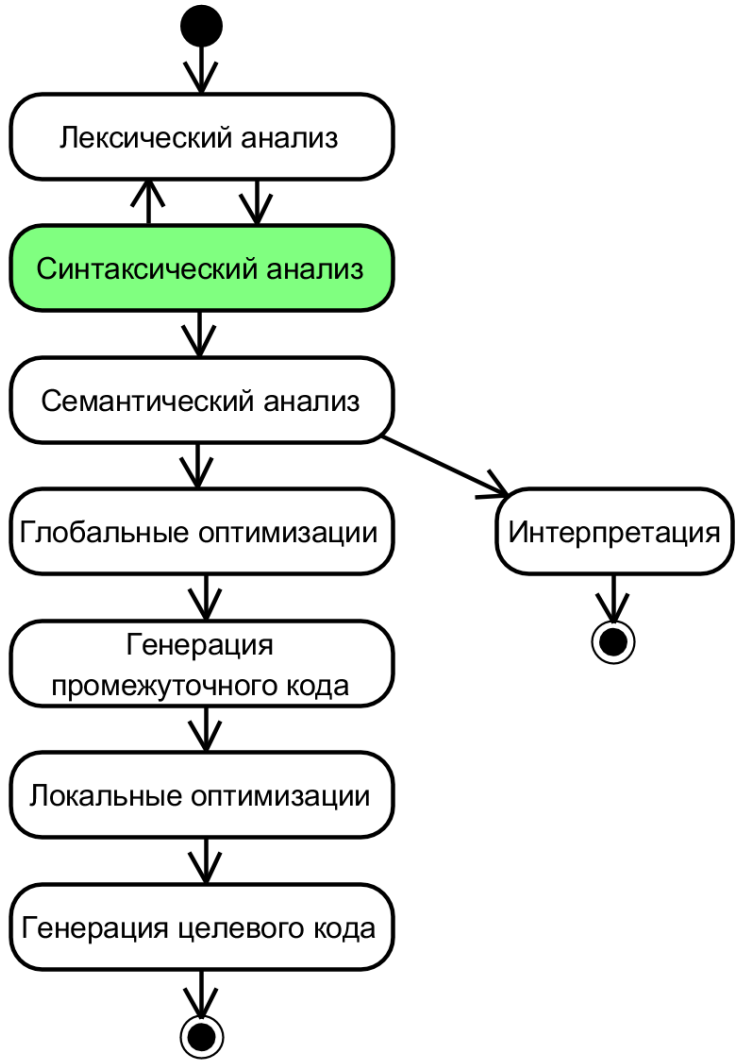
Синтаксический анализ

Часть 1: синтаксический анализ вообще

Юрий Литвинов
y.litvinov@spbu.ru

17.04.2025

Фазы компиляции

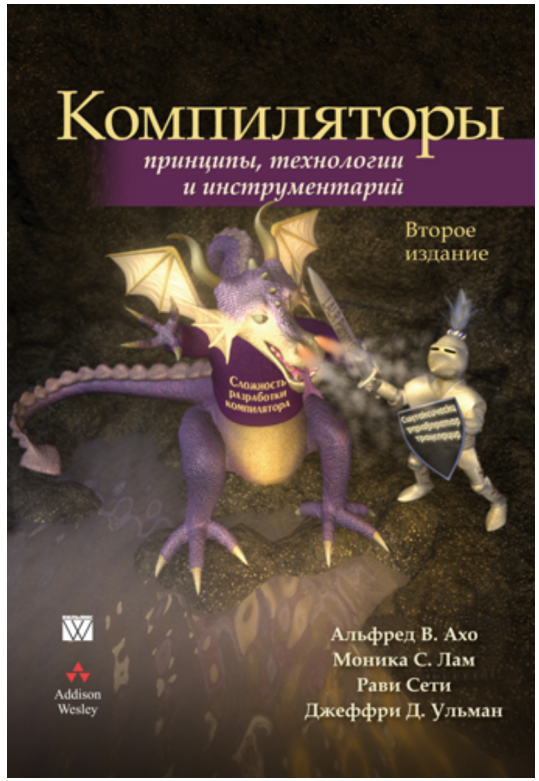


Книжка

Must read

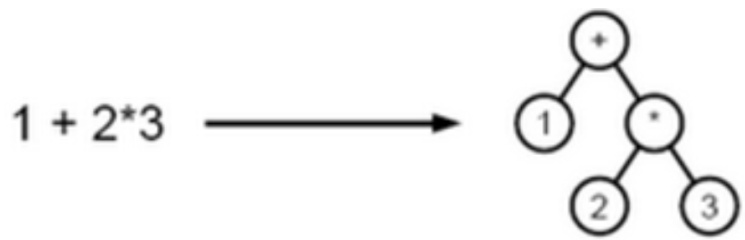
А. Ахо, Р. Сети, Дж. Ульман, М. Лам. Компиляторы. Принципы, технологии, инструменты.

- Так же известна как “Книга дракона” (“Dragonbook”)



Синтаксический анализ

- Анализ последовательности токенов с целью выяснить синтаксическую структуру
  - Сопоставление с формальной грамматикой
- Строит структуру данных, представляющую разобранный по синтаксическим правилам документ
  - Чаще всего, абстрактное синтаксическое дерево (Abstract Syntax Tree, AST)
  - Бывает ещё дерево разбора (Parse tree) — содержит все токены из входной строки, в явном виде обычно не строится



Другие задачи синтаксического анализа

- Диагностика ошибок
- Восстановление после ошибок
  - Режим паники
  - Коррекция
  - Грамматические правила, обнаруживающие ошибки
    - “Предсказание ошибок”
- Привязка — определение для каждой синтаксической конструкции её места в коде

Формальные грамматики

- Терминал — символ входной строки для синтаксического анализатора (токен)
  - Для лексического анализа входная строка состоит из букв, для синтаксического — из токенов
- Нетерминал — объект, представляющий сложную синтаксическую конструкцию
- Грамматика, формально: (Σ, N, P, S), где
  - Σ — множество терминалов
  - N — множество (алфавит) нетерминальных символов
  - P — продукции, функции вида «цепочка символов» $\rightarrow$ «цепочка символов», где слева в цепочке есть хотя бы один нетерминал
    - $P : (\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$
  - S — стартовый символ, $S \in N$

Пример грамматики

E ::= E + E
| E - E
| -E
| (E)
| NUMBER

NUMBER ::= 1 | 2 | 3 | ... | 9

Иерархия Хомского

- Регулярные языки (языки типа 3) — задаются регулярными выражениями, разбираются конечными автоматами
- Контекстно-свободные грамматики — грамматики, у которых слева в продукциях может быть только один символ (нетерминал)
  - Пример с предыдущего слайда — КС-грамматика
  - Разбираются стековыми автоматами (например, рекурсивным спуском)
- Контекстно-зависимые грамматики — в левой части может быть нетерминал и “контекст”, нетерминал раскрывается в правой части
  - Разбираются линейно ограниченными недетерминированными машинами Тьюринга (то есть, всё плохо)
- Языки типа 0 — грамматики без ограничений на вид продукций
  - Разбираются машинами Тьюринга (то есть всё очень плохо)

В реальной жизни

- Регулярные языки — регэкспы, весь лексический анализ
  - Не умеют считать, поэтому грамматики вида $a^n b^n$ (скобочные последовательности) им не под силу
  - Не могут в иерархические структуры, никогда не парсите регэкспами HTML
- Контекстно-свободные грамматики — грамматики большинства современных языков программирования
  - Не могут в анализ типов
- Контекстно-зависимые грамматики — грамматика C++ и некоторых неаккуратных мест в других языках
  - Пример: A<B> c; — либо **class** **A**<T> {}, либо **int** A; **int** B; **int** c;
- Языки типа 0 — естественные языки (да, их тоже анализируют грамматиками, и вообще, Хомский был лингвистом)

Вывод в грамматике

- Формально, если есть грамматика $G = (\Sigma, N, P, S)$, то вывод, $\Rightarrow_G$ — бинарное отношение на строках
  - $x \Rightarrow_G y \iff \exists u, v, p, q \in (\Sigma \cup N)^* : (x = upv) \wedge (p \to q \in P) \wedge (y = uqv)$
  - Неформально, шаг вывода — применение одной из продукций
- $\Rightarrow_G^*$ — рефлексивное транзитивное замыкание $\Rightarrow_G$
  - $x \Rightarrow_G^* y$ — существует конечная последовательность применений продукций грамматики, которая по x делает y
  - Говорят, «y выводится из x»
- *Порождение* — последовательность шагов вывода
- $L(G)$ — $\{w \in \Sigma^* | S \Rightarrow_G^* w\}$ — язык, порождаемый грамматикой G

Пример

Грамматика:

E ::= E + E
| E * E
| -E
| (E)
| id

Входная строка: -(id + id)

Порождения:

- Левое: E => -E => -(E) => -(E + E) => -(id + E) => -(id + id)
- Правое: E => -E => -(E) => -(E + E) => -(E + id) => -(id + id)

Левая рекурсия

Проблема:

$$A \to Aa \mid b$$

Пример:

$$E \to E + T \mid T$$

$$T \to T * F \mid F$$

$$F \to (E) \mid id$$

Решение:

$$A \to bA'$$

$$A' \to aA' \mid \epsilon$$

Пример:

$$E \to TE'$$

$$E' \to +TE' \mid \epsilon$$

$$T \to FT'$$

$$T' \to *FT' \mid \epsilon$$

$$F \to (E) \mid id$$

Неоднозначность

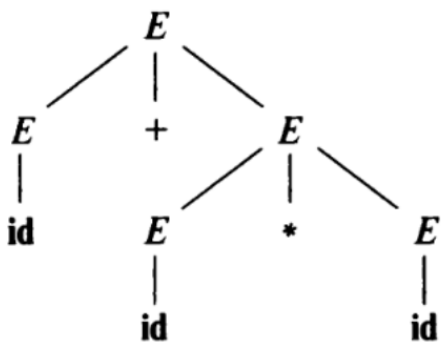
Строка: id + id * id

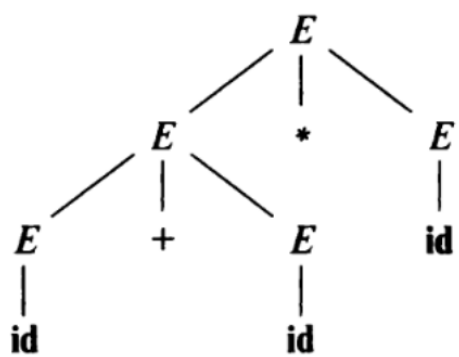
Грамматика (как была): E ::= E + E | E * E | -E | (E) | id

Вывод:

$$E \Rightarrow E + E$$
$$\Rightarrow id + E$$
$$\Rightarrow id + E * E$$
$$\Rightarrow id + id * E$$
$$\Rightarrow id + id * id$$

$$E \Rightarrow E * E$$
$$\Rightarrow E + E * E$$
$$\Rightarrow id + E * E$$
$$\Rightarrow id + id * E$$
$$\Rightarrow id + id * id$$



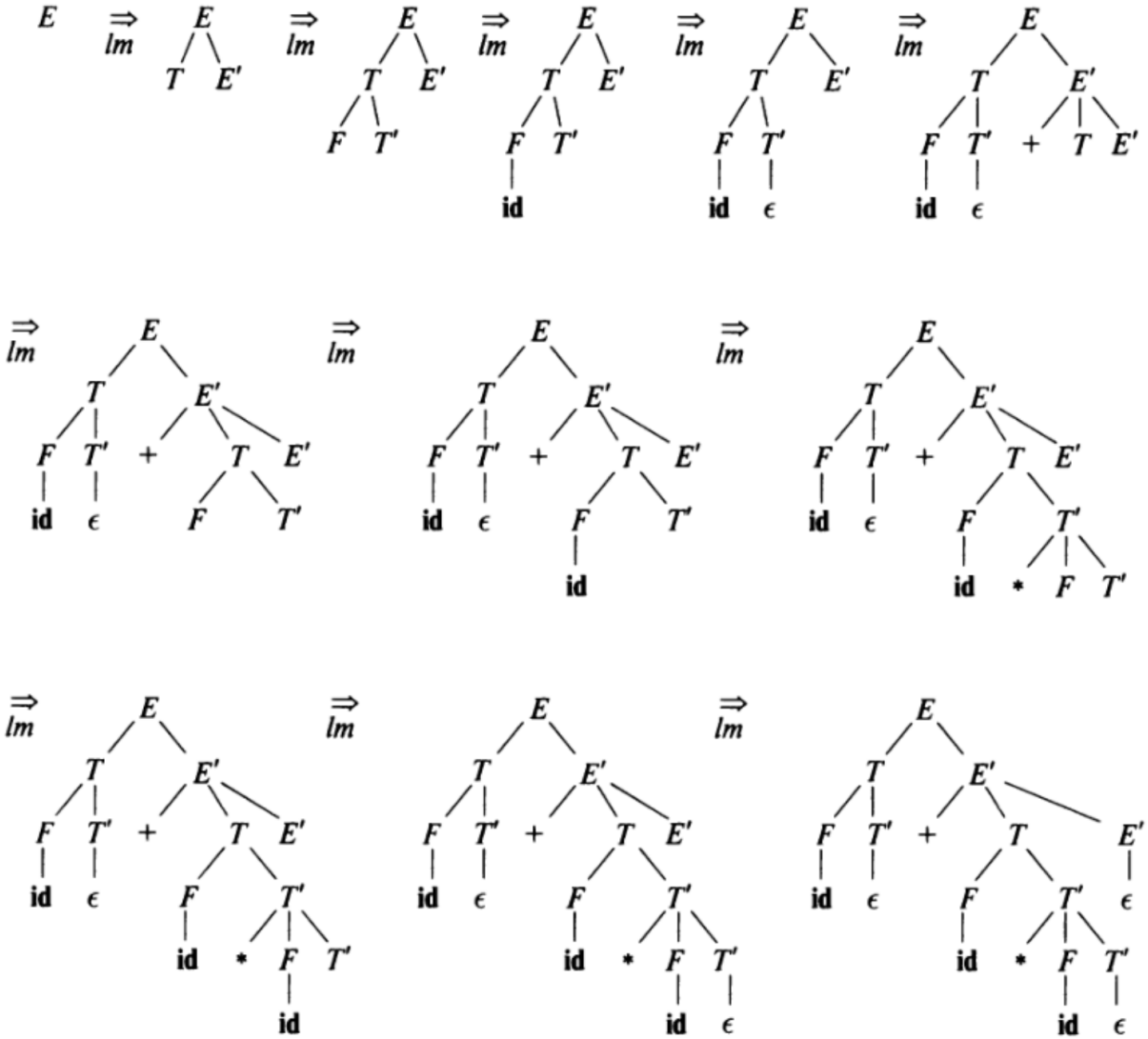


Алгоритмы разбора

- Нисходящий разбор — начинаем со стартового нетерминала, пытаемся построить входную строку
  - Рекурсивный спуск
  - LL-анализ
- Восходящий разбор — пытаемся найти во входной строке последовательность терминалов и нетерминалов и свернуть её в нетерминал
  - LR-анализ

Пример

Нисходящий разбор с построением левого порождения



FIRST(α) и FOLLOW(α)

Пусть α — строка из нетерминалов и терминалов

- FIRST(α) — множество всех терминалов, с которых может начинаться α
  - Считается рекурсивно, раскрытием нетерминальных символов
- FOLLOW(α) — множество всех терминалов, которые могут стоять за α в выводе в грамматике G
  - Считается через FIRST во всех цепочках выводов, в которых может встречаться α
  - ϵ-продукции требуют особого внимания

Зачем:

- FIRST позволяет выбрать из альтернативных продукций
- FOLLOW — чтобы выбрать между ϵ-продукцией и какой-то другой

Рекурсивный спуск

- По одной функции на нетерминал
- Просмотр строки слева направо

Выбираем продукцию $A \to X_1 X_2 \ldots X_k$;
for i от 1 до k **do**
  if X_i – нетерминал **then**
    Вызов функции $X_i()$;
  else if X_i равно текущему символу a **then**
    Переходим к следующему символу;
  else
    Обнаружена ошибка;
  end
end

BNF

Форма Бэкуса-Наура

- В угловых скобках — нетерминал (<literal>)
- ::= — определение (<brackets> ::= ’(’ | ’)’)
- | — альтернатива

Пример: <expr> ::= <term>|<expr><addop><term>

Пример

BNF, записанная в синтаксисе BNF

```
<syntax> ::= <rule> | <rule> <syntax>
<rule> ::= <opt-whitespace> "<" <rule-name> ">" <opt-whitespace>
       "::=" <opt-whitespace> <expression> <line-end>
<opt-whitespace> ::= " " <opt-whitespace> | ""
<expression> ::= <list> | <list> <opt-whitespace> "|" <opt-whitespace> <expression>
<line-end> ::= <opt-whitespace> <EOL> | <line-end> <line-end>
<list> ::= <term> | <term> <opt-whitespace> <list>
<term> ::= <literal> | "<" <rule-name> ">"
<literal> ::= '"' <text1> '"' | "'" <text2> "'"
<text1> ::= "" | <character1> <text1>
<text2> ::= '' | <character2> <text2>
<character> ::= <letter> | <digit> | <symbol>
<character1> ::= <character> | "'"
<character2> ::= <character> | '"'
<rule-name> ::= <letter> | <rule-name> <rule-char>
<rule-char> ::= <letter> | <digit> | "-"
```

Расширенная форма Бэкуса-Наура

- { } — 0 или более повторений
- [ ] — 0 или 1 раз (опционально)
- ( ) — группировка
- , — конкатенация
- Вариантов синтаксиса EBNF больше, чем звёзд на небе

Пример

EBNF, записанная в синтаксисе EBNF

```
character = letter | digit | symbol | "_" ;
identifier = letter , { letter | digit | "_" } ;
terminal = "'" , character , { character } , "'"
         | '"' , character , { character } , '"' ;
lhs = identifier ;
rhs = identifier
    | terminal
    | "[" , rhs , "]"
    | "{" , rhs , "}"
    | "(" , rhs , ")"
    | rhs , "|" , rhs
    | rhs , "," , rhs ;
rule = lhs , "=" , rhs , ";" ;
grammar = { rule } ;
```

Парсер-комбинаторы

- Основная идея — а давайте рассматривать парсер как композицию более простых парсеров
  - Определим примитивные парсеры и комбинаторы, строящие парсеры по парсерам
- По сути, удобная запись рекурсивного спуска
  - Не всегда, иногда используются “настоящие” преобразования грамматик
    - Например, Meerkat
- Пример — FParsec
  - Порт известной библиотеки Parsec (Haskell)
  - Рассмотрим http://www.quanttec.com/fparsec/tutorial.html