**Введение в теорию компиляторов**

*Трансляторы*

Транслятор – программа или техническое средство, выполняющее трансляцию программы

Трансляция программы – преобразование программы, представленной на одном из языков программирования, в программу на другом языке и, в определенном смысле, равносильную первой.

Трансляторы:

* Компиляторы (на вход программа, выход – исполняемый модуль)
* Интерпретаторы (на вход – программа, исходные данные, выход -результат выполнения)

FORTRAN I – первый компилятор

*Структура компилятора*

1. Лексический анализ
2. Синтаксический анализ (парсинг)
3. Семантический анализ
4. Оптимизация
5. Генерация кода

Шаг 1: распознавание слов

* Наименьшая единица программы после букв

На этапе лексического анализа программа разбивается на «слова» - лексемы

Шаг 2: понимание структуры предложения

Парсинг (синтаксический анализ)

Шаг 3: понимание смысла предложения (это сложно!)

Компиляторы выполняют ограниченный семантический анализ:

* Обнаружение несоответствий

Шаг 4: оптимизация

Не имеет аналога в естественных языках

Похожая операция – правка текста

Автоматическое изменение программ таким образом, чтобы они:

* Выполнялись быстрее
* Использовали меньше памяти

Шаг 5: генерация кода

Перевод на другой язык

*Лексический анализ*

Класс лексемы:

* В русском языке
  + Существительное
  + Глагол
  + Прилагательное
  + И др.
* В языке программирования:
  + Идентификатор
  + Ключевое слово
  + (
  + )
  + Число
  + И др.

Классы лексем соответствуют множествам строк

Идентификатор:

* Строка буква и цифр, начинающаяся с буквы

Целое число:

* Непустая строка из цифр

Ключевое слово:

* Else, if, begin и т.д.

Пробельные символы

* Непустая последовательность пробелов, переводов строк, табов и т.д.

Цели:

* Выделение «слов»
* Классификация подстрок-«слов» в соответствии с их ролью в программе (классы лексем)
* Передача токенов парсеру (на стадию синтаксического анализа)



Х = 42

<идентификатор, «х»> <операция, «=»> <число, «42»>

Классы лексем:  
Пробелы | (  
ключевые слова | )  
идентификаторы | ;  
числа | =  
операции сравнения |

FORTRAN I

* Пробельные символы игнорируются

VAR1 и VA R1

Цели – разбить строку на лексемы

Разбиение осуществляется путем чтения слева направо и распознавания по одной лексеме за каждый шаг

Может потребоваться lookahead (backtracking) – предпросмотр

PL/I

* Ключевые слова не являются зарезервированными

- Неограниченный предпросмотр

Лексическая структура языка = классы лексем

Необходимо определить, какое **множество строк** образует каждый класс лексем

* Регулярные языки

*Регулярные выражения*

Одиночный символ

‘c’ = {“c”}

Эпсилон

e = {“”} (не пустое множество)

Объединение

A+B = {a|a э A}^{b|b э B}

Конкатенация

AB = {ab|a э A, b э B}

Итерация

A\*= ^ Ai Ai = A…A

i>=0 A0 = e

Регулярные выражения над алфавитом ∑ – наименьшее множество выражений, включающее:

R = e c э e

| ‘c’

| R+R

| R\*R

| R\*

∑ = {0, 1}

1\* = ^ Ai (i>= 0)

“” + “1” + “11” + … = все строки из единиц

(1+0)1 = {ab|a э 1 + 0, b э 1} = {11, 01}

0\* + 1\* = {0i|I >=0} ^ {1i|i>=0}

(0 + 1)\* = ^(0+1)i

“”, (0+1), (0+1)^2, (0+1)^3, … ∑\*

Регулярные выражения задают регулярные языки.

* Регулярное выражение – синтаксис.
* Регулярные язык – множество строк.

5 конструкций:

* 2 базовые конструкции:
  + Пустая и 1-символььная строки;
* 3 составные конструкции:
  + Объединение, конкатенация, итерация.

*Формальные языки*

Пусть ∑ - множество символов (алфавит)

Формальный язык над алфавитом ∑ - множество строк, состоящих из символов алфавита ∑ (и удовлетворяющих определенным правилам)

*Алфавит = буквы русского языка  
Язык = предложения на русском языке*

*Алфавит = ASCII  
Язык = программы на языке С*

*Функция значения (meaning function)*

Функция значения L(x) задает взаимное соответствие синтаксиса и семантики

L(e) = M

Регулярное выражение = множество строк

L(e) = {“”} (не пустое множество)  
L(‘c’) = {“c”}  
L(A+B) = {a|a э L(A)}^{b|b э L(B)}  
L(AB) = {ab|a э L(A), b э L(B)}  
L(A\*) = ^ L(Ai)   
 i>=0

L: Выражение 🡪 Множество строк

Функция значения:

* Позволяет разделить синтаксис и семантику
* Позволяет отделить проблему выбора нотации от остальных вопросов
* Выражение и их значение (смысл) не всегда однозначно соответствуют друг другу

**Лексические спецификации**

Первый этап разработки лексического анализатора – разработка лексической спецификации языка

* Т. е. описания правил записи лексем языка

Часто для записи лексической спецификации используют регулярные выражения

Ключевые слова

(“if”, “then”, “else” и т.д.)

‘i’f’ + ‘t’’h’’e’’n’ + … (фуу, какашка)

‘if’ + ‘then’ + ‘else’ + … (воо, уже лучше)

Целое число – непустые строки из цифр

digit = ‘0’ + ‘1’ + … + ‘9’

digit digit\* AA\*

digit + A+

Идентификатор – множество строк, состоящих из буквы и цифр и начинающихся с буквы.

letter = ‘a’ + ‘b’ + …

letter = [a-zA-Z]

letter (letter + digit)

Пробельные символы – непустая строка из пробелов, переводов строк и Tab-символов.

(‘ ‘ + ‘\n’ + ‘\t’)+

Вещественные числа:

digit = [0-9]  
digits = digit+  
opt\_frac = (‘.’ digits) + e = (‘.’ digits)?  
opt\_exp = (‘E’ (‘+’ + ‘-‘ + e) digits) + e = (‘E’ (‘+’ + ‘-‘)? digits)?  
num = digits opt\_frac opt\_exp

Один и более: А+ = АА\*

Перечисление: А | B = A + B

Необязательный: A? = A + e

Диапазон: [a-z] = ‘a’ + ‘b’ + …

Дополнение: [^a-z]

*Регулярные выражения и RegExp*

Во многих ЯП есть возможность, которая также называется «регулярные выражения» (часто – просто regexp’ы)

* Есть также библиотеки аналогичного назначения

То, что реализуется такими языками и библиотеками, как правило НЕ является регулярными выражениями

* И почти всегда используется неоправданно и не к месту

ВНИМАНИЕ!!!

Во всех вопросах по дисциплине обсуждаются НЕ regexp’ы, а регулярные выражения

Ответ, в котором описываются синтаксис и назначение regexp’ов, равнозначен **отсутствию ответа**

Регулярные выражения могут использоваться для описания многих полезных языков

Регулярные выражения – спецификация языка.

* По-прежнему нужна реализация.

Задача:

* Как, имея строку s и регулярное выражение R определить, верно ли, что  
  s e L(R)

Множество строк

c1c2c3c4|c5c6c7|c8c9…

1. Написать регулярные выражения для каждого класса сексем

* Number = digit+
* Keyword = ‘if’ + ‘else’ + …
* Identifier = letter (letter + digit)\*
* OpenParen = ‘(‘
* …

1. Построить регулярное выражение R, задающее все возможные лексемы всех классов лексем

R = Keyword + Identifier + Number + … = R1 + R2 + …

1. Пусть входная строка – x1…xn

Для 1 <= I <= n проверить:  
x1…xi e L(R)

1. Если условие выполняется, то:

x1…xi e L(Rj)

1. Удалить x1…xi из входной строки перейти к 3

Сколько символов входной строки использовать?

x1…xi e L(R)

x1…xj e L(R)

i != j

 ==

“Maximal munch” (не всегда, но лучше, чтоб оно проблем на создавало в языке)

К какому класс отнести лексему?

x1…xi e L(R)

x1…xi e L(Rj)  
x1…xi e L(Rk)

Keywords = ‘if’ + ‘else’ + …  
Identifiers = letter (letter + digit)\*

if e L(Keywords)  
if e L(Identifiers)

**Выбирать класс, заданный раньше**

Что делать, если не подходит ни одно правило?

Error = все строки, не описанные в лексической спецификации

Регулярные выражения – удобный инструмент для описания множеств строк

Для их использования в лексическом анализе необходимо:

* Избавиться от неопределенностей
* Обрабатывать ошибки

Хорошие алгоритмы:

* Используют один проход по входной строке
* Выполняют минимальное количество операций для каждого символа (table lookup)

**Конечные автоматы**

Регулярные выражения – спецификация.

Конечные автоматы – реализация.

Пример: (представь светофор)

Состояния: К, К+Ж, З, Мигающий З, Ж.

🡪

Сигналы таймера 🡪 Светофор 🡪

🡪

Конечный автомат:

* Алфавит входных сигналов **∑**
* Множество состояний **S**
* Начальное состояние **n**
* Множество заключительных состояний **F ( S**
* Множество переходов  
  Состояние 🡪 *сигнал* состояние

Переход

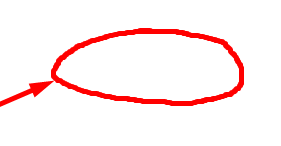
S1 🡪*a* S2

Означает:

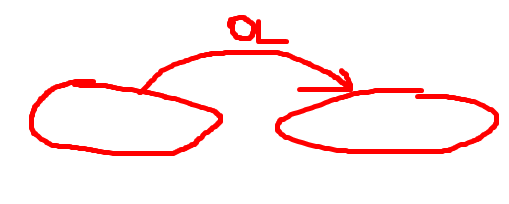
* Из состояния **S1** при поступлении входного сигнала **a** перейти в состояние **S2**

Если достигнут конец входной строки и автомат находится в заключительном состоянии, то автомат **допускает** строку.

Состояние 

Начальное состояние 

Заключительное состояние 

Переход

Автомат, допускающий только строку «1»



|  |  |
| --- | --- |
| Состояние | Входной сигнал |
| А | |1 |
| B | 1| |

Accept

|  |  |
| --- | --- |
| Состояние | Входной сигнал |
| А | |0 |

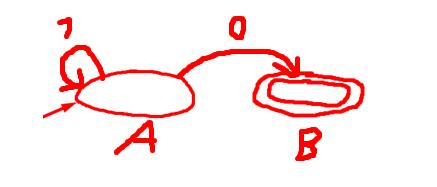
Reject

|  |  |
| --- | --- |
| Состояние | Входной сигнал |
| А | |10 |
| B | 1|0 |

Reject

Автомат, допускающий строку из произвольного количество 1 с последующим одиночным 0

Алфавит: {0, 1}



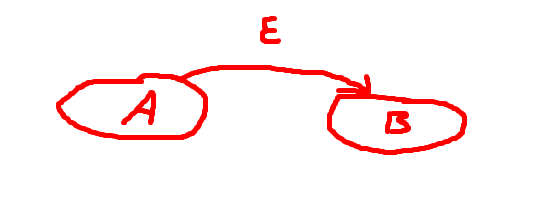
|  |  |
| --- | --- |
| Состояние | Входной сигнал |
| А | |110 |
| A | 1|10 |
| A | 11|0 |
| B | 110| |

Accept

|  |  |
| --- | --- |
| Состояние | Входной сигнал |
| А | |100 |
| A | 1|00 |
| B | 10|0 |

Reject

Особый тип переходов: e-переходы

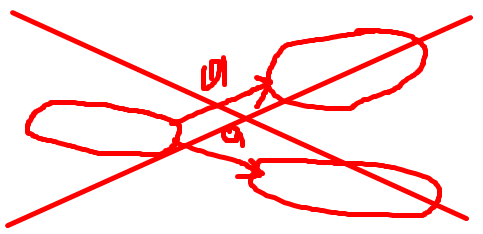


|  |  |
| --- | --- |
| Состояние | Входной сигнал |
| А | X1|X2X3 |
| B | X1|X2X3 |

*Детерминированные конечные автоматы (ДКА)*

ДКА:

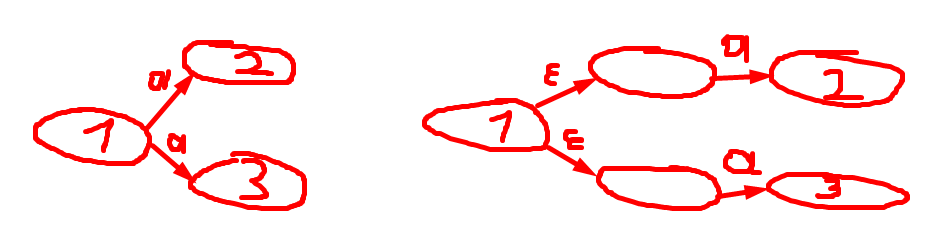
* Для каждой пары «состояние-сигнал» есть только один переход
* Нет e-переходов



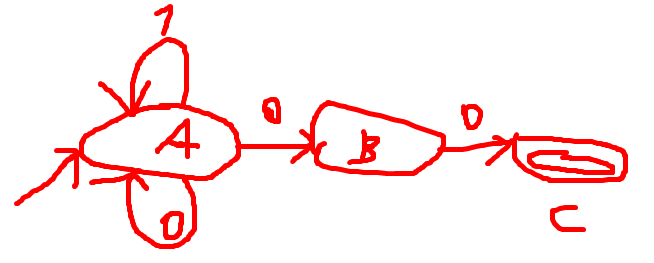
*Недетерминированные конечные автоматы (НКА)*

НКА:

* Для каждой пары «состояние-сигнал» может быть несколько переходов
* Могут быть e-переходы



**НКА допускает строку, если хотя бы один набор решений (выборов) приводит в заключительное состояние**



Входная строка: 1 0 0

Состояния: {A} {A,B} {A,B,C}

НКА и ДКА распознают одно и то же множество языков:

* Регулярные языки

ДКА работают быстрее:

* Не нужно выбирать между несколькими вариантами

НКА в общем случае компактнее:

* Экспоненциально

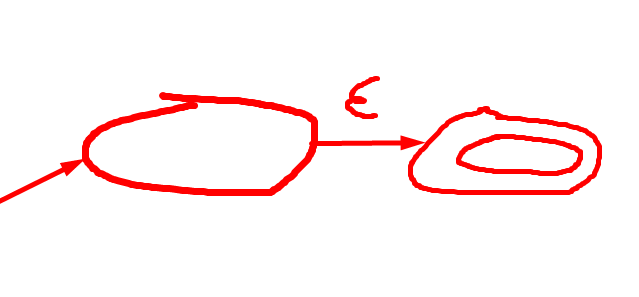
*Построение лексического анализатора*

Лексическая спецификация 🡪 Регулярные выражения 🡪 НКА 🡪 ДКА 🡪 Табличная реализация автомата

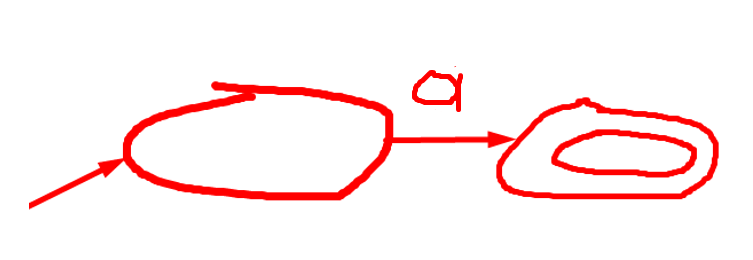
НКА для регулярного выражения М:



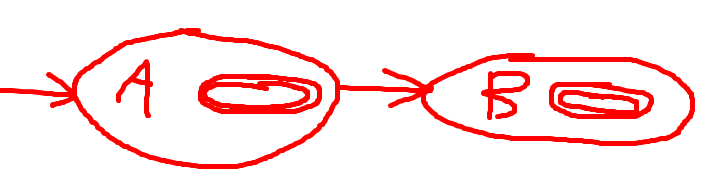
НКА для e:

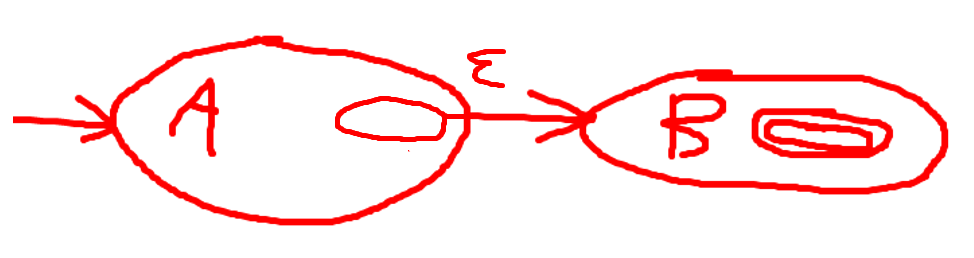


НКА для одиночного символа а:

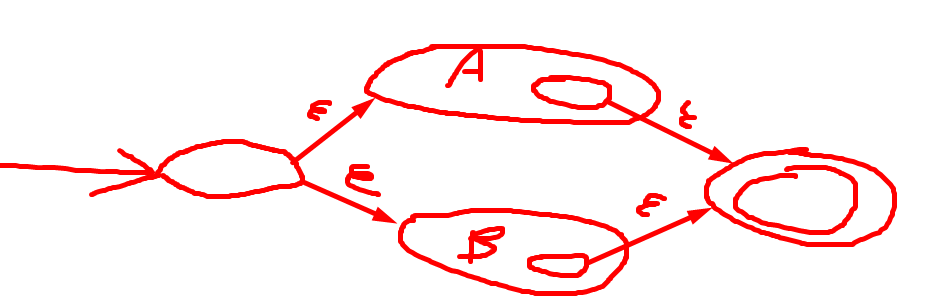


НКА для выражения AB:

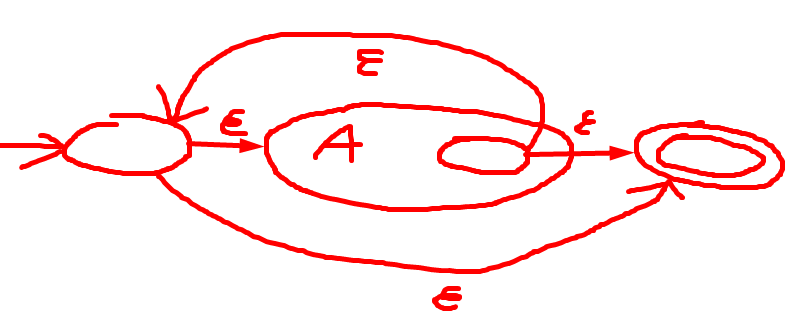




НКА для выражения A + B:

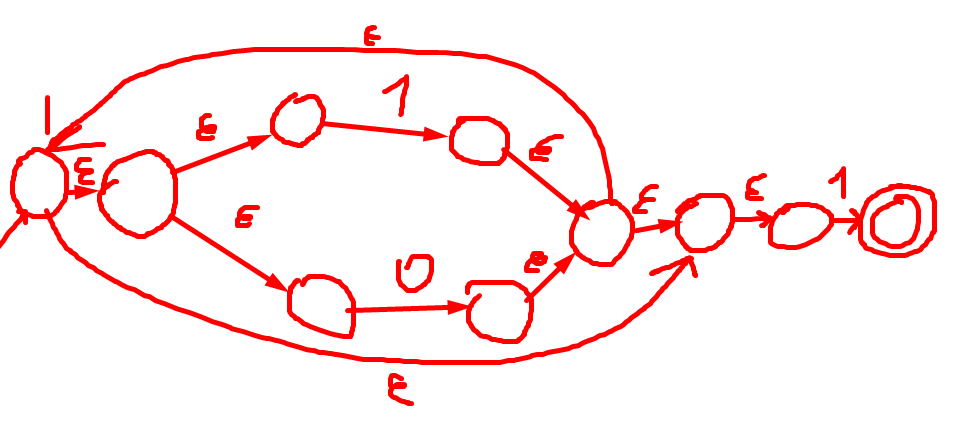


НКА для выражения А\*:

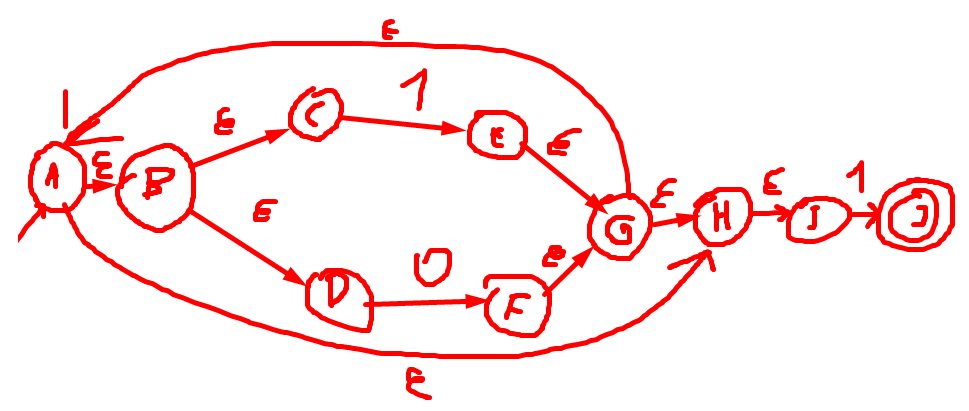


*Построение НКА для сложного регулярного построения*

(1+0)\*1



*e-замыкание*



е-closure(B) = {B, C, D}

e-closure(G) = {A, B, C, D, G, H, I}

Преобразование НКА в ДКА

НКА может одновременно находиться в нескольких состояниях

Сколько есть наборов (множеств) состояний, в которых может находиться НКА?

N состояний  
|S| <= N  
2^N – 1

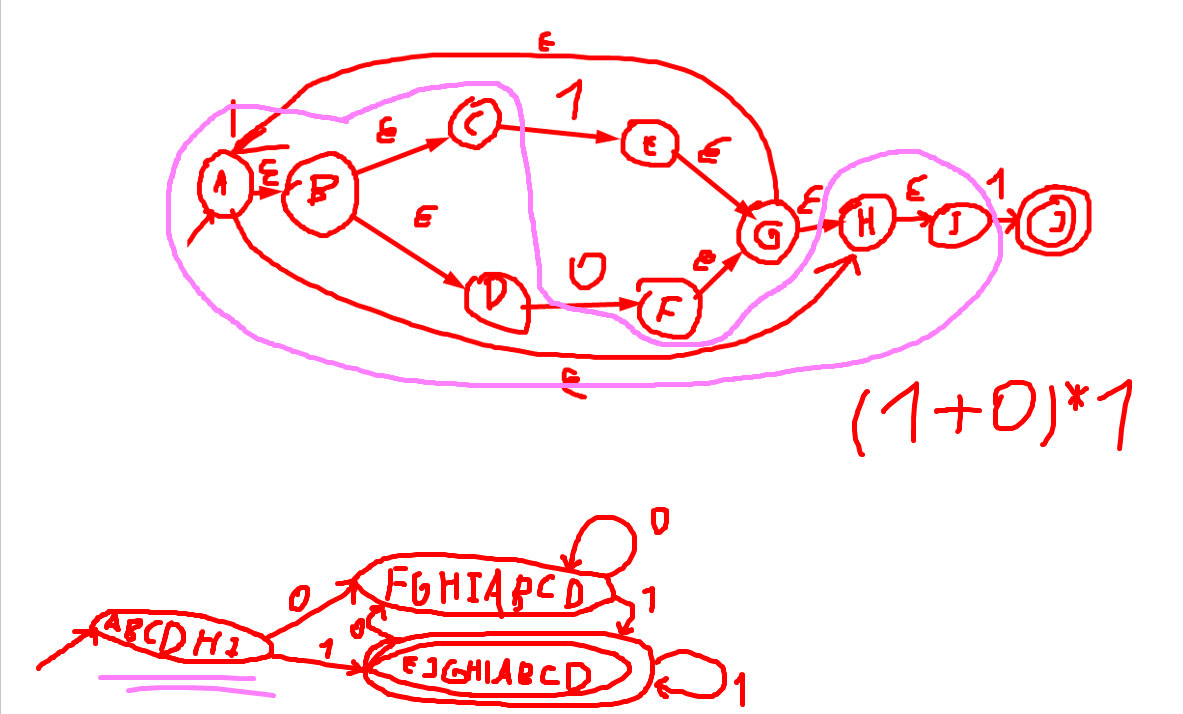
НКА  
Множество состояний S  
Начальное состояние s e S  
Заключительные состояния F ( S   
a(X) = {y|x e X ^ x 🡪a y}

e-clos

ДКА  
Множество состояний непустые подмножества S  
Начальное состояние e-clos(s)  
Заключительные состояния {X|X ^ F != ~~0~~} (~~0~~ - пустое множество)

X 🡪a Y

Y = e – clos(a(X))

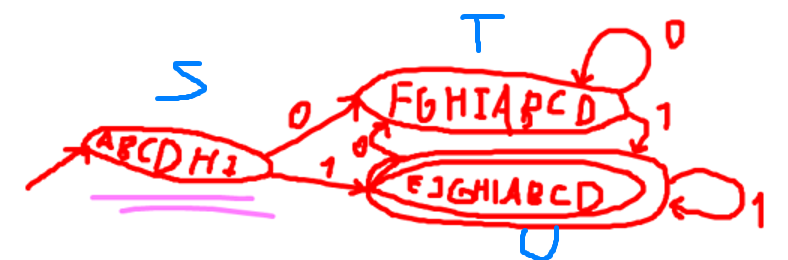


**Реализация конечного автомата**

*Табличная реализация ДКА*

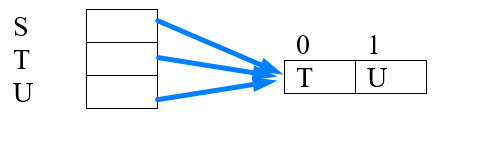
ДКА можно реализовать на основе 2-мерной матрицы Т:

* Два измерения:
  + Состояния
  + Символ из входного потока
* Каждому переходу Si 🡪a Sk ставится в соответствие элемент T[i,a] = k



|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| S | T | U |
| T | T | U |
| U | T | U |

i = 0;  
state = 0;  
while (input[i])  
{  
 state = A[state, input[i++]];  
}



**Введение в синтаксический анализ**

*Иерархия Хомского*

Формальные языки:

* Неограниченные
* Контекстно-зависимые
* Контекстно-свободные
* Регулярные

Регулярные языки:

* Самые ограниченные формальные языки
* Много применений

{()| i >= 0}

() ((1+2)\*3)

(())

((()))

*Синтаксический анализатор*

Токены 🡪 **[Синтаксический анализатор (парсер)]** 🡪 дерево разбора (parse tree) (или AST)

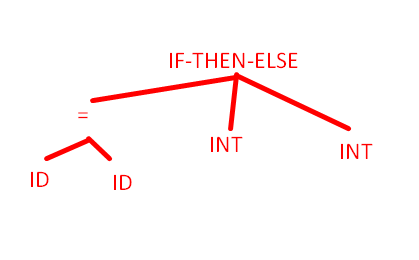
Язык:

if x = y then 1 else 2 fi

Входные данные для парсера:

IF ID = ID THEN INT ELSE INT FI

Результат работы парсера:



|  |  |  |
| --- | --- | --- |
| **Фаза компиляции** | **Входные данные** | **Выходные данные** |
| Лексический анализ | Строки символов | Токены |
| Синтаксический анализ | Токены | Дерево разбора (может быть не явным) |

Не все последовательности токенов являются программами…

Парсер должен различать корректные и некорректные последовательности токенов

Нужны:

* Язык описания корректных последовательностей токенов
* Метод проверки корректности последовательностей токенов

Языки программирования имеют рекурсивную структуру.

Например, STATEMENT – это if EXPR then STATEMENT else STATEMENT while EXPR do STATEMENT repeat STATEMENT until EXPR

Контекстно-свободные грамматики – удобный способ задания таких рекурсивных структур

Контекстно-свободная грамматика:

* Множество терминальных символов T
* Множество нетерминальных символов N
* Начальный символ S S e N
* Множество продукций   
  X 🡪 Y1…Yn  
  X e N  
  Yi e T ^ N ^ {e}

S 🡪 (S) N = {S}

S 🡪 e T = {(,)}

Продукции можно рассматривать как правила подстановки:

S 🡪 (S)

1. Начать со строки, состоящей из единственного стартового символа S
2. Заменить любой нетерминальный символ X на правую часть какой-либо из его продукций X 🡪 Y1…Yn
3. Повторять шаг 2, пока остаются нетерминальные символы

X1…XiXXi+1…Xn 🡪 X1…XiY1…YkXi+1Xn

Продукция: X 🡪 Y1…Yk

S 🡪 \_\_ 🡪 \_\_ 🡪 a0 🡪 a1 🡪 a2 🡪 … 🡪 an

a0 🡪\* an (за 0 или более шагов)

Пусть G – контекстно-свободная грамматика с начальным символом S.

Тогда язык L(G) – это:

…

Терминальные символы – символы, для которых в грамматике нет правил подстановки.

Появившись в строке, терминалы больше не могут быть ничем заменены

При написании транслятора терминалы – лексемы языка.

* Обычно, но не обязательно

EXPR 🡪 if EXPR then EXPR else EXPR fi

| while EXPR loop EXPR pool

| id

Простые арифметические выражения:

E 🡪 E + E

| E \* E

| (E)

| id

Идея контекстно-свободных грамматик – важный шаг. НО:

* Принадлежность строки определяется в виде «да»/«нет»; еще необходимо строить дерево разбора
* Необходимо обрабатывать ошибки на входных данных (программах)
* Нужна реализация контекстно-свободных грамматик

Способ записи грамматики имеет значение:

* Многие грамматики порождают один и тот же язык
* Инструменты чувствительны к способу записи грамматики

*Порождения (цепочки вывода)*

Порождение (derivation) – последовательность решений о применении продукций:

S 🡪 … 🡪 … 🡪 … 🡪 …

Порождение может быть представлено деревом:

* Стартовый символ – корень дерева.
* Для продукций X 🡪 Y1…Yn к узлу X добавляются дочерние узлы Y1,…,Yn

Грамматика:

E 🡪 E + E

| E \* E

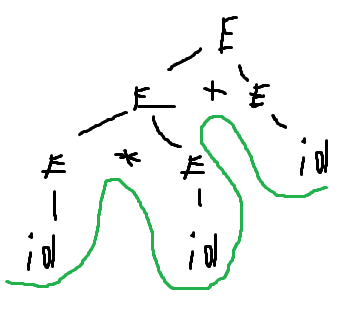
| (E)

| id

Строка:

id \* id + id

E  
🡪 E + E  
🡪 E \* E + E  
🡪 id \* E + E  
🡪 id \* id + E  
🡪 id \* id + id



Синтаксическое дерево состоит из:

* Терминалов – листьев
* Нетерминалов – промежуточных узлов

Последовательный обход листьев позволяет восстановить исходную строку

Показывает ассоциативность операций (в отличие от исходной строки)

Пример – левое порождение (left-most derivation):

* На каждом шаге заменяется «самый левый» нетерминал

Существует также правое порождение, где на каждом шаге производится замена «самого правого» нетерминала

Левому и правому порождениям соответствует одно и то же дерево разбора

Интерес представляет не только ответ на вопрос, верно ли, что s e L(G)

* Необходимо построить синтаксическое дерево для S

Порождение определяет дерево разбора

* Но одно и то же дерево разбора может иметь много порождений

Левые и правые порождения играют важную роль в синтаксическом анализе

*Неоднозначность*

E  
🡪 E + E  
🡪 E \* E + E  
🡪 id \* E + E  
🡪 id \* id + E  
🡪 id \* id + id

И

E  
🡪 E \* E  
🡪 E \* E + E  
🡪 id \* E + E  
🡪 id \* id + E  
🡪 id \* id + id

Неоднозначная грамматика – формальная грамматика, которая может породить некоторую строка более чем одним способом

* Т. е. для одной строки есть более одного дерева разбора
* Равносильно тому, что для строки существует более одного левого и правого порождения

Неоднозначность – это **ПЛОХО:**

* Значение некоторых программ может оставаться неопределенным
* Синтаксический анализ неоднозначной грамматики детерминированным анализатором затруднен
* Синтаксический анализ неоднозначной грамматики недетерминированным анализатором – это **МЕДЛЕННО**

Грамматики некоторых языков программирования неоднозначны

При разборе таких языков необходимо учитывать семантическую информацию для выбора правильного разбора

C – неоднозначный язык

x \* y;

Оператор-выражение, в котором x умножается на y, а результат игнорируется

…

Задача определения, является ли грамматика неоднозначной, алгоритмически неразрешима

Большинство конструкций, требующих синтаксического анализа, может быть распознано однозначными грамматиками

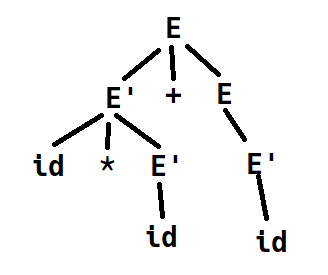
Некоторые неоднозначные грамматики могут быть преобразованы в однозначные

Существует несколько способов устранения неоднозначности

Самый простой способ – переписать грамматику:

E 🡪 E’ + E | E’   
E ‘ 🡪 id \* E’  
 | id  
 | (E) \* E’  
 | (E)

id \* id + id



E 🡪 E’ + E 🡪 E’ + E’ + E 🡪 … 🡪 E’ + E’ + E’ + … + E’ + E’

E’ 🡪 id \* E’ 🡪 id \* id \* E’ 🡪 … 🡪 id \* id \* id \* … \* id \* id

E 🡪 if E then E

| if E then E else E

| OTHER

else (как правило) относится к ближайшему if, имеющему только ветвь then

E 🡪 MIF // matched if  
 | UIF // unmatched if

MIF 🡪 if E then MIF else MIF

| OTHER

UIF 🡪 if E then E

| if E then MIF else UIF

(неоднозначные грамматики)   
Невозможно автоматически преобразовать в однозначные

При осторожном использовании неоднозначности позволяют упростить грамматику:

* Иногда позволяют записать продукции в более естественной форме
* В этом случае нужны механизмы разрешения неоднозначностей

Вместо переписывания грамматики:

* Используется более естественная (неоднозначная) грамматика
* Вместе с описаниями, позволяющими разрешить неоднозначность.

Большинство инструментов позволяет использовать описания, задающие порядок выполнения и ассоциативность.

*Ассоциативность*

В математике:

* Свойств бинарной операции, при котором результат ее последовательного применения не зависит от расстановки скобок

В программировании:

* Свойство операций, позволяющее восстановить последовательность их выполнения при:
  + Отсутствии явных указаний на очередность
  + Равном приоритете

Приоритеты Left + / Left \*

*Обработка ошибок*

Задачи компилятора:

* Выявлять некорректные программы
* Транслировать корректные программы

Существуют разные виды ошибок

|  |  |  |
| --- | --- | --- |
| Вид ошибки | Пример | Обнаруживает |
| Лексическая | … $ … | Лексер |
| Синтаксическая | … x \*% … | Парсер |
| Семантическая | int x;  y = x(3); | Контроль типов |
| Логическая | Любая программа | Тестировщик или пользователь |

Простейший способ – прерывать компиляцию при обнаружении первой ошибки

Обработчик ошибок парсера:

* Должен ясно и точно сообщать о наличии ошибок
* Должен обеспечивать быстрое восстановление после ошибки
* Не должен существенно замедлять компиляцию корректных программ

Восстановление в режиме паники  
Восстановление на уровне фразы  
Продукции ошибок  
Автоматическая локальная или глобальная коррекция

*Восстановление в режиме паники*

Самый простой и популярный метод

При обнаружении ошибки:

* Пропускать токены, пока не встретится токен определенного назначения
* Продолжить с этого места

Основан на поиске **синхронизирующих** токенов:

* Обычно токены, завещающие операцию или выражение

Пример:

(1 + + 2) + 3

Восстановление:

* Пропустить все токены до следующего целого числа

Bison: использовать специальный терминальный символ error для указания, сколько токенов пропускать

E 🡪 int | E + E | (E) | error int | (error)

*Восстановление на уровне фразы*

При обнаружении ошибки выполняется локальная коррекция входного потока.

* Парсер «притворяется», что коррекция выполнена, но анализируемый текст все равно считается ошибочным.

Пример:

* Перед точкой с запятой, разделяющей операторы, автоматически закрывать все круглые скобки.

*Продукции ошибок*

Известные распространенные синтаксические ошибки включаются в грамматику

Пример:

* Для записи 5 х вместо 5 \* х
* Включить в грамматику продукцию E 🡪 … | E E

Недостаток:  
грамматика становится сложнее

*Автоматическая коррекция ошибок*

Идея: находить (подбирать) похожую (близкую) программу, не содержащую ошибки

* Использовать вставку/удаление токенов
* Использовать полный перебор

Пример: PL/C – компилятор языка PL/I с поддержкой автоматической коррекции

*Обработка ошибок*

Прошлое:

* Длинный цикл компиляции/запуска программы (часы, дни)
* Нужно находить за один цикл как можно больше ошибок

Настоящее:

* Короткий цикл компиляции/запуска
* Тенденция к исправлению одной ошибки за каждый цикл
* Сложные механизмы обработки ошибок менее востребованы

*Абстрактные синтаксические деревья*

Парсер отслеживает порождение последовательности токенов

Но остальным частям компилятора требуется структурное представление программы

Абстрактные синтаксические деревья

* Похожи на деревья разбора, но не содержат некоторых деталей.
* Сокращение: AST.

Грамматика:

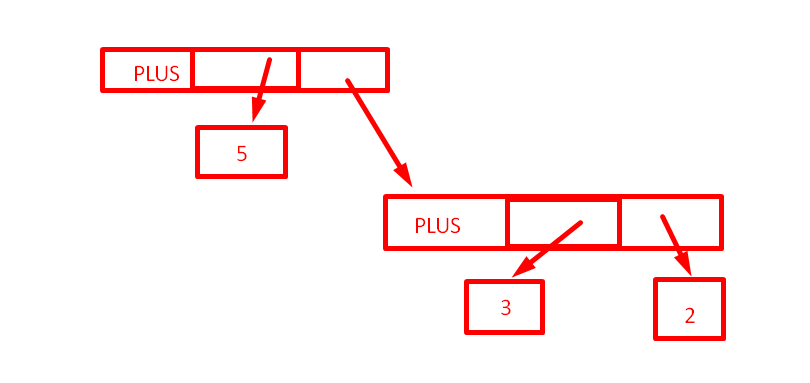
E 🡪 int | (E) | E + E

Строка:

5 + (2 + 3)

После лексического анализа:

int5 ‘+’ ‘(‘ int2 ‘+’ int3 ‘)’



*Метод рекурсивного спуска*

Нисходящие:

* Метод рекурсивного спуска
* LL-анализ

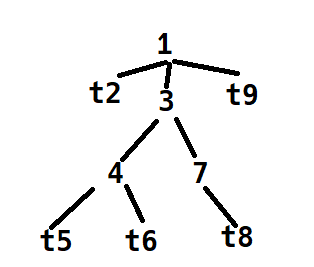
Восходящие:

* LR-анализ
* GLR-анализаторы

Дерево разбора стоится:

* Начиная с вершины (корня)
* Слева направо

Терминалы появляются в порядке их появления в потоке токенов



*Parsing Expression Grammar (PEG)*

Метод синтаксического разбора, похожий на метод рекурсивного спуска

Отличия:

* Продукция нетерминалов получают приоритеты в порядке следования в записи грамматики
* Это обозначается разделением продукций символом \ вместо |

Грамматика:

E 🡪 T | T + E  
T 🡪 int | int \* T | (E)

Поток токенов:

( int5 )

Разбор начинается со стартового символа E

* Поочередно перебираются все правила для символа E

E 🡪 T | T + E

T 🡪 int | int \* T | (E)

TOKEN – перечисляемый тип

* Значения: INT, OPEN, CLOSE, PLUS, TIMES…

next – глобальная переменная, указывающая на следующий токен из входной последовательности.

Функции, которые проверяют соответствие входной последовательности:

* Токену-терминалу:

bool term(TOKEN tok) {return \*next++ == tok; }

* n-й продукции S:

bool Sn() {…}

* всем продукциям S:

bool S() {…}

Для продукции E 🡪 T:  
bool E1() { return T(); }

Для продукции E 🡪 T + E;  
bool E2() {return T() && term(PLUS) && E(); }

Для всех продукций E:  
bool E()  
{  
 TOKEN \*save = next;  
 return (next = save, E1())  
 || (next = save, E2());  
}

Запуск парсера:

* инициализация next (должна указывать на первый токен)
* вызвать E()

Удобно реализовывать вручную

*Левая рекурсия*

Продукция S 🡪 S a:  
bool S1() { return S() && term(a); }  
bool S() { return S1(); }

Вызов S() зацикливается.

Леворекурсивная грамматика – грамматика, в которой есть такой нетерминал S, что:

S 🡪+ Sa для некоторого a

**S 🡪 Sa 🡪 Saa 🡪 … 🡪 Sa…a**

Грамматика:

S 🡪 Sa | B  
S 🡪 Sa 🡪 Saa 🡪 Saaa 🡪 … 🡪 Sa…a 🡪 Ba…a

S порождает строки, начинающиеся с B, за которым следует произвольное количество a.

Можно переписать грамматику с использованием правой рекурсии.

S 🡪 BS’  
S’ 🡪 aS’ | e

S 🡪 BS’ 🡪 BaS’ 🡪 BaaS’ 🡪 … 🡪 Ba…aS’ 🡪 Ba…a

В общем случае:

S 🡪 Sa1 | … | San | B1 | … | Bm

Все порождаемые строки начинаются с B1, …, Bm и продолжаются несколькими a1, …, an

Можно переписать как:

S 🡪 B1S’ | … | BmS’  
S’ 🡪 a1S’ | … | anS’ | e

Грамматика

S 🡪 Aa | b  
A 🡪 SB

тоже леворекурсивна, т.к.  
S 🡪+ SBa

Такую левую рекурсию тоже можно устранить  
Общий алгоритм – в «Книге Дракона»

*Метод рекурсивного спуска*

Простая и универсальная стратегия синтаксического разбора

Необходимо избегать левой рекурсии…  
… но это может быть сделано автоматически.

Используется в разработке компиляторов:

* Например, gcc.

**Предиктивный анализ**

Как и метод рекурсивного спуска, является нисходящим методом…  
… но может «предсказывать» какую продукцию лучше выбрать:

* Заглядывая на несколько токенов вперед
* Не возвращаясь назад (backtracking)

Предиктивные анализаторы работают с LL(k)-грамматиками

**L**eft-to-right scan **L**L(k)  
**L**eft-most derivation L**L**(k)  
**k** tokens lookahead LL(**k**)

Метод рекурсивного спуска:

* На каждом этапе осуществляется выбор из множества продукций
* При неправильном выборе осуществляется возврат на предыдущий шаг.

LL(k):

* На каждом этапе существует не более 1 подходящей продукции

wAB Следующий токен: t

Одна продукция:  
A 🡪 a

Грамматика:

E 🡪 T + E | T  
T 🡪 int | int \* T | (E)

Сложно «предсказывать», т.к.:

* Для T две продукции начинаются с int
* Для Е вообще непонятно, как предсказывать:  
  продукции начинаются с нетерминалов

Нужно выполнить  
**левую факторизацию** грамматики

E 🡪 T + E | T  
T 🡪 int | int \* T | (E)

E 🡪 T X  
X 🡪 + E | e

T 🡪 int Y | (E)  
Y 🡪 \* T | e

Грамматика:

E 🡪 T X X 🡪 + E | e

T 🡪 int Y | (E) Y 🡪 \* T | e

Таблица LL(1)-разбора

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | int | \* | + | ( | ) | $ |
| E | T X |  |  | T X |  |  |
| X |  |  | + E |  | e | e |
| T | int Y |  |  | (E) |  |  |
| Y |  | \* T | e |  | e | e |

LL(1)-разбор:

* Для крайнего слева нетерминала S
* Просматривается следующий токен a
* И выбирается продукция [S, a]

Стек содержит крону (yield) или границу (frontier) дерева разбора:

* Нетерминалы, которые предстоит раскрыть
* Терминалы, которые предстоит сопоставить с потоком токенов
* Вершина стека – крайний слева подлежащий обработке терминал/нетерминал

Reject:

* При достижении ошибочного состояния

Accept:

* Если достигнут конец строки токенов и стек пуст

|  |  |  |
| --- | --- | --- |
| Стек | Поток токенов | Действие |
| E $ | Int \* int $ | T X |
| T X $ | Int \* int $ | Int Y |
| int Y X $ | Int \* int $ | Терминал |
| Y X $ | \* int $ | \* T |
| \* T X $ | \* int $ | Терминал |
| T X $ | int $ | int Y |
| Int Y X $ | int $ | Терминал |
| Y X $ | $ | e |
| X $ | $ | e |
| $ | $ | ACCEPT |

*FIRST и FOLLOW*

Нетерминал А, продукция А 🡪 a, токен t.

T[A, t] = a в двух случаях:

* Если a 🡪 \* tB
  + a может порождать t на первой позиции
  + Говорят, что t First(a)
* Если A 🡪 a и a 🡪 \* ε и S 🡪\* BAtb
  + Случай, когда вершина стека – A, следующий токен – t, но A не порождает t
  + В этом случае надо избавиться от A (путем порождения, превращающего A в ε)
    - Возможно, если t следует за A хотя бы в одном порождении
  + Говорят, t Follow(a)

*FIRST*

Определение:  
First(X) = {t | X 🡪\* ta} ^ {ε | X 🡪\* ε}

Алгоритм:

1. First(t) = {t}
2. ε First(X)
   1. Если X 🡪 ε
   2. Если X 🡪 A1…An и ε First(Ai)
3. First(a) C First(X)  
   если X 🡪 A1…An a и ε First(Ai)

Грамматика:

E 🡪 T X X 🡪 + E | e

T 🡪 int Y | (E) Y 🡪 \* T | e

First(+) = {+}  
First(\*) = {\*}  
…

First(E) ) First(T)  
First(T) = {(, int}

First(E) = First(T)  
First(T) = {(, int}  
First(X) = {+, ε}  
First(Y) = {\*, ε}

*FOLLOW*

Определение:

Follow(X) = {t | S 🡪\* BXtb}

Идеи вычисления:

1. Если T 🡪 AB, то First(B) C Follow(A)  
   Follow(T) C Follow(B)
   1. Если B 🡪\* ε, то Follow(T) C Follow(A)
2. Если S – стартовый символ, то $ Follow(S)

Алгоритм

1. $ Follow(S)
2. First(B) – {ε} C Follow(X)
   1. Для каждой продукции вида A 🡪 aXB
3. Follow(A) C Follow(X)
   1. Для каждой продукции вида A 🡪aXB, где ε First(B)

Грамматика:

E 🡪 T X X 🡪 + E | e

T 🡪 int Y | (E) Y 🡪 \* T | e

Follow(E) = {$, )

Follow(X) C Follow(E)  
Follow(E) C Follow(X)

Follow(E) = Follow(X)  
Follow(X) = {$, )

Follow(T) = {+…  
Follow(E) C Follow(T)  
Follow(T) = {+, $, )}

Follow(Y) C Follow(T)  
Follow(T) C Follow(Y)

Follow(T) = Follow(Y)  
Follow(Y) = {+, $, )}

Follow(‘(‘) = {(, int}  
Follow(‘)’) = {+, $, )}  
Follow(‘+’) = {(, int}  
Follow(‘\*’) = {(, int}  
Follow(int) = {\*, +, $, )}

*Построение таблицы LL(1)-разбора*

Для каждой продукции A 🡪 a:

* Для каждого терминала t First(a):
  + T[A, t] = a
* Если ε First(a), для t Follow(A):
  + T[A, t] = a
* Если ε First(a) и $ Follow(A):
  + T[A, $] = a

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ( | ) | + | \* | int | $ |
| E | T X |  |  |  | T X |  |
| T | (E) |  |  |  | int Y |  |
| X |  | ε | + E |  |  | ε |
| Y |  | ε | ε | \* T |  | ε |

Грамматика:

S 🡪 Sa | b

First(S) = {b}

Follow(S) = {$, a}

|  |  |  |  |
| --- | --- | --- | --- |
|  | a | b | $ |
| S |  | b  Sa |  |

А работать не будет(((

[ТУТ БУДЕТ КОНЕЦ]

Умные мысли

Лексема – подстрока, фрагмент исходного кода программы.  
Токен – представление лексемы в памяти транслятора.

е-замыкание – множество состояний НКА, достижимых из заданного за 0 или более е-переходов.

При обсуждении грамматик:

1. Заглавные латинские буквы – нетерминалы
2. Строчные латинские буквы – терминалы
3. Буквы греческого алфавита – произвольные последовательности из терминалов и/или нетерминалов

КОНЕЦ умных мыслей