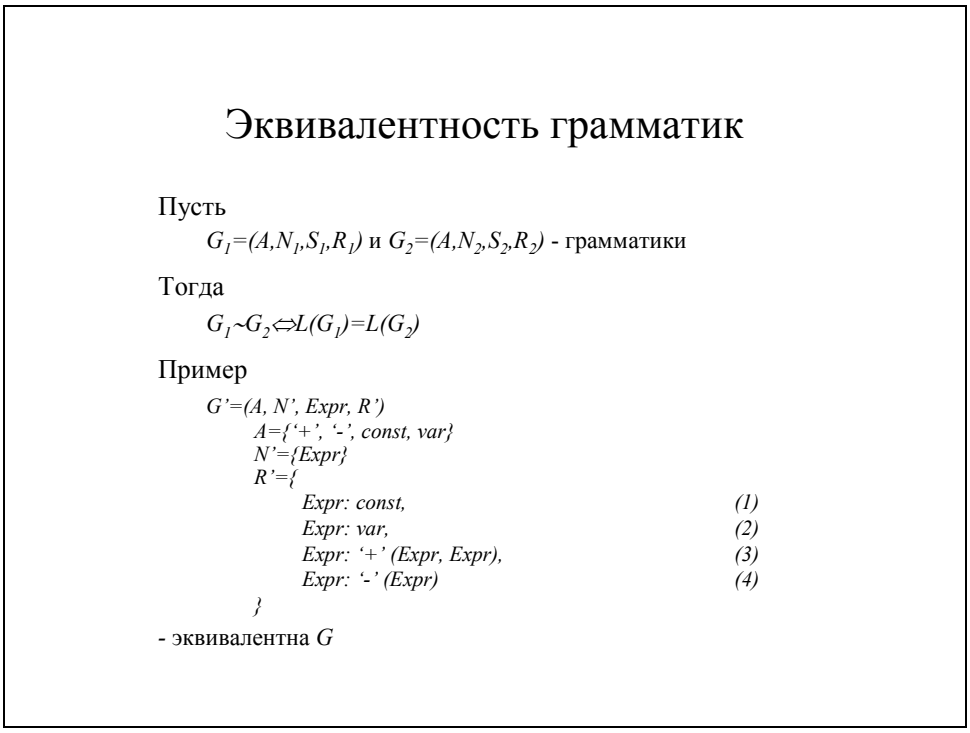
Язык, определяемый грамматикой

Пусть есть грамматика *G=(A,N,S,R)*. Языком, определяемым *G* (обозначение *L(G)*) назовем подмножество множества всех деревьев в алфавите A, для которых непусто множество выводов из тривиального дерева, единственная вершина которого помечена стартовым нетерминалом грамматики. Заметим, что такое дерево, так же, как и произвольное дерево в алфавите *A*, являются образцами.

Неформально говоря, грамматика определяет язык, состоящий из тех деревьев, которые можно получить подстановкой образцов из правых частей правил грамматики вместо листьев, помеченных нетерминалом из левой части этих правил до тех пор, пока в дереве существуют листья, помеченные нетерминалами.

На иллюстрации приведен пример грамматики, которая порождает деревья выражений из констант, переменных, унарных и бинарных операций. Видно, таким образом, что деревянные грамматики предоставляют удобный механизм описания различных деревьев, включая деревья внутреннего представления программы.

Можно показать, что существуют деревянные языки, для которых невозможно их задание с помощью грамматики приведенного вида.



Эквивалентность грамматик

Две грамматики назовем эквивалентными, если совпадают порождаемые ими языки. Грамматика, приведенная на данной иллюстрации, эквивалентна грамматике, приведенной на предыдущей.

Можно увидеть, что эта грамматика получена из предыдущей с помощью устранения цепных правил. Как и в случае обычных грамматик, любая деревянная грамматика может быть приведена к эквивалентной, не содержащей цепных правил.

Проблема определения эквивалентности грамматик разрешима, хотя для неоднозначных грамматик (определение неоднозначных грамматик будет дано ниже) это может потребовать экспоненциального времени.

Нормализация грамматик

Пусть

$G=(A,N,S,R)$ - грамматика с правилами вида

$N:p$, где p - образец произвольного вида

Тогда

$\exists G'=(A,N',S,R')$ - грамматика в нормальной форме, такая, что $G' \sim G$

Преобразование:

1. первоначально $N'=N$, $R'=R$
2. пусть $R_i=N:p \in R'$, $p=a(t_1,t_2,...,t_k)$ где $t_{i_1},t_{i_2},...,t_{i_n}$ - подобразцы общего вида
3. тогда

$N'=N' \cup \{K_1,...,K_n\}$, где K_i - новые нетерминалы,

$R'=R' \cup \{N:a(t'_1,t'_2,...,t'_k),K_1:t_{i_1},...,K_n:t_{i_n}\}$ - новые правила, где

$t'_m=K_p$, если $m=i_p$

$t'_m=t_m$, если $m \notin \{i_1,...,i_n\}$

Нормализация грамматик

Покажем, что для деревянной грамматики, у которой в правой части правил находятся образцы произвольного вида, существует эквивалентная грамматика в нормальной форме.

Пусть G=(A,N,S,R) – грамматика произвольного вида. Построим по ней грамматику G'=(A,N',S,R') в нормальной форме, действуя следующим образом:

1. первоначально множества нетерминалов и правил для новой грамматики совпадают с таковыми для старой;
2. выберем среди правил новой грамматики правило $R=N:p$, у которого образец p в правой части не находится в нормальной форме, и удалим его. Если такого правила нет, то грамматика уже находится в нормальной форме;
3. добавим в грамматику столько новых нетерминалов, сколько сыновей у *root(p)*, которые не являются тривиальными образцами (т.е. листьями, помеченными нетерминалами);
4. добавим в множество правил дополнительные правила, которые содержат в левой части новые нетерминалы, а в правой – те поддеревья p, которые не являются тривиальными образцами;
5. добавим в множество правил правило $N=p'$, где p' – образец, полученный из p заменой тех его поддеревьев, которые не являются тривиальными образцами, на листья, помеченные соответствующими новыми нетерминалами;
6. перейдем к шагу 2.

Пример нормализации грамматики

Легко видеть, что процесс, описанный выше, всегда завершится, и что полученная грамматика будет грамматикой в нормальной форме, эквивалентной исходной.

В качестве примера рассмотрим грамматику, показанную на иллюстрации. Здесь второе правило изначально не находится в нормальной форме, поскольку образец в его правой части содержит нетривиальный подобразец (лист, помеченный терминальным символом *b*).

В данном случае нормализация приводит к введению дополнительного нетерминала K_1 и двух правил

K_1:*b* и
K: *a*(K_1, *K*)

вместо второго правила.

Представление выводов

Пусть

$G=(A, N, S, R)$ - грамматика

$t \in L(G)$ - дерево

Разметка

$\forall v \in t, \ \forall K \in N$

$C[v][K]=\{R_1,\ldots,R_k\}$:

$K \rightarrow_G t(v)$?

$\forall R_i, \ 1 \le i \le k, (K, K, R_i),\ldots \in D_G(K, t(v))$

Представление выводов

Следующей нашей задачей будет построение множества выводов для данного дерева t в данной грамматике G.

Для представления множества выводов построим разметку C, которая вершине дерева v и нетерминалу K сопоставляет множество правил, каждое из которых начинает вывод образца $t(v)$ из образца K в грамматике G (правило R начинает вывод образца $t(v)$ из образца K тогда и только тогда, когда в множестве всех выводов $t(v)$ из K $D_G(K,t(v))$ существует вывод, который начинается тройкой (K, K, R), т.е. применением правила R к единственной вершине, помеченной нетерминалом K.)

Построение выводов

Алгоритм:

```
void Label (root: Tree)
{
    for each s∈sons(root) do Label (s);
    for each K∈N do C[root][K]=∅;
    for each Ri=K:p∈R do
        if (Match (Ri, root)) then begin
            if (C[root][K]=∅) then BuildClosure (root,K);
            C[root][K]=C[root][K]∪{Ri}
        end
}
```

Построение выводов

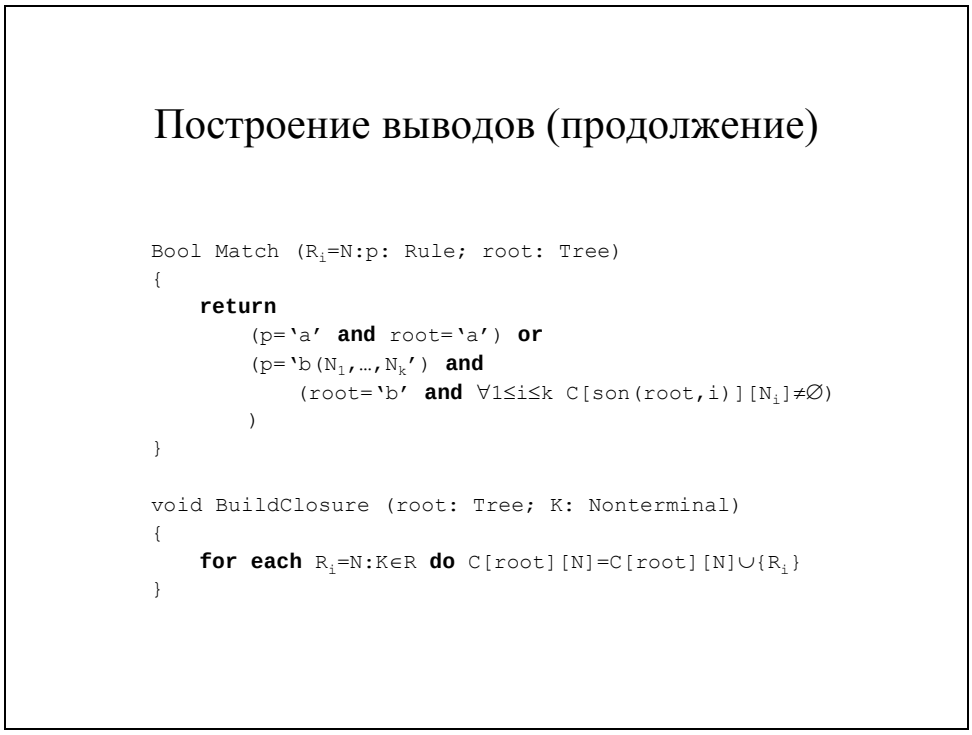
Приведем алгоритм построения разметки *C* для грамматики в нормальной форме.

Данный алгоритм обходит дерево снизу-вверх. При этом он пытается применить все возможные правила к текущей вершине *root*. Если какое-либо правило *R=N:p* применимо, то в разметку для пары *C[root][N]* добавляется правило *R*.

Для того, чтобы проверить применимость правила для текущей вершины, проверяется соответствие этой вершины образцу в правой части правила (этим занимается функция `Match`).

Кроме того, после вывода нового нетерминала в разметке *C* строится ее замыкание относительно цепных правил (это делает функция `BuildClosure`).

Заметим, что формально говоря всегда можно рассматривать грамматики без цепных правил, поскольку любую грамматику можно преобразовать в эквивалентную, не содержащую цепных правил. Однако такое преобразование увеличивает число правил, в которых требуется нетривиальное сопоставление с образцом.



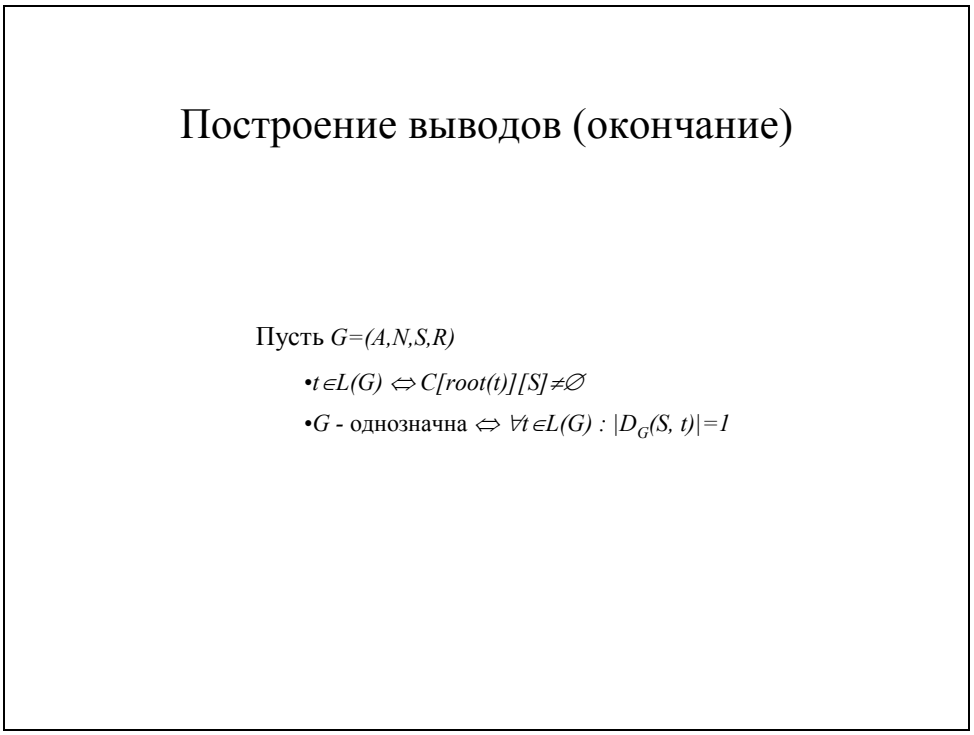
Построение выводов (продолжение)

Рассмотрим задачу сопоставления с образцом для вершины дерева (функция `Match`). Нас в данном случае интересуют образцы двух видов: лист, помеченный неким терминальным символом и дерево высоты один, корень которого помечен неким терминальным символом, а листья – нетерминалами. Кроме того, возможны образцы цепных правил, но они будут учтены при построении замыкания. Этим набором исчерпываются все образцы грамматики в нормальной форме.

Что касается образца первого вида, то данная вершина соответствует ему в том и только том случае, когда она является листом, помеченным тем же самым терминальным символом.

Для соответствия вершины образцу второго вида необходимо и достаточно, чтобы она была помечена нужным терминалом, а ее сыновья выводились в грамматике G из соответствующих нетерминалов образца. Поскольку построение разметки C происходит снизу-вверх, последнее условие может быть трактовано как непустота множества правил для данной вершины и данного нетерминала.

Наконец, функция `BuildClosure`, получая в качестве аргумента вновь выведенный нетерминал, строит замыкание разметки C относительно цепных правил, просто добавляя их в разметку для данной вершины и выведенного по цепному правилу нетерминала.

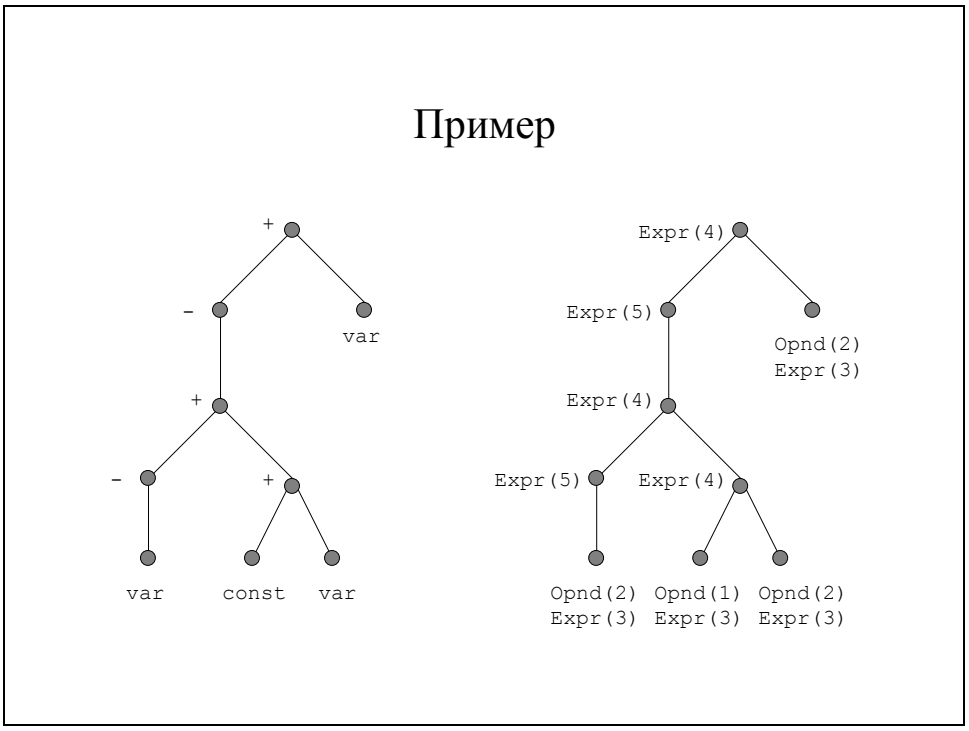


Построение выводов (окончание)

Пусть задана грамматика G и дерево t. Индукцией по числу шагов можно доказать, что приведенный алгоритм действительно строит разметку, обладающую заявленными свойствами. В частности, дерево t выводится в грамматике G тогда и только тогда, когда $C[root(t)][S]$ непусто, где S – стартовый нетерминал G.

Данный алгоритм имеет сложность, пропорциональную произведению числа правил на число вершин дерева.

Будем называть грамматику G однозначной тогда и только тогда, когда для произвольного дерева из порождаемого ею языка существует в точности один вывод, и неоднозначной в противном случае.



Пример

На иллюстрации показан пример построения разметки для дерева в соответствии с грамматикой, рассмотренной выше. Здесь слева изображено само дерево, а справа – дерево, снабженное разметкой *C*. Эта разметка представлена именами нетерминалов, в скобках указаны номера правил.

Системы восходящего переписывания деревьев (BURS)

Основная идея:

использовать деревянные грамматики для описания выбора инструкций

Мотивация:

выбор инструкций - это сопоставление с образцом

Интерпретация:

нетерминалы - типы операндов (режимы адресации)

цепные правила - преобразования типов или пересылки

язык - формат промежуточного представления

вывод - последовательность инструкций

Требования:

однозначность грамматики или возможность выбора одного вывода из нескольких

Системы восходящего переписывания деревьев

Деревянные грамматики лежат в основе систем восходящего переписывания деревьев (Bottom-Up Rewriting Systems, BURS), которые на сегодняшний день являются одним из наиболее распространенных способов описания кодогенераторов. Точнее, BURS позволяет построить алгоритм выбора инструкций, который при определенных допущениях строит оптимальный код.

Деревянные грамматики представляются естественным выбором как механизм описания выбора команд, поскольку являются удобной формализацией сопоставления с образцом.

Для использования в качестве формализма описания выбора команд понятия теории деревянных грамматик интерпретируются следующим образом:

1. нетерминалы обозначают режимы адресации или классы размещения значений (регистр, непосредственный операнд и т.д.)
2. цепные правила обозначают преобразования типов или пересылки
3. язык, определяемый грамматикой – это формат внутреннего представления программы перед выбором команд
4. вывод дерева в данной грамматике – последовательность команд

Из последнего пункта интерпретации следует, что грамматика должна быть однозначной. Если все же существует несколько выводов для данного дерева, и, следовательно, несколько последовательностей машинных команд, которые его вычисляют, то должна быть реализована возможность выбора какого-то одного вывода.

Грамматики восходящего переписывания

Грамматика $G=(A,N,S,R,c)$, где c - функция стоимости:

$c : R \to \{0,1,2\ldots\}$

Пусть $t \in L(G)$ - дерево, $d=(S, S, R_1),\ldots,(t,v,R_k) \in D_G(S,t)$ - его вывод

Тогда

$C(d)=\sum_{i=1,\ldots,k-1} c(R_i)$ - стоимость вывода

$D^+_G(S,t)=\{d \in D_G(S,t) : \forall d' \in D_G(S,t)\ C(d) \le C(d')\}$ - множество оптимальных выводов

Грамматики восходящего переписывания

Поскольку системы команд предоставляют большое разнообразие способов вычислений, ожидается, что практически любая грамматика будет неоднозначной. Однако небольшой модификацией этих неоднозначностей можно избежать.

Именно, снабдим каждое правило стоимостью и определим стоимость вывода как сумму стоимостей правил, входящих в его состав. Далее среди всех выводов нас будут интересовать только те, которые имеют наименьшую стоимость. Кроме того, будем считать, что выводы наименьшей стоимости для нас неразличимы, то есть нам совершенно все равно, какой из них выбрать. Таким образом грамматика становится однозначной с точностью до вывода минимальной стоимости.

Неформально говоря, каждое правило описывает либо машинную команду, либо ее операнд. Стоимость при этом отображает некоторое представление о сложности команды или операнда.

Представление вывода в BURS

Пусть

$G=(A, N, S, R, c)$ - грамматика

$t \in L(G)$ - дерево

Разметка

$\forall v \in t, \ \forall K \in N$

$C[v][K]=(R_i, c)$, если

$K \rightarrow_G t(v)$ и

$d=(K, K, R_i),... \in D^+_G(K, t(v))$ и

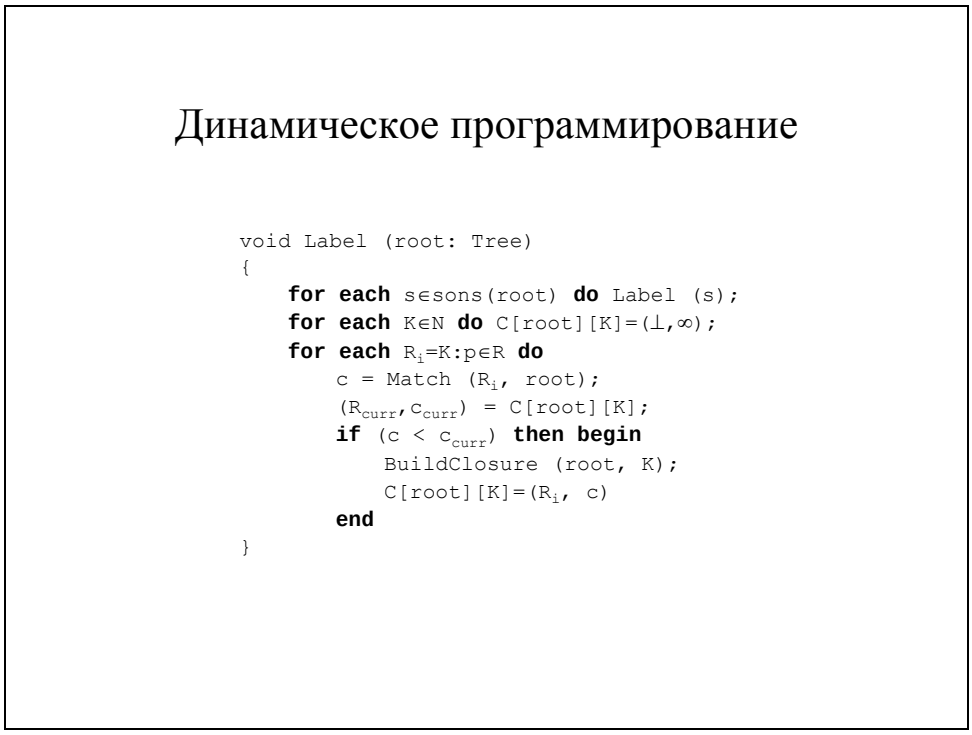
$c=C(d)$

$C[v][K]=(\perp, \infty)$, иначе

Представление вывода в BURS

Аналогично обычным деревянным грамматикам, построим разметку дерева, описывающую его вывод в BURS-грамматике. Поскольку BURS-грамматика однозначна, каждый нетерминал может быть выведен только одним правилом (точнее, достаточно помнить только одно правило, которое доставляет вывод наименьшей стоимости). Кроме того, в разметку будет входить число, соответствующее стоимости данного вывода.

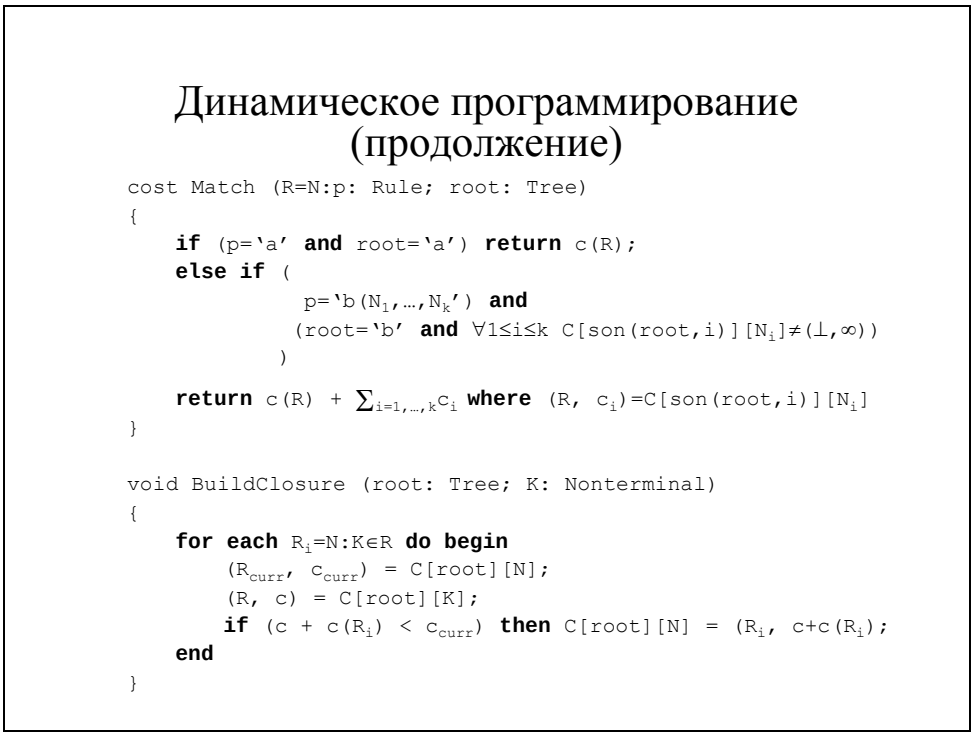
Если же поддерево с корнем в текущей вершине невыводимо из данного нетерминала, то будем считать, что существует вывод бесконечной стоимости.



Динамическое программирование

Наша задача теперь заключается в построении вывода минимальной стоимости. Поскольку стоимость вывода аддитивна относительно стоимостей правил, входящих в него, можно воспользоваться принципом динамического программирования, составляя оптимальный вывод для текущей вершины путем выбора среди всех применимых правил и оптимальных выводов для соответствующих поддеревьев (заметим, что это возможно в силу неотрицательности функции стоимости).

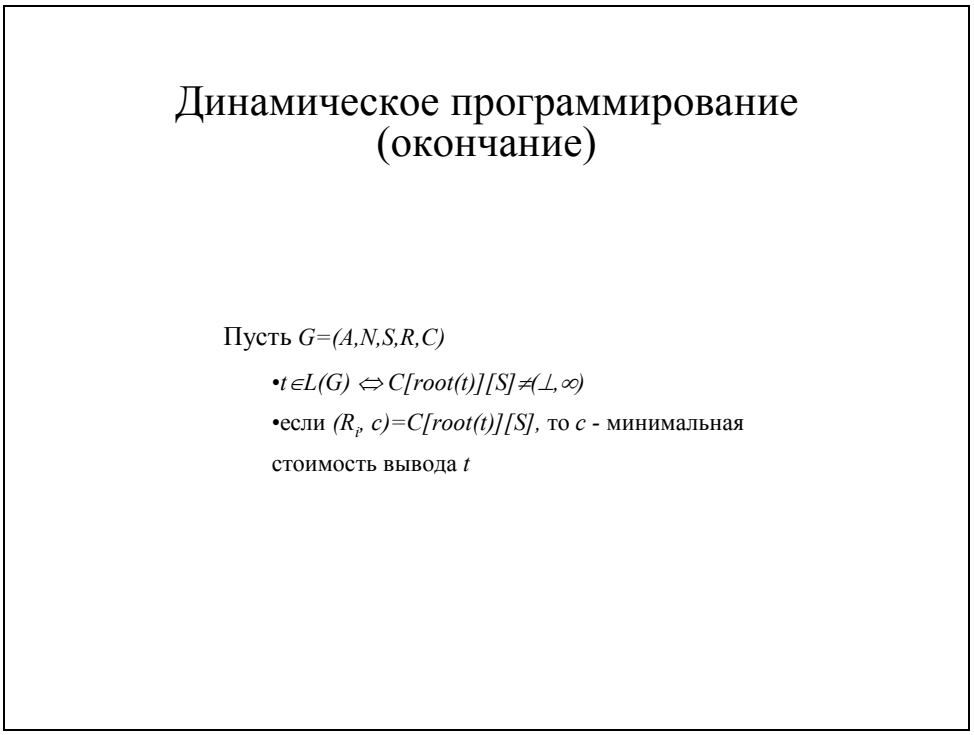
Код алгоритма восходящего переписывания деревьев приведен на иллюстрации. Теперь функция проверки соответствия поддерева образцу возвращает стоимость вывода этого поддерева из этого образца. Кроме того, вывод попадает в разметку только в том случае, если он минимален относительно всех выводов данного поддерева из данного нетерминала.



Динамическое программирование (продолжение)

Условия соответствия поддерева образцу остались прежними, но теперь функция `Match` заодно считает стоимость вывода поддерева из образца, суммируя стоимость собственно правила и стоимости минимальных выводов поддеревьев из нужных для соответствия образцу нетерминалов.

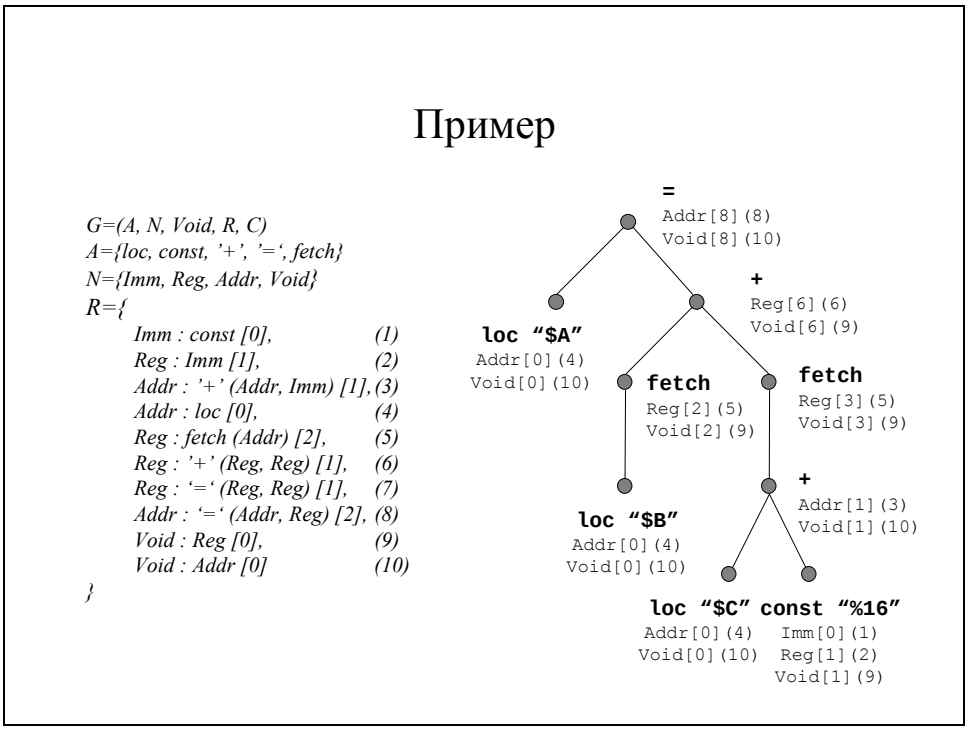
Построение замыкания по цепным правилам также аналогично обычному замыканию в деревянных грамматиках, однако здесь замыкание выводит нетерминал только в том случае, когда вывод по этому правилу доставляет минимальную стоимость.



Динамическое программирование (окончание)

Можно доказать, что приведенный алгоритм действительно строит разметку, обладающую заявленными свойствами. Его сложность также пропорциональна произведению числа правил на количество вершин дерева.

Отсюда следует, что если дерево выводимо в данной грамматике, то разметка сопоставляет паре (корень дерева, стартовый нетерминал) стоимость минимального вывода и правило, которое этот вывод начинает.



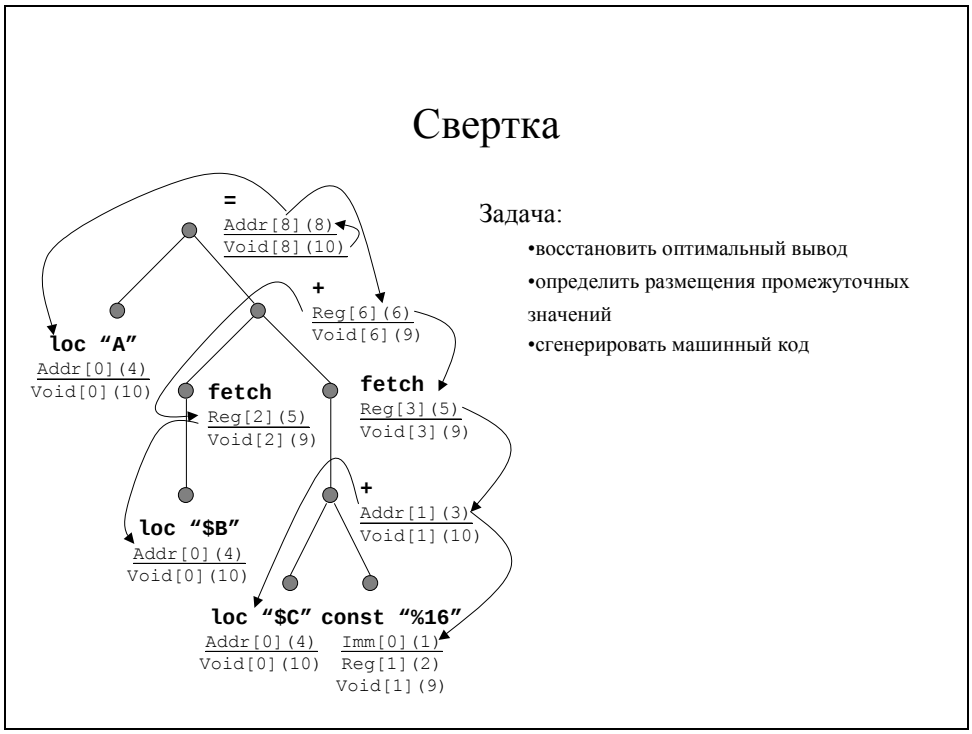
Пример

В качестве примера рассмотрим грамматику, приведенную на иллюстрации. Здесь в квадратных скобках указаны стоимости правил, в круглых – их номера.

Неформально говоря данная грамматика используется для выбора команд для дерева выражения, состоящего из констант (*const*), переменных (*loc*), бинарного сложения (*'+'*), присваивания (*'='*) и косвенной адресации (*fetch*).

Нетерминалы грамматики имеют следующий смысл: *Imm* – непосредственный операнд, *Reg* – регистр, *Addr* – адрес в памяти, *Void* – стартовый нетерминал. Таким образом, мы видим, что язык, определяемый грамматикой, действительно описывает внутреннее представление программы, а нетерминалы и правила машиннозависимы и отражают систему команд и элементы архитектуры устройства.

Разметка дерева в соответствии с этой грамматикой приведена в правой части иллюстрации. Здесь также в квадратных скобках указана стоимость минимального вывода для данного нетерминала, в круглых скобках – номер правила, доставляющего минимальный вывод.



Свертка

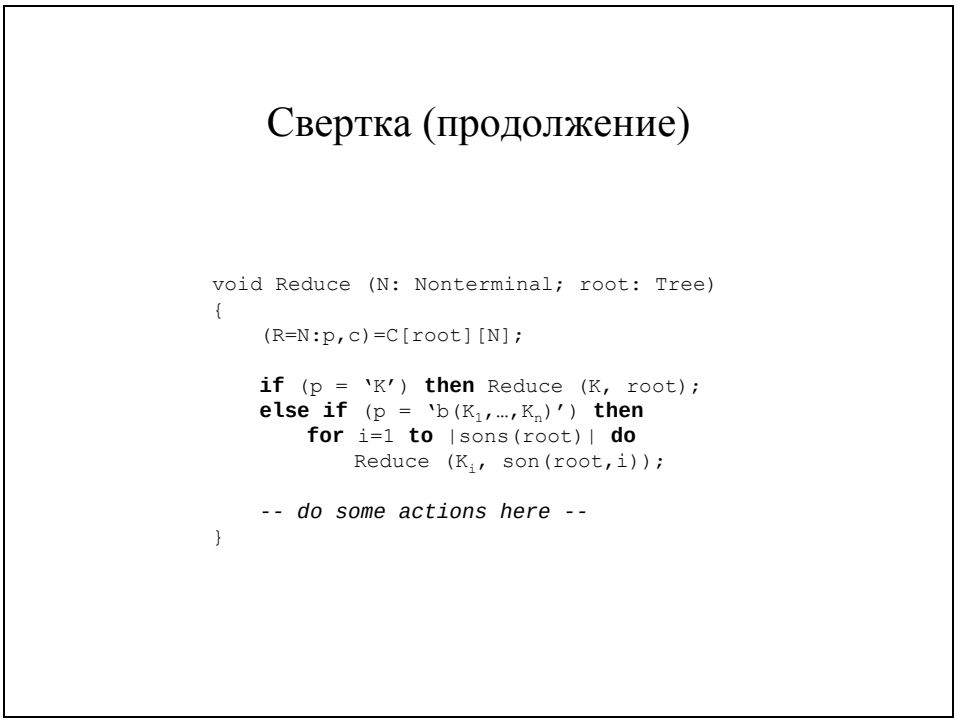
После того, как приведенным алгоритмом была построена разметка, необходимо извлечь из нее оптимальный вывод. Этот шаг называется *сверткой*.

Извлечение оптимального вывода происходит следующим образом. Разметка сопоставляет каждой вершине и каждому нетерминалу правило, которое входит в минимальный вывод поддерева данной вершины из данного нетерминала.

Предположим, что нетерминал, вывод из которого нас интересует, как-то задан. Тогда правило, которое возвращает нам разметка, однозначно определяет нетерминалы, из которых в минимальном выводе должны быть выведены поддеревья текущей вершины. В свою очередь, разметка для этих поддеревьев и этих нетерминалов доставляет правила, присутствующие в минимальном выводе этих поддеревьев, и так далее.

Теперь осталось заметить, что нас интересует вывод всего дерева из стартового нетерминала грамматики. Таким образом, сначала мы выбираем правило, которое начинает минимальный вывод дерева из стартового нетерминала, а затем поступаем так, как описано выше. Последовательность извлеченных таким образом правил и будет минимальным выводом данного дерева.

На иллюстрации последовательность выбора нетерминалов в оптимальном выводе отражена с помощью сплошных стрелок, а сами эти нетерминалы подчеркнуты.



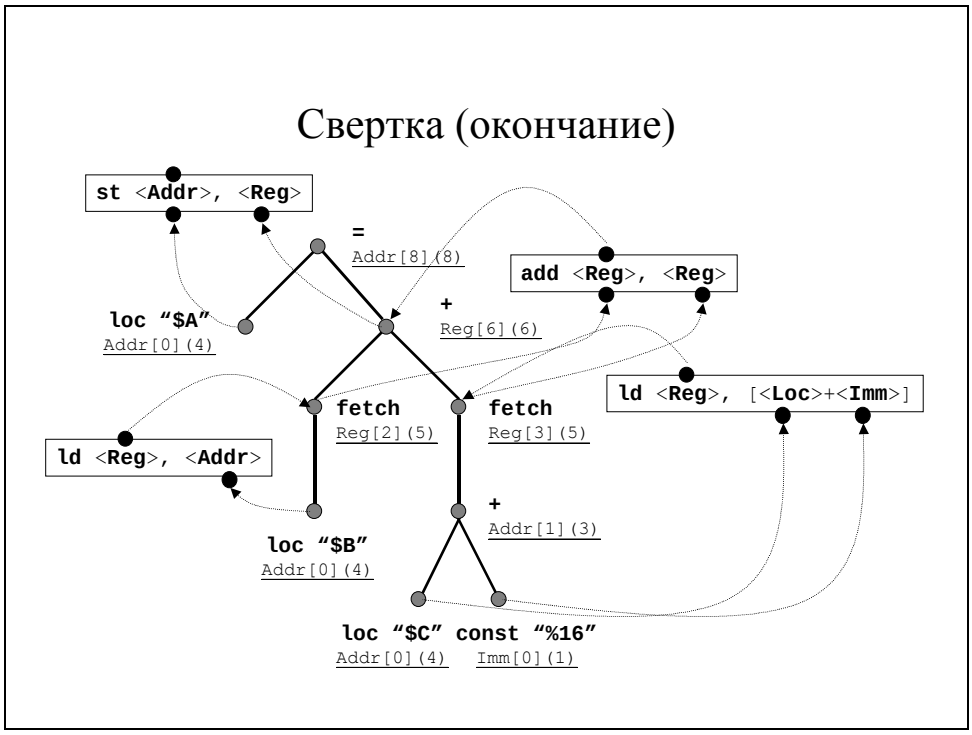
Свертка (продолжение)

Конечно, при генерации кода нас интересует не столько сам минимальный вывод, сколько его интерпретация с точки зрения машинных команд. Поэтому фактически наряду с восстановлением вывода должны быть предприняты какие-то действия по конструированию машинных команд и их операндов.

На иллюстрации приведен алгоритм свертки, который после восстановления вывода производит некоторые неформальные действия, призванные сконструировать машинный код. Суть этих действий мы разберем ниже, а пока прокомментируем коротко этот алгоритм.

Как и было объявлено, построение вывода определяется для корня дерева и некоторого предзаданного нетерминала. Первым делом извлекается правило, которое начинает вывод минимальной стоимости дерева из этого нетерминала (то, что такое правило всегда существует, будет видно по индукции из дальнейшего). Затем разбираются случаи устройства образца в правой части этого правила. Если правило было цепным, то производится светртка того же поддерева по нетерминалу в правой части правила, если нет, то выполняется свертка поддеревьев данной вершины по тем нетерминалам, которыми помечены соответствующие листья образца в правой части.

Наконец, изначально корень дерева сворачивается по стартовому нетерминалу.



Свертка (окончание)

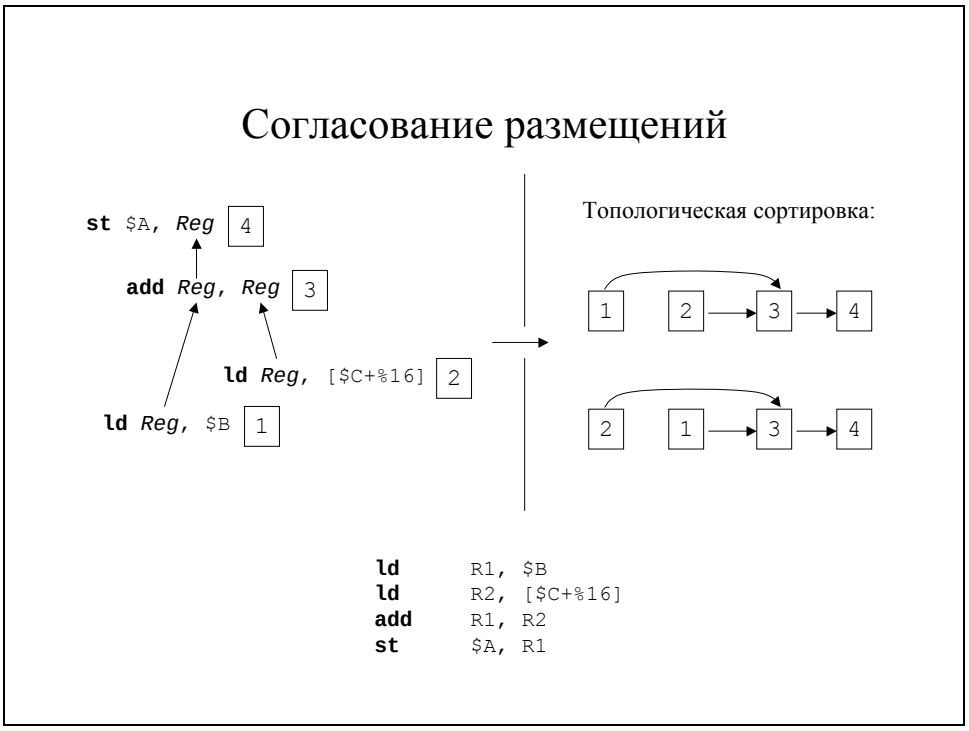
Здесь мы рассмотрим суть неформальных действий, которые выполняются при свертке для того, чтобы построить машинный код. Как было объявлено раньше, нетерминалы соответствуют классам размещений значений, а правила – операндам машинных инструкций или самим машинным инструкциям.

Каждый шаг свертки обладает следующей информацией:

1. текущим правилом (и, следовательно, машинной командой или операндом, который должен быть порожден)
2. текущим нетерминалом (и, следовательно, размещением результата текущей команды или операнда)
3. списком нетерминалов, из которых выводятся поддеревья текущей вершины (и, следовательно, списком размещений аргументов текщей команды или операнда)
4. вообще говоря, результатами свертки поддеревьев (то есть машинным кодом, построенным для их вычисления)

Все это дает возможность разметить дерево шаблонами машинных инструкций, которые содержат мнемонику, режим адресации и ссылки на узлы дерева, которые являются аргументами и результатами команды.

На иллюстрации приведен пример такой разметки. Здесь аргументы команд обозначены точками снизу команды, результаты – точками сверху, пунктирные стрелки указывают на соответствующие вершины дерева.

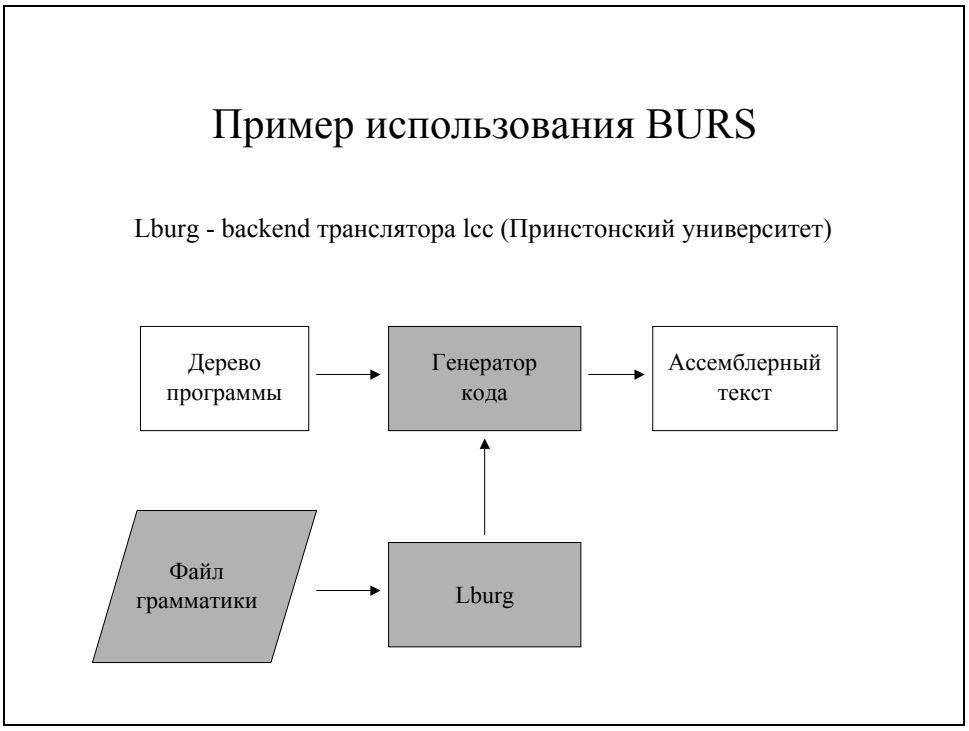


Согласование размещений

Действия на предыдущем шаге позволяют от исходного размеченного дерева перейти к дереву, в котором вершины уже соответствуют машинным инструкциям, но их операнды еще известны не полностью. Каждая машинная инструкция уже обладает мнемоникой и теми операндами, которые определены на предыдущем шаге (таковыми операндами, например, являются глобальные переменные, регистры, которые необходимо использовать из соображений соглашений о связях и вызовах и т.д.) Дуги в этом дереве обозначают передачу данных через временные значения и таким образом представляют необходимую информацию для согласования размещений.

Для получения финального кода требуется построить подходящую топологическую сортировку этого дерева (возможные сортировки показаны справа) и осуществить окончательное распределение регистров.

Финальный код после этих двух преобразований показан внизу иллюстрации. Заметим, что в случае неудачного распределения регистров выбор команд не может быть сохранен в изначальном виде (потребуется изменение размещений каких-то значений) и код, таким образом, станет субоптимальным.

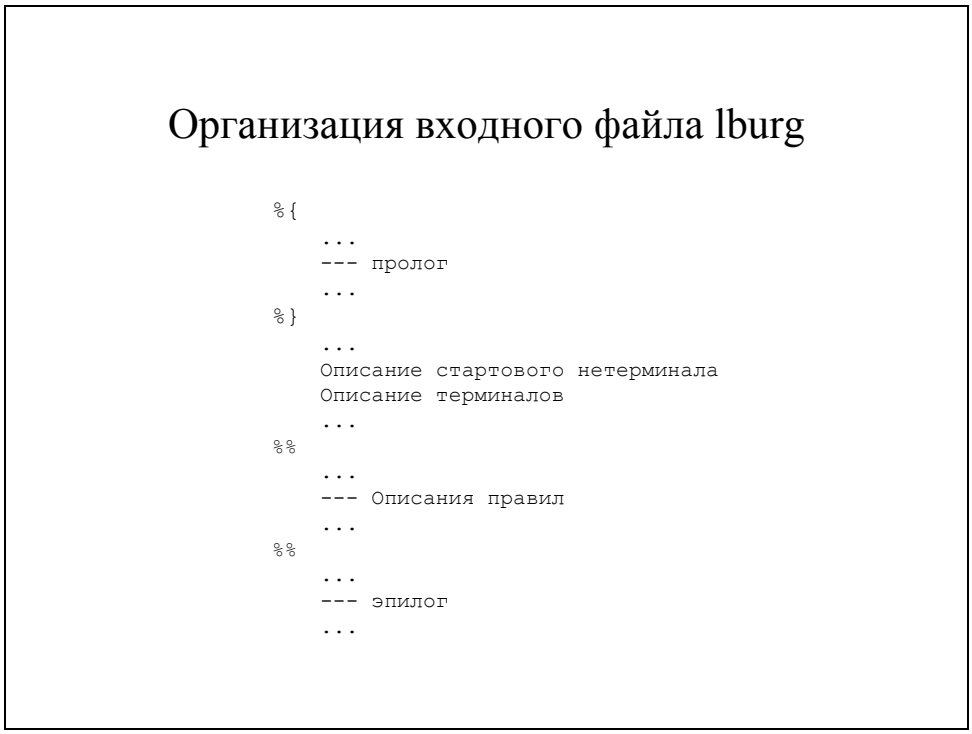


Пример использования BURS

Использование технологии BURS мы продемонстрируем на примере организации backend'а транслятора lcc, разработанного в Принстонском университете.

В основе backend'а лежит средство lburg, которое порождает текст генератора кода на языке C по входному файлу, содержащему описание BURS-грамматики и некоторых вспомогательных функций, необходимых, например, для оформления ассемблерного текста и т.д.

Заметим, что реализация backend'а в трансляторе lcc содержит встроенный механизм распределения регистров. Таким образом, чтобы осуществить раскрутку транслятора на новую архитектуру, необходимо поменять один только файл.

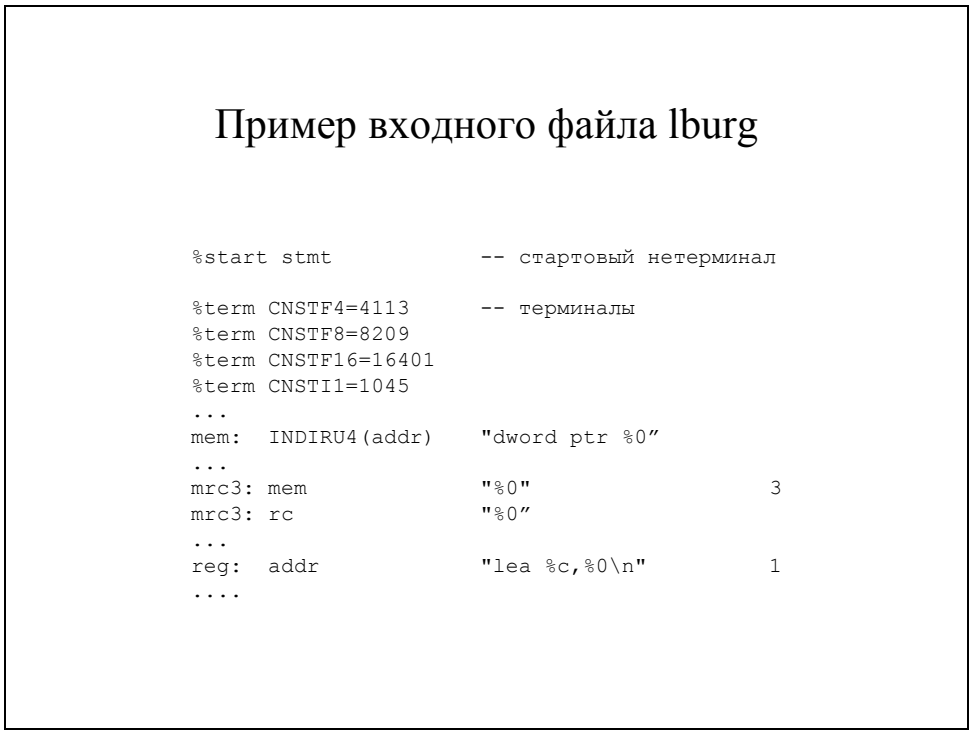


Организация входного файла lburg

Входной файл lburg организован согласно традиционной схеме представления грамматик в средствах типа YACC (см. лекцию 8). Он поделен на следующие секции:

1. пролог – содержит произвольный текст на языке реализации (в данном случае C)
2. описание терминалов и стартового нетерминала грамматики
3. описания правил, снабженных семантиками
4. эпилог, также содержащий неформальный текст на языке реализации

При обработке lburg порождает по описанию грамматики функции разметки и свертки (при этом свертка использует заданные в правилах семантики), а пролог и эпилог соответственно просто копируются в начало и конец порожденного генератора кода.



Пример входного файла lburg

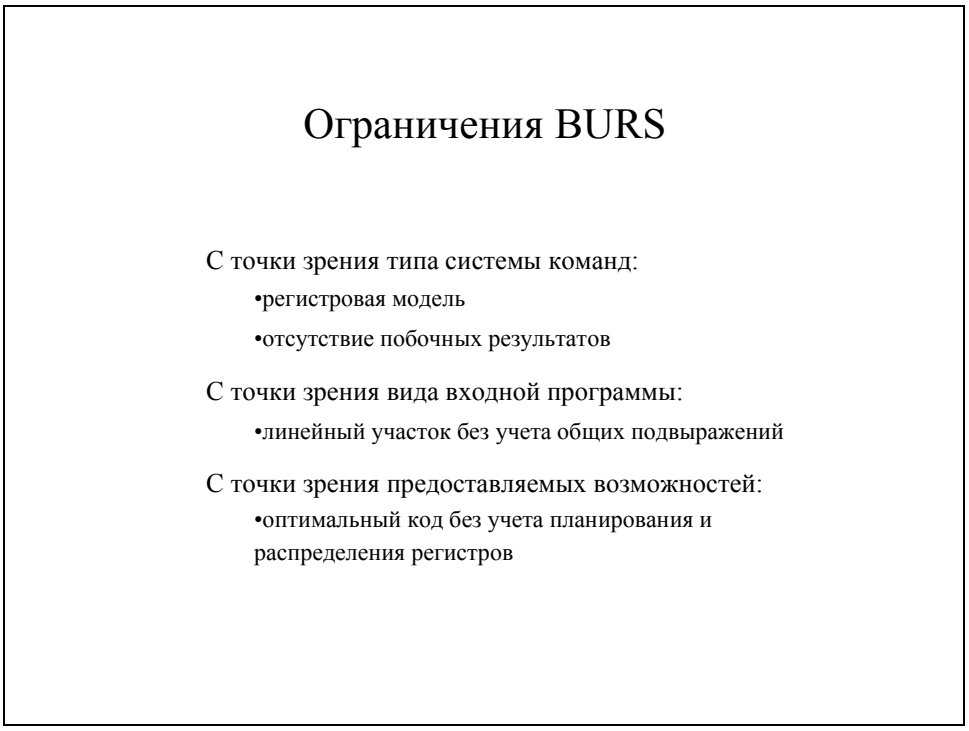
Рассмотрим формат описания основных элементов в грамматике для lburg.

Традиционно нетерминалы обозначаются идентификаторами, состоящими из строчных букв, терминалы – идентификаторами, состоящими из прописных букв. Конструкция `%start <nonterminal>` служит для описания стартового нетерминала. Конструкция `%term <terminal>=<constant>` определяет терминал, при этом числовое значение справа от знака равенства соответствует внутреннему коду узла дерева в представлении программы, выдаваемом frontend'ом.

Наконец, правило состоит из нетерминала, который им выводится, деревянного образца в нормальной форме, который отделен от нетерминала двоеточием, шаблоном вывода, который используется при свертке и необязательной стоимости (в случае ее отсутствия правилу приписывается стоимость 0).

В данном случае приведено четыре правила из описания кодогенератора для процессора x86: первое порождает команду косвенной адресации для четырехбайтового адреса, второе и третье выводит нетерминал, соответствующий режиму адресации mcr (memory+constant+register) из нетерминалов, соответствующих регистру и адресу в памяти, четвертое описывает команду загрузки содержимого памяти в регистр.

Форматная строка напоминает таковую в стандартных функциях обмена, однако асортимент форматных символов несколько иной: так, "%0" обозначает результат свертки 1-го сына текущего узла, "%c" – результат свертки текущего узла и т.д.



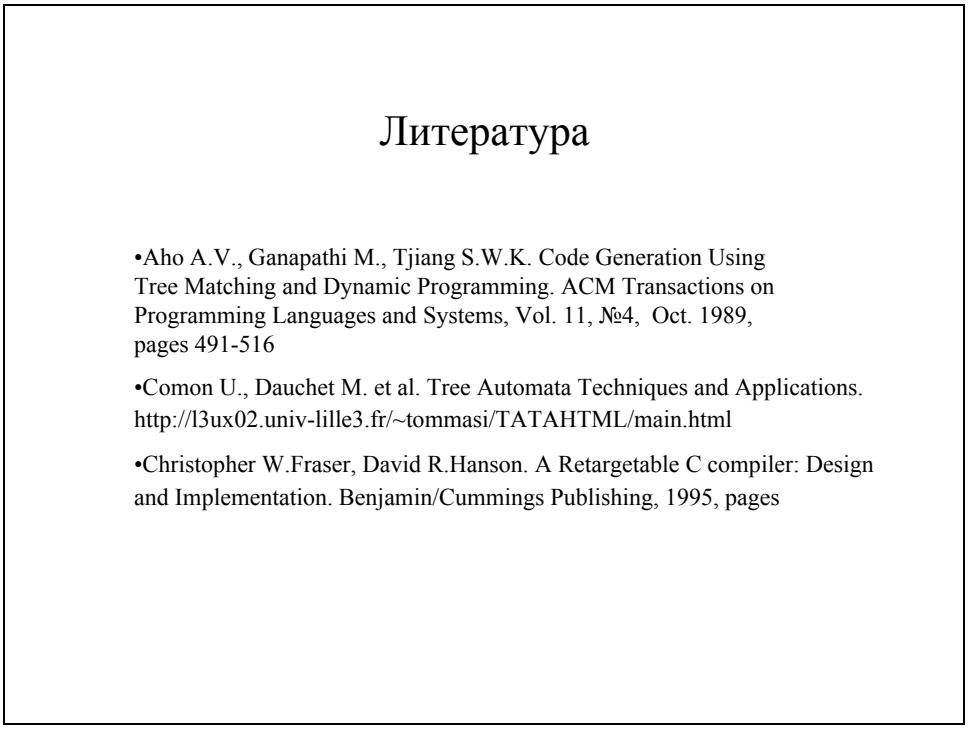
Ограничения BURS

Несмотря на простоту описания системы команд с помощью формализма BURS, построение кодогенератора с его помощью требует дополнительных усилий. Использование BURS кардинально упрощает стадию сопоставления с образцом и выбора режимов адресации, однако такие задачи, как линеаризация дерева (построение топологической сортировки) и распределение регистров должны быть решены отдельно. BURS представляется наилучшим выбором для CISC архитектур, в которых разнообразие команд и режимов адресации действительно велико. Для RISC же архитектур задачи распределения регистров и планирования оказываются более важными.

Кроме того, в BURS-схеме плохо выражаются нерегистровые системы команд (например, стековая модель вычислений) и вообще ситуации, когда машинные команды имеют побочные эффекты, поскольку в такой ситуации выводимый нетерминал перестает однозначно сответствовать размещению результата.

Еще одним существенным ограничением BURS является то, что их применение ограничивается деревьями, т.е. наборами выражений без общих подвыражений. Можно доказать, что построение оптимального кода для выражений с общими подвыражениями с помощью BURS труднорешаемо (тем не менее существуют эвристические подходы к генерации кода в такой ситуации).

Наконец, как уже упоминалось выше, BURS-код оптимален при условии, что распределение регистров не заденет результатов выбора команд (т.е. что запаса регистров хватит для его реализации).



Литература

1. Aho A.V., Ganapathi M., Tjiang S.W.K. Code Generation Using Tree Matching and Dynamic Programming. ACM Transactions on Programming Languages and Systems, Vol. 11, №4, Oct. 1989, pages 491-516
2. Comon U., Dauchet M. et al. Tree Automata Techniques and Applications. http://l3ux02.univ-lille3.fr/~tommasi/TATAHTML/main.html
3. Christopher W.Fraser, David R.Hanson. A Retargetable C compiler: Design and Implementation. Benjamin/Cummings Publishing, 1995, pages 564