1

Курс простой. По нему зачёт.
Нужно решить 4 лабораторные работы:

- Регулярные выражения в языке **Perl**. Самостоятельное изучение материала и сдача в PCMS. Если сдавать в 2022, то можно решить N-3 задач. До конца сессии -2. После -1 (все, кроме одной).
- До 22:00 в пн нужно будет записываться на сдачу лабораторных 2,3,4. Показываешь, что работает. Получаешь модификацию (их много рарных на одно и то же задание). Потом показываешь уже модификацию.
- ЛР2 — ручное построение нисходящих парсеров
- ЛР3 — использование автоматических генераторов парсеров. ANTLR, Happy, Bison (YACC)
- ЛР4 — написать автоматический генератор парсеров

Компиляция: программа → парсинг → генерация исполнимого кода.
Здесь пройдём первый основы парсинга (довольно глубоко), а генерацию кода только базовую основу.
Если вы пойдёте в магистратуру, там будет курс компиляторов. Там фокус наоборот на генерации.

В курсе нет дедлайнов. НО удачи вам с тайм-менеджментом без них :v) Курс по софт-скиллам, you're welcom

Что такое пасрер?

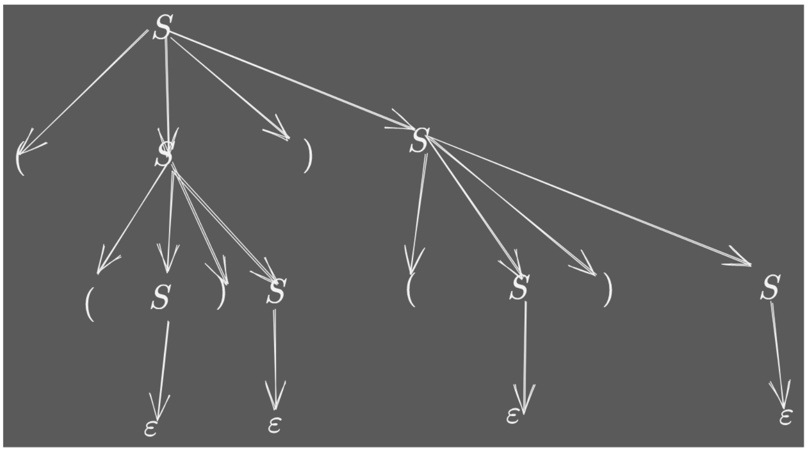
текстовое представление информации → внтреннее представление компьютера. Чаще всего дерево разбора в некоторой контекстно-свободной грамматике.

Регулярное выражение — способ записать регулярные языки.

Контекстно свободная грамматика — алфавит, множество нетерминалов, стартовый нетерминал и правила.

```
неоднозначная:
S → (S)
S → SS
S → e

однозначная:
S → (S)S
S → e
```



Токенизация: инпут в *любом* формате $\to$ $c_1, c_2, \ldots, c_n$, где c_i это токены.

Синтаксический/Лексический анализ (парсинг)

Лексический анализатор:
Есть входной алфавит парсера. Есть совсем входной алфавит A (ASCII условно).
Пример: арифметические выражения $\Sigma = \{+, -, *, (,), number\}$. Числа здесь не нужны. 2+3, 5+5, 17+231. Всё это считается одним выражением. Поэтому есть один токен для числа.

```
+ +
- -
* *
) )
( (
n (1|2|3|4|5|6|7|8|9)(0|1|2| ... |9)*|0

17+231
n
 n
 n+
 n+n
 n+ n
 n+  n
```

Задача: у вас есть ЯП довольно мощный. Вы прочитали токен. Вам нужно выдать номер токена. Чтобы делать это быстро нужно Perfect < ... > Mapping

---

Парсеры

Алгоритм Кока-Янгера-Касами
Работает только для грамматик в нормальной форме Хомского (A $\to$ BC). Можно снять ограничение, но это неприятно. Работает за куб от длины слова. Если бы мы парсили за куб ... было бы очень грустно

```
dp_A[l][r] — можно ли получить s[l..r] из A
```

Парсить хочется быстро.

Снизу вверх vs Сверху вниз.
Направление построения дерева. Его можно строить от корня или от листьев (спрашивая какой у них может быть родитель)

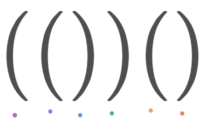
Рекурсивный парсер сверху вниз = рекурсивный спуск.
Есть языки, которые так не распарсить.

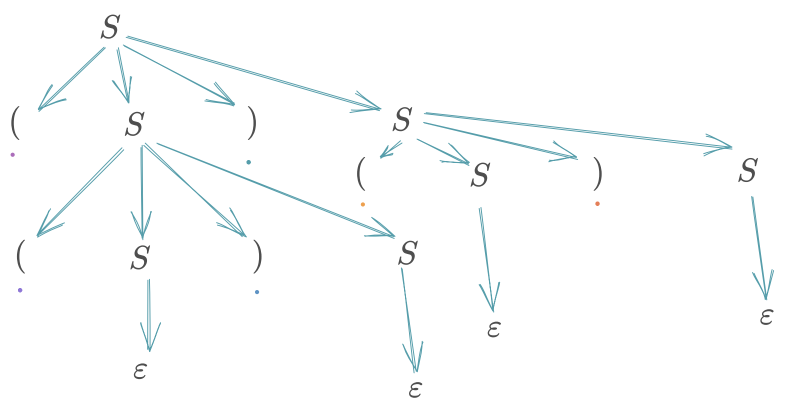
^ это было введение

Нисходящие методы разбора

```
S → (S)S
S → e

(())()
```





LL(1) грамматика — знание первого нераскрытого терминала и первого терминала даёт понять следующий шаг разбора.
$S \implies {}^*xA\alpha \implies x\xi\alpha \implies {}^*xc\alpha' \implies {}^*xcy*$
$S \implies {}^*xA\beta \implies x\eta\beta \implies {}^*xc\beta' \implies {}^*xcz*$

LL(1) грамматика — Если есть два таких вывода, то $\xi = \eta$

Патч: все символы уже подвесили, а первый нераскрытый терминал всё ещё есть. Нужно раскрывать в ε..

Добавим символ конец ввода $

Сейчас определение выше рассматривает бесконечное число выражений (импликаций). Можно сделать, чтобы конечное, об этом позже.

1

Методы Трансляции с 2022-09-06

2

FIRST: $(N \cup \Sigma) \to \mathbb{P}(\Sigma \cup \varepsilon)$
FIRST(α) = $\{c \mid \alpha \implies c\beta\} \cup \{\varepsilon \mid \alpha \implies \varepsilon\}$

Лемма:

- FIRST(ε) = $\{\varepsilon\}$
- FIRST($c\xi$) = $\{c\}$ $\quad c \in \Sigma$
- FiFIRSTrst($A\xi$) = (First[A]$\setminus\{\varepsilon\}$)$\cup$(First(ξ) if $\varepsilon \in$ First[A]

Алгоритм 1:
FIRST = $\{A \implies \emptyset \; for A \in N\}$

```
while changes
    for A → alpha \in Γ
        FIRST[A] U= FIRST(alpha)
```

Note: 2126 (140) - прыскалка для доски

Лемма 2: Алгоритм 1 корректно строит First[]

Пусть $c \in$ First(A)\First[A]. $A \implies c\xi$ за k шагов (минимальное k)

- $k \neq 1$
- $k > 1 \quad A \implies B\eta \underset{k-1}{\implies} c\xi$

$FOLLOW : N \to \mathbb{P}(\Sigma \cup \{\$\})$
$FOLLOW(A) = \{c \mid S \underset{*}{\implies} \alpha A c \beta\} \cup \{\$ \mid S \underset{*}{\implies} \alpha A\}$

Алгоритм 2:
FOLLOW = {A ⇒ ∅ for A ∈ N}

```
while changes:
    for A → alpha \in Γ:
        for B: alpha = xi N eta:
            FOLLOW[B] U= (FIRST[eta] \ epsilon) U
                         (FOLLOW[A] if epsilon \in FIRST(eta)
```

#png картинка дерева разбора

Пример:

```
E → T + E — первое слагаемое к сумме всего остального
^ это не то что мы хотим

что мы хотим:
E → E + T
E → T
T → T * F
T → F
F → eta
F → (E)
```

Σ = $\{(,),+,*,\eta\}$
N = $\{E,T,F\}$

FIRST[E] = FIRST[T] = FIRST[F] = $\eta($

$$\begin{array}{ccc} & FIRST & FOLLOW \\ E & \eta(& \$+) \\ T & \eta(& \$+*) \\ F & \eta(& \$+*) \end{array}$$

Теорема: грамматика Γ является LL1 тогда и только тогда, когда:

- $A \to \alpha, A \to \beta \implies FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
- $A \to \alpha, A \to \beta, \varepsilon \in FIRST(\alpha) \implies FIRST(\beta) \cap FOLLOW(A) = \emptyset)$
  Предположим, что грамматика LL1, но одно из предположений не выполняется.
  Пусть первое неверно:
- $FIRST(\alpha) \cap FIRST(\beta) \ni c \in \Sigma$
  $S \underset{*}{\implies} xA\xi \implies x\alpha\xi \underset{*}{\implies} xc\alpha'\xi$
  $S \underset{*}{\implies} xA\xi \implies x\beta\xi \underset{*}{\implies} xc\beta'\xi$
- $\varepsilon \in FIRST(\alpha) \cap FIRST(\beta)$
  $S \underset{*}{\implies} xA\xi \implies x\alpha\xi \underset{*}{\implies} x\xi \underset{*}{\implies} xc\eta$
  $S \underset{*}{\implies} xA\xi \implies x\beta\xi \underset{*}{\implies} x\xi => xc\eta$
  Пусть второе неверно:
  $\varepsilon \in FIRST(\alpha) \quad c \in FIRST(\beta)$
  ...

Пусть $\Gamma \notin LL(1)$
...

Литература:
Ахо, Сети, Ульман "Компиляторы: принципы построения, ... " на ней нарисован дракон (Dragon book)

Две проблемы, которые делают грамматику не LL1 (есть ещё, но вот есть вот эти две):

- Если в грамматике Γ есть левая рекурсия
  A $\underset{+}{\Longrightarrow} A\alpha$
- Если в грамматике Γ есть правое ветвление
  $A \to \alpha\beta \quad A \to \alpha\gamma \quad \exists x \neq \varepsilon : \alpha \underset{*}{\Longrightarrow} x$

2

Методы Трансляции с 2022-09-13

3

...
A → alpha_1, A → alpha_2, ... , A → alpha_k

```
FIRST_1(alpha) = FIRST(alpha) \ eps U (FOLLOW(A) if eps \in
FIRST(alpha))

Node A()
    Node res = Node('A')
    switch (token)
        case FIRST_1(alpha_1)
            // alpha_1[1] \in N, B = alpha_1[1]
            Node n1 = B()
            res.children.push_back(n1)
            // alpha_1[2] \in \Sigma, c = alpha_1[2]
            ensure(token = c;"expected".c."found".token)
            Node n2 = Node(token)
            res.children.push_back(n2)
            next_token()
        ... // end of switch
        default:
            error("unexpected".token)
    return res
```

```
E  → TE'
E' → +TE'
E' → eps
T  → FT'
T' → * FT'
T' → eps
```

```
F  → n
F  → (E)
```

```
Node E()
    Node res = Node('E')
    switch(token):
        case '(', 'n':
            n1 = T()
            res.children.push_back(n1)
            n2 = E'()
            res.children.push_back(n2)
            break
        default:
            error( ... )
    return res

Node E'()
    Node res = Node('E')
    switch(token)
        case '+':
            ensure (token = '+')
            res.children.push_back(Node('+'))
            next_token()
            res.children.push_back(T())
            res.children.push_back(E'())
           break
        case '$', ')':
            break
        default:
            error
    return res

Node F()
    Node res = Node(F)
    switch(token)
        case 'n':
            res.children.push_back(n)
            nextToken()
            break
        case '(':
            res.children.push_back(Node('('))
            nextToken()
            res.children.push_back(E)
            ensure(token = ')')
            res.children.push_back(Node(')'))
            nextToken()
```

```
A → beta A'
A'→ alpha A'
```

```
A'→ eps
```

```
Node A()
    Node res = Node('A')
    process beta
    while (token \in FIRST(alpha)
        t = res
        res = Node('A')
        res.children.push_back(t)
        process alpha
    ensure (token \in FOLLOW(A))
    return res
```

3

Методы Трансляции с 2022-09-20