# Практическое занятие №1

Густов Владимир Владимирович
gutstuf@gmail.com

Цитата дня: Остерегайтесь трясин Тьюринга, в которых можно сделать всё, но ничего интересного нельзя сделать просто. (с) Алан Джей Перлис

# План курса

Вас ждёт:
- 9 практических занятий;
- 4 лабораторных занятия;
- 1 курсовая работа;
- 4 практических работы (в зачёт лабораторных);

Структура курсовой:
- введение (общее);
- внешний проект (общее: грамматика, список ошибок);
- реализация (индивидуальное, только своя часть);
- тестирование (общее и индивидуальное);
- заключение (общее);
- список литературы (классика).

Вывод: читайте больше специализированной литературы и выполняйте практические работы.

# Эзотерические ЯП

Эзотерические языки программирования придумываются с целью изучения и практики техник программирования, но в силу специфики своего синтаксиса (и прочих свойств) не используются в разработке коммерческого ПО.

Эзотерические ЯП могут классифицироваться по:
- полнота по Тьюрингу – можно ли (хотя бы в теории) с помощью языка реализовать любую вычислимую функцию или нет;
- цели создания – решение конкретной задачи, обфускация, проверка концепции;
- just for fun – забавы ради.

Не все Тьюринг полные языки могут быть применены на практике в силу плохой семантики и примитивного синтаксиса (так называемая Тьюринговская трясина).

# Arnold C

Компилируемый эзотерический ЯП использующий в качестве синтаксических конструкций фразы из фильмов с участием Арнольда Шварценеггера.
Написан на Scala.

```
a = (b || c) && d
```

```
GET TO THE CHOPPER a
HERE IS MY INVITATION b
CONSIDER THAT A DIVORCE c
KNOCK KNOCK d
ENOUGH TALK
```

```
IT'S SHOWTIME
TALK TO THE HAND "hello world"
YOU HAVE BEEN TERMINATED
```

# Brainfuck

Интерпретируемый эзотерический язык программирования, имеющий всего 8 команд.  Язык оперирует набором ячеек. Размер ячейки — один байт, количество ячеек 30 000. Ввод и вывод из ячеек в формате ASCII.

| Команда | Значение |
|---|---|
| > | Переход к следующей ячейке |
| < | Переход к предыдущей ячейке |
| + | Инкремент значения в текущей ячейке |
| - | Декремент значения |
| . | Вывод значения |
| , | Ввод значения |
| [ | Начало цикла |
| ] | Конец цикла |

70, 100, 30, 10

```
++++++++++
[>+++++++>++++++++++>+++>+<<<<-]
>++.>+.+++++++..+++.>++.<<++++++
++++++++.>.+++.------.--------
.>+.>.
```

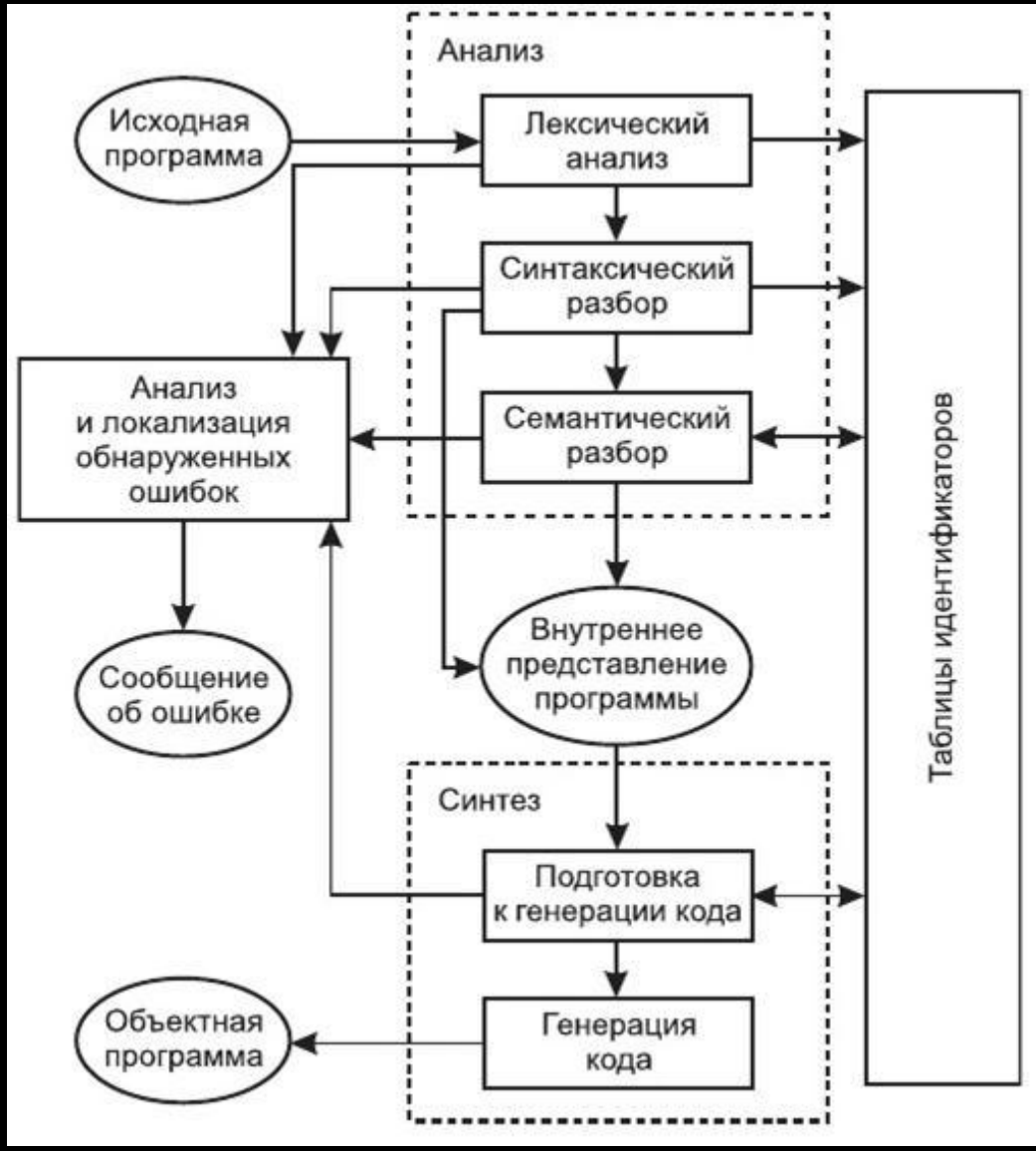
Nooooooo, you can't use Brainfuck as a real programming language, it's not practical!!!!!!!!!!!!!

```
++++++++++[>+++++++>++++++++
++>+++>+<<<<-]>++.>+.++++++
+..+++.>++.<<+++++++++++++++
.>.+++.------.--------.>+.>.
```

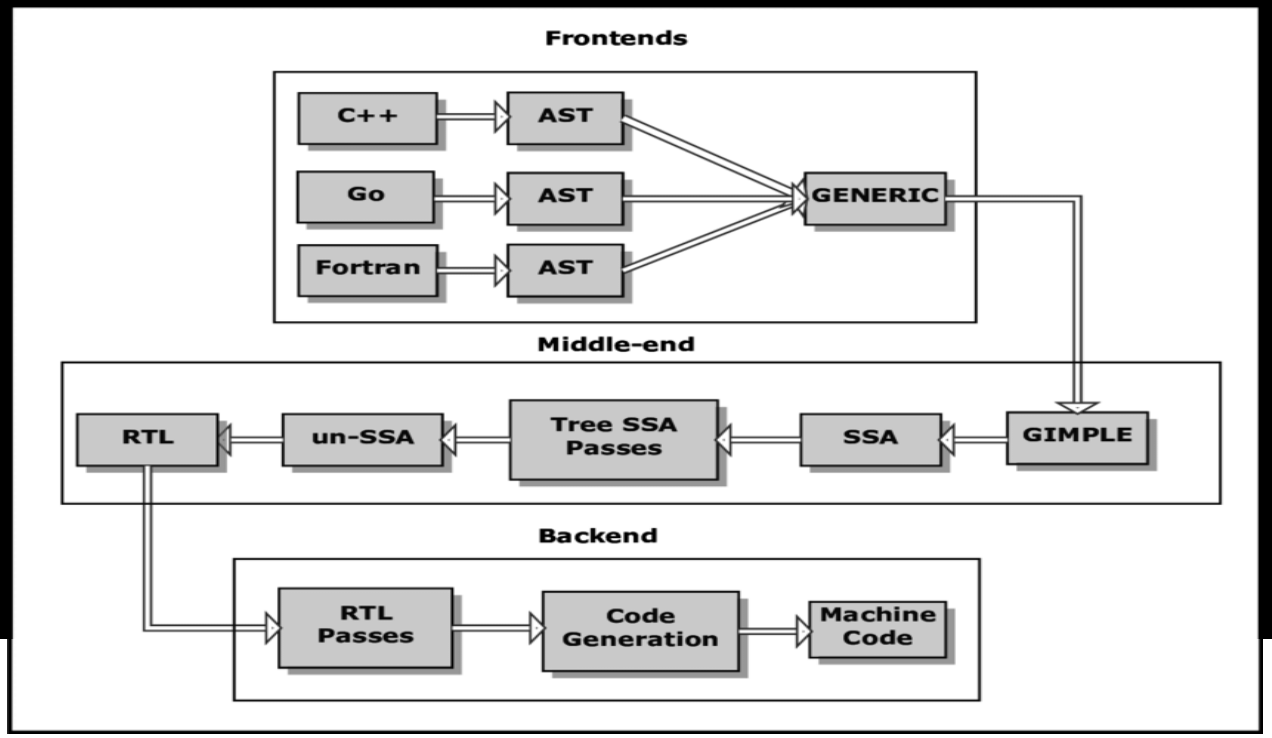
# Компилятор

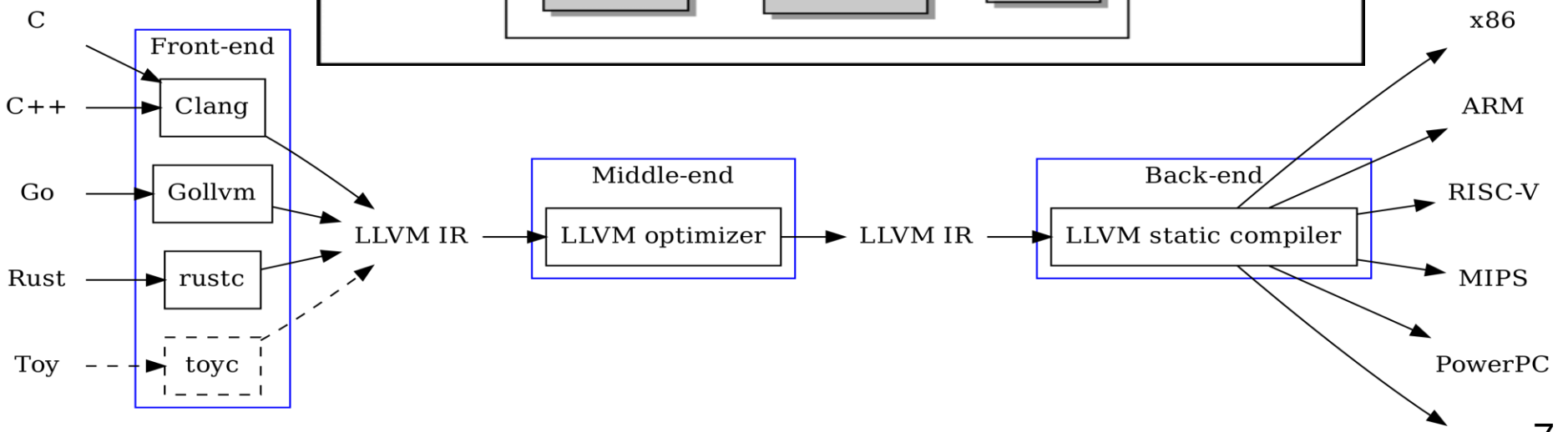
**Задачи компилятора:**

- Трансляция кода из ЯВУ в язык ассемблера;
- Валидация программы, вывод сообщений об ошибках;
- Статический анализ, вывод предупреждений;
- Оптимизация;
- Инструментальная обработка программы (профилирование, динамические проверки);
- Генерация отладочной информации
- Взаимодействие со сторонними программами (система сборки, отладчик, IDE)

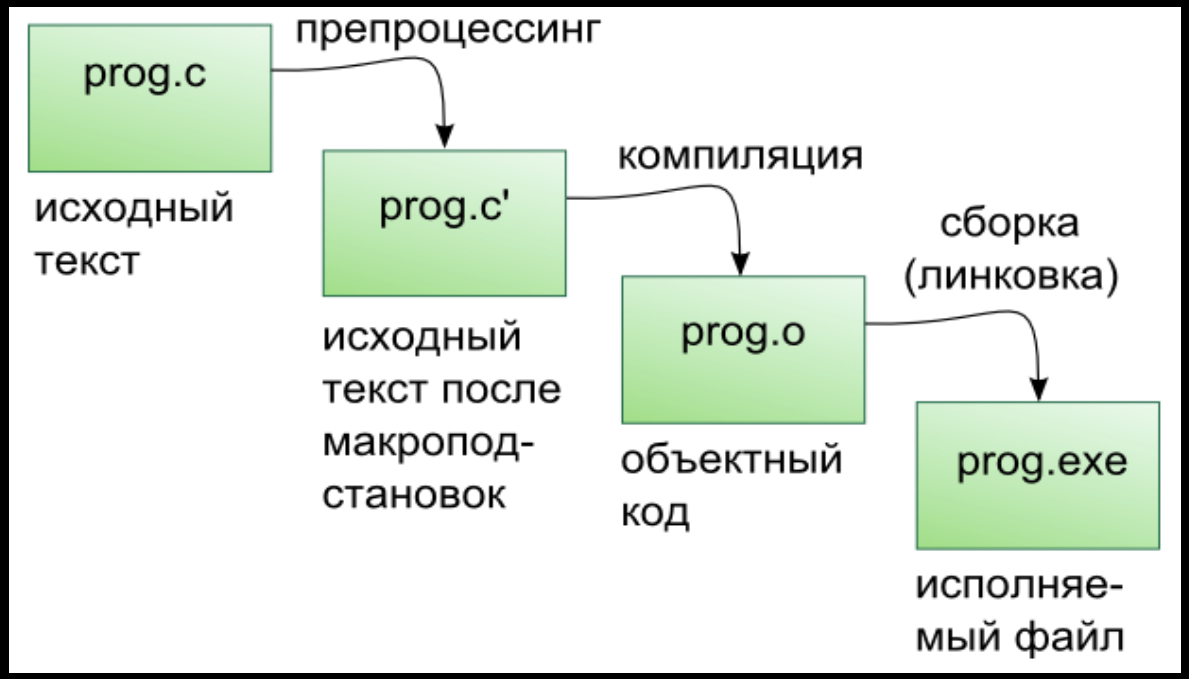


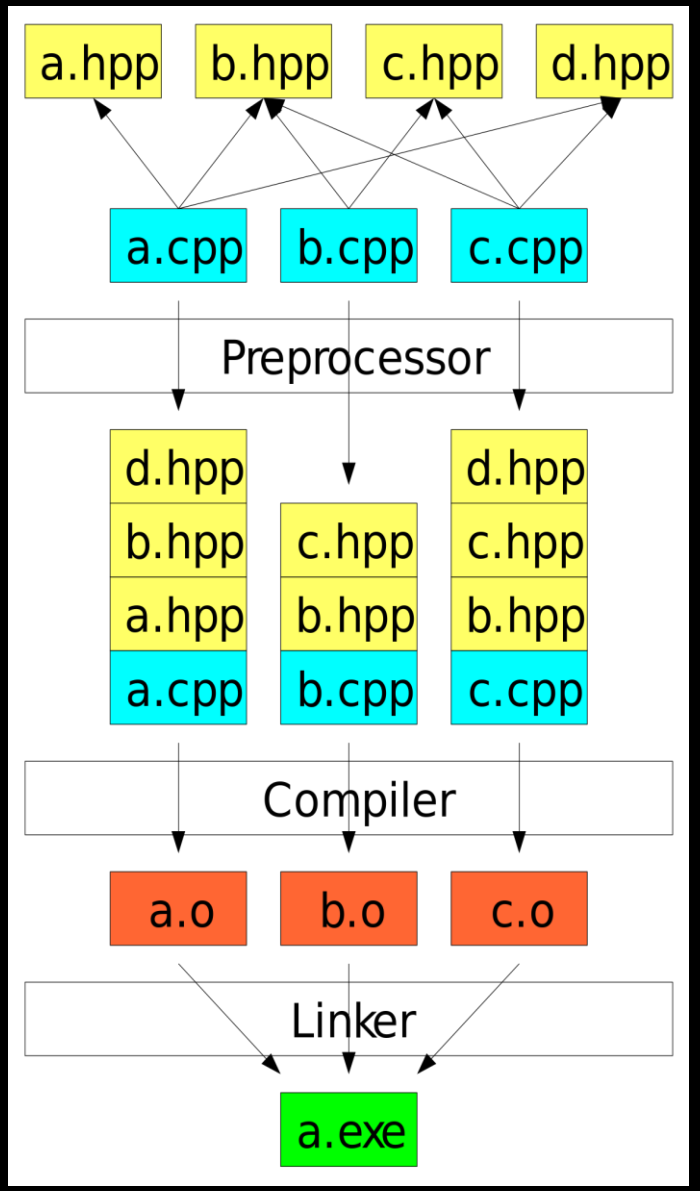
# GCC (GNU Compiler Collection) & LLVM (Low Level Virtual Machine)



# Компиляция C/C++ программ




GCC состоит из:
- cc/cpp – препроцессор;
- as – ассемблер;
- gcc/g++ - компилятор;
- ld – линкер.

# Этапы

- препроцессинг – работа с препроцессорными директивами. Добавление хэдеров, замена макросов их значениями, выбор нужных кусков кода (#if...);
- компиляция – преобразование кода (с подстановками) в ассемблерный код;
- ассемблирование – перевод ассемблерного кода в машинный. Генерирует объектный файл;
- компоновка (линковка) – связка всех объектных файлов и библиотек в единый исполняемый файл.

Промежуточное представление программы, после препроцессинга:

```
g++ -E main.cpp
cpp main.cpp
```

Представление ассемблированной программы:

```
g++ -S main.cpp
```

Компиляция и ассемблирование программы без линковки:

```
g++ -c main.cpp
```

# Грамматика

G = {N, T, P, S}, где
G – грамматика языка;
N – нетерминальные (неосновные) символы грамматики G;
T – терминальные (основные/конечные) символы грамматики G;
P – набор (множество) правил грамматики G;
S – начальный (обязательно нетерминальный) символ грамматики G.

Пример грамматики описывающей арифметические выражения, использующие операции сложения и умножения, а так же скобки и идентификаторы:

```
Exp → Ter
Exp → Exp + Ter
Ter → Mul
Ter → Ter * Mul
Mul → Id
Mui → ( Exp )
```

# БНФ/РБНФ

(Расширенная) Бэкус-Наурова форма – нотация используемая для описания синтаксиса языков программирования.

Свойства:
- вместо символа → («это есть») используется знак ::=;
- правила с одинаковыми левыми частями объединяют в одно, используя в качестве разделителя | («или»);
- нетерминальные символы всегда заключаются в угловые скобки < и >;
- фигурные скобки { и } означают, что заключённая в них конструкция может повторяться ноль и более раз;
- квадратные скобки [ и ] означают, что заключённая в них конструкция может входить, а может и не входить в правило;
- круглые скобки ( и ) используются для группировки символов.

**«Классическая»**

```
Exp → Ter
Exp → Exp + Ter
Ter → Mul
Ter → Ter * Mul
Mul → Id
Mui → ( Exp )
```

**РБНФ**

```
<Exp> ::= <Ter> | <Exp> + <Ter>
<Ter> ::= <Mul> | <Ter> * <Mul>
<Mul> ::= Id | ( <Exp> )
```

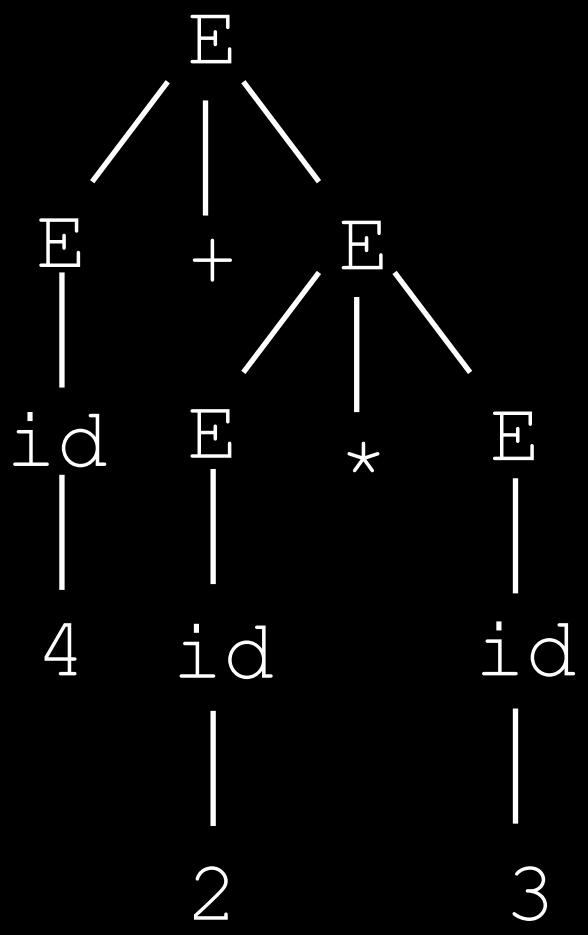
Если правило в грамматике имеет вид `U ::= Uu`, то она называется *прямо леворекурсивной* и *прямо праворекурсивной* при наличии правила вида: `U ::= uU`, где U нетерминал.

**Леворекурсивная грамматика** может привести синтаксический анализатор, работающий **методом рекурсивного спуска**, к **бесконечному циклу**, т.е. когда мы попытаемся развернуть нетерминал U, то в конечном счёте можем найти этот же нетерминал и прийти к попытке развернуть U, так и не взяв ни одного символа из входного потока.

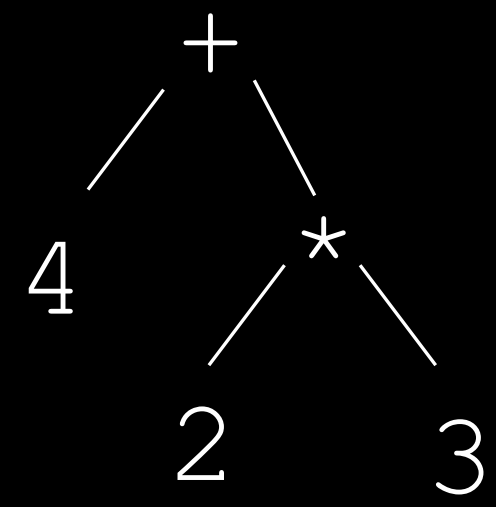
# Дерево разбора

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

4 + 2 * 3



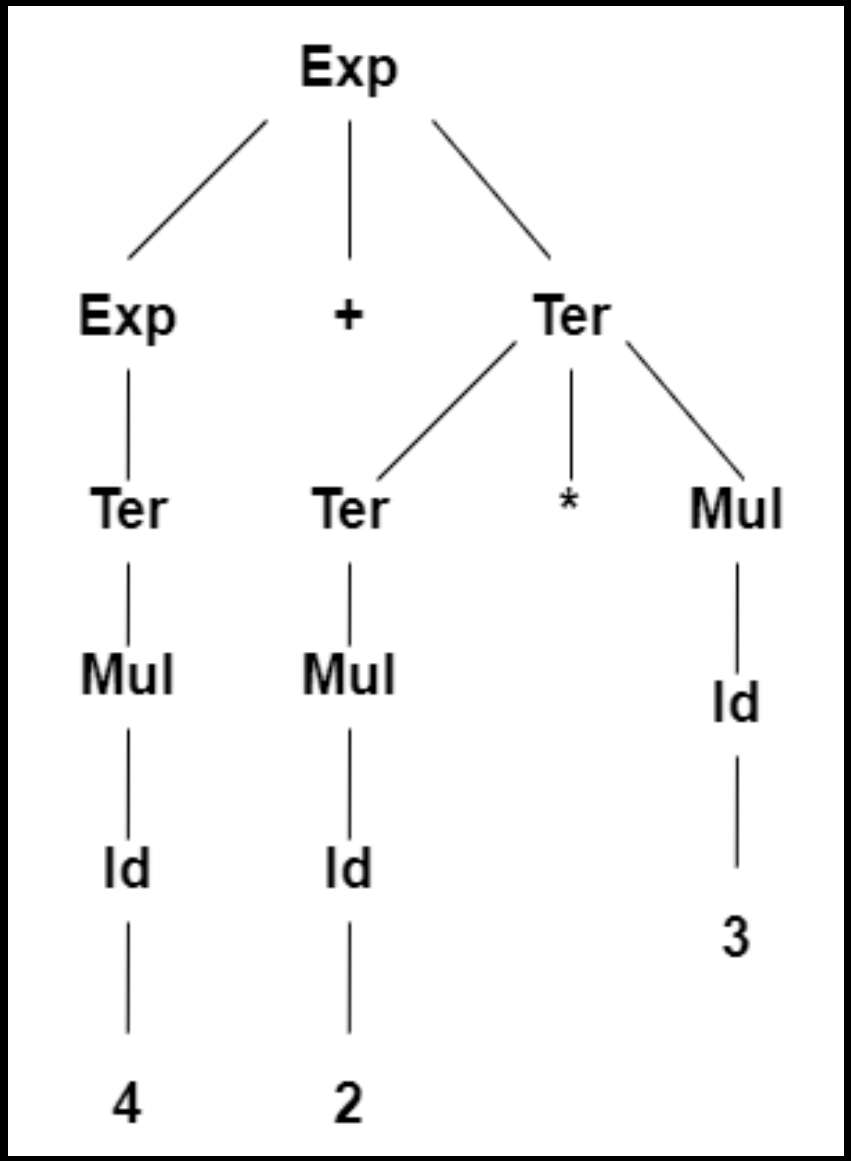

(дерево синтаксического разбора)




(абстрактное синтаксическое дерево)

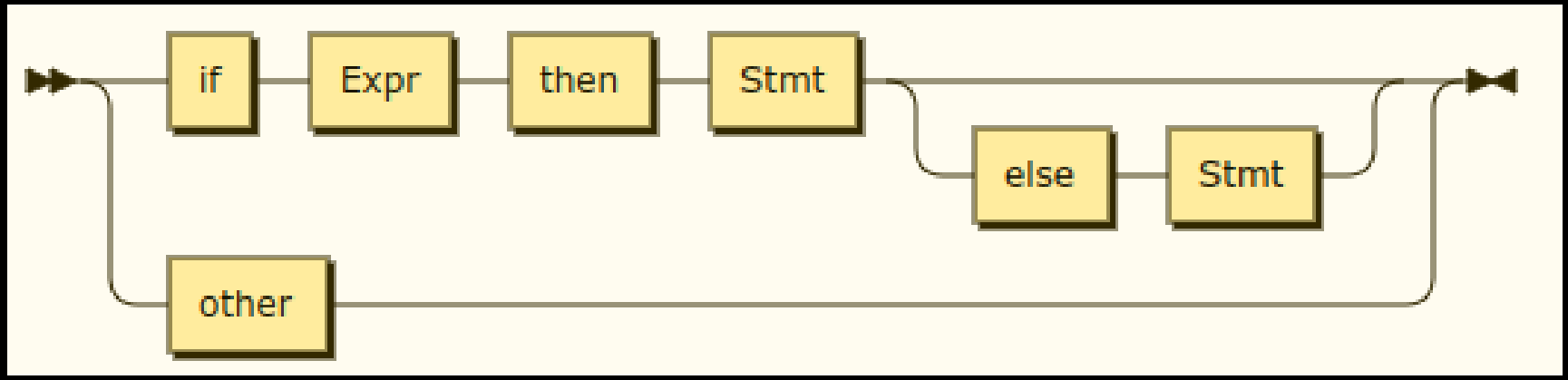
```
<Exp> ::= <Ter> | <Exp> + <Ter>
<Ter> ::= <Mul> | <Ter> * <Mul>
<Mul> ::= Id | ( <Exp> )
```

4 + 2 * 3



# Неоднозначность

```
Stmt ::= if Expr then Stmt
       | if Expr then Stmt else Stmt
       | other
```



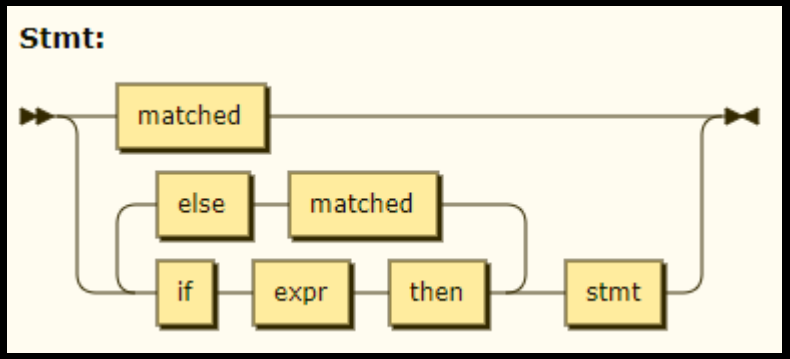

1) if E1 then other else if E2 then other else other
2) if E1 then if E2 then other else other

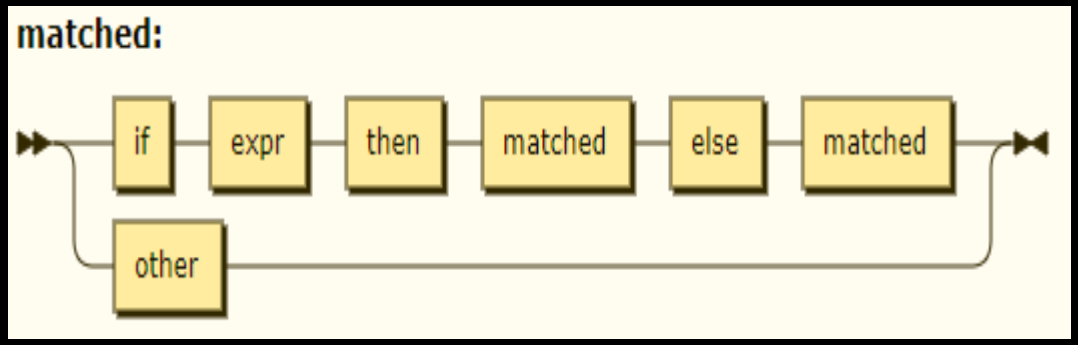
**Было:**

```
Stmt ::= if Expr then Stmt | if Expr then Stmt else Stmt
                           | other
```

**Стало:**

```
Stmt    ::= matched | open
matched ::= if expr then matched else matched | other
open    ::= if expr then stmt | if expr then matched else
                                                    open
```




1) if E1 then other else if E2 then other else other

2) if E1 then if E2 then other else other

# Факторизация

TL;DR: расширяем грамматику до тех пор, пока не станет ясно, какую продукцию выбирать

Было:

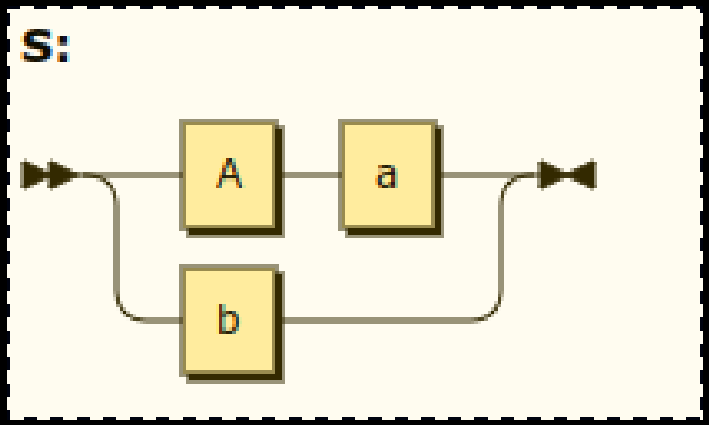
```
S ::= if E then S | if E then S
else S | other
```

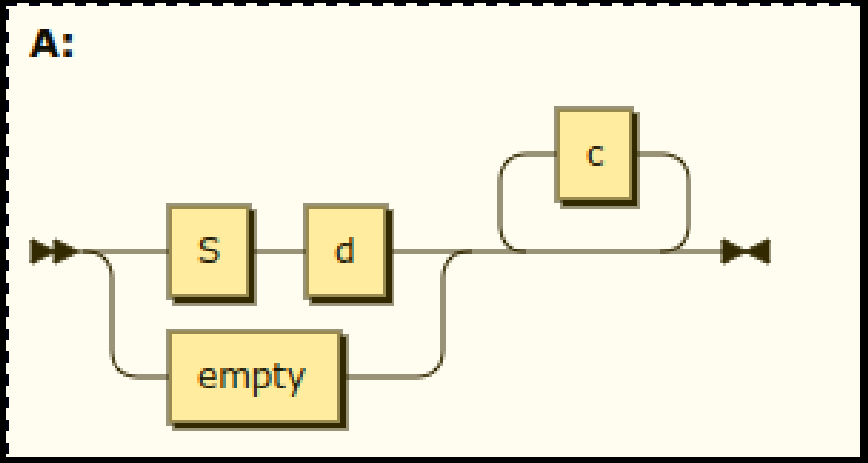
Стало:

```
S  ::= if E then S S` | other
S` ::= else S | empty
```

# Левая рекурсия

```
S ::= A a | b
A ::= A c | S d | empty
```




Непосредственная рекурсия – когда нетерминал A выводится из нетерминала B, с которого начинается вывод нетерминала A (A->B->A).

Прямая рекурсия – когда из нетерминала A выводится нетерминал A с дополнительными (не)терминалами (A->A).

0) S ::= A a | b
   A ::= A c | S d | empty

1) A ::= A c | A a d | b d | empty

2) A ::= b d A′ | A′
   A′::= c A′ | a d A′ | empty

**Итого:**

S ::= A a | b
A ::= b d A′| A′
A′::= c A′| a d A′ | empty

**Было:**

```
E ::= E + M | E - M | M
M ::= M * A | M / A | A
A ::= var | num | (E)
```

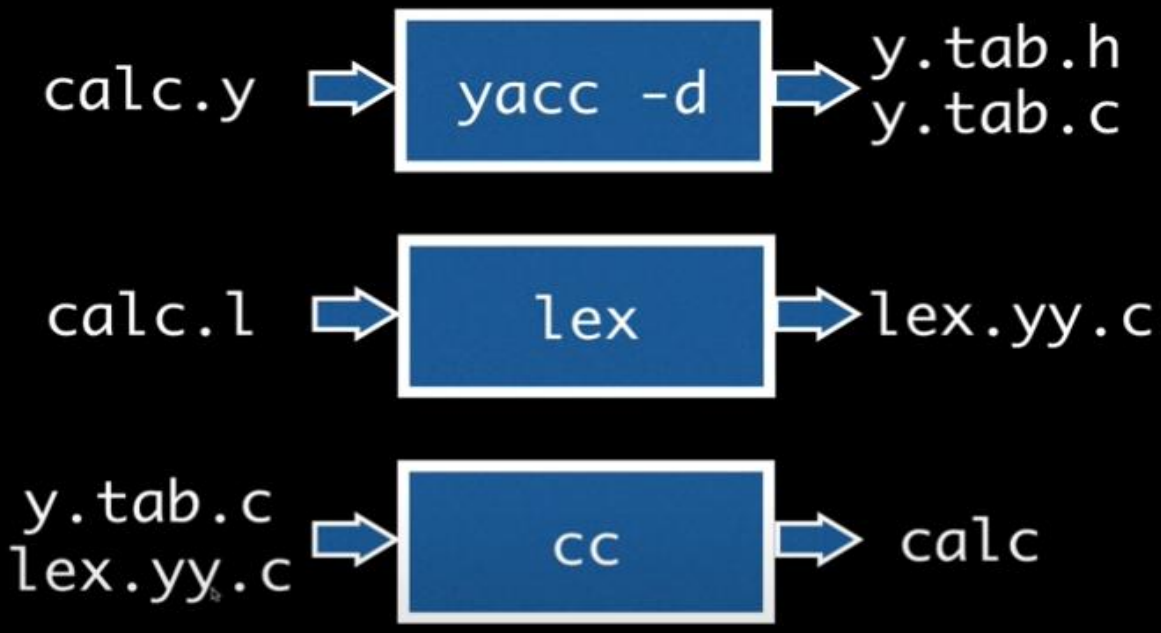
**Стало:**

```
E  ::=   M E′
E′ ::= + M E′ | - M E′ | empty
M  ::=   A M′
M′ ::= * A M′ | / A M′ | empty
A  ::= var | num | (E)
```

# Yacc Bison Flex Lex

Yacc/bison – генераторы синтаксического анализатора.
Lex/flex – генераторы лексического анализатора.

Файлы разделяются на три блока (определения, правила, код реализации на C/C++), результат работы – парсер указанной грамматики на языке C/C++.



# Ссылки

- про эзотерические яп [1];
- про эзотерические яп [2];
- примеры на Piet;
- вики по эзотерическим языкам;
- компилятор для ArnoldC;
- интерпретатор Brainfuck с визуализацией работы;
- устройство GCC;
- GIMPLE;
- SSA;
- Процесс компиляции C++;
- LLVM;
- пример работы с yacc;
- пример работы с flex и bison;
- о работе Lex и Yacc;
- построение графов грамматики по РБНФ;

# Just for fun

Кругом ограничения!
Напишите программу на языке C, которая будет выводить числа от 1 до 1000. В программе нельзя использовать ключевые слова if, switch, do, while, for, goto и подобные, логические операции, побитовую логику. Естественно, программа не должна содержать тысячу вызовов printf, нужно компактное решение. Постарайтесь не применять макросы.

* компактное решение – листинг не более 60 строк (при длине строки в 80 символов);
* в данном случае тернарные операции считаются логическими.