Содержание

Аннотация

Как читать это пособие? Каждый раздел разбит условно на 3 части: основную, первую часть, которая, как правило повествуется на лекции; затем, опционально, идут комментарии к практике, а затем – некоторые важные ремарки и дополнительные примеры. Некоторые подразделы также могут быть разбиты на 3 части, как правило, это подразделы с большим объемом материала, по каждому из которых было отдельное занятие.

1 Языки и их свойства, операции над языками

1.1 Введение понятия языка

Назовём множество абстрактных объектов – символов – алфавитом Σ. Пусть алфавит конечный. Пустой и бесконечный алфавиты нам неинтересны.

Введём слово над алфавитом $\Sigma : w(A) = a_i, a_i \in \Sigma, \forall i = 0..|w(A)|$ – последовательность (строка) символов из алфавита $, 0 \leq |w(\Sigma)| < +\infty$.

Чтобы оперировать словами длины 0, вводят специальный символ длины $0 - \varepsilon : |\varepsilon^n| = 0, n = 0.. + \infty$; Его называют пустым.

Обозначим множество таких последовательностей из символов алфавита Σ, включая слово длины 0, как Σ^*.Тогда некоторый язык $L(\Sigma)$ над алфавитом Σ можно задать как подмножество слов над алфавитом: $Ł(\Sigma) \subset (\Sigma^*)$. Таким образом, математически мы определили объекты, с которыми будем работать, это последовательности конечной длины и множества.

Теория формальных языков – математический способ конструктивного описания множеств последовательностей (слов) элементов некоторых множеств (алфавитов). Почему конструктивного? Потому что, в принципе, все слова языка можно просто перечислить, если:

1. любое слово – конечной длины.
2. множество слов конечно.
3. нет ограничений на временную сложность алгоритмов, используемых в работе с таким языком.

Нарушения 1) и 2), соответственно, говорят о том, что мы будем перечислять слова бесконечно, 3) это пожелание для применения на практике – нам нужны алгоритмы, которые работают, по крайней мере, за полином небольшой степени и по времени, и по памяти, а лучше за линию, так как мы хотим иметь дело с относительно мощными языками, и нам важна масштабируемость.

В нашем курсе 1) будет всегда выполняться: считаем, что любое слово языка – конечной длины. Но пусть 2) не выполняется, а 3) нас просят строго соблюсти. Тогда задача конструктивного, то есть 'сжатого' и точного описания множества слов обретает куда более глубокую практическую значимость.

Кроме перечисления, можно предложить еще 2 способа задания языка:

- Распознаватель – все слова языка можно распознать некоторой вычислительной машиной.
- Генератор – все слова языка можно вывести посредством формальной процедуры переписывания строк по системе правил. Система математических объектов, позволяющих это сделать, называется формальной грамматикой.

С этими двумя способами теория формальных языков и работает. Мы начнём с первого, в последствии переключимся на второй, а затем синхронно двинемся дальше с обеими способами, усложняя и рассматриваемые методы, подходы и задачи.

1.2 Операции над языками

Начнём с базовых операций над элементами языков – словами.

1.2.1 Операции над словами

Опр. 1.1 *Конкатенация – склеивание*[1] *строк. Если* $u = a_1 \dots a_m$ *и* $v = b_1 \dots b_n$ *— две строки, то их конкатенация — это строка* $u \cdot v = uv = a_1 \dots a_m b_1 \dots b_n$*. Знак* $\cdot$*, как правило, опускают.*

Конкатенация строки сама с собой обозначается как возведение в степень: w^n – n раз повторяемая w. $w^1 = w, w^0 = \varepsilon$, то есть конкатенация играет роль умножения с единицей ε, и превращает язык в свободную группу.

Опр. 1.2 *Взятие префикса – из любой строки* s *длины* l *можно взять префикс* $s[:n]$ *длины* $n, n \in 0..l$*,* $s[:0] = \varepsilon, s[:l] = s$*.*

Опр. 1.3 *Взятие суффикса* $s[n:]$ *вводится по аналогии с взятием префикса.*

Опр. 1.4 *Взятие подстроки* $s[n:m], n \leq m$ *можно ввести, например, как* $(s[n:])[:(m-n)]$*, либо* $(s[:m])[n:]$*.*

Конечно, существует множество других интересных, широкоиспользуемых либо экзотических операций, вроде инверсии слова, но оставим их за рамками данного пособия.

1.2.2 Операции над языками как множествами

Объединение, пересечение, вычитание, дополнение – как с обычными множествами ... Нам они понадобятся, в особенности, при проверке свойств принадлежности языка некоторому классу.

1.2.3 Операции над языками как множествами, содержащими последовательности

Опр. 1.5 *Конкатенация языков* $L_1(\Sigma_1), L_2(\Sigma_2) \subset (\Sigma_1 \cup \Sigma_2)^*$ *– это операция склеивания всех возможных слов языков:* $L_1 \cdot L_2 = \{uv | u \in L_1, v \in L_2\}$*.*

[1] устоявшегося русского термина пока нет, увы

Можно сконкатенировать не 2, а любое неотрицательное число языков k. Если язык конкатенируют сам с собой, то это обозначают L^k. Для $k < 2$ операцию определяют так: если $k = 0$, то это будет язык $\{\varepsilon\}$, что соответствует определению $x^0 = 1$ для чисел. Если $k = 1$, то это будет сам L. Как видим, конкатенация играет роль умножения[2].

Опр. 1.6 *Итерация языка* L *:* $L^* = \bigcup_{k=0}^{\infty} L^k$.

Заметим, что множество слов Σ^* – итерация языка Σ.

1.3 О приложениях теории формальных языков

В общем и целом, формальные языки имеют приложения таких направлениях науки и техники, как:

- Построение и синтаксический анализ языков программирования
- Построение автоматизированных систем управления (особенно автоматные языки)
- Извлечение информации из текста (на естественном или формальном языке) с учётом его (некоторой) синтаксической структуры
- Вычисление запросов к графовым БД
- Анализ цепочек аминокислот
- Статический анализ ПО
- прочее...

2 Конечные автоматы

Конечный автомат – математическая модель вычислителя с конечной памятью.

Опр. 2.1 *Недетерминированный конечный автомат (НКА) – это кортеж* $\langle Q, \Sigma, \Delta, q_0, F \rangle$*:*

- $Q, |Q| < \infty$ – *множество состояний*
- Σ – *алфавит*
- $\Delta \subset Q \times \Sigma^* \times Q$ – *множество переходов*[3]
- $q_0 \in Q$ – *стартовое состояние*

[2] это и правда умножение в некотором полукольце с единицей ε (вопрос: а какая операция – сложение в этом полукольце?)

[3] Δ задаёт множество двухместных отношений на Q, помеченных элементами Σ^*.

- $F \subset Q$ – *множество финальных состояний*

Существует эквивалентное определение автомата, где вместо Δ задают функцию перехода $\delta : Q \times \Sigma^* \to 2^Q$; будем пользоваться «более графовым» определением через Δ, хотя функция перехода нам ещё понадобится.

Способ распознавания строки автоматом уже лежит в его определении: представим граф автомата. Вершины – это состояния, рёбра – переходы. Если мы находимся в стартовом состоянии, и нам подают на вход строку, то нам достаточно брать по символу/слову из Σ^*, смотреть, по каким рёбрам графа мы можем перейти (если ε – перейти можем спонтанно), совершать переход(ы), брать следующий символ/слово из Σ^*, смотреть, куда мы по нему можем перейти из текущего состояния, и так далее. Слово распозналось, если мы дошли до какого либо финишного состояния и обработали всё слово. Распознавание строки автоматом – суть проверка достижимости по рёбрам его графа из q_0 в одно из состояний в F.

Основным недостатком КА служит то, что мы в каждый момент времени знаем только текущее состояние и в какие мы можем из него перейти. У нас нет данных о том, что происходило ранее, и это накладывает ограничения на выразительность[4]. К примеру, нельзя составить КА, распознающий язык $a^n b^n, \forall n \in [0, +\infty)$, хотя для любого фиксированного множества n – можно (Рис. 1).

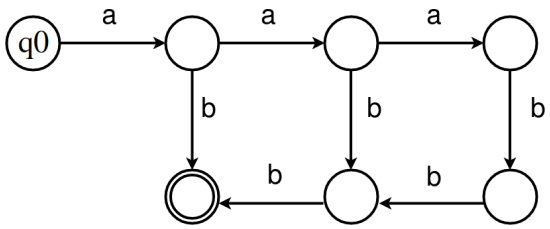


Рис. 1: КА, распознающий язык $a^n b^n, n \in [1, 3]$

О достижимости проще говорить в терминах пар $\langle q_x, v \rangle \in Q \times \Sigma^*$, где q_x – текущее состояние, а v – недоразобранная подстрока входной строки. Такая пара называется конфигурацией автомата[5]. Введём отношение достижимости на конфигурациях.

Опр. 2.2 *Достижимость* ($\vdash$) – *наименьшее рефлексивное транзитивное отношение над* $Q \times \Sigma^*$, *такое что:*

1. $\forall w \in \Sigma^* : (\langle q_1, w \rangle \to q_2) \in \Delta \Rightarrow \langle q_1, w \rangle \Rightarrow \langle q_2, \varepsilon \rangle$
2. $\forall u, v \in \Sigma^* : \langle q_1, u \rangle \vdash \langle q_2, \varepsilon \rangle, \langle q_2, v \rangle \vdash \langle q_3, \varepsilon \rangle \Rightarrow \langle q_1, uv \rangle \vdash \langle q_3, \varepsilon \rangle$

[4]тем не менее, конечные автоматы широко применяются в технике вокруг нас. Примеры: светофор, лифт, кодовый замок, система контроля воздуха в помещении, компьютерная мышь, аудиоплеер, веб-форма и т.д.

[5]по мере усложнения моделей вычислителей, мы будем добавлять новые параметры в конфигурацию – например, появится параметр, описывающий стек, и т.д.

3. $\forall u \in \Sigma^* : \langle q_1, u\rangle \vdash \langle q_2, \varepsilon\rangle \Rightarrow \forall v \in \Sigma^* \langle q_1, uv\rangle \vdash \langle q_2, v\rangle$

Используя это определение, несложно задать язык, распознаваемый КА.

Опр. 2.3 *Пусть дан* $M = \langle Q, \Sigma, \Delta, q_0, F\rangle$*. Язык, распознаваемый автоматом* M *–* $L(M) = \{w \in \Sigma^* | \exists q \in F : \langle q_0, w\rangle \vdash \langle q, \varepsilon\rangle\}$*.*

Опр. 2.4 *Язык* L *называется автоматным, если существует КА* M *:* $L = L(M)$*. Множество таких языков* L *образует класс автоматных языков.*

На практике гораздо приятнее работать с детерминированным конечным автоматом (ДКА).

Опр. 2.5 *(Неформально) НКА* $M = \langle Q, \Sigma, \Delta, q_0, F\rangle$ *называется детерминированным КА, если*

- *Все переходы – однобуквенные:* $\forall(\langle q_1, w\rangle \to q_2) \in \Delta : |w| = 1$
- $\forall a \in \Sigma, q \in Q |\delta(q, a)| \leq 1$*, где* $\delta(q, a)$ *– множество состояний, достижимых из* q *по символу* a*. Задание: расписать* $\delta(q, w)$ *аккуратнее через конфигурации.*

Иными словами, для любых фиксированных букв, для любого состояния, переход приводит только в одно результирующее состояние.

Можно ввести ДКА-автоматный язык L_{NFA} по аналогии с тем, как вводили $L(M) = L_{DFA}$. Очевидно, что $L_{DFA} \subseteq L_{NFA}$, так как ДКА – это частный случай НКА.

Если мы покажем, что произвольный НКА сводится к ДКА, то $L_{DFA} = L_{NFA}$.

2.1 Сведение НКА к ДКА

Л. 2.1 *(«Построение подмножеств», Рабин и Скотт [1959]). Пусть* $B = (\Sigma, Q, q_0, \Delta, F)$ *— произвольный. Тогда* $\exists$ *DFA* $A = (\Sigma, 2^Q, Q_0, \Delta', F')$*, состояния которого — помножества* Q*, который распознаёт тот же язык, что и* B*. Его переход в каждом состоянии-подмножестве* $s \subseteq Q$ *по каждому символу* $a \in \Sigma$ *ведет во множество состояний, достижимых по* a *из некоторого состояния* s*.*

Произведём серию упрощений НКА.

Утв. 2.1 *В опредлении НКА можно считать все переходы – однобуквенными. Для этого нужно перестроить множества* Δ *и* Q*.*

Утв. 2.2 *В опредлении НКА можно считать* $|F| = 1$*.*

Утв. 2.3 *(* ε*-замыкание) От переходов по* ε *можно избавиться, применив некоторые преобразования (см.Рис. 2).*

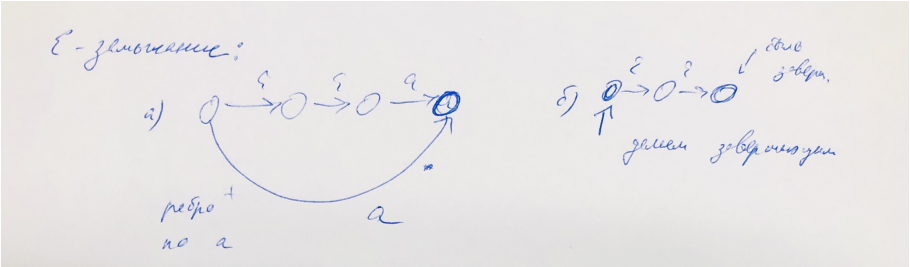


Рис. 2: Основные преобразования при построении ε-замыкания: последовательность переходов $\varepsilon \dots \varepsilon a$ заменить на переход a (а), состояние, из которого существует переход $\varepsilon \dots \varepsilon$ в финальное состояние – обозначить как финальное (б)

Эти утверждения доказываются технически, не будем этим заниматься сейчас (рекомендуется попробовать доказать дома или посмотреть в классических книгах и курсах).

TODO: доказательство Л2.1, алгоритм на базе метода «построение подмножеств»

Утв. 2.4 *(о корректности Л2.1). Для любой строки $w \in \Sigma^*$, состояние-подмножество, достигаемое DFA по прочтении строки w, содержит элемент q тогда и только тогда, когда хотя бы одно из вычислений NFA на w заканчивается в состоянии q.*

Доказывается индукцией по длине строки w.

Далее из утверждения о правильности выводится, что построенный DFA распознаёт строку $w \in \Sigma^*$ тогда и только тогда, когда распознаёт исходный NFA. Построение переводит NFA с n состояниями в DFA с 2^n состояниями-подмножествами. На практике, многие из них обычно бывают недостижимы. Поэтому хороший алгоритм должен строить только подмножества, достижимые из уже построенных, начиная с q_0.

2.2 Минимизация ДКА

Говорят, что состояния u, v различаются словом s, если одно из них по s переводит автомат в финальное состояние, а другое нет.

Если состояния не различаются никакой строкой, они называются неразличимыми. На Рис.2 изображен ДКА, в котором есть такие: действительно, окажемся мы в финальном состоянии или нет, зависит только от количества нулей в строке, следовательно, B и C – неразличимы.

Л. 2.2 *Отношение неразличимости суть отношение эквивалентности.*

Рефлексивность очевидна, симметричность следует из определения (попробуйте заменить u и v местами).

Транзитивность: u и v неразличимы, v и w неразличимы, следовательно, u и w неразличимы, тоже очевидно.

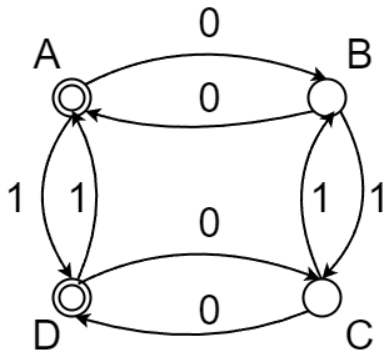


Рис. 3: ДКА, в котором есть неразличимые состояния (найдите их)

По индукции по длине строки доказывается, что модификация автомата как на Рис. 3, если состояния A и B не различимы, не меняет распознаваемый им язык.

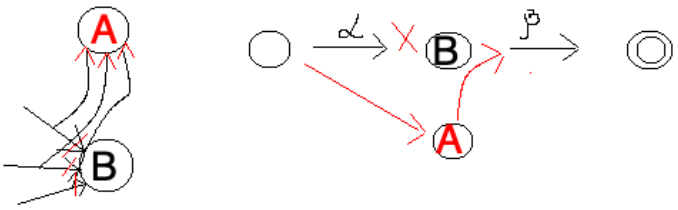


Рис. 4: Вспомогательный рисунок

Повторяя процедуру модификации для всех классов эквивалентности, оставляя какую-то одну вершину для каждого класса, получим некий автомат с возможно меньшим числом состояний. Можно доказать, что это число состояний – минимально.

Л. 2.3 *Пусть у ДКА M все состояния различимы и любое достижимо из стартового. Тогда M – минимальный автомат для $L(M)$*

Т. 2.1 *Для любого ДКА существует и единственный с точностью до изоморфизма ДКА с минимальным числом состояний.*

Интуитивно, для выполнения минимизации нужно выделить:

- Недостижимые состояния – их нужно удалить[6]
- Неразличимые состояния – их можно объединить в одно для каждого класса эквивалентности

Существует, как минимум, 3 способа выделить и схлопнуть неразличимые состояния:

- Наивный алгоритм основан на построении классов эквивалентности и объединении эквивалентных состояний [2], и рассматривается на семинаре. Он работает за $O(n^2)$.
- Алгоритм Хопкрофта, позволяющий решить задачу за $O(nlog(n))$ [3].
- Также существует алгоритм Бржозовского, который строит минимальный ДКА и из НКА [4]

3 Регулярные выражения и языки

3.1 Регулярные выражения

Опр. 3.1 *(Клини [1951]). Регулярные выражения над алфавитом Σ определяются так:*

- *ε — регулярное выражение.*
- *Всякий символ a, где $a \in \Sigma$ — регулярное выражение.*
- *Если α, β — регулярные выражения, то тогда $(\alpha|\beta), (\alpha\beta)$ и $(\alpha)^*$ — тоже регулярные выражения.*

Всякое регулярное выражение α определяет язык над алфавитом Σ, обозначаемый через $L(\alpha)$.

Всякий символ из Σ обозначает одноэлементное множество, состоящее из односимвольной строки: $L(a) = \{a\}$

Оператор выбора задает объединение множеств: $L(\alpha|\beta) = L(\alpha) \cup L(\beta)$.

Конкатенация задает конкатенацию языков: $L(\alpha\beta) = L(\alpha)L(\beta)$.

Символ ε определяет пустое множество.

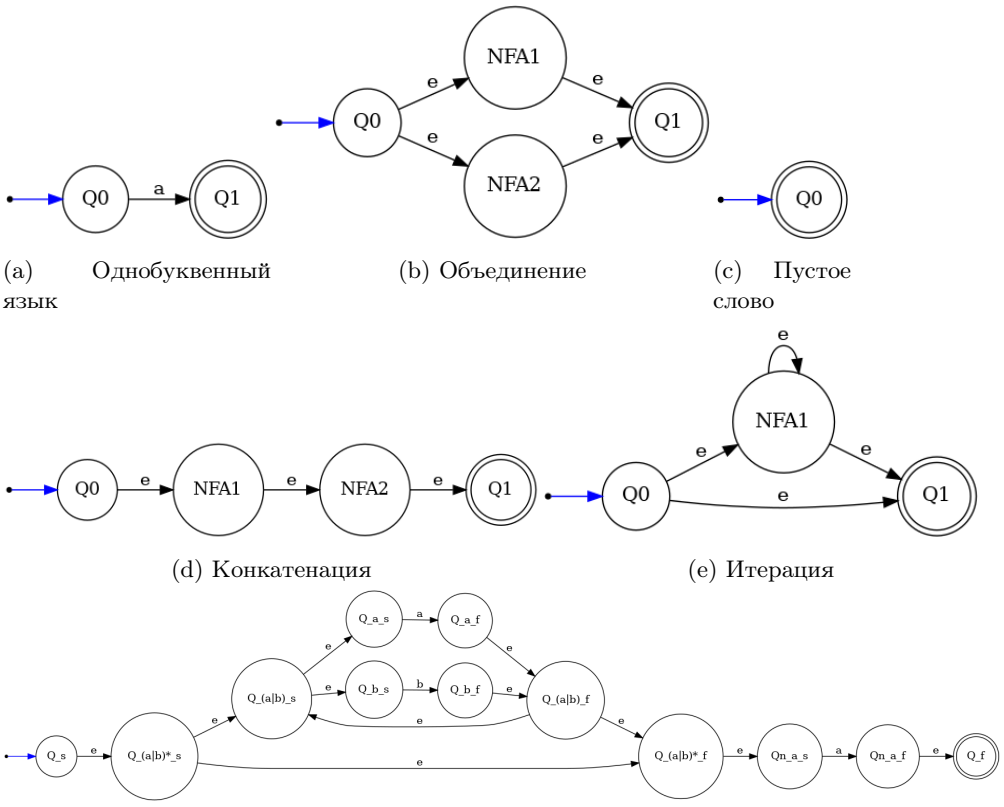
Оператор итерации задает итерацию: $L(\alpha^*) = L(\alpha)^*$.

Приоритеты операций: сперва итерация, затем конкатенация, затем выбор.

Синтаксис регулярных выражений на практике часто расширяется, к примеру:

- повторение один и более раз $(\alpha+)$, $(\alpha+) = \alpha\alpha^*$

[6] если этого еще не сделали на этапе построения ДКА, то можно обойти его граф из стартового состояния, например, в глубину, и собрать список достижимых состояний, а остальные удалить, модифицируя при этом остальные элементы автомата



(a) Однобуквенный язык (b) Объединение (c) Пустое слово

(d) Конкатенация (e) Итерация

(f) Пример: композиция для (a|b)*a

Рис. 5: Базовые автоматы построения Томпсона и пример композиции

- необязательная конструкция ($[\alpha]$, что означает «α или ничего»), $[\alpha] = \alpha|\varepsilon = \alpha|\varepsilon^*$

Л. 3.1 *(«построение Томпсона»). Для всякого регулярного выражения α, существует NFA C_α с одним начальным и одним принимающим состояниями, распознающий язык, задаваемый α.*

Доказательство производится индукцией по структуре регулярного выражения, структурные единицы представлены на Рис. 5.

Схема пошагового перехода от регулярного выражения к ДКА:

$$regexp \to NFA \to NFA_{simplified} \to DFA \to DFA_{min}, \tag{1}$$

была разобрана в разделах 1-3. Данная схема лежит в основе подавляющего большинства библиотек для работы с регулярными выражениями.

3.2 Регулярные языки

Любое регулярное выражение $reg(\Sigma)$ над алфавитом Σ задает регулярный язык L_{reg}.

Утв. 3.1 *Любой регулярный язык задаётся грамматикой* $\langle \Sigma, N, S, P \rangle$*, где правила из* P *имеют вид* $A \to a, A \to \gamma, A \to \varepsilon$*, где* γ *- либо* aB *(правая регулярная грамматика), либо* Ba *(левая регулярная грамматика),* $a \in \Sigma$*,* $A, B, S \in N$*.*

3.2.1 Свойства замкнутости регулярных языков

Операции, сохраняющие регулярность: ...объединение, пересечение, дополнение, разность, обращение, итерация, конкатенация, гомоморфизм, обратный гомоморфизм.

Так как, по определенению, класс регулярок замкнут относительно этих операций, то и композицию этих операций даёт и регулярку, и НКА, её распознающую. Тем не менее, так ли регулярны регулярные выражения в современных ЯП?...

3.2.2 Проверка на нерегулярность

Лемма о накачке (разрастании)

3.3 Регулярные выражения на практике

Регулярные выражения, входящие в современные языки программирования (в частности, PCRE в Perl), имеют больше возможностей, чем то, что мы рассмотрели: в них есть нумерованные обратные ссылки и т.д. Это позволяет задавать ими не только регулярные языки, но и более сложные, в частности, контекстно-свободные [5].

Пример (из [5]): `/^(a(?1)?b)$/` задаёт язык $a^n b^n, n \in [1 \dots \infty)$

Это регулярное выражение очень простое: `(?1)` ссылается на первую подмаску — `(a(?1)?b)`. Можно заменить `(?1)` подмасками, формируя таким образом рекурсивную зависимость:

```
/^(a(?1)?b)$/
/^(a(a(?1)?b)?b)$/
/^(a(a(a(?1)?b)?b)?b)$/
/^(a(a(a(a(?1)?b)?b)?b)?b)$/
```

...

Очевидно, это выражение способно описать любую строку с одинаковым количеством a и b, но конечный автомат, распознающий язык всех таких строк, построить нельзя.

4 Лексический анализ

Следующее приложение, о котором мы будем говорить – лексический анализ – это выделение во входном тексте характерных подстрок, «значащих» что-то, для дальнейших действий.

Опр. 4.1 *Лексема – последовательность символов, удовлетворяющая некоторому заданному требованию.*

Основная проблема выделения лексем – их может быть много и разных. Давайте работать не с лексемами, а с их «классами», на которые они делятся по смыслу нашей задачи.[7]

Опр. 4.2 *Токен – последовательность символов, «осмысленно» описывающая класс некоторой лексемы.*

Пример: $int \to TYPE$ (int – лексема, $TYPE$ – токен).
Для задания токенов, как правило, используют регулярные выражения.

Опр. 4.3 *Лексер, лексический анализатор, сканер – транслятор, преобразующий входную строку в последовательность токенов.*

4.1 Комментарии к практике

- Примеры работы с генератором лексических анализаторов `flex` были приведены на семинаре.
- В контесте 2 есть задачи, подразумевающие генерацию лексера по спецификации. И еще есть задача, которая демонстрирует, что в частных случаях («найти все вхождения слов в некоторый текст», «найти слово наименьшей длины, содержащее все подслова данного», и т.д.) можно, но не нужно писать регулярки, а лучше строить автомат по известной заранее структуре[8].
- Существует ряд подходов к оптимизации представления регулярных выражений, например, префиксное сжатие [6] и пр. Понятно, что в случае компиляции в минимальный ДКА для дальнейшего использования, этот подход никакого выигрыша в производительности не даст, так как ДКА будет одним и тем же с точностью до изоморфизма. Тем не менее, такой подход может повлиять на производительность промежуточных преобразований автоматов, так как НКА, полученный с оптимизацией, может отличаться от такового без оптимизации.

Итого:

[7] Здесь считаем такую классификацию однозначной.
[8] Например, суффиксный бор в случае с алгоритмом Ахо-Корасик (1975)

- В практических приложениях обычно используют библиотеки регулярных выражений или встроенные средства ЯП. Эти средства, как правило, реализуются по схеме, описанной выше, за исключением некоторых технических нюансов и ухищрений.
- При этом в лексическом анализе ЯП зачастую используют ручное написание лексеров. По крайней мере, в некоторых промышленных компиляторах (например, в Clang или OpenArkCompiler лексеры написаны вручную). Почему так – рассказано в разделе про компиляторы.
- Тем не менее, лексический анализ – довольно общая задача, и существуют инструменты построения лексеров по спецификациям, например, flex.
- В очень частных случаях («найти все вхождения слов в некоторый текст», «найти слово наименьшей длины, содержащее все подслова данного», и т.д.) можно, но не нужно писать регулярки, а лучше строить автомат по известной заранее структуре.

5 КС-грамматики и языки

5.1 Грамматики как системы переписывания

Опр. 5.1 *Формальная грамматика – кортеж* $G = (\Sigma, N, R, S)$*:*

- Σ *– терминальный алфавит – алфавит определяемого языка.*
- N *– нетерминальный алфавит*[9] *– алфавит промежуточных символов.*
- *Конечное множество правил* R *вида* $\alpha \to \beta, \alpha \in \{\Sigma \cup N\}^*, \beta \in \{\Sigma \cup N\}^* \cup \{\varepsilon\}$ *– каждое из которых описывает возможную структуру строк* β *со свойством* α*.*
- *Начальный символ* $S \in N$*.*

Грамматика при этом является системой переписывания строк, и системой порождения слов языка, где каждое слово порождается за конечное число шагов. Шаг порождения $w'\alpha w'' \to w'\beta w''$ состоит в замене α на подцепочку β в соответствии с правилом порождения $\alpha \to \beta$. Иначе говоря, если имеется некоторая цепочка и некоторая ее подцепочка является левой частью какого-то правила грамматики, то мы имеем право заменить эту левую часть правила на правую. Конечная последовательность шагов порождений называется порождением. Нуль или более порождений будет обозначать знаком $\to^*$. Обозначение $\alpha \to^* \beta$ говорит о том, что цепочка β получена из цепочки α конечным числом подстановок на основе правил

[9] В лингвистике нетерминалы называются синтаксическими категориями

порождения. В этом обозначении может быть так, что подстановка не была применена ни разу, в этом случае цепочка $alpha = beta$.

Язык, задаваемый (порождаемый) грамматикой G – это множество слов, составленных из терминальных символов и порожденных из начального символа грамматики $L = \{w | S \to^* w\}$.

Опр. 5.2 *Грамматики G_1 и G_2 называются эквивалентными, если они задают один и тот же язык: $L(G_1) = L(G_2)$*

Понятие регулярной грамматики уже вводилось в разделе 3. Ниже будет введено понятие контекстно-свободной грамматики. Эти два типа грамматик являются наиболее исследованными типами иерархии Хомского (типами 3 и 2 соответственно), о которой мы будем говорить позже, и наиболее интересными нам в данном курсе.

5.1.1 Выводимость в грамматике

Опр. 5.3 *Отношение непосредственной выводимости. Последовательность терминалов и нетерминалов $\gamma\alpha\delta$ непосредственно выводится из $\gamma\beta\delta$ при помощи правила $\alpha \to \beta$ ($\gamma\alpha\delta \Rightarrow \gamma\beta\delta$), если*

- $\alpha \to \beta \in P$
- $\gamma, \delta \in \{\Sigma \cup N\}^* \cup \varepsilon$

Опр. 5.4 *Рефлексивно-транзитивное замыкание отношения — это наименьшее рефлексивное и транзитивное отношение, содержащее исходное.*

Опр. 5.5 *Отношение выводимости является рефлексивно-транзитивным замыканием отношения непосредственной выводимости*

- $\alpha\beta$ *означает* $\exists\gamma_0, \dots \gamma_k : \ \alpha[]\gamma_0[]\gamma_1[] \dots []\gamma_{k-1}[]\gamma_k[]\beta$
- *Транзитивность:* $\forall\alpha, \beta, \gamma \in \{\Sigma \cup N\}^* \cup \varepsilon : \ \alpha\beta, \beta\gamma \Rightarrow \alpha\gamma$
- *Рефлексивность:* $\forall\alpha \in \{\Sigma \cup N\}^* \cup \varepsilon : \ \alpha\alpha$
- $\alpha\beta$ — *α выводится из β*
- $\alpha[k]\beta$ — *α выводится из β за k шагов*
- $\alpha[+]\beta$ — *при выводе использовалось хотя бы одно правило грамматики*

Опр. 5.6 (Вывод слова в грамматике) *Слово $\omega \in \Sigma^*$ выводимо в грамматике $\langle \Sigma, N, P, S \rangle$, если существует некоторый вывод этого слова из начального нетерминала $S\omega$.*

Частные случаи вывода:

Опр. 5.7 *Левосторонний вывод. На каждом шаге вывода заменяется самый левый нетерминал.*

Опр. 5.8 *Правосторонний вывод. На каждом шаге вывода заменяется самый правый нетерминал.*

5.2 КС-грамматики

Опр. 5.9 *Контекстно-свободная грамматика – кортеж* $G = (\Sigma, N, R, S)$*:*

- Σ *– терминальный алфавит.*
- N *– нетерминальный алфавит.*
- *Конечное множество правил* R *вида* $N_i \to \alpha, N_i \in N, \alpha \in \{\Sigma \cup N\}^* \cup \{\varepsilon\}$
- *Начальный символ* $S \in N$*.*

То есть, исходя из общего определения[10] формальной грамматики (5.1), КС грамматика – такая грамматика, в которой каждое правило порождения позволяет явно установить свойство подстроки как промежуточный символ, либо вывести подстроку с заданным свойством только из промежуточного символа, вне зависимости от того, что стоит слева или справа в строке в процессе переписывания. Далее будем называть промежуточные символы нетерминальными, и, чтобы не было путаницы, потребуем $\Sigma \cap N = \emptyset$.

При спецификации грамматики часто опускают множества терминалов и нетерминалов, оставляя только множество правил. При этом нетерминалы часто обозначаются прописными латинскими буквами, терминалы — строчными, а стартовый нетерминал обозначается буквой S. Мы будем следовать этим обозначениям, если не указано иное.

Опр. 5.10 *Грамматика называется однозначной, если для любого порождённого по ней слова последовательность порождения – единственна.*

Иными словами, для слова, порождаемого однозначной КС-грамматикой, существует единственное дерево разбора.

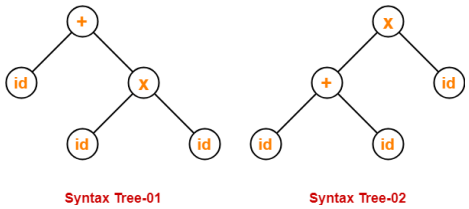


Рис. 6: Неоднозначный разбор арифметического выражения

Соответственно, множество языков, порождаемое КС-грамматиками, называется КС-языками.

Л. 5.1 (Лемма о накачке для КС-языков) *Для каждого КС-языка* $L \subseteq \Sigma^*$ *существует такая константа* $p \geq 1$*, что для любой строки* $w \in L$*, для которой* $|w| > p$*, существует разложение* $w = xuyvz$*, где* $|uv| > 0$ *и* $|uyv| \leq p$*, для которого* $xu^iyv^iz \in L$ *при всех* $i \geq 0$*.*

[10] и значения

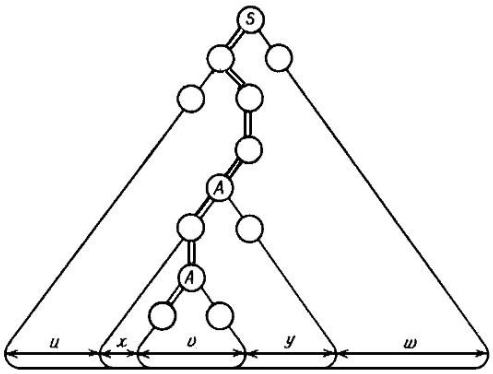


Рис. 7: К лемме о накачке: структура дерева вывода для uvwxy

5.2.1 Формы представления КС-грамматик

В зависимости от вида правил, КС-грамматики подразделяются на формы, свойства и алгоритмы анализа которых зачастую существенно различаются. Опишем некоторые из форм, которыми будем пользоваться.

Опр. 5.11 *Грамматика находится в Нормальной форме Хомского (НФХ, CNF), если любое правило имеет один из трех видов:*

1. $S \to \varepsilon$
2. $N_i \to N_j N_k, N_i, N_j, N_k \in N$
3. $N_i \to t, N_i \in N, t \in A$

Замечание: в НФХ стартовый нетерминал не встречается в правых частях правил, ε-правила только для стартового нетерминала.

Опр. 5.12 *Грамматика находится в Ослабленной Нормальной форме Хомского (weak-CNF), если...*

Л. 5.2 *Любую КС-грамматику можно привести к НФХ.*

Алгоритм приведения к НФХ был разобран на семинаре.

5.2.2 Об алгоритмах синтаксического анализа КС-языков

КС-языки, наравне с регулярными – наиболее полно исследованный класс формальных языков, для которых существует целое разнообразие алгоритмов разбора различной сложности. На практике, в особенности при анализе языков программирования, основным требованием к алгоритму разбора является его вычислительная эффективность, даже если он не годится для произвольных КС-грамматик. Поэтому зачастую применяются алгоритмы

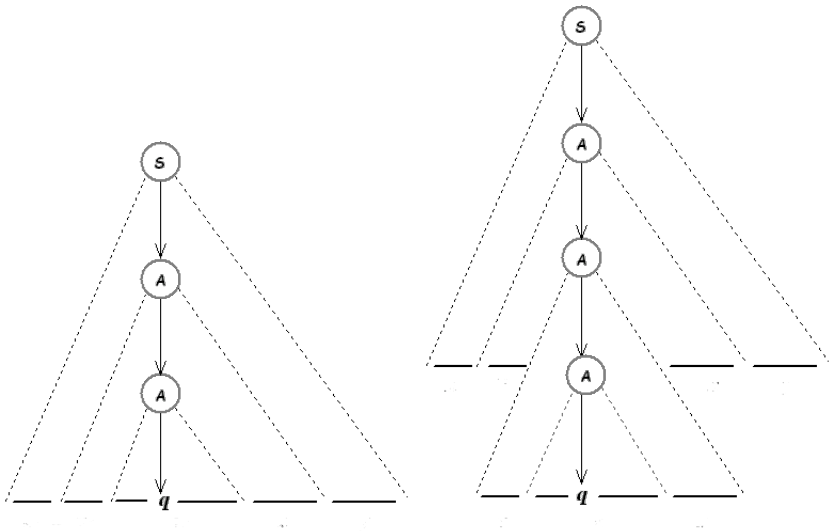


Рис. 8: К доказательству леммы о накачке: «1 шаг накачки».

с временной сложностью порядка $O(n)$ на слове длины n, в частности, алгоритмы LL, LR-семейства, которые рассмотрены далее в соответствующих разделах.

Мы же начнём повествование с алгоритма кубичной сложности, позволяющего осуществлять разбор слов, порождаемых КС-грамматиками (даже неоднозначными), заданными в специальной форме, к которой можно привести любую КС-грамматику (размер полученной грамматики – количества правил и нетерминалов – при этом может сильно разрастаться по сравнению с исходной) – нормальной форме Хомского. Алгоритм носит имя создателей – Кока, Янгера и Касами (CYK)[7].

5.2.3 Алгоритм CYK для строк

CYK (Кока, Янгера и Касами, Cocke-Younger-Kasami) — один из классических алгоритмов синтаксического анализа. Его асимптотическая сложность в худшем случае — $O(n^3 \cdot |N|)$, где n — длина входной строки, а N — количество нетерминалов во входной граммтике [**?**].

Для его применения необходимо, чтобы подаваемая на вход грамматика находилась в Нормальной Форме Хомского (НФХ). Других ограничений нет, следовательно,данный алгоритм применим для работы с произвольными контекстно-своболными языками.

В основе алгоритма лежит принцип динамического программирования. Алгоритм строится из следующих соображений:

1. Для правила вида $A \to a$ очевидно, что из A выводится ω (с применением этого правила) тогда и только тогда, когда $a = \omega$:

$$A\omega \iff \omega = a$$

2. Для правила вида $A \to BC$ понятно, что из A выводится ω (с применением этого правила) тогда и только тогда, когда существуют две цепочки ω_1 и ω_2 такие, что ω_1 выводима из B, ω_2 выводима из C и при этом $\omega = \omega_1\omega_2$:

$$A[]BC\omega \iff \exists\omega_1, \omega_2 : \omega = \omega_1\omega_2, B\omega_1, C\omega_2$$

Или в терминах позиций в строке:

$$A[]BC\omega \iff \exists k \in [1 \dots |\omega|] : B\omega[1 \dots k], C\omega[k+1 \dots |\omega|]$$

В процессе работы алгоритма заполняется булева трехмерная[11] матрица M размера $n \times n \times |N|$ таким образом, что

$$M[i, j, A] = true \iff A\omega[i \dots j]$$

.

Первым шагом инициализируем матрицу, заполнив значения $M[i, i, A]$:

- $M[i, i, A] = true$, если в грамматике есть правило $A \to \omega[i]$.
- $M[i, i, A] = false$, иначе.

Далее используем динамику: на шаге $m > 1$ предполагаем, что ячейки матрицы $M[i', j', A]$ заполнены для всех нетерминалов A и пар $i', j' : j' - i' < m$. Тогда можно заполнить ячейки матрицы $M[i, j, A]$, где $j - i = m$ следующим образом:

$$M[i, j, A] = \bigvee_{A \to BC} \bigvee_{k=i}^{j-1} M[i, k, B] \wedge M[k, j, C]$$

По итогу работы алгоритма значение в ячейке $M[0, |\omega|, S]$, где S — стартовый нетерминал грамматики, отвечает на вопрос о выводимости цепочки ω в грамматике.

Рассмотрим пример работы алгоритма CYK на грамматике правильных скобочных последовательностей в Нормальной Форме Хомского.

Пример.

Пусть дана грамматика:

$$\begin{array}{lll} S \to AS_2 \mid \varepsilon & S_2 \to b \mid BS_1 \mid S_1S_3 & A \to a \\ S_1 \to AS_2 & S_3 \to b \mid BS_1 & B \to b \end{array}$$

[11] Можно считать матрицу двухмерной, а не трехмерной булевой, но содержащей в элементах не биты, а списки соответствующих нетерминалов, как показано ниже в примере

Проверим выводимость строки $\omega = aabbab$ в ней.

Будем иллюстрировать работу алгоритма двумерными матрицами размера $n \times n$, где в ячейках указано множество нетерминалов, выводящих соответствующую подстроку.

Шаг 1. Инициализируем матрицу элементами на главной диагонали:

$$\begin{pmatrix} \{A\} & & & & & \\ & \{A\} & & & & \\ & & \{B, S_2, S_3\} & & & \\ & & & \{B, S_2, S_3\} & & \\ & & & & \{A\} & \\ & & & & & \{B, S_2, S_3\} \end{pmatrix}$$

Шаг 2. Заполняем диагональ, находящуюся над главной:

$$\begin{pmatrix} \{A\} & & & & & \\ & \{A\} & lightgray\{S_1\} & & & \\ & & \{B, S_2, S_3\} & & & \\ & & & \{B, S_2, S_3\} & & \\ & & & & \{A\} & lightgray\{S_1\} \\ & & & & & \{B, S_2, S_3\} \end{pmatrix}$$

В двух ячейках появились нетерминалы S_1 благодаря присутствию в грамматике правила $S_1 \to AS_2$.

Шаг 3. Заполняем следующую диагональ:

$$\begin{pmatrix} \{A\} & & & & & \\ & \{A\} & \{S_1\} & red\{S_2\} & & \\ & & \{B, S_2, S_3\} & & & \\ & & & \{B, S_2, S_3\} & & lightgray\{S_2, S_3\} \\ & & & & \{A\} & \{S_1\} \\ & & & & & \{B, S_2, S_3\} \end{pmatrix}$$

Шаг 4. И следующую за ней:

$$\begin{pmatrix} \{A\} & & & lightgray\{S_1, S\} & & \\ & \{A\} & \{S_1\} & \{S_2\} & & \\ & & \{B, S_2, S_3\} & & & \\ & & & \{B, S_2, S_3\} & & \{S_2, S_3\} \\ & & & & \{A\} & \{S_1\} \\ & & & & & \{B, S_2, S_3\} \end{pmatrix}$$

Шаг 5 Заполняем предпоследнюю диагональ:

$$\begin{pmatrix} \{A\} & & & \{S_1, S\} & & \\ & \{A\} & \{S_1\} & \{S_2\} & & lightgray\{S_2\} \\ & & \{B, S_2, S_3\} & & & \\ & & & \{B, S_2, S_3\} & & \{S_2, S_3\} \\ & & & & \{A\} & \{S_1\} \\ & & & & & \{B, S_2, S_3\} \end{pmatrix}$$

Шаг 6. Завершаем выполнение алгоритма:

$$\begin{pmatrix} \{A\} & & & \{S_1, S\} & & lightgray\{S_1, S\} \\ & \{A\} & \{S_1\} & \{S_2\} & & \{S_2\} \\ & & \{B, S_2, S_3\} & & & \\ & & & \{B, S_2, S_3\} & & \{S_2, S_3\} \\ & & & & \{A\} & \{S_1\} \\ & & & & & \{B, S_2, S_3\} \end{pmatrix}$$

Стартовый нетерминал находится в верхней правой ячейке, а значит цепочка $aabbab$ выводима в нашей грамматике.

Теперь выполним алгоритм на цепочке $\omega = abaa$.

Шаг 1. Инициализируем таблицу:

$$\begin{pmatrix} \{A\} & & & \\ & \{B, S_2, S_3\} & & \\ & & \{A\} & \\ & & & \{A\} \end{pmatrix}$$

Шаг 2. Заполняем следующую диагональ:

$$\begin{pmatrix} \{A\} & lightgray\{S_1, S\} & & \\ & \{B, S_2, S_3\} & & \\ & & \{A\} & \\ & & & \{A\} \end{pmatrix}$$

Больше ни одну ячейку в таблице заполнить нельзя и при этом стартовый нетерминал отсутствует в правой верхней ячейке, а значит эта строка не выводится в данной грамматике.

5.3 Обобщение синтаксического анализа со строк на графы

5.3.1 CYK для графов

Первым шагом на пути к обобщению CYK для поиска путей, задаваемых языками меток рёбер на графах, является модификация представления входа. Прежде мы сопоставляли каждому символу слова его позицию во входной цепочке, поэтому при инициализации заполняли главную диагональ

матрицы. Вместо этого, обозначим числами позиции между символами. В результате, слово можно представить в виде линейного графа следующим образом(в качестве примера рассмотрим слово $aabbab$):

$$0 \xrightarrow{a} 1 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 6$$

Что нужно изменить в описании алгоритма, чтобы он продолжал работать при подобной нумерации? Каждая буква теперь идентифицируется не одним числом, а парой — номера слева и справа от нее. При этом чисел стало на одно больше, чем при прежнем способе нумерации.

Возьмем матрицу $(n+1) \times (n+1) \times |N|$ и при инициализации будем заполнять не главную диагональ, а диагональ прямо над ней. Таким образом, мы начинаем наш алгоритм с определения значений $M[i, j, A]$, где $j = i+1$. При этом наши дальнейшие действия в рамках алгоритма не изменятся.

Для примера 5.2.3 на шаге инициализации матрица выглядит следующим образом:

$$\begin{pmatrix} \{A\} & & & & & \\ & \{A\} & & & & \\ & & \{B, S_2, S_3\} & & & \\ & & & \{B, S_2, S_3\} & & \\ & & & & \{A\} & \\ & & & & & \{B, S_2, S_3\} \end{pmatrix}$$

А в результате работы алгоритма имеем:

$$\begin{pmatrix} \{A\} & & & \{S_1, S\} & & \{S_1, S\} \\ & \{A\} & \{S_1\} & \{S_2\} & & \{S_2\} \\ & & \{B, S_2, S_3\} & & & \\ & & & \{B, S_2, S_3\} & & \{S_2, S_3\} \\ & & & & \{A\} & \{S_1\} \\ & & & & & \{B, S_2, S_3\} \end{pmatrix}$$

Мы представили входную строку в виде линейного графа, а на шаге инициализации получили его матрицу смежности. Добавление нового нетерминала в язейку матрицы можно рассматривать как нахождение нового пути между соответствующими вершинами, выводимого из добавленного нетерминала. Таким образом, шаги алгоритма напоминают построение транзитивного замыкания графа. Различие заключается в том, что мы добавляем новые ребра только для тех пар нетерминалов, для которых существует соответстующее правило в грамматике.

Алгоритм можно обобщить и на произвольные графы с метками, рассматриваемые в этом курсе. При этом можно ослабить ограничение на форму входной грамматики: она должна находиться в ослабленной Нормальной Форме Хомского.

5.3.2 Алгоритм (Y. Hellings, 2015) с рабочими множествами для графов

Можно заметить, что CYK производит много избыточных итераций. Можно модифицировать алгоритм, чтобы не просматривались заведомо пустые ячейки. Данная модификация была предложена Хеллингсом [9] в именном алгоритме, но также фигурирует и в более ранних работах [8]. В основе алгоритма лежит обработка двух рабочих множеств: текущего и конечного.

Идеологически, на каждом шаге алгоритма:

- Просматривается какой-то путь, полученный на текущем шаге.
- Нужно попробовать приконкатенировать к нему какую-то из существовавших ранее подцепочек слева, и справа.
- Просмотрев все текущие пути, перейти на новую итерацию цикла.

Процесс повторяется, пока текущее множество не опустеет.

Несмотря на то, что мы храним не матрицу в явном виде, а рабочее множество, можно хранить и матрицу, тогда пути восстанавливаются более естественным способом [8]. Псевдокод алгоритма Хеллингса представлен в листинге 1.

Algorithm 1 Алгоритм Хеллингса

```
1: function HellingsAlgo(G = ⟨Σ, N, P, S⟩, 𝒢 = ⟨V, E, L⟩)
2:     r ← {(N_i, v, v) | v ∈ V ∧ N_i → ε ∈ P} ∪ {(N_i, v, u) | (v, t, u) ∈ E ∧ N_i →
       t ∈ P}
3:     m ← r
4:     while m ≠ ∅ do
5:         (N_i, v, u) ← m.pick()
6:         for (N_j, v', v) ∈ r do
7:             for N_k → N_j N_i ∈ P таких что ((N_k, v', u) ∉ r) do
8:                 m ← m ∪ {(N_k, v', u)}
9:                 r ← r ∪ {(N_k, v', u)}
10:            end for
11:        end for
12:        for (N_j, u, v') ∈ r do
13:            for N_k → N_i N_j ∈ P таких что ((N_k, v, v') ∉ r) do
14:                m ← m ∪ {(N_k, v, v')}
15:                r ← r ∪ {(N_k, v, v')}
16:            end for
17:        end for
18:    end while
19:    return r
20: end function
```

Несмотря на то, что теоретически худшие случаи должны при таком

подходе давать временную асимптотику, как у CYK для графов, на практике, как правило, данный алгоритм отрабатывает быстрее.

5.3.3 Другие алгоритмы

5.4 КС-достижимость

5.4.1 Постановка задач

Пусть $\boldsymbol{L}(G)$ – язык сконкатенированных меток рёбер графа $G = (V, E, L)$: V, E, L – вершины, рёбра, метки, $E \subseteq V \times L \times V$, $\boldsymbol{L}(G) = \{w\}$,
$w = w(v_0 l_0 v_1, v_1 l_1 v_2, ...)$, $v_i, l_j, v_k \in E$, то классически ставятся следующие задачи:

- Восстановить все пары вершин, служащих началом и концом путей, заданных данной КС-грамматикой.
- Восстановить все пары вершин, служащих началом и концом путей, заданных данной КС-грамматикой, и восстановить само множество путей. Сложности:
  - Пути нужно где-то хранить
  - Путей может оказаться формально бесконечное количество, даже если граф конечен. Решение – пути хранятся в специальной структуре данных, именуемой сжатый лес разбора (Shared packed parsing forest, SPPF).
- Для заданной пары вершин, проверить, есть ли между ними путь, заданный данной КС-грамматикой.
- Проверить пустоту пересечения $L(G)$ и некоторого другого языка.

5.4.2 Классические подходы решения

5.4.3 Вспомогательные структуры данных

Сжатый лес разбора (Shared packed parsing forest, SPPF). Впервые подобная идея была предложена Джоаном Рекерсом в его кандидатской диссертации [**?**]. В дальнейшем она нашла широкое применение в обобщённом синтаксическом анализе и получила серьёзное развитие [12]. Оптимальное асимптотическое поведение достигается при использовании бинаризованного SPPF [**?**] — в этом случае объём леса составляет $O(n^3)$, где n — это длина входной строки.

Рассмотрим способ построения SPPF на примере.

Во-первых, заметим, что в дереве вывода каждая вершина соответствует выводу какой-то подстроки с известными позициями начала и конца. Давайте будем сохранять эту информацию в вершинах дерева. Таким образом, метка любой вершины – это тройка (i, q, j), где i — координата начала подстроки, соответствующей этой вершине, j — координата конца, $q \in \Sigma \cup N$ —

метка как в исходном определении. Так как такие вершины содержат символ, терминальный или нетерминальный, их в терминологии лесов разбора принято называть *символьными*.

Во-вторых, заметим, что любая внутренняя вершина со своими непосредственными потомками связаны продукцией в грамматике: вершина появляется благодаря применению конкретной продукции в процессе вывода. Давайте занумеруем все продукции в грамматике и добавим в дерево вывода ещё один тип вершин – *дополнительные*, или *промежуточные* вершины – в которых будем хранить номер применённой продукции. Получим следующую конструкцию: непосредственный предок дополнительной вершины — это левая часть продукции, а непосредственные потомки дополнительной вершины — это правая часть продукции.

Построим модифицированное дерево вывода цепочки ${}_0a_1b_2a_3b_4a_5b_6$ в грамматике

$$G_0 = \langle \{a, b\}, \{S\}, S, \{ \\ (0)S \to a\ S\ b\ S, \\ (1)S \to \varepsilon \\ \}\rangle$$

Сохраняемая нами дополнительная информация позволит переиспользовать вершины в том случае, если деревьев вывода оказалось несколько (в случае неоднозначной грамматики). При этом мы можем не бояться, что переиспользование вершин приведёт к появлению ранее несуществовавших деревьев вывода, так как дополнительная информация позволяет делать только “безопасные” склейки и затем восстанавливать только корректные деревья. Таким образом, мы можем представить лес вывода в виде единой структуры данных без дублирования информации.

Сжатие леса разбора. Построим несколько деревьев вывода цепочки ${}_0a_1b_2a_3b_4a_5b_6$ в грамматике

$$G_1 = \langle \{a, b\}, \{S\}, S, \{ \\ (0)S \to SS, \\ (1)S \to a\ S\ b, \\ (2)S \to \varepsilon \\ \}\rangle$$

Предположим, что мы строим левосторонний вывод. Тогда после первого применения продукции 0 у нас есть два варианта переписывания первого нетерминала: либо с применением продукции 0, либо с применением продукции 1:

$$\mathbf{S} \xrightarrow{0} \mathbf{S}S \xrightarrow{0} \mathbf{S}SS \xrightarrow{1} a\mathbf{S}bSS \xrightarrow{2} ab\mathbf{S}S \xrightarrow{1} aba\mathbf{S}bS \xrightarrow{2} abab\mathbf{S} \xrightarrow{1} ababa\mathbf{S}b \xrightarrow{2} abababa$$

$$\mathbf{S} \xrightarrow{0} \mathbf{S}S \xrightarrow{1} a\mathbf{S}bS \xrightarrow{2} ab\mathbf{S} \xrightarrow{0} ab\mathbf{S}S \xrightarrow{1} aba\mathbf{S}bS \xrightarrow{2} abab\mathbf{S} \xrightarrow{1} ababa\mathbf{S}b \xrightarrow{2} abababa$$

Сначала рассмотрим первый вариант (применили переписывание по продукции 0). Все остальные шаги вывода деретерминированы и в результате мы получим следующее дерево разбора:

Теперь рассмотрим второй вариант — применить продукцию 1. Остальные шаги вывода всё также детерминированы. В результате мы получим следующее дерево вывода:

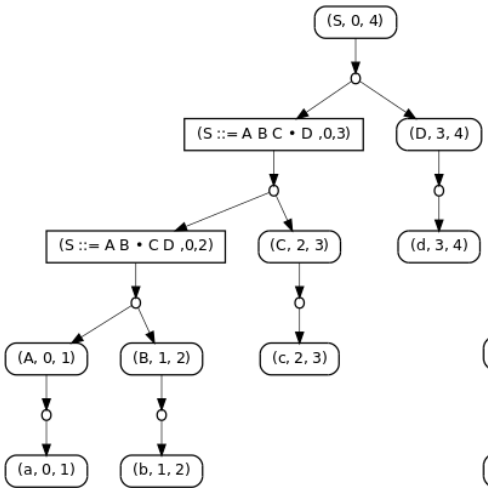
В двух построенных деревьях большое количество одинаковых узлов. Построим структуру, которая содержит оба дерева и при этом никакие нетерминальные и терминальные узлы не встречаются дважды. В результате мы получим следующий граф:

Мы получили очень простой вариант сжатого представления SPPF.

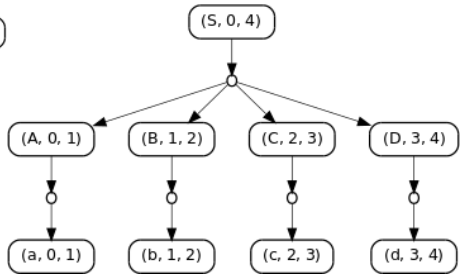
Пример 2 Рассмотрим грамматику:

$S \to ABCD \quad A \to a \quad B \to b \quad C \to c \quad D \to d.$

и разбор в ней слова *abcd*. Построим лес разбора:



(a) SPPF



(b) SPPF без промежуточных вершин

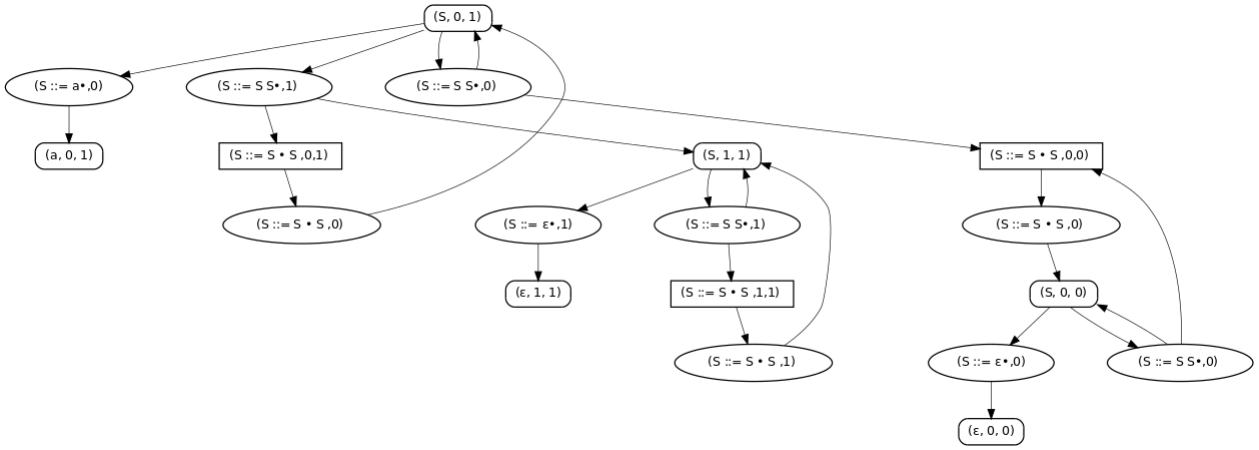
Пример 3 Рассмотрим грамматику:

$S \to SS \mid a \mid \varepsilon$. Очевидно, она циклическая. Рассмотрим SPPF для слова a в такой грамматике:

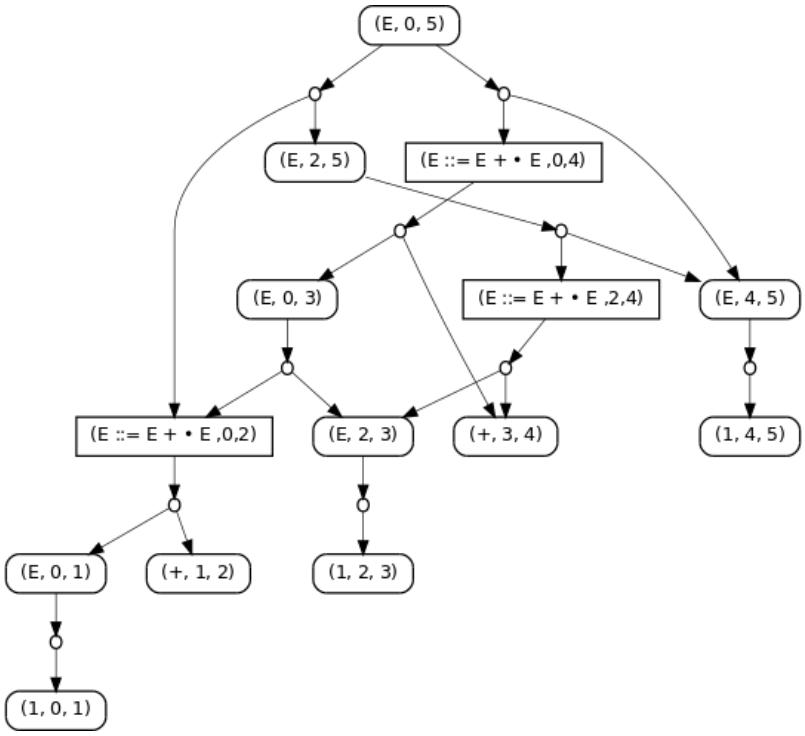
Пример 4. Рассмотрим неоднозначную грамматику $E \to E + E \mid 1$ и входную строку $1+1+1$. Лес разбора называется неоднозначным, если хранит хотя бы одну неоднозначную конструкцию (несколько деревьев разбора одной строчки). Построим такой лес:

В нем корневая вершина $(E, 0, 5)$ имеет 2 сжатых вершины-потомка. следовательно, минимум 2 различных дерева разбора начинается с этой вершины – это деревья, выводящие $(E+(E+E))$ и $((E+E)+E)$ соответственно.

Множество деревьев разбора, содержащихся в SPPF, находится по следующей процедуре: стартуя в корне SPPF, обходим его как дерево, посещая



(a) Пример 3: SPPF для a в $S \to SS \mid a \mid \varepsilon$



(a) Пример 4: SPPF для $E \to E + E \mid 1$ и $1 + 1 + 1$

каждую сжатую вершину под текущей, а затем посещая каждую дочернюю вершину под сжатыми рекурсивно.

Структурные свойства SPPF.

- At first note that each symbol node (E, j, i) with $E \in T \cup N \cup \{\varepsilon\}$ is unique, то есть SPPF не может содержать 2 символьные вершины (A, k, l) и (B, m, n) , где $A = B, k = m$ и $l = n$.

- Ноды для терминалов не содержат потомков и являются листовыми в лесе разбора. Ноды для символов-терминалов (A, j, i) имеют сжатые вершины потомки с метками вида $(A ::= \gamma\cdot, k)$, где $j \leq k \leq i$, и возможное число потомков не ограничено двумя.

- Промежуточные вершины (t, j, i) имеют потомками сжатые вершины с метками (t, k), где $j \leq k \leq i$.

- Сжатые вершины (t, k) имеют 1 или 2 потомка. The right child is a symbol node (x, k, i) and the left child (if it exists) is a symbol or intermediate node with label (s, j, k), where $j \leq k \leq i$. Packed nodes have always exactly one parent which is a symbol node or intermediate node.

- It is useful to observe that the SPPF is a bipartite graph, with on the one hand the set of intermediate and symbol nodes and on the other hand the set of packed nodes. Therefore edges always go from a node of the first type to a node of the second type, or the other way round. As a consequence, cyles in the SPPF are always of even length.

Преобразование в абстрактное синтаксическое дерево

Наконец, зачастую, на выходе CA бывает важно получить абстрактное синтаксическое дерево (AST), а не что-либо иное. Причем нас может интересовать только одно AST, и нужно ободнозначнить получение дерева разбора, из которого AST и получается. Конечно, применяются и другие простейшие трансформации, типа удаления пробелов и т.д. Подходы к такому ободнозначниванию бывают различные, например, внедрение специальных фильтров, которые позволяют сделать что-то наподобие PEG, кроме того, существуют подходы, которые (в особенности при проведении обобщенного LL-анализа) позволяют избегать добавления неоднозначных результатов разбора непосредственно при построении леса разбора [14].

5.4.4 КС-достижимость через операции линейной алгебры

Из данных выше материалов следует, что и разбор, и вывод слов языка по грамматике суть исчисление над термами. С другой стороны, алгоритмы типа CYK демонстрируют процесс разбора как последовательность преобразований специальных матриц. Возникает идея свести разбор к матричному исчислению с хитро заданными операциями сложения и умножения:

инструмент матриц намного лучше исследован и оптимизирован человечеством для различных вычислительных задач, существует огромное количество эффективных его реализаций, в конце концов, работа с матрицами более привычна для инженеров, исследователей и студентов, нежели работа с языками и грамматиками.

Ранее нами был разобран алгоритм для решения задачи КС достижимости на основе CYK. Заметим, что обход матрицы напоминает умножение матриц, в ячейках которых хранятся множества нетерминалов:

$$M_3 = M_1 \times M_2$$
$$M_3[i,j] = \sum_{k=1}^{n} M[i,k] * M[k,j]$$

, где сумма — это объединение множеств:

$$\sum_{k=1}^{n} = \bigcup_{k=1}^{n}$$

, а поэлементное умножение определено следующим образом:

$$S_1 * S_2 = \{N_1^0 ... N_1^m\} * \{N_2^0 ... N_2^l\} = \{N_3 \mid (N_3 \to N_1^i N_2^j) \in P\}.$$

Таким образом, алгоритм решения задачи КС достижимости может быть дан в терминах перемножения матриц над полукольцом с соответствующими операциями.

Для частного случая этой задачи, синтаксического анализа линейного входа, существует алгоритм Валианта [**?**], использующий эту идею. Однако он не обобщается на графы из-за того, что существенно использует возможность упорядочить обход матрицы (см. разницу в CYK для линейного входа и для графа). Поэтому, хотя для линейного случая алгоритм Валианта и является алгоритмом синтаксического анализа для произвольных КС грамматик за субкубическое время, его обобщение до задачи КС достижимости в произвольных графах с сохранением асимптотики является нетривиальной задачей [**?**]. В настоящее время алгоритм с субкубической сложностью получен только для частного случая — языка Дика с одним типом скобок — Филипом Брэдфордом [**?**].

В случае с линейным входом, отдельного внимания заслуживает работа Лиллиан Ли (Lillian Lee) [**?**], где она показывает, что задача перемножения матриц сводима к синтаксическому анализу линейного входа. Аналогичных результатов для графов на текущий момент не известно.

Поэтому рассмотрим более простую идею, изложенную в статье и диссертации Рустама Азимова [**?**]: будем строить транзитивное замыкание графа через наивное (не через возведение в квадрат) умножение матриц.

Пусть $\mathcal{G} = (V, E)$ — входной граф и $G = (N, \Sigma, P)$ — входная грамматика. Тогда алгоритм может быть сформулирован как представлено в листинге 2.

Algorithm 2 Context-free recognizer for graphs

```
1: function contextFreePathQuerying(𝒢, G)
2:     n ← количество узлов в 𝒢
3:     E ← направленные ребра в 𝒢
4:     P ← набор продукций из G
5:     T ← матрица n × n, в которой каждый элемент
6:     for all (i, x, j) ∈ E do                    ▷ Инициализация матрицы
7:         T_{i,j} ← T_{i,j} ∪ {A | (A → x) ∈ P}
8:     end for
9:     for all i ∈ 0 ... n − 1 do      ▷ Добавление петель для нетерминалов,
   порождающих пустую строку
10:        T_{i,i} ← T_{i,i} ∪ {A ∈ N | A → ε}
11:    end for
12:    while матрица T меняется do
13:        T ← T ∪ (T × T)              ▷ Вычисление транзитивного замыкания
14:    end while
15:    return T
16: end function
```

Особенности реализации

Переход к матричным операциям позволяет с минимальными затратими получить эффективную параллельную реализацию алгоритма для решения задачи КС достижимости в графах. Благодаря этому, хотя асимптотически приведенные алгоритмы и имеют большую сложность чем, скажем, алгоритмы CYK и Хеллингса, в результате эффективного распараллеливания на практике они работают быстрее однопоточных алгоритов с лучшей сложностью.

Далее рассмотрим некоторые детали, упрощающие реализацию с использованием современных библиотек и аппаратного обеспечения.

Так как множество нетерминалов и правил конечно, то мы можем свести представленный выше алгоритм к булевым матрицам: для каждого нетерминала заведём матрицу, такую что в ячейке стоит 1 тогда и только тогда, когда в исходной матрице в соответствующей ячейке сожержится этот нетерминал. Тогда перемножение пары таких матриц, соответсвующих нетерминалам A и B, соответствует построению путей, выводимых из нетерминалов, для которых есть правила с правой частью вида AB.

5.4.5 Комментарии к практике

5.4.6 Пример: КС достижимость при анализе программ

Пусть по-прежнему $\boldsymbol{L}(G)$ – язык сконкатенированных меток рёбер графа $G = (V, E, L)$, V, E, L – вершины, рёбра, метки. Если G явдяется некоторым представлением программы p: $G = (V, E, L) = G(p)$, $E \subseteq V \times L \times V$,

$\boldsymbol{L}(G) = \{w(p)\}$, $w(p) = w(v_0 l_0 v_1, v_1 l_1 v_2, ...)$, $v_i, l_j, v_k \in E$, то можно рассмотреть следующие задачи:

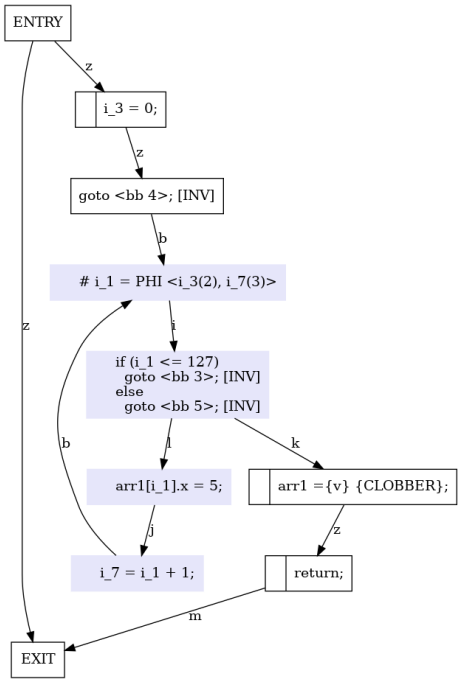


Рис. 12: Пример графа, полученного из программы в промежуточном представлении GCC GIMPLE [13]. Сиреневым подсвечен цикл, детектированный посредством решения задачи КС-достижимости (путь по рёбрам $iljb$).

1. Поиск паттерна. Найти все пути в G, содержащие слова из L': $\boldsymbol{L'}(G) \subseteq \boldsymbol{L}(G) : \{P_G^{patterns}\} = \{P_G | w(P_G) \in \boldsymbol{L'}(G)\}$.
2. Проверка на анти-паттерн: пусто ли пересечение языка графа с «языком анти-паттернов» $\boldsymbol{L''}(G)$: $\{P_G \cap \boldsymbol{L''}(G)\}? \equiv \emptyset$.
3. Классическая задача достижимости – найти все пары вершин (состояний программы, точек останова и т.д.), таких, что между ними существует нужный путь?
4. Подзадача классической задачи достижимости – существует ли нужный путь из точки A в B в программе?

Возможна и постановка последовательной проверки на КС-достижимость: сначала выделяется множество путей по (1), а далее проверяется (2). Такие паттерны (2) для (1) назовём «ограничивающими».

5.5 Нисходящий разбор

5.5.1 LL(1)-анализ

Удаление бесплодных и недостижимых символов.

First/Follow – построение.

Рассмотрим построение $First_1/Follow_1$ множеств как частного случая построения $First/Follow$. В виду того, что мы рассматриваетм только LL(1), следовательно, мы будем предпросматривать строку только на 1 символ вперёд, и нам этого будет достаточно.

5.5.2 Комментарии к практике

На семинаре (`fltp/p10/`) были разобраны:

- Рекурсивный спуск
- LL(1)-анализ:
  - $First_1/Follow_1$ - построение
  - Построение таблицы разбора
  - Построение и тестирование анализатора
- Устранение левой рекурсии (`fltp/p10/left_recursion_elimination`)

5.6 Восходящий разбор: LR

5.6.1 LR(0)

5.6.2 SLR

Автомат – такой же, как в LR(0). Таблица отличается только тем, что reduce выполняется только там, где это имеет смысл.

5.6.3 (C)LR(1)

Канонический LR.

5.6.4 LALR

Наиболее часто реализуемый на практике подход.

https://github.com/meyerd/flex-bison-example

Пусть есть грамматика, не разбираемая из-за конфликтов сдвиг-свертка или свертка-свертка по алгоритму SLR.

В этом случае грамматика преобразуется следующим образом:

- ищется нетерминал, на котором возникла вызвавшая конфликт свертка. Обозначим его A.
- вводятся новые нетерминалы $A_1, A_2, \ldots, A_n$, по одному на каждое появление A в правых частях правил.
- везде в правых частях правил A заменяется на соответствующее A_k.

- набор правил с A в левой части повторяется n раз по разу для каждого A_k.
- правила с A в левой части удаляются, тем самым полностью удаляя A из грамматики. Для преобразованной грамматики (она порождает такой же язык, что и исходная) повторяется попытка построения SLR(1) таблицы разбора.

Действие основано на том, что Follow(A) есть объединение всех Follow(A_k). В каждом конкретном состоянии новая грамматика имеет уже не A, а одно из A_k, то есть множество Follow для данного состояния имеет меньше элементов, чем для A в исходной грамматике.

Это приводит к тому, что для LALR(1) совершается меньше попыток поставить «приведение» в клеточку таблицы разбора, что уменьшает риск возникновения конфликтов с приведениями, иногда вовсе избавляет от них и делает грамматику, не разбираемую по SLR(1), разбираемой после преобразования.

Множество Follow(A_k) называется lookahead set для A и k-той встречи в правилах, отсюда название алгоритма.

5.6.5 Комментарии к практике

На семинаре (`p11/`) было разобрано несколько примеров работы с Flex/Bison для лексического и синтаксического анализа с вычислениями по ходу рабора соответственно.

5.7 О применении синтаксического анализа на практике

Как правило, в ходе синтаксического анализа мы не желаем просто узнавать, что это программа – синтаксически корректная программа на ЯП / строка какого-то языка; мы хотим что-то скомпилировать / извлечь и тд. То есть получить ее синтаксическую структуру, и с ней уже работать.

Тем не менее, бывает интересна и сама процедура вывода, если требуется что-то делать по ходу этой процедуры. Механизм выполнения действий во время разбора называется синтаксически управляемой трансляцией, и рассматривается в следующем разделе.

Пример: напишем грамматику арифметических выражений с $+, *, (,)$

```
S -> S + S
S -> S * S
S -> (S)
S -> n
```

Данная грамматика действительно задаёт указанные выражения. Но чем она плоха с точки зрения их вычислений?[12] И чем грамматика, выписанная ниже, лучше на практике?

[12] Для ответа на этот вопрос нарисуйте дерево разбора в данной грамматике какого-нибудь выражения

```
E -> T
E -> E + T
T -> F
T -> T * F
F -> n
F ->(E)
```

По данной грамматике уже можно однозначно выполнить арифметические действия на основании полученной структуры и того, что записано в терминалах. Записи можно считать «значениями» или «атрибутами».

Но можно пойти дальше и считать, что у нетерминалов тоже есть атрибуты... С одним, частным вариантом их обработки мы уже познакомились, когда разрабатывали парсер на Bison – в нём можно было производить вычисления атрибутов и выполнять действия по-восходящей по мере разбора. О том, обстоят дела в общем случае, будет рассказано в главе «Синтаксически управляемая трансляция».

6 О выразительности языков и грамматик

6.1 Иерархия Хомского

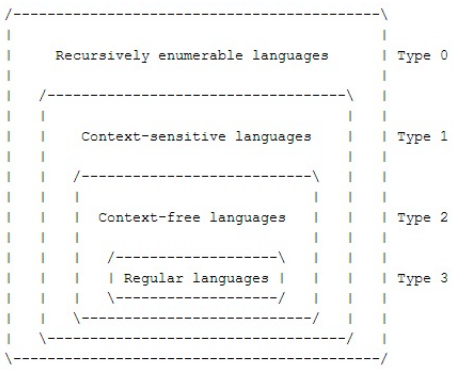


Рис. 13: Иерархия Хомского

6.2 О некоторых грамматиках промежуточных типов

6.2.1 Грамматики с контекстами

7 Синтаксически управляемая трансляция

7.1 Введение

Сначала мы работали с задачей разпознавания – принадлежит ли исследуемое слово языку – да / нет. Потом нам понадобилось строить дерево

разбора – извлекать синтаксическую структуру из слова в языке. Теперь нам и этого станет мало.

Заметим, что дерево разбора – это тоже цепочка в некотором языке (любое дерево кодируется как $root[child_1[...], clild_2[...], ...]$).

Опр. 7.1 *Трансляция - преобразование некоторой входной строчки в выходную.* $\tau : L_i \Rightarrow L_o$, $L_i \in \Sigma_i^*, L_o \in \Sigma_o^*$

Примеры:

- Вычисление арифметического выражения
- Преобразование арифметического выражения
- Любое преобразование программы в компиляторе
- Восстановление дерева по коду Прюфера

То есть, фактически, синтаксический анализ – это трансляция[13].

Зачем же урезать модели трансляции, если у нас есть ЯП общего назначения (Тьюринг-полный)? В теории, чтобы можно было гарантировать некоторые свойства транслятора.

Опр. 7.2 (Нестрогое) *Синтаксически управляемая трансляция (англ. Syntax-directed translation, SDT, СУТ) — преобразование текста в последовательность команд через добавление таких команд в правила грамматики*

В этом месте может возникнуть резонный вопрос – почему бы просто не разобрать слово, а потом обойти полученное дерево разбора, и выполнить необходимые вычисления? Действительно, зачастую в алгоритмах преобразования различных графоструктурированных данных (например, в преобразованиях компилятора) именно так и поступают. Однако, существует минимум две причины так не делать:

- Экономия памяти – как минимум, можно не хранить всё дерево разбора в памяти. Проблема – больше историческая.
- Актуальная проблема: есть логика выражений, в которой мы что-то делаем с атрибутами; если мы запишем дерево, а потом сделаем visitor по дереву, нам снова придется описать всю логику работы внутри обходчика еще раз – получается дублирование функциональности.

В то же время, СУТ позволяет и логику действий, и синтаксис описать в одном месте.

[13] В задачах обобщения на графы это не всегда так – нас могут интересовать пересечения, пустота, etc

7.2 Атрибутные грамматики

Расширим понятие грамматики атрибутами и семантическими действиями.

- Пусть каждый символ в $X \in \Sigma \cup N$ в грамматике может иметь атрибуты, которые содержат данные[14]. Это может быть $key : value$ словарь, структура или union, не принципиально. Пусть, для определённости, для X с атрибутом t обращение к атрибуту может выглядеть как $X.t$, а ко всему атрибутам $X.attr$. Грамматика, содержащая такие «расширенные» символы, называется атрибутной грамматикой.

- Дополним атрибутную грамматику $G = (\Sigma, N, P, S)$ сематническими действиями – множеством функций $A - G = (\Sigma, N, P, S, A)$, где $\forall a \in A \exists p \in P : a(\{l.attr : l \in L\}, \{r.attr : r \in R\})$, l, r – всевозможные символы в соответственно левой и правой частях правила p, вызывается тогда и только тогда, когда применяется правило p. Говорят, что такая грамматика задаёт схему трансляции. Далее будем рассматривать только КС-грамматики, поэтому $|L| = 1$.

7.2.1 Типы атрибутов

Типы атрибутов вводятся с точки зрения действия над ними семантических операций в ходе разбора.

Опр. 7.3 *Синтезированные атрибуты – атрибуты, высчитываемые из правых частей правил.*

Синтезированные атрибуты содержат информацию, подтягиваемую вверх по ходу восходящего разбора (либо возврата из рекурсивного спуска, etc), в общем, вычисляются по мере восхождения от терминалов к корню дерева разбора: в момент сворачивания по некоторому правилу, мы знаем атрибуты правой части, но ещё не знаем атрибуты левой. Они-то и «синтезируются» на основе атрибутов правой части[15].

Пример: вычисления на синтезируемых атрибутах:

```
E -> E+T  { E.val = E.val + T.val then print (E.val)}
E -> T    { E.val = T.val}
T -> T*F  { T.val = T.val * F.val}
T -> F    { T.val = F.val}
F -> Id   {F.val = id}
```

Другие примеры с синтезируемыми атрибутами были рассмотрены на паре про Flex/Bison.

Опр. 7.4 *Наследуемые атрибуты – атрибуты, высчитываемые из соседних либо родительских вершин дерева разбора.*

[14] Обычно такие атрибуты могут включать в себя тип переменной, значение выражения, и т.п.

[15] В этом месте становится понятно, почему Bison работает именно на синтезированных атрибутах, и вычисления происходят именно так

Пример: присвоение типа переменным при создании (`int a,b,c;`). Пример грамматики составить самостоятельно.

7.3 Более общая формулировка

Возьмем понятие трансляции из прошлого подраздела. Введем СУ схему как:

Опр. 7.5 *СУТ – это пятерка* (Σ, N, P, S, Π)*, где*

- Π– *выходной алфавит*
- P *– конечное множество правил вида* $A \to \alpha, \beta, \alpha \in (N \cup \Sigma)*, \beta \in (N \cup \Pi)*$,
- *вхождения нетерминалов в цепочку* β *образуют перестановку нетерминалов их цепочки* α
- *Если нетерминалы повторяются более одного раза, их различают по индексам*

В таком виде мы можем задавать, как преобразовывать цепочку. Получается, СУТ-схема задает синхронный вывод 2 цепочек.

- Если $A \to (\alpha, \beta) \in P$, то $(\gamma A^i \delta, \gamma' A^i \delta') \Rightarrow (\gamma \alpha^i \delta, \gamma' \beta^i \delta')$
- Рефлексивно-транзитивное замыкание отношения $\Rightarrow$ называется отношением выводимости $\Rightarrow *$
- Трансляцией называется множество пар $\{(\alpha, \beta) | (S, S) \Rightarrow *(\alpha, \beta), \alpha \in \Sigma^*, \beta \in \Pi^*\}$
- Схема называется простой, если в любых правилах вида $A \to (x, y)$ нетерминалы x, y встречаются в одном и том же порядке.
- Схема называется однозначной, если не существует двух правил $A \to a, b, A \to a, c$, таких, что b, c – разные символы.

Т. 7.1 *Выходная цепочка однозначной СУТ-схемы может быть сгенерирована при одностороннем выводе.*

Также существует понятие обобщенной СУТ-схемы.

Там, фактически, параллельно строятся два дерева разбора:

Для каждой внутренней вершины дерева, соответсующей нетерминалу A, с каждым A_j связвывается цепочка (трансляция) символа A_j

TODO: дописать

$$
\begin{array}{lcll}
E & \to & E+T & , \; E_1 = E_1 + T_1 \\
 & & & , \; E_2 = E_2 + T_2 \\
 & | & T & , \; E_1 = T_1, \; E_2 = T_2 \\
T & \to & T*F & , \; T_1 = T_1 * F_1 \\
 & & & , \; T_2 = T_1 * F_2 + T_2 * F_1 \\
 & | & F & , \; T_1 = F_1, \; T_2 = F_2 \\
F & \to & (E) & , \; F_1 = (E_1) \\
 & & & , \; F_2 = (E_2) \\
 & | & sin(E) & , \; F_1 = sin(E_1) \\
 & & & , \; F_2 = cos(E_1) * E_2 \\
 & | & cos(E) & , \; F_1 = cos(E_1) \\
 & & & , \; F_2 = -sin(E_1) * E_2 \\
 & | & x & , \; F_1 = x, \; F_2 = 1 \\
 & | & n & , \; F_1 = n, \; F_2 = 0
\end{array}
$$

Рис. 14: Обобщенная СУТ, позволяющая описать простейшее дифференцирование

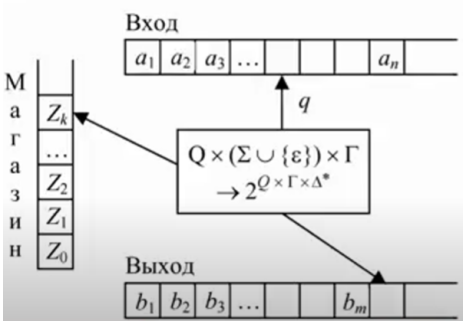


Рис. 15: МП-преобразователь

7.4 Магазинный преобразователь

Под механизмом СУТ лежит (или может лежать) формальный вычислитель – магазинный преобразователь, представляющий собою выходную ленту + МП автомат, который на каждый шаг что-нибудь на выходную ленту печатает.

Доказывается, что, так как МП-преобразователь не может что-нибудь переставить на своем стеке, то класс трансляций МП-автомата не шире класса простых СУ-схем.

Также доказывается, что по любой простой СУ-схеме можно простроить МП-преобразователь, то есть классы простых СУ-схем и МП-преобразователей совпадают.

8 Компиляторные технологии

8.1 Представление кода в виде дерева

Дерево разбора (именуемое ещё «concrete syntax tree» в книгах по компиляторам [10]) – подробно разобранная нами в разделе о грамматиках структура представления синтаксиса. В компиляторах его использование, вернее, использование его как явного представления программы, избыточно, так как некоторые синтаксические конструкции могут быть удалены или слиты воедино после синтаксического разбора.

Опр. 8.1 *Abstract syntax tree (AST) – упрощённое представление синтаксической структуры программы – помеченное ориентированное дерево, в котором внутренние вершины помечены операторами языка программирования, а листья – соответствующими операндами.*

Таким образом, листья *AST* являются пустыми операторами и представляют только переменные и константы.

AST отличается от дерева разбора тем, что в нём отсутствуют узлы и рёбра для тех синтаксических правил, которые не влияют на семантику программы. Например:

- отсутствует информация о скобках – она задается структурой дерева
- вышеупомянутое упрощение $numterm - -leafnode \rightarrow leafnode : val$

Обычно всё незначимая подцепочка просто заменяется на значение(я) из терминала(ов).

```
TranslationUnitDecl 0x58e128 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x58e9c0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x58e6c0 '__int128'
|-TypedefDecl 0x58ea30 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x58e6e0 'unsigned __int128'
|-TypedefDecl 0x58ed38 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
| `-RecordType 0x58eb10 'struct __NSConstantString_tag'
|   `-Record 0x58ea88 '__NSConstantString_tag'
|-TypedefDecl 0x58edd0 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x58ed90 'char *'
|   `-BuiltinType 0x58e1c0 'char'
|-TypedefDecl 0x58f0c8 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag [1]'
| `-ConstantArrayType 0x58f070 'struct __va_list_tag [1]' 1
|   `-RecordType 0x58eeb0 'struct __va_list_tag'
|     `-Record 0x58ee28 '__va_list_tag'
`-FunctionDecl 0x5ebe50 <1.c:1:1, line:3:1> line:1:5 f 'int (int, int)'
 (|-ParmVarDecl 0x5ebcf8 <col:7, col:11> col:11 used a 'int'
  |-ParmVarDecl 0x5ebd78 <col:14, col:18> col:18 used b 'int')
{}`-CompoundStmt 0x5ec048 <col:21, line:3:1>
    `-ReturnStmt 0x5ec038 <line:2:2, col:15>
      `-BinaryOperator 0x5ec018 <col:9, col:15> 'int' '/'
        |-ParenExpr 0x5ebfd8 <col:9, col:13> 'int'
        | `-BinaryOperator 0x5ebfb8 <col:10, col:12> 'int' '+'
        |   |-ImplicitCastExpr 0x5ebf88 <col:10> 'int' <LValueToRValue>
        |   | `-DeclRefExpr 0x5ebf48 <col:10> 'int' lvalue ParmVar 0x5ebcf8 'a' 'int'
        |   `-ImplicitCastExpr 0x5ebfa0 <col:12> 'int' <LValueToRValue>
        |     `-DeclRefExpr 0x5ebf68 <col:12> 'int' lvalue ParmVar 0x5ebd78 'b' 'int'
        `-IntegerLiteral 0x5ebff8 <col:15> 'int' 2
```

Рис. 16: Clang AST для функции целочисленного осреднения 2 целых чисел

Понятно, что структура элементов дерева укладывается в иерархии. Здесь следует отметить 2 момента по программированию:

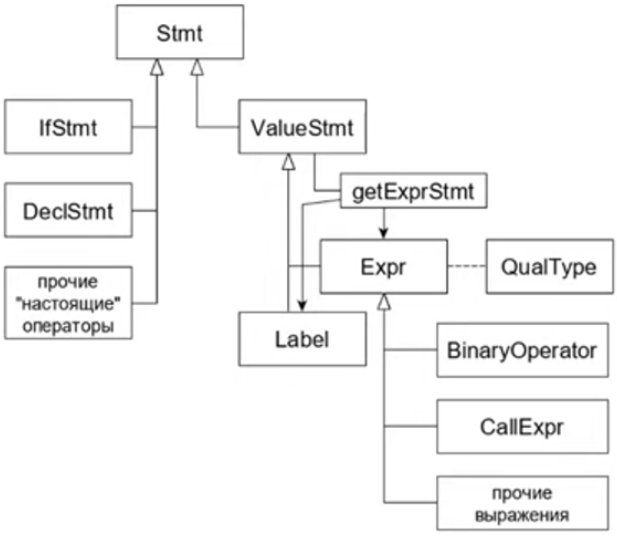


Рис. 17: Clang FE: Иерархия Stmt в Clang AST

- У 2 разных корней (ноды разных категорий) может не быть общего предка, и на практике они наследованы от разных базовых классов. То есть иерархически по классам дерево получается не деревом, а лесом. И методы для каждого дерева из леса могут быть различными.
- Представим, мы находимся в вершине дерева A, и нам нужно сделать кодогенерацию для дочерней вершины B, которая может быть типов $C_1, C_2, C_3, .., C_n$. Следовательно, в функцию $CG :: GenerateCodeA$ придётся вставить switch-case на n элементов для каждой из альтернатив C_i. Но так придётся делать для каждой из функций!

Напрашивается способ, как решить вышеуказанные моменты изящно. Для этого служит паттерн ООП «Visitor»[11], который мы рассматривать не будем. Любой модуль, использующий AST для своих целей ($ASTConsumer$) реализует в себе такой «Visitor».

8.2 Синтаксический разбор

Как правило, в компиляторах на данный момент доминируют три способа построения AST по входной программе:

- LALR(1)-парсинг + модификации парсера для специфических операций типа «составление таблицы символов», etc. Использовался ранее в GCC до версии 3.X.X, затем был переработан во вручную написанный рекурсивный спуск.

- Рекурсивный спуск, написанный вручную. Используется в Clang, современном GCC, Rust C и др[16].
- GLR-анализ – обобщенный LR-разбор, как правило, использующий GLR-парсеры общего назначения, частично доработанные. Пример – Elsa C++ Parser.

Если смотреть по соотношению в индустрии, подход №2 с рекурсивным спуском существенно доминирует. Почему так? Этому есть, как минимум, 4 причины:

1. Языки C и C++ на самом деле не контекстно-свободные
2. Но большинство конструкций при этом вообще регулярные – язык «почти регулярный», и следует ожидать длинные цепочки вывода
3. Стандарт – довольно строгий, и содержит много частных случаев
4. Частные случаи и правила для «почти регулярных» цепочек легче прописывать и отлаживать вручную, чем городить LR-грамматику.

8.3 Лексический анализ C-подобных языков

```
int 'int'        [StartOfLine]  Loc=<1.c:1:1>
identifier 'f'   [LeadingSpace] Loc=<1.c:1:5>
l_paren '('             Loc=<1.c:1:6>
int 'int'               Loc=<1.c:1:7>
identifier 'a'   [LeadingSpace] Loc=<1.c:1:11>
comma ','               Loc=<1.c:1:12>
int 'int'        [LeadingSpace] Loc=<1.c:1:14>
identifier 'b'   [LeadingSpace] Loc=<1.c:1:18>
r_paren ')'             Loc=<1.c:1:19>
l_brace '{'      [LeadingSpace] Loc=<1.c:1:21>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<1.c:2:2>
l_paren '('      [LeadingSpace] Loc=<1.c:2:9>
identifier 'a'          Loc=<1.c:2:10>
plus '+'                Loc=<1.c:2:11>
identifier 'b'          Loc=<1.c:2:12>
r_paren ')'             Loc=<1.c:2:13>
slash '/'               Loc=<1.c:2:14>
numeric_constant '2'            Loc=<1.c:2:15>
semi ';'                Loc=<1.c:2:16>
r_brace '}'      [StartOfLine]  Loc=<1.c:3:1>
eof ''          Loc=<1.c:3:2>
```

Рис. 18: Токены для функции целочисленного осреднения 2 целых чисел (получены командой clang -Xclang -dump-tokens main.c)

Проблемы:

[16]На момент проведения занятия весной 2022 г. было выяснено, что MSVC тоже использует рекурсивный спуск, о других проприетарных компиляторах автору ничего не известно.

- Как было сказано ранее для C и C++, реальные C-подобные языки синтаксически сложны, наделены большим количеством корнеркейсов, и, как правило, контекстно зависимы. Интуитивно, если представить бесконтекстный парсинг таких языков рекурсивным спуском, в лексере будут откаты – мы что-то считали из потока лексем, интерпретировали это как-то, потом подчитали ещё что-то, поняли, что ошиблись, и вернули подчитанное обратно во входной поток с целью дальнейшего анализа.

- Есть 2 типа токенов (или даже больше! В Clang существуют аннотирующие токены, которые парсер внедряет во входную последовательность с целью указать, что некоторая подпоследовательность им уже проанализирована) – Token и PreprocessingToken (для макроопределений)

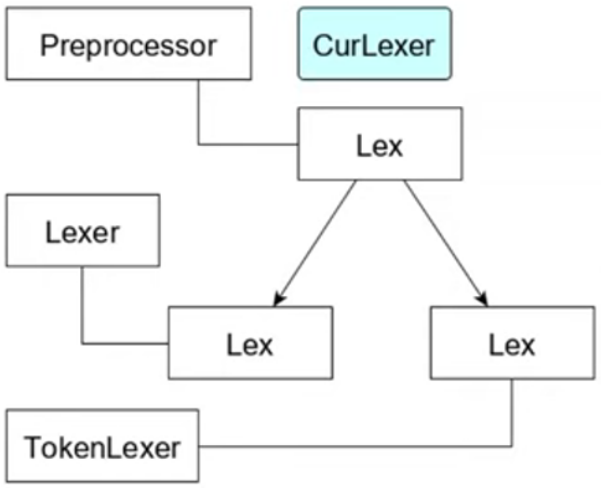


Рис. 19: Два лексера в Clang

Интуитивно бы сделать 1 лексер на 2 класса токенов. Но в некоторых компиляторах, например в Clang, все с точностью до наоборот – 2 лексера (Lexer и TokenLexer) и один класс токенов (Token)! Как результат, при обработке, например, `#include`, нужно поддерживать целый стек лексеров, какие-то из которых просто лексеры, а какие-то TokenLexer.

8.4 Взаимодействие компонент фронтенда

В учебниках по компиляторам [10] и различных курсах часто пишут, что взаимодействие компонент фронтенда выглядит как конвейер:

лексический –> синтаксический –> семантический анализ

Это крайне грубое представление о работе современных компиляторных фронтендов. В следующем подразделе мы покажем, что в деталях это совсем не так.

8.5 Clang как фронтенд

При вызове `clang -cc1` создаётся экземпляр класса `Clang::CompilerInstance` в методе `cc1_main`, в нём выставляется базовое действие, которое должен сделать фронтенд[17]. Действие активируется `Act`, после чего Clang его выполняет.

8.5.1 Иерархия базовых действий

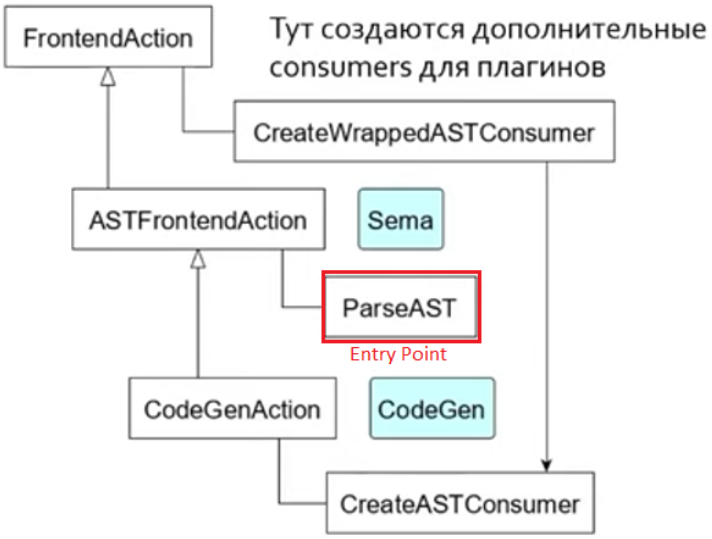


Рис. 20: Clang FE: Иерархия действий при парсинге

Как правило, мы хотим что-то выводить – у нас в качестве действий используются вызовы методов `Emit<actionname>Action`. Стоит отметить, что:

- Все такие действия наследуют от `CodeGenAction`
- CodeGenAction делает CodeGen консьюмером для AST
- ASTFrontendAction добавляет использование семантического анализа
- Точка входа при таких действиях: `ParseAST`.

[17] только одно, поэтому clang не может одновременно, например, скомпилировать программу (-emit-obj) и сдампить AST (-ast-dump)

8.5.2 Парсинг в Clang

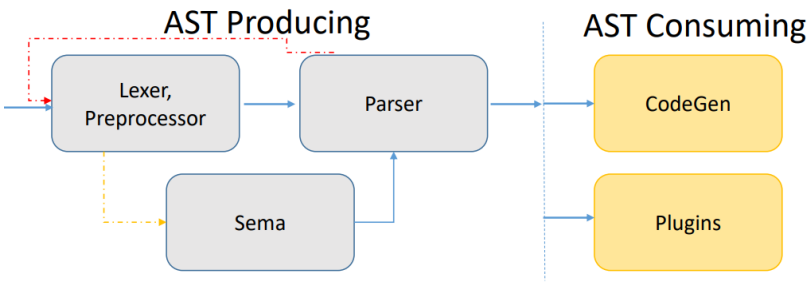


Рис. 21: Clang FE: Диаграмма зависимостей при парсинге / консьюминге AST

Главный модуль в парсинге – Parser. Его задача – подготовить AST, далее включаются все продьюсеры, потребляющие AST. Лексер – однопроходный и по умолчанию не зависит от парсера[18]. Это 2 лексера, описанных выше. Стек лексеров хранит объект класса Preprocess, он и является настоящим лексером в Clang. Семантический модуль Sema, по теории, не должен зависеть от лексера, но он от него зависит! (Ужас!).

Парсер – это рукописный рекурсивный спуск, как было сказано ранее. То есть написан набор методов `Parser:Parse<XYZ>`, по функции для каждого нетерминала. Если в ходе парсинга происходит ошибка, в принципе, предпостроенная часть AST имеет право на существование, а в месте, в котором возник затык, вставляется вершина с записью о возможной ошибке (ошибки вставляются по сопоставлению с большим enum'ом). Также есть опция `-fixit`, позволяющая исправлять простейшие синтаксические ошибки.

LALR(1) не используется, потому что C/C++ языки, для которых:

- Грамматика «ну почти» регулярная – довольно простая[19]
- При этом язык (на самом деле) контекстно-зависимый
- Потенциально мало бектрекинга
- Довольно строгий стандарт
- Много особых случаев, которые гораздо проще прописывать вручную

Проблемы:

- Невозможность раннего определения идентификаторам категории (лексер даже не пытается). Поток токенов ну оооочень простой. Парсер

[18] Это не совсем так, в виду возможности махинаций с токенами и возможностью бектрекинга

[19] По крайней мере, большинство правил

должен по грамматике догадаться по грамматике, что это. А сам язык сложный.

- Бектрекинг может быть необходим при таком подходе!

Бектрекинг в лексере: интерфейс (завёрнутый в `TentativeParsingAction`-объект)

- EnableBacktrackAtThisPos[20] – запомнить точку отката
- CommitBacktrackedTokens – забыть
- Backtrack – откатиться

Следовательно: лексер поддерживает бектрекинг, после которого подпоследовательность снова считывается, и снова разбирается парсером. Это довольно накладно по производительности, поэтому придумали ещё один тип токенов – аннотирующие. Как правило, их используют для `typename`, `scope_identifiers`. Парсер внедряет этот токен в последовательность токенов для указания, что уже понял, что это за тип и т.д. (проверка: `if TryAnnotateTypeOrScopeToken()`, установка: `setTypeAnnotation(tok,ty)`).

Мало того, парсер способен внедрять не только аннотирующие, а и вообще любые токены (см. метод `ExpectAndConsume`). Иногда это используется для обработки ошибок.

8.5.3 Семантический анализ

Утверждение Языки C/C++ (и многие другие) КС-языками не являются. Наиболее известным примером не-контекстно-свободности ЯП является конструкция

if cond then stmt1 else stmt2.

В виду того, что парсер осуществляет анализ в КС-приближении, а ЯП по сути КС-языками не являются, в компиляторы включается семантический модуль, позволяющий производить обработку контекстно-зависимых правил. Подхода к разбору бывает два:

- Непосредственно "налету по ходу разбора, парсер вызывает семантический модуль, который преобразует AST в ходе построения. Это позволяет существенно сэкономить память и не реализовывать логику проходов дважды.
- После получения результатов КС-разбора, семантический модуль трансформирует их в AST.

Clang использует первый подход, выполняя и синтаксический и семантический анализ в один проход. Семантический анализ выполняется модулем Sema по вызову из модуля Parser.

[20] Данный вызов укладывает позиции в стек: откатываясь к n-й контрольной точке, далее можно откатиться к $n-1$-й и так далее

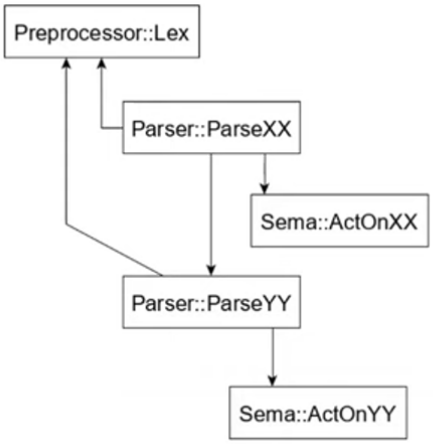


Рис. 22: Clang FE: Parser + Sema

Семантический анализ происходит по схеме:

Parse(XX) –> Sema::ActOn(XX) –> Ok? –> change AST / No? –> Error, причём AST строит именно семантический анализатор. То есть модуль Sema по сути решает 2 задачи:

- Ищет ошибки
- Строит AST

Заметим, что схема вызова семантического модуля похожа на вызов сематнического правила на атрибутах в Bison, которое строило бы AST "снизу–вверх".

8.5.4 Выводы

Кратко сформулируем выводы о внутреннем устройстве Clang-фронтенда[21]:

- Парсинг осуществляется рекурсивным спуском
- AST строит именно семантический модуль, на основании того, что разобрал парсер
- В лексере присутствует бектрекинг, лексеров – 2 типа
- Лексер, парсер и семантический модуль имеют куда более хитрые созависимости, нежели описано в классической схеме построения компиляторов.

[21] Для Clang версии 12.0.0

8.6 Обработка AST

Положим, у нас есть AST, построенное компилятором. Какие дальнейшие действия нужно предпринять, чтобы превратить его в исполняемый код?

9 Приложение

9.1 Необходимые определения из близких областей

9.1.1 Графы

В данном курсе мы будем рассматривать только конечные ориентированные помеченные графы, подразумевая под «графами» именно такие графы, если не указано противное.

Опр. 9.1 *Граф $G = (V, E, L)$, где V — конечное множество вершин, E — конечное множество рёбер, L — множество меток.*

Опр. 9.2 *Отношением достижимости на графе в смысле нашего определения называется двухместное, транзитивно-рефлексивное,*

Опр. 9.3 *Транзитивным замыканием графа называется транзитивное замыкание отношения достижимости по всему графу.*

9.2 Ссылки на контесты и дополнительные материалы

Контест 1: http://judge2.vdi.mipt.ru/cgi-bin/new-register?contest_id=220221
Контест 2: http://judge2.vdi.mipt.ru/cgi-bin/new-register?contest_id=220222

Список литературы

[1] Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. Санкт-Петербург : Вильямс, 2008.

[2] `http://neerc.ifmo.ru/wiki/index.php?title=Минимизация_ДКА,_алгоритм_за_O(n%5E2)_с_построением_пар_различимых_состояний`

[3] `http://neerc.ifmo.ru/wiki/index.php?title=Минимизация_ДКА,_алгоритм_Хопкрофта_(сложность_O(n_log_n))`

[4] `http://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Бржозовского`

[5] Истинное могущество регулярных выражений. Хабр. https://habr.com/ru/post/171667, 2013 (перевод, оригинал также доступен в Интернете).

[6] Префиксное сжатие регулярных выражений. Хабр. https://habr.com/ru/post/117177

[7] Younger, Daniel H. Recognition and parsing of context-free languages in time n3 (англ.) // Information and Computation. — Vol. 10, no. 2. — P. 189–208. — doi:10.1016/s0019-9958(67)80007-x

[8] Melski, David and T. Reps. “Interconvertibility of a class of set constraints and context-free-language reachability.” Theor. Comput. Sci. 248 (2000): 29-98.

[9] Hellings, «Path Results for Context-free Grammar Queries on Graphs», 2015.

[10] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Compilers: Principles, Techniques, and Tools.

[11] Fluentcpp: Design Patterns vs Design Principless: Visitor. https://www.fluentcpp.com/2022/02/09/design-patterns-vs-design-principles-visitor

[12] https://grammarware.net/text/2016/sppf.pdf

[13] https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html

[14] van der Sanden, L.J. Parse Forest Disambiguation, 2014. https://pure.tue.nl/ws/portalfiles/portal/46998704/784691-1.pdf