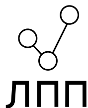


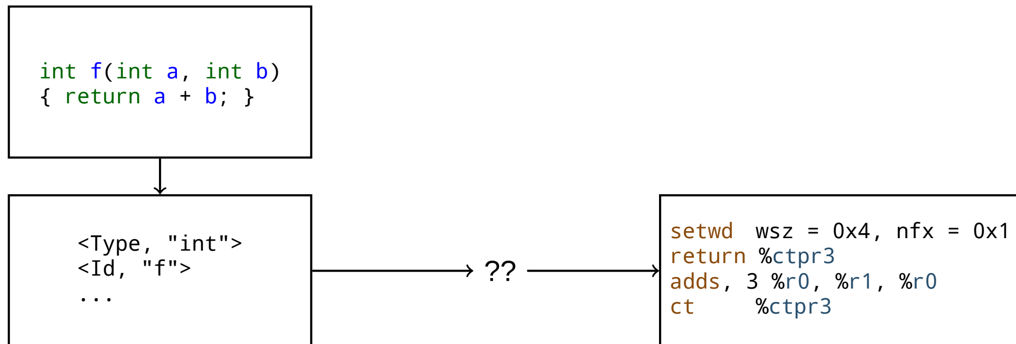
Проектирование компиляторов

Лекция 3.
Синтаксический анализ.

Маркин А. Л.
master@lab-pp.ru

2023

Компиляция программы



Проблема

Как составить регулярное выражение, распознающее цепочки вложенных конструкций `if-else`?

```
if( cond )
{
   if( cond )
   {
      ...
   } else
   {
      ...
   }
} else
{
...
}
```

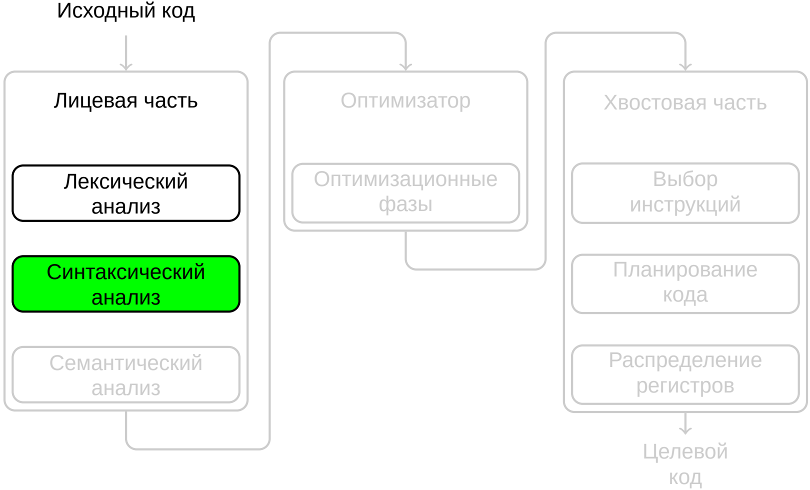
Проблема

Проблемы регулярных языков:

1. Не умеют распознавать рекурсивные языки
2. Не обеспечивают информации о структуре программы

```
if( cond )
{
   if( cond )
   {
      ...
   } else
   {
      ...
   }
} else
{
...
}
```

Компиляция программы



Контекстно-свободные грамматики

Языки программирования имеют рекурсивную структуру:

```
if( EXPR ){ STATEMENT }
else { STATEMENT }
```

`STATEMENT` может быть другой конструкцией `if-else`

Для описания структуры подобных языков используется механизм **КСГ — контекстно-свободных грамматик** (CFG — Context Free Grammar).

Контекстно-свободные грамматики

Грамматика G задаётся четвёркой $G = (V_T, V_N, P, S)$, где

V_T — алфавит, символы которого называют **терминальными**

V_N — алфавит, символы которого называют **нетерминальными**.
$V_T \cap V_N = \varnothing$

P — множество **продукций** (правил вывода)

S — символ предложения (аксиома грамматики)

$L(G)$ — язык L, выводимый грамматикой G представляет из себя все слова, выводимые из стартового состояния.

Контекстно-свободные грамматики

Пример контекстно-свободной грамматики для языка $\{a^n b^n\}$:

$V_T = \{a, b\}$ — алфавит терминальных символов

$V_N = \{S\}$ — алфавит нетерминальных символов

$$P = \begin{cases} S & \to aSb \\ S & \to \varepsilon \end{cases}$$ — продукции

S — аксиома

Контекстно-свободные грамматики

Как работает контекстно-свободная грамматика:

1. Сначала создаём строку, содержащую только символ аксиомы S
2. Заменяем любой нетерминал V_N в соответствии с какой-либо продукцией: $X \to Y_1 \dots Y_n$
3. Повторяем пункт 2 до тех пор пока не получим строку, состоящую только из терминальных символов

Контекстно-свободные грамматики

Пример работы регулярной грамматики для языка $\{a^n b^n\}$:

$V_T = \{a, b\}$ — алфавит терминальных символов

$V_N = \{S\}$ — алфавит нетерминальных символов

$$P = \begin{cases} S & \to aSb \\ S & \to \varepsilon \end{cases}$$ — продукции

S — аксиома

Процесс выведения языка при помощи заданной грамматики:

Строка	Продукция	Результат
S	$S \to aSb$	aSb
S	$S \to \varepsilon$	ε
aSb	$S \to \varepsilon$	ab
aSb	$S \to aSb$	$aaSbb$

$\dots$

Контекстно-свободные грамматики

Такой способ описания языка называется **порождающим**.

В КСГ не существует продукций, способных заменять терминальные символы.

Терминальные символы по сути являются токенами языка.

Классификация грамматик и языков

Грамматики и формальные языки можно разделить на несколько типов.

Тип 3: регулярные грамматики

- Имеют правила вида $A \to aB$ или $A \to a$ (где a - терминал; A, B - нетерминалы)
- Описываются конечным автоматом
- Подходят для разбора нерекурсивных языков
- Применяются для поиска в тексте по шаблону или разбора языков ассемблера и макроопределений

Классификация грамматик и языков

Тип 2: контекстно-свободные грамматики

- Имеют правила вида $A \rightarrow \alpha$ (где α - строка терминалов и/или нетерминалов; A нетерминал)
- Описываются автоматом с магазинной памятью
- Подходят и применяются для разбора большинства языков программирования

Классификация грамматик и языков

Тип 1: контекстно-зависимые грамматики

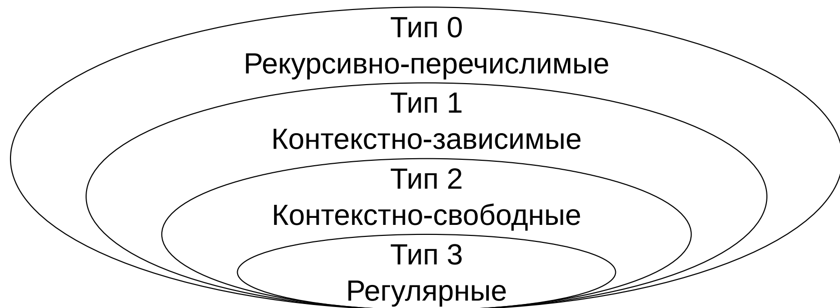
- Имеют правила вида $\alpha A\beta \to \alpha\gamma\beta$ (где α, β, γ - строка терминалов и/или нетерминалов; *A* нетерминал)
- Описываются линейно-ограниченной недетерминированной машиной Тьюринга
- Могут применяться для анализа естественных языков, практически не применяются для анализа языков программирования в силу своей сложности

Классификация грамматик и языков

Тип 0: рекурсивно-перечислимые

- Имеют правила вида $\gamma \to \alpha$ (где α, γ - строка терминалов и/или нетерминалов)
- Описываются машиной Тьюринга
- В силу своей сложности практическое применение отсутствует

Иерархия Хомского



В 1956 г. Хомский описывает иерархию формальных языков

Форма Бэкуса-Наура

Для описания грамматики языков программирования (а также языков разметки, и прочих нужд) используется специальный мета-язык. Он отображает правила вывода терминалов из нетерминалов и позволяет строить синтаксические анализаторы на своей основе.

На практике такое описание формальной грамматики впервые было внедрено Джоном Бэкусом в 1959 году для спецификации языка АЛГОЛ 58. В 1963 году при дальнейшем развитии языка АЛГОЛ, Питер Наур назвал такой мета-язык «Нормальной формой Бэкуса». Позже Дональд Кнут настоял на названии «форма Бэкуса-Наура».

С тех пор форма Бэкуса-Наура была доработана, и выведен международный стандарт Iso 14977, описывающий расширенную форму Бэкуса-Наура.

Примеры

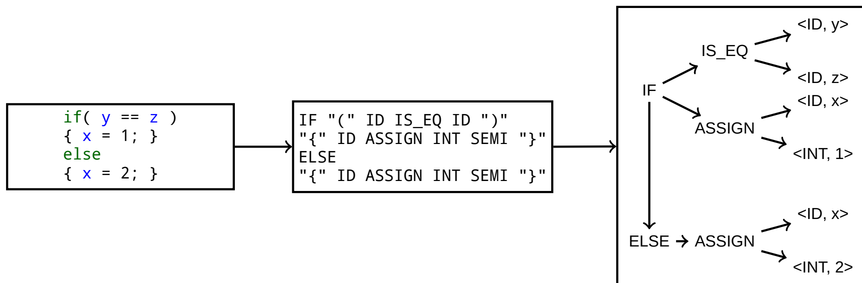
Пример участка грамматики языка C в синтаксисе Bison:

```
selection_statement
    : IF '(' expression ')' statement
    | IF '(' expression ')' statement ELSE statement
    | SWITCH '(' expression ')' statement
    ;

iteration_statement
    : WHILE '(' expression ')' statement
    | DO statement WHILE '(' expression ')' ';'
    | FOR '(' expression_statement expression_statement ')' statement
    | FOR '(' expression_statement expression_statement expression ')' statement
    | FOR '(' declaration expression_statement ')' statement
    | FOR '(' declaration expression_statement expression ')' statement
;
```

Деревья разбора

КСГ умеет определять принадлежит ли строка языку, однако на практике недостаточно механизма только допускающего строки. Необходимо уметь представлять программу в виде, пригодном для обработки.



Деревья разбора

Вывод (derivation) — это последовательность продукций $S \to \ldots \to \ldots \to \ldots$

Вывод может быть представлен в виде дерева, где:

- Стартовый символ является корнем
- Для продукции $X \to Y_1 \ldots Y_n$ к узлу X добавляются дети $Y_1 \ldots Y_n$

Деревья разбора

Формальное определение:

Деревом разбора для грамматики $G = (V_T, V_N, P, S)$ будет дерево, обладающее следующими свойствами:

1. Каждый внутренний узел отмечен переменной из V_N
2. Каждый лист отмечен либо переменной, либо терминалом, либо ε. Если лист отмечен ε, он должен быть единственным сыном своего родителя
3. Если внутренний узел отмечен A, и его сыновья отмечены слева направо $X_1, X_2, \ldots, X_k$, то $A \rightarrow X_1 X_2 \ldots X_k$ является продукцией в P

Деревья разбора

Составим грамматику для языка выражений, допускающих операцию $/$:

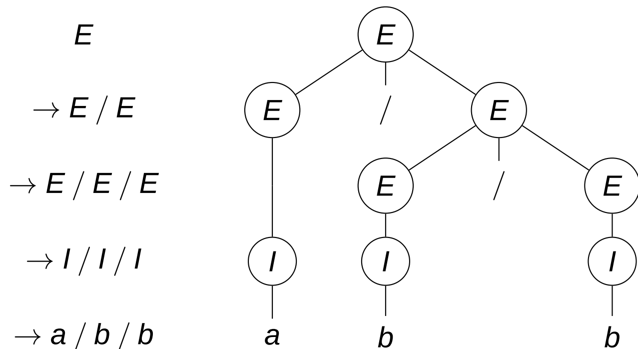
Деревья разбора

Составим грамматику для языка выражений, допускающих операцию $/$:

$$
\begin{aligned}
E &\rightarrow E \mathbin{/} E \\
  &\mid I \\
I &\rightarrow a \\
  &\mid b
\end{aligned}
$$

Деревья разбора

Для составленной грамматики построим дерево разбора выражения $a \mathbin{/} b \mathbin{/} b$:



Деревья разбора

Крона дерева — цепочка листьев дерева разбора, последовательно выписанная слева направо.

Крона дерева разбора полностью соответствует вводимой строке.

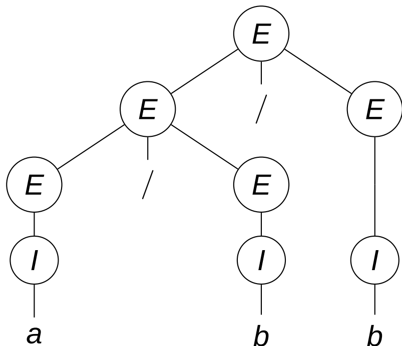
Вывод может быть **правосторонним** (right-most) или **левосторонним** (left-most), но оба способа дают одинаковый результат.

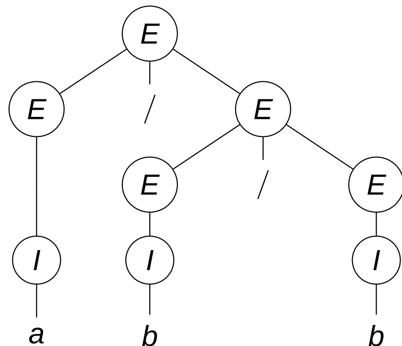
Неоднозначность грамматик

Для строки $a\ /\ b\ /\ b$ грамматика

$$\begin{aligned} E &\rightarrow E\ /\ E | I \\ I &\rightarrow a | b \end{aligned}$$

может создавать два дерева разбора:





Неоднозначность грамматик

Грамматику называют **неоднозначной** если для какой-либо строки она допускает более одного дерева разбора.

Это не мешает распознавать строки, однако для языка программирования неоднозначность грамматики недопустима.

Неоднозначность грамматик

Для практического разрешения неоднозначности вводится понятие **ассоциативности операций**.

Ассоциативность может быть правой, может быть левой, а может отсутствовать. Она отвечает за группировку выражений вида `x op y op z`.

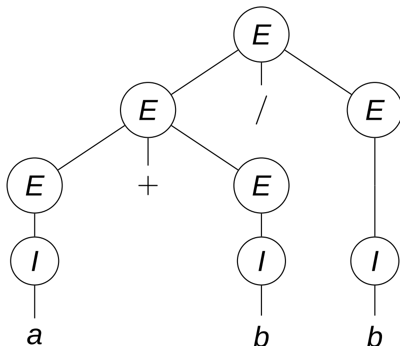
Например, для арифметических выражений вида $+$, $-$, $*$, $/$ принято выставлять левую ассоциативность, т.е.: `(x op y) op z`. А для выражений возведения в степень — правую: `x op (y op z)`.

Неоднозначность грамматик

Для левоассоциативных операций выражения $a + b \;/\; b$ рассмотрим по правилам грамматики

$$\begin{array}{lcl} E & \rightarrow & E + E | E \;/\; E | I \\ I & \rightarrow & a | b \end{array}$$

построение дерева разбора:



Неоднозначность грамматик

Для построения правильного с точки зрения арифметики дерева необходимо указать приоритет операций. Один из способов добиться этого — изменить грамматику:

Было:

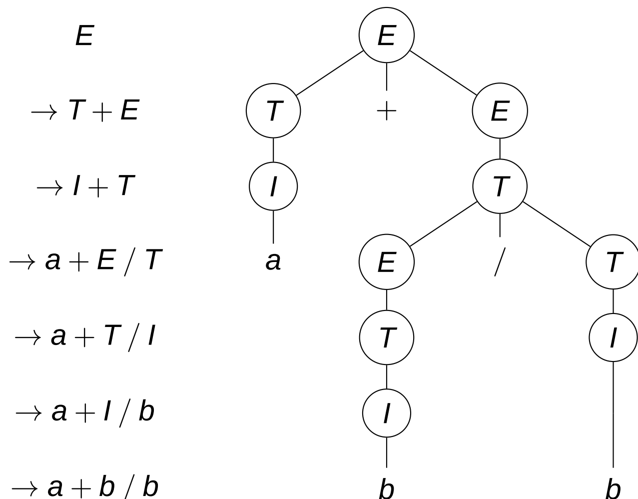
$$
\begin{array}{lcl}
E & \to & E + E \\
  & \mid & E \,/\, E \\
  & \mid & I \\
I & \to & a \\
  & \mid & b
\end{array}
$$

Стало:

$$
\begin{array}{lcl}
E & \to & T \\
  & \mid & T + E \\
T & \to & E \,/\, T \\
  & \mid & I \\
I & \to & a \\
  & \mid & b
\end{array}
$$

Неоднозначность грамматик

Для модифицированной грамматики построим дерево разбора выражения $a + b \,/\, b$:



Неоднозначность грамматик

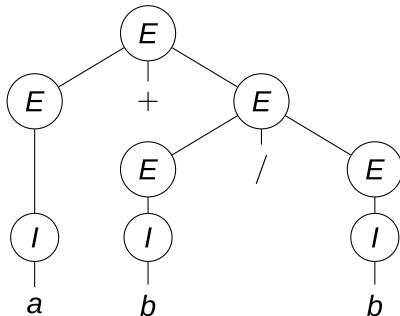
Для неоднозначной грамматики

$$\begin{array}{lcl} E & \rightarrow & E + E | E \, / \, E | I \\ I & \rightarrow & a | b \end{array}$$

можно договориться что приоритет операции $/$ будет выше приоритета операции $+$.

Неоднозначность грамматик

Тогда для строки $a + b \,/\, b$ данная грамматика будет задавать единственное дерево разбора:



Неоднозначность грамматик

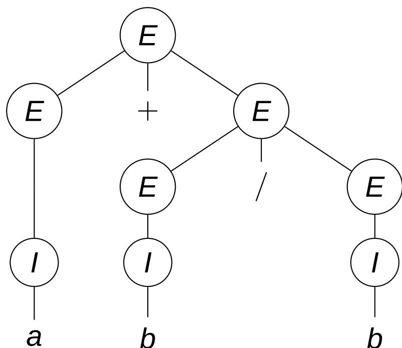
Не существует алгоритма трансформации неоднозначной грамматики в однозначную.

Иногда удобнее иметь неоднозначную грамматику и разрешать неоднозначность техническими (установка ассоциативности и приоритетов) способами.

Такие подходы применяются при работе с генераторами синтаксических анализаторов.

Упрощение дерева разбора

На практике не имеет смысла строить все узлы дерева разбора, более того лишние узлы будут только мешать.

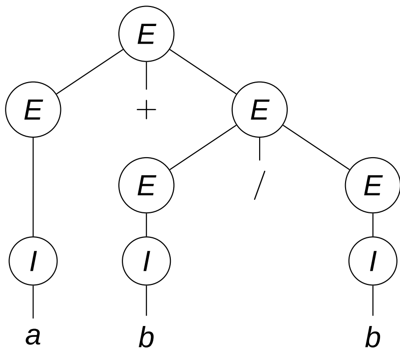


В данном дереве узлы E и I не несут в себе никакой полезной информации, а значит их можно не строить.

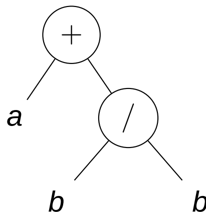
Упрощение дерева разбора

При построении дерева разбора мы стремимся избавляться от узлов, не влияющих на семантику программы.

Было:



Стало:



Упрощение дерева разбора

Пример участка дерева разбора, создаваемый clang для программы:

```
int foo(int a, int b) { return a + b / b; }
```

```
`-FunctionDecl 0x55c03fe78228 <t.c:1:1, col:43> col:5 foo 'int (int, int)'
  |-ParmVarDecl 0x55c03fe780c0 <col:9, col:13> col:13 used a 'int'
  |-ParmVarDecl 0x55c03fe78140 <col:16, col:20> col:20 used b 'int'
  `-CompoundStmt 0x55c03fe78418 <col:23, col:43>
    `-ReturnStmt 0x55c03fe78408 <col:25, col:40>
      `-BinaryOperator 0x55c03fe783e8 <col:32, col:40> 'int' '+'
        |-ImplicitCastExpr 0x55c03fe783d0 <col:32> 'int' <LValueToRValue>
        | `-DeclRefExpr 0x55c03fe78320 <col:32> 'int' lvalue ParmVar 0x55c03fe780c0 'a' '
        `-BinaryOperator 0x55c03fe783b0 <col:36, col:40> 'int' '/'
          |-ImplicitCastExpr 0x55c03fe78380 <col:36> 'int' <LValueToRValue>
          | `-DeclRefExpr 0x55c03fe78340 <col:36> 'int' lvalue ParmVar 0x55c03fe78140 'b'
          `-ImplicitCastExpr 0x55c03fe78398 <col:40> 'int' <LValueToRValue>
            `-DeclRefExpr 0x55c03fe78360 <col:40> 'int' lvalue ParmVar 0x55c03fe78140 'b'
```

Ошибки проектирования языков

В некоторых языках грамматика является неоднозначной. Например в языке Си существует классическая проблема «висящего `else`»:

```
if (a) if (b) s; else s2;
```

Ошибки проектирования языков

В некоторых языках грамматика является неоднозначной. Например в языке Си существует классическая проблема «висящего `else`»:

```
if (a) if (b) s; else s2;
```

В данном случае невозможно сказать к какому именно `if` относится `else`. Проблему можно было бы решить, обязав всегда обрамлять тело сравнения фигурными скобками.

Обработка ошибок

На вход компилятору могут подаваться некорректные программы. Задача компилятора обработать ошибку и сообщить пользователю правильную диагностику.

Например для программы с пропущенной точкой с запятой:

```
class A { }
int main(){return 0;}
```

Компилятор msvc-19 печатает следующее сообщение:

```
<source>(2): error C2628: 'A' followed by 'int' is illegal (did you forget a ';'?)
<source>(2): error C3874: return type of 'main' should be 'int' instead of 'A'
<source>(2): error C2440: 'return': cannot convert from 'int' to 'A'
<source>(2): note: No constructor could take the source type, or
constructor overload resolution was ambiguous
```

Обработка ошибок

Пример качественного понятного сообщения о той же ошибке от gcc-10.2:

```
<source>:1:12: error: expected ';' after class definition
    1 | class A { }
      |            ^
      |            ;
```

Отличным примером того как не надо делать диагностические сообщения является система вёрстки LaTeX.

Обработка ошибок

Обработка ошибок в компиляторе должна придерживаться следующих принципов:

- Сообщения об ошибках должны быть точными и понятными
- Восстановление из ошибочного состояния должно быть быстрым
- Обработчик ошибок не должен замедлять компиляцию корректных программ

Обработка ошибок

Стратегии поведения при обнаружении ошибки:

- Режим паники:
  - Пропускаем все токены, которые не можем распознать. Обычно ищутся токены завершения текущего выражения.
  - Продолжаем обработку с найденного токена.
- Продукции для ошибок: создаются специальные правила для наиболее часто встречающихся ошибок.
- Автоматическое исправление ошибок — на данный момент не используется

Режим паники

Обработку ошибок можно заложить в описание грамматики.

В генераторе синтаксических анализаторов `bison` существует специальный терминал `error`, определяющий пропускаемые токены:

$$E \quad \to \quad \mathit{int} | E + E | (E) | \mathit{error}\ \mathit{int} | (\mathit{error})$$

Такое правило позволит корректно обработать ошибочную запись:

```
( 1 + + 2 ) + 3
```

Продукции для ошибок

В грамматику можно заложить некоторые типы встречающихся ошибок. Например:

$$E \rightarrow EE$$

Позволит выявлять ошибку вида `5 x` вместо `5 * x`.

Данный метод черезмерно усложняет грамматику.

Автоматическое исправление ошибок

Основная идея — нахождение «ближайшей программы» методом перебора комбинаций с добавлением и удалением токенов.

Проблемы:

- Сложно сделать
- Замедляет компиляцию программы
- Невозможно угадать что имел ввиду человек

Заключение

- Большинство языков программирования возможно разобрать только контекстно-свободной (или даже контекстно-зависимой) грамматикой.
- Грамматика языка должна быть однозначной.
- Синтаксический анализатор должен те просто проверить правильность программы, но и построить дерево разбора, а также выдать понятное сообщение об ошибке.

Литература

- *А. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман* Компиляторы: принципы, технологии и инструменты.
- *K. D. Cooper, L. Torczon* Engineering a compiler.
- *A. Aiken* CS143 Compilers. Stanford lectures.
- *Д. Хопкрофт, Р. Мотвани, Д. Ульман* Введение в теорию автоматов, языков и вычислений
- *Р. Хантер* Основные концепции компиляторов