

Содержание

1 Языки и их свойства, операции над языками	2
1.1 Введение	2
1.2 Операции над языками	3
1.2.1 Операции над словами	3
1.2.2 Операции над языками как множествами	3
1.2.3 Операции над языками как множествами, содержащими последовательности	4
2 Конечные автоматы	4
2.1 Сведение НКА к ДКА	6
2.2 Минимизация ДКА	7
3 Регулярные выражения и языки	9
3.1 Регулярные выражения	9
3.2 Регулярные языки	11
3.2.1 Свойства замкнутости регулярных языков	11
3.2.2 Проверка на нерегулярность	11
3.3 Регулярные выражения на практике	11
4 Лексический анализ	11
4.1 Комментарии к практике	12
5 КС-грамматики и языки	12
5.1 Граматики как системы переписывания	12
5.2 КС-грамматики	13
5.3 Обобщение со строк на графы	14
5.3.1 СУК для графов	14
5.3.2 Алгоритм (Y. Hellings) с рабочими множествами для графов	14
5.3.3 Другие алгоритмы	14
5.3.4 Пример: КС достижимость при анализе программ	14
5.4 Восходящий разбор: LR	15
5.4.1 LR(0)	15
5.4.2 SLR	15
5.4.3 (C)LR(1)	15
5.4.4 LALR	15
5.5 О применении синтаксического анализа на практике	16
6 Синтаксически управляемая трансляция	17
6.1 Введение	17
6.2 Атрибутные грамматики	18
6.2.1 Типы атрибутов	18
6.3 Более общая формулировка	19
6.4 Магазинный преобразователь	20

7	Компиляторные технологии	21
7.1	Представление кода в виде дерева	21
7.2	Синтаксический разбор	22
7.3	Лексический анализ C-подобных языков	23
7.4	Взаимодействие компонент фронтенда	24
7.5	Clang как фронтенд	24
7.5.1	Иерархия базовых действий:	24
7.5.2	Парсинг в Clang	25
7.5.3	Семантический анализ	27
8	Приложение	27
8.1	Необходимые определения из близких областей	27
8.1.1	Графы	27

Аннотация

Это вводный абзац в начале документа.

1 Языки и их свойства, операции над языками

1.1 Введение

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: $L(\Sigma) \subset (\Sigma^*)$. 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 практическая хотелка – нам нужны алгоритмы, которые работают, по крайней мере, за полином небольшой степени и по

времени, и по памяти, так как мы хотим работать 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:

- Формальный 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 – склеивание¹ 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*

Опр. 1.3 *Взятие суффикса*

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

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.

¹устоявшегося русского термина пока нет, увы

1.2.3 Операции над языками как множествами, содержащими последовательности

Опр. 1.4 Конкатенация языков $L_1(\Sigma_1), L_2(\Sigma_2) \subset (\Sigma_1 \cup \Sigma_2)^*$ – это операция склеивания всех возможных слов языков: $L_1 \cdot L_2 = \{uv | u \in L_1, v \in L_2\}$.

Можно взять не 2, а другое 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 умножения².

Опр. 1.5 Итерация языка $L : L^* = \bigcup_{k=0}^{\infty} L^k$.

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

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 переходов³
- $q_0 \in Q$ – стартовое состояние
- $F \subset Q$ – множество финальных состояний

Существует эквивалентное определение автомата, где вместо Δ задают функцию перехода $\delta : Q \times \Sigma^* \rightarrow 2^Q$; 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 подадут 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 распознано, если мы дошли до какого-либо финального состояния и обработали

²Это 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^* .

всё слово. То есть 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 выразительность⁴. 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).

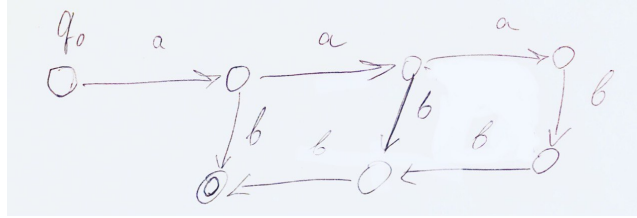


Рис. 1: КА, распознающий язык $a^n b^n, n \in [1, 3]$

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 автомата⁵. Введём отношение достижимости на конфигурациях.

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

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

Теперь 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).

⁴тем 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.

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

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

- Все переходы – однобуквенные: $\forall (\langle q_1, w \rangle \rightarrow q_2) \in \Delta : |w| = 1$
- $\forall a \in \Sigma, q \in Q |\delta(q, a)| \leq 1$, где $\delta(q, a)$ – множество состояний, достижимых из q по символу a . Задание: расписать $\delta(q, w)$ ажуратнее 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 – множества Q , который распознаёт тот же язык, что и B . Его переход в каждом состоянии-подмножестве $s \subseteq Q$ по каждому символу $a \in \Sigma$ ведёт во множество состояний, достижимых по a из некоторого состояния s .

Произведём серию упрощений НКА.

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

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

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

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 .

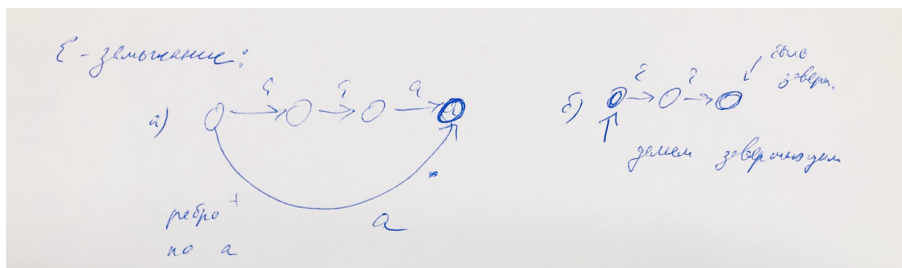


Рис. 2: Основные преобразования при построении ерс-замыкания: последовательность переходов $\varepsilon \dots \varepsilon a$ заменить на переход a (а), состояние, из которого существует переход $\varepsilon \dots \varepsilon$ в финальное состояние – обозначить как финальное (б)

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, В и С – 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.

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

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 $L(M)$*

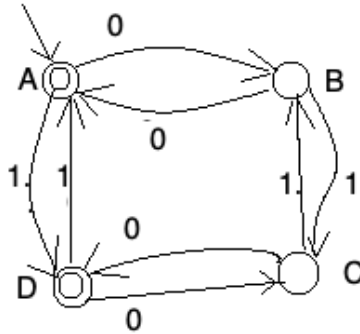


Рис. 3: ДКА, в котором есть неразличимые состояния (найдите их)

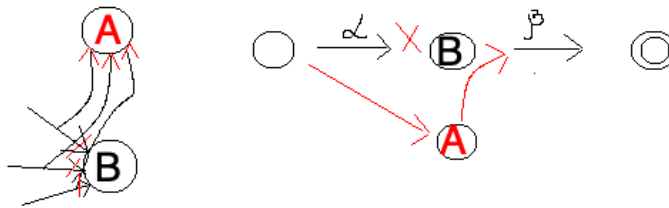


Рис. 4: Вспомогательный рисунок

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 выкинуть⁶
- 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 $O(n \log(n))$ [3].

⁶если 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

- Также существует алгоритм Бржозовского, который строит минимальный ДКА 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 $(\alpha+)$, $(\alpha+) = \alpha\alpha^*$
- 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. 4:

Так как, по определению, класс регулярных 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?...

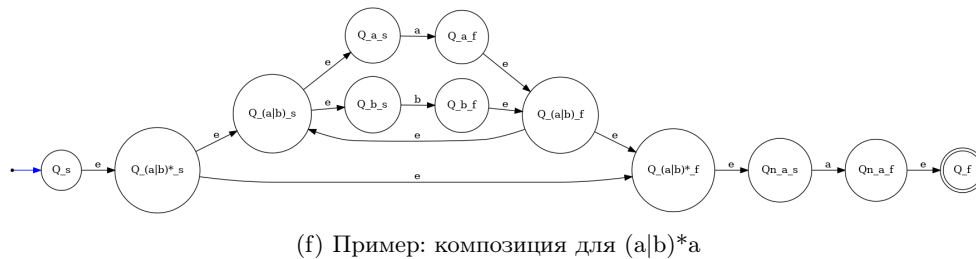
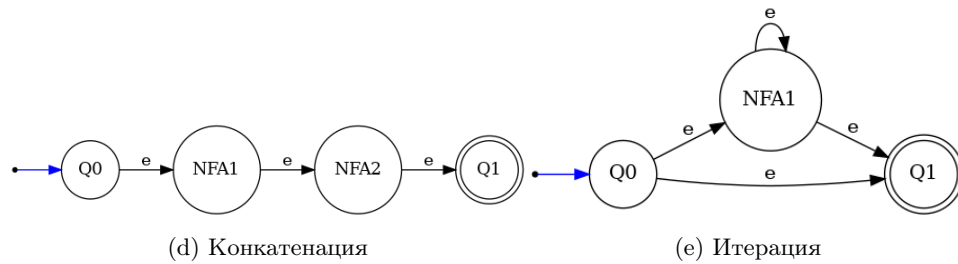
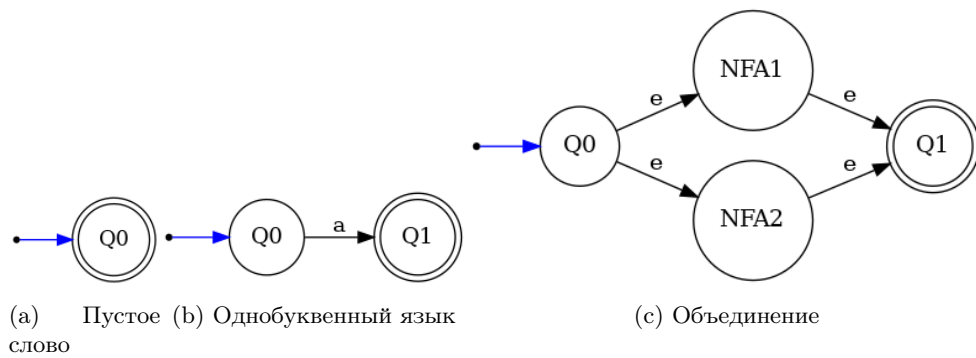


Рис. 5: Базовые автоматы построения Томпсона

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 $A \rightarrow a, A \rightarrow \gamma, A \rightarrow \epsilon$, 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.

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]): $/^{\wedge}(a(?1)?b)\$/$ 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:

```
/^{\wedge}(a(?1)?b)\$/  
/^{\wedge}(a(a(?1)?b)?b)\$/  
/^{\wedge}(a(a(a(?1)?b)?b)?b)\$/  
/^{\wedge}(a(a(a(a(?1)?b)?b)?b)?b)\$/  
...
```

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

4 Лексический анализ

Следующее приложение регулярных языков, о котором мы будем говорить – лексический анализ – 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: $int \rightarrow TYPE$ (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.*

Схема перехода от регулярки к ДКА: $regex \rightarrow NFA \rightarrow NFA_{simplified} \rightarrow DFA \rightarrow DFA_{min}$ была разобрана в разделах 1-3.

4.1 Комментарии к практике

- Примеры работы с генератором лексических анализаторов **flex** были приведены на семинаре.
- В контексте 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 под слова данного», и т.д.) можно, но не нужно писать регулярки, а лучше строить автомат по известной заранее структуре⁸.
- Существует ряд подходов к оптимизации представления регулярки, например, префиксное сжатие [6] и пр. Понятно, что в случае компиляции регулярки 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.

5 КС-грамматики и языки

5.1 Грамматики как системы переписывания

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

⁷Здесь считаем такую классификацию однозначной.

⁸Например, суффиксный бор в случае с алгоритмом Ахо-Корасик (1975)

- Σ – терминальный алфавит – алфавит определяемого языка.
- N – нетерминальный алфавит⁹ – алфавит промежуточных символов.
- Конечное множество правил R вида $\alpha \rightarrow \beta, \alpha \in \{\Sigma \cup N\}^*, \beta \in \{\Sigma \cup N\}^* \cup \{\varepsilon\}$ – 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 $w'\alpha w'' \rightarrow w'\beta w''$ состоит в замене α на подцепочку β в соответствии с правилом порождения $\alpha \rightarrow \beta$. 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 \rightarrow^* . Обозначение $\alpha \rightarrow^* \beta$ говорит о том, что цепочка β получена из цепочки α конечным числом подстановок на основе правил 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 $L = \{w | S \rightarrow^* w\}$.

Понятие регулярной грамматики уже вводилось в разделе 3. Ниже вводится понятие контекстно-свободной грамматики. Эти 2 класса грамматик являются наиболее исследованными типами грамматик иерархии Хомского (типами 3 и 2 соответственно), о которой мы будем говорить позже.

5.2 КС-грамматики

Опр. 5.2 Контекстно-свободная грамматика – кортеж $G = (\Sigma, N, R, S)$:

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

То есть, исходя из общего определения¹⁰ формальной грамматики (5.1), КС грамматика – такая грамматика, в которой каждое правило порождения

⁹В лингвистике нетерминалы называются синтаксическими категориями

¹⁰и значения

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 $\Sigma \cap N = \emptyset$.

Опр. 5.3 *Грамматика называется однозначной, если для любого порождённого по ней слова последовательность порождения – единственна.*

$$formula \tag{1}$$

5.3 Обобщение со строк на графы

5.3.1 СΥК для графов

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

Можно заметить, что СΥК производит много избыточных итераций. Можно модифицировать алгоритм, чтобы не просматривались заведомо пустые ячейки. Данная модификация была предложена Хеллингсом [1] в его именном алгоритме, но также фигурирует и в более ранних работах [2]. 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.

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 [1].

5.3.3 Другие алгоритмы

5.3.4 Пример: КС достижимость при анализе программ

Пусть $L(G)$ – язык сконкатенированных меток графа $G = (V, E, L)$, V, E, L – вершины, рёбра, метки. Если G является представлением программы p :

$$G = (V, E, L) = G(p), E \subseteq V \times L \times V,$$

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

1. Поиск паттерна. Найти все пути в G , содержащие слова из L' : $L'(G) \subseteq L(G) : \{P_G^{patterns}\} = \{P_G | w(P_G) \in L'(G)\}$.

2. Проверка на анти-паттерн: пусто ли пересечение языка графа с «языком анти-паттернов» $L''(G): \{P_G \cap L''(G)\}^? \equiv \emptyset$.
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».

5.4 Восходящий разбор: LR

5.4.1 LR(0)

5.4.2 SLR

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

5.4.3 (C)LR(1)

Канонический LR.

5.4.4 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, \dots, 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 $\text{Follow}(A)$ есть объединение всех $\text{Follow}(A_k)$. 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 $\text{Follow}(A_k)$ называется lookahead set для A и k -той встречи в правилах, отсюда название алгоритма.

5.5 О применении синтаксического анализа на практике

Как правило, в ходе синтаксического анализа мы не желаем просто узнать, 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, нам бывает интересна и сама процедура вывода, если мы хотим что-то делать по ходу этой процедуры. Это процедура называется синтаксически управляемой трансляцией, и рассматривается в следующем разделе.

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

$S \rightarrow S + S$

$S \rightarrow S * S$

$S \rightarrow (S)$

$S \rightarrow n$

Данная грамматика действительно задаёт указанные выражения. Но чем она плоха с точки зрения их вычислений?¹¹ И чем грамматика, написанная ниже, – лучше?

$E \rightarrow T$

$E \rightarrow E + T$

$T \rightarrow F$

$T \rightarrow T * F$

$F \rightarrow n$

$F \rightarrow (E)$

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

Но можно пойти дальше и считать, что и у нетерминалов тоже есть атрибуты...

¹¹ Для ответа на этот вопрос нарисуйте дерево разбора в данной грамматике какого-нибудь выражения

6 Синтаксически управляемая трансляция

6.1 Введение

Сначала мы работали с задачей распознавания – да / нет. Потом нам понадобилось строить дерево разбора. Теперь нам и этого станет мало.

Заметим, что дерево разбора – это тоже цепочка в некотором языке (любое дерево кодируется как $root[child_1[...], child_2[...], ...]$).

Опр. 6.1 *Трансляция - преобразование некоторой входной строки 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 кода в компиляторе
- Восстановление дерева по коду Прюфера

То есть, фактически, синтаксический анализ – это трансляция¹².

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

Опр. 6.2 (Нестрогое) *Синтаксически управляемая трансляция (англ. *Syntax-directed translation, SDT, CYT*) – преобразование текста в последовательность команд через добавление таких команд в правила грамматики*

А почему бы не разобрать слово, а потом обойти полученное дерево разбора, и посчитать? Действительно, зачастую в алгоритмах преобразования различных графоструктурированных данных (например, в оптимизационных проходах компилятора) именно так и поступают. Однако, существует минимум 2 мотивации так не делать:

- Экономия памяти – как минимум, мы можем не хранить дерево разбора 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

А СУТ позволяет логику и синтаксис описать в одном месте.

¹²В задачах обобщения на графы это не всегда так – нас могут интересовать пересечения, пустота, etc

6.2 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 данные¹³. 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 $G = (\Sigma, N, P, S)$ семантическими 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 $|L| = 1$.

6.2.1 Типы атрибутов

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

Опр. 6.3 *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 части¹⁴.

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

¹³Обычно такие атрибуты могут включать в себя тип переменной, значение выражения, и т.п.

¹⁴В 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.

6.3 Более общая формулировка

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

Опр. 6.5 *СУТ – это пятерка (Σ, N, P, S, Π) , где*

- Π – выходной алфавит
- P – конечное множество правил вида $A \rightarrow \alpha, \beta, \alpha \in (N \cup \Sigma)^*, \beta \in (N \cup \Pi)^*$,
- вхождения нетерминалов в цепочку β образуют перестановку нетерминалов из 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 $A \rightarrow (\alpha, \beta) \in P$, 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 $A \rightarrow (x, y)$ нетерминалы x, y встречаются в одном и том же порядке.
- Схема называется однозначной, если не существует двух правил $A \rightarrow a, b, A \rightarrow a, c$, таких, что b, c – разные символы.

Т. 6.1 *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, соответствующей нетерминалу A , с каждым A_j связывается цепочка (трансляция) символа A_j

TODO: дописать

$E \rightarrow E + T$, $E_1 = E_1 + T_1$
	, $E_2 = E_2 + T_2$
T	, $E_1 = T_1, E_2 = T_2$
$T \rightarrow T * F$, $T_1 = T_1 * F_1$
	, $T_2 = T_1 * F_2 + T_2 * F_1$
F	, $T_1 = F_1, T_2 = F_2$
$F \rightarrow (E)$, $F_1 = (E_1)$
	, $F_2 = (E_2)$
$\sin(E)$, $F_1 = \sin(E_1)$
	, $F_2 = \cos(E_1) * E_2$
$\cos(E)$, $F_1 = \cos(E_1)$
	, $F_2 = -\sin(E_1) * E_2$
x	, $F_1 = x, F_2 = 1$
n	, $F_1 = n, F_2 = 0$

Рис. 6: Обобщенная СУТ, позволяющая описать простейшее дифференцирование

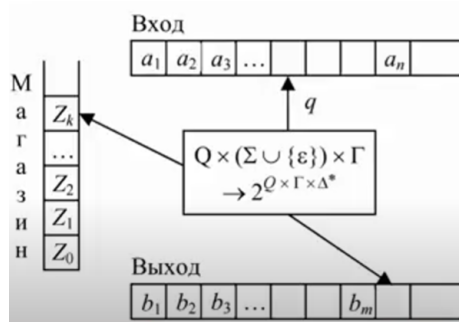


Рис. 7: МП-преобразователь

6.4 Магазинный преобразователь

Под этим лежит (может лежать) формальный вычислитель – магазинный преобразователь – выходная лента + МП автомат, который на каждый шаг на выходную ленту что-нибудь 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-трансляций.

Также доказывается, что по любой простой СУ-схеме можно построить МП-преобразователь, то есть классы совпадают.

7 Компиляторные технологии

7.1 Представление кода в виде дерева

Дерево разбора (именуемое ещё «concrete syntax tree» в книгах по компиляторам [7]) – подробно разобранный нами в разделе 5 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. 7.1 *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>
- TypedDefDecl 0x58e9c0 <invalid sloc> <invalid sloc> implicit __int128_t '__int128'
  - BuiltinType 0x58e6c0 '__int128'
- TypedDefDecl 0x58ea30 <invalid sloc> <invalid sloc> implicit __uint128_t 'unsigned __int128'
  - BuiltinType 0x58e6e0 'unsigned __int128'
- TypedDefDecl 0x58ed38 <invalid sloc> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
  - RecordType 0x58eb10 'struct __NSConstantString_tag'
    - Record 0x58ea88 '__NSConstantString_tag'
- TypedDefDecl 0x58edd0 <invalid sloc> <invalid sloc> implicit __builtin_ms_va_list 'char *'
  - PointerType 0x58ed90 'char *'
    - BuiltinType 0x58e1c0 'char'
- TypedDefDecl 0x58f0c8 <invalid sloc> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag [1]'
  - ConstantArrayType 0x58f070 'struct __va_list_tag [1]' 1
    - RecordType 0x58eb00 'struct __va_list_tag'
      - Record 0x58ee28 '__va_list_tag'
- FunctionDecl 0x58e50 <1.c:1:1, line:3:1> line:1:5 f 'int (int, int)'
  (|- ParmVarDecl 0x58ebcf8 <col:7, col:11> col:11 used a 'int'
    |- ParmVarDecl 0x58ebd8 <col:14, col:18> col:18 used b 'int')
  |- CompoundStmt 0x58ec948 <col:21, line:3:1>
    - ReturnStmt 0x58ec038 <line:2:2, col:15>
      - BinaryOperator 0x58ec018 <col:9, col:15> 'int' '/'
        - ParenExpr 0x58ebfd8 <col:9, col:13> 'int'
          - BinaryOperator 0x58ebfb8 <col:10, col:12> 'int' '+'
            - ImplicitCastExpr 0x58ebf88 <col:10> 'int' <LValueToRValue>
              - DeclRefExpr 0x58ebf48 <col:10> 'int' lvalue ParmVar 0x58ebcf8 'a' 'int'
            - ImplicitCastExpr 0x58ebfa0 <col:12> 'int' <LValueToRValue>
              - DeclRefExpr 0x58ebf68 <col:12> 'int' lvalue ParmVar 0x58ebd8 'b' 'int'
            - IntegerLiteral 0x58ebff8 <col:15> 'int' 2
```

Рис. 8: Clang AST для функции целочисленного осреднения 2 целых чисел

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

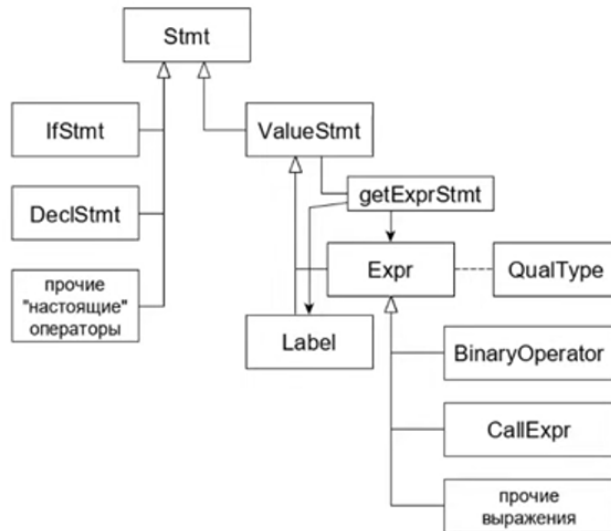


Рис. 9: Clang FE: Иерархия Stmt в Clang AST

- 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 $C_1, C_2, C_3, \dots, C_n$. 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» [8], который мы рассматривать не будем. Любой модуль, использующий AST для своих целей (AST Consumer) реализует в себе такой «Visitor».

7.2 Синтаксический разбор

Как правило, в компиляторах на данный момент доминируют 3 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 др¹⁵.
- 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:

7.3 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. 10: Токены для функции целочисленного осреднения 2 целых чисел

Проблемы:

- Есть 2 типа токенов (или даже больше!) – Token и PreprocessingToken (для макроопределений)

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.

¹⁵На момент проведения занятия весной 2022 г. было выяснено, что MSVC тоже использует рекурсивный спуск, о других проприетарных компиляторах автору ничего не известно.

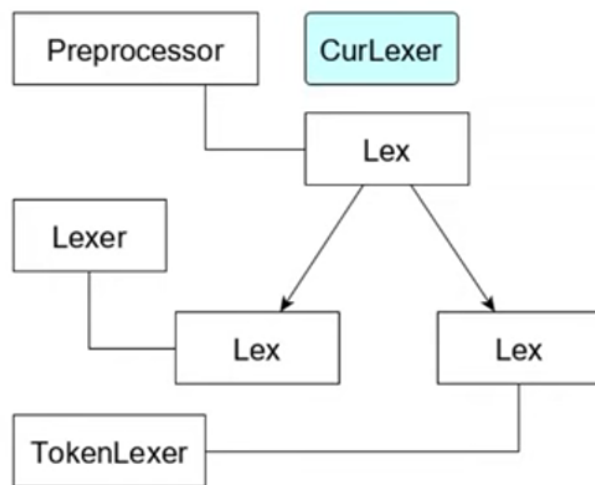


Рис. 11: Два лексера в Clang

7.4 Взаимодействие компонент фронтенда

В учебниках по компиляторам [7] 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.

7.5 Clang как фронтенд

При вызове `clang -cc1` создаётся экземпляр класса `Clang::CompilerInstance` в методе `cc1_main`, в нём выставляется базовое действие, которое должен сделать фронтенд¹⁶. Действие активируется `Act`, после чего Clang его выполняет.

7.5.1 Иерархия базовых действий:

Как правило, мы хотим что-то выводить – у нас `Emit<actionname>Action`. Стоит отметить, что:

- Все такие действия наследуют от `CodeGenAction`
- `CodeGenAction` делает `CodeGen` консьюмером для `AST`

¹⁶только одно, поэтому clang не может одновременно, например, скомпилировать программу (`-emit-obj`) и сдать AST (`-ast-dump`)

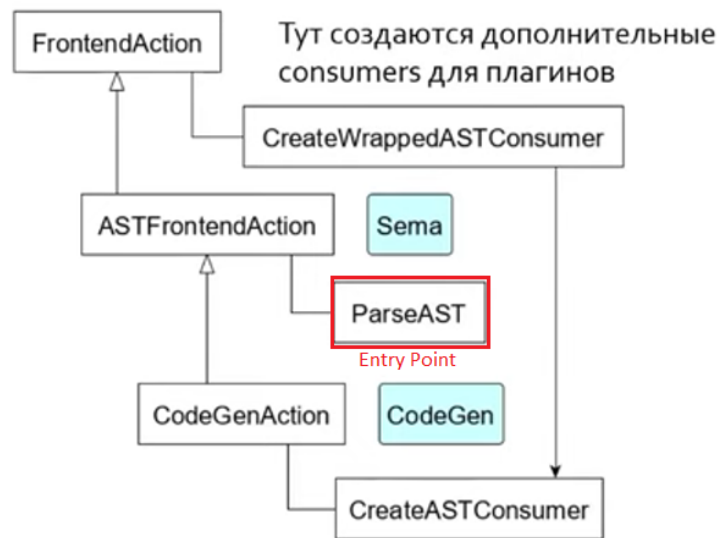


Рис. 12: Clang FE: Иерархия действий при парсинге

- ASTFrontendAction добавляет использование семантического анализа
- Точка входа при таких действиях: ParseAST.

7.5.2 Парсинг в Clang

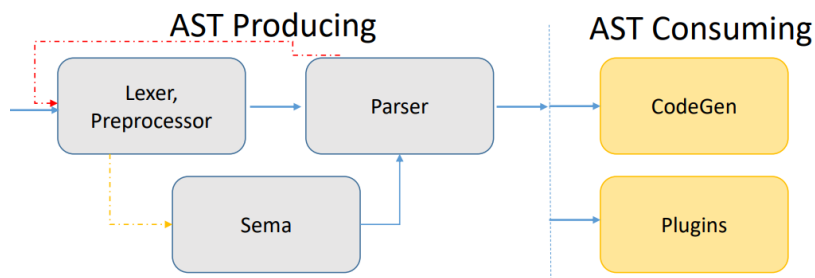


Рис. 13: Clang FE: Диаграмма зависимостей при парсинге / консьюминге AST

Главный модуль в парсинге – Parser. Его задача – подготовить AST, далее включаются все продьюсеры, потребляющие AST. Лексер – однопоточный и по умолчанию не зависит от парсера¹⁷. Это 2 лексера, описанных

¹⁷Это не совсем так, в виду возможности махинаций с токенами и возможностью бектрекинга

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 простая¹⁸
- 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 ооочень простой. Парсер 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`¹⁹ – 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

¹⁸По крайней мере, большинство правил

¹⁹Данный вызов укладывает позиции в стек: откатываясь к n -й контрольной точке, далее можно откатиться к $n - 1$ -й и так далее

понял, что это за тип и т.д. (проверка: `if TryAnnotateTypeOrScopeToken()`, установка: `setTypeAnnotation(tok,ty)`).

Мало того, парсер может вставлять не только такие токены, а и вообще любые. `ExpectAndConsume`. Иногда это используется для обработки ошибок.

7.5.3 Семантический анализ

Семантический анализ выполняется модулем `Sema` по вызову из модуля `Parser`.

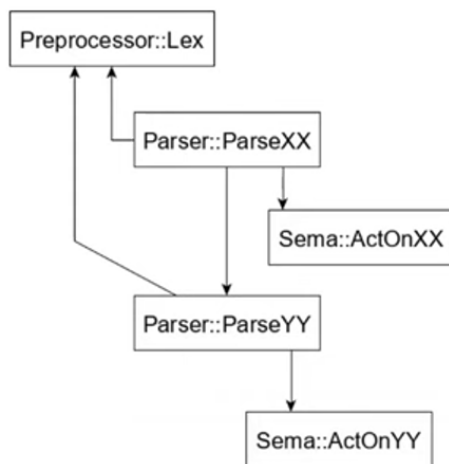


Рис. 14: Clang FE: Parser + Sema

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

8 Приложение

8.1 Необходимые определения из близких областей

8.1.1 Графы

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

Опр. 8.1 Граф $G = (V, E, L)$, где V — конечное множество вершин, E — конечное множество рёбер, L

Опр. 8.2 Отношением достижимости на графе в смысле нашего определения называется двуместное,

Опр. 8.3 Транзитивным замыканием графа называется транзитивное замыкание отношения достижимости по всему графу.

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_0\(n%5E2\)_с_построением_пар_различимых_состояний](http://neerc.ifmo.ru/wiki/index.php?title=Минимизация_ДКА,_алгоритм_за_0(n%5E2)_с_построением_пар_различимых_состояний)
- [3] [http://neerc.ifmo.ru/wiki/index.php?title=Минимизация_ДКА,_алгоритм_Хопкрофта_\(сложность_0\(n_log_n\)\)](http://neerc.ifmo.ru/wiki/index.php?title=Минимизация_ДКА,_алгоритм_Хопкрофта_(сложность_0(n_log_n)))
- [4] http://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Бржозовского
- [5] <https://habr.com/ru/post/171667>, 2013 (перевод, оригинал тоже гуглится).
- [6] Префиксное сжатие регулярных выражений. Хабр. <https://habr.com/ru/post/117177/>
- [7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Compilers: Principles, Techniques, and Tools.
- [8] Fluentcpp: Design Patterns vs Design Principless: Visitor. <https://www.fluentcpp.com/2022/02/09/design-patterns-vs-design-principles-visitor/>