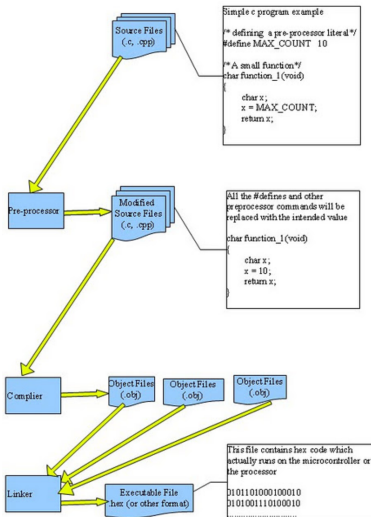


Автоматы и лексический анализ

Юрий Литвинов
y.litvinov@spbu.ru

07.12.2021

Напоминание о процессе компиляции C++-кода



Фазы компилятора



Лексический анализ

- ▶ Преобразование потока символов в поток токенов
- ▶ Например, `position := initial + rate * 60`
 - ▶ Идентификатор `position`
 - ▶ Символ присвоения
 - ▶ Идентификатор `initial`
 - ▶ Знак сложения
 - ▶ Идентификатор `rate`
 - ▶ Знак умножения
 - ▶ Число 60
- ▶ Токен представляется в виде структуры из типа токена и его значения

Формальные языки, определения

- ▶ Алфавит — произвольное множество. Элементы множества называются *символами* алфавита
- ▶ Строка (она же *цепочка*) — конечная последовательность символов из алфавита
- ▶ Длина строки s (обозначается как $|s|$) — количество символов в строке
- ▶ Пустая строка (ϵ) — строка, которая не содержит символов
- ▶ Конкатенация строк — две строки, записанные друг за другом
- ▶ Язык — множество строк. Например, пустой язык (обозначается $\{\}$), множество всех корректных программ на C++, множество всех грамматически корректных предложений русского языка

Операции над языками

Позволяют собрать из простых более сложные

- ▶ Объединение L и M ($L \cup M$) — $L \cup M = \{s \mid s \in L \vee s \in M\}$
- ▶ Конкатенация L и M (LM) — $LM = \{st \mid s \in L \wedge t \in M\}$
- ▶ Замыкание Клини или *итерация* (она же *звёздочка Клини*) (L^*) — $L^* = \bigcup_{i=0}^{\infty} L^i$, где L^i — это $LL \dots L$ i раз
 - ▶ В том числе, $i = 0$, то есть ϵ всегда принадлежит L^*
 - ▶ То есть, пустая строка всегда часть замыкания
- ▶ Позитивное замыкание L (L^+) — LL^*
 - ▶ Или $L^* = \bigcup_{i=1}^{\infty} L^i$, то есть замыкание без пустой строки

Регулярные выражения

Формальный язык для записи языков

- ▶ Регулярное выражение ϵ задаёт язык $\{\epsilon\}$
- ▶ Регулярное выражение a задаёт язык $\{a\}$ (язык из одного символа)
- ▶ Пусть r и s — регулярные выражения, задающие языки $L(r)$ и $L(s)$. Тогда
 - ▶ Регулярное выражение $(r) \mid (s)$ задаёт объединение, $L(r) \cup L(s)$
 - ▶ Регулярное выражение $(r)(s)$ задаёт конкатенацию, $L(r)L(s)$
- ▶ Регулярное выражение $(r)^*$ задаёт замыкание Клини, $(L(r))^*$
- ▶ Регулярное выражение (r) задаёт $L(r)$ (то есть можно ставить скобки)
- ▶ Регулярное выражение $r?$ задаёт $L(r) \cup \epsilon$
- ▶ Регулярное выражение r^+ задаёт $L(r)^+$
- ▶ Регулярное выражение $[a..z]$ задаёт $L(a) \cup L(b) \cup \dots \cup L(z)$

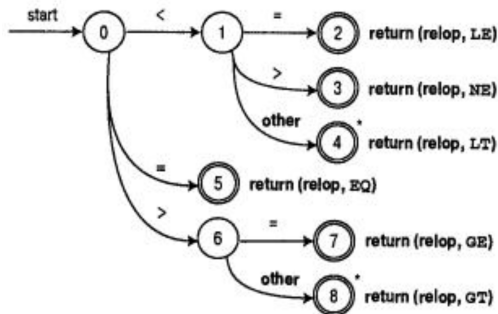
Регулярные определения

Именуем регулярные выражения для удобства записи

- ▶ $letter \rightarrow A|B|\dots|Z|a|b|\dots|z$
- ▶ $digit \rightarrow 0|1|\dots|9$
- ▶ $id \rightarrow letter(letter|digit)^*$
- ▶ $num \rightarrow digit + (.digit+)^?(E(+|-)?digit+)^?$
- ▶ $relop \rightarrow < | <= | = | <> | > | >=$

Диаграммы переходов

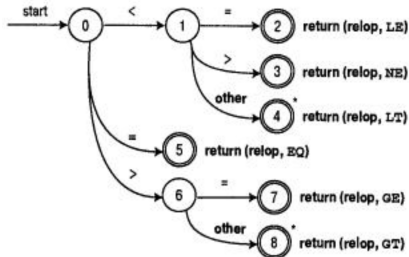
relop $\rightarrow < \mid <= \mid = \mid <> \mid > \mid >=$



- ▶ Двойной кружок — *принимаящее состояние*
- ▶ Звёздочка — вернуть последний символ во входной поток

Как это закодировать

relop → < | <= | = | <> | > | >=



```

Token nextToken()
{
    while (true) {
        switch (state) {
            case 0:
                c = nextChar();
                if (c == blank || c == tab || c == newline) {
                    state = 0;
                    lexeme_beginning++;
                } else if (c == '<') {
                    state = 1;
                } else if (c == '=') {
                    state = 5;
                } else if (c == '>') {
                    state = 6;
                } else {
                    state = fail;
                    break;
                }
            case 1: ...

```

Конечные автоматы

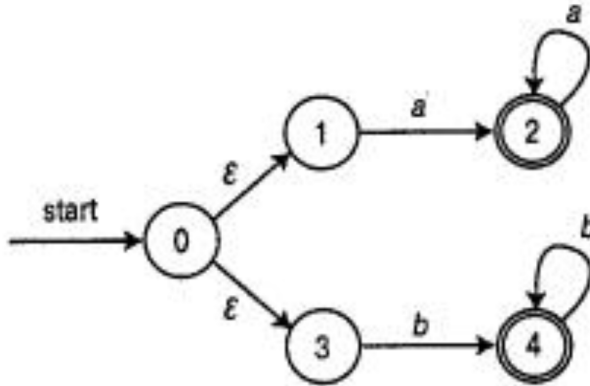
- ▶ Конечный автомат — $(S, \Sigma, move, s_0, F)$
 - ▶ S — множество состояний
 - ▶ Σ — входной алфавит
 - ▶ $move$ — функция переходов
 - ▶ s_0 — начальное состояние
 - ▶ F — множество допускающих состояний
- ▶ Неформально автомат имеет состояние, в зависимости от которого может по-разному реагировать на входные символы (или события)
- ▶ Применяются не только в лексическом анализе, но и в сетевых протоколах, пользовательских интерфейсах и т.д. и т.п.
- ▶ Автомат принимает язык, если для любой строки языка после выполнения всех переходов он оказывается в допускающем состоянии

ДКА и НКА

- ▶ Детерминированный конечный автомат (ДКА) —
 $move : S \times \Sigma \rightarrow S$
- ▶ Недетерминированный конечный автомат (НКА) —
 $move : 2^S \times \Sigma \cup \epsilon \rightarrow 2^S$
 - ▶ НКА может находиться в нескольких состояниях одновременно и делать переход одновременно в несколько разных состояний
 - ▶ Или, другая точка зрения — он не знает, в каком именно состоянии сейчас находится
 - ▶ И есть эпсилон-переходы — спонтанные переходы, без входного символа

Пример НКА

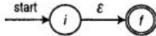
Пусть есть язык, задаваемый регулярным выражением $aa^* \mid bb^*$.
Тогда его принимает НКА:



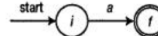
Построение НКА по регулярному выражению

Теорема: по любому РВ можно построить НКА, принимающий язык РВ. Доказательство:

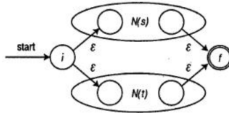
1. ε



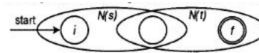
2. a из Σ



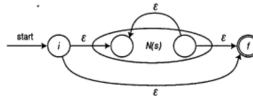
3. $s|t$



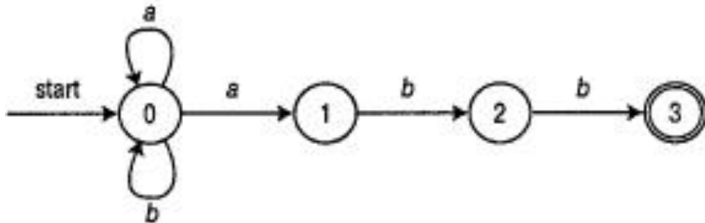
4. st



5. s^*



Моделирование НКА, таблица переходов



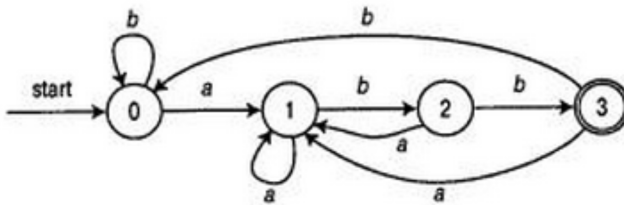
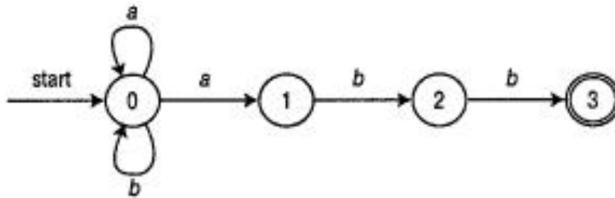
Состояние	a	b
0	{0, 1}	{0}
1	—	{2}
2	—	{3}

Построение ДКА по НКА

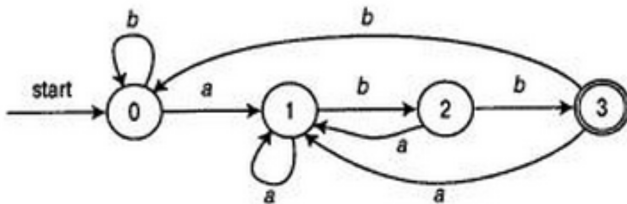
Теорема: по любому НКА можно построить ДКА, принимающий в точности тот же язык (то есть НКА и ДКА эквивалентны друг другу в плане выразительности).

Без доказательства.

Пример



Моделирование ДКА



Состояние	a	b
0	1	0
1	1	2
2	1	3
3	1	0

Как это выглядит в коде

- ▶ switch-case вполне вариант
- ▶ Интерпретировать таблицу состояний
 - ▶ Лучше, меньше кода и гибче

```
while c <> eof do begin
```

```
    s := move(s, c);
```

```
    c := nextchar();
```

```
end;
```

```
return (s in F);
```

Материалы

Книжка:

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



Конструктор автоматов:

- ▶ Веб-версия: <https://spbu-se.github.io/WebAutomataConstructor>
- ▶ GitHub: <https://github.com/spbu-se/WebAutomataConstructor>
- ▶ Desktopная версия:
<https://github.com/spbu-se/DesktopAutomataConstructor>
- ▶ Экспериментальная десктопная версия: <https://github.com/IlyaMuravjov/automaton-constructor/tree/develop>