5. Лексический анализ

- Основные задачи лексического анализа
- Регулярные выражения
- Использование конечных автоматов для лексического анализа
- Утилита Lex
- Использование механизма регулярных выражений .NET

Лекция 5. Лексический анализ

В этой лекции рассматриваются следующие вопросы:

- Основные задачи лексического анализа
- Регулярные выражения
- Использование конечных автоматов для лексического анализа
- Утилита Lex
- Использование механизма регулярных выражений .NET

Лексический анализ

- Задача лексического анализа – разбиение исходной программы на лексемы
- Типичные лексические классы:
  - Ключевые слова (if, switch, for…)
  - Идентификаторы
  - Строковые литералы
  - Числовые константы

Лексический анализ

Исходное текстовое представление программы совсем не очень для работы компилятора, поэтому во время анализа программа прежде всего разбивается на последовательность строк, или, как принято говорить, *лексем (lexeme)*. Множество лексем разбивается на непересекающиеся подмножества (лексические классы). Лексемы попадают в один лексический класс, если они неразличимы с точки зрения синтаксического анализатора. Например, во время синтаксического анализа все идентификаторы можно считать одинаковыми.

Размеры лексических классов различны. Например, лексический класс идентификаторов, вообще говоря, бесконечен. С другой стороны, есть лексические классы, состоящие только из одной лексемы, например, подмножество, состоящее из лексемы if. В большинстве языков программирования имеются следующие лексические классы: ключевые слова (по одному на каждое ключевое слово), идентификаторы, строковые литералы, числовые константы. Каждому подмножеству сопоставляется некоторое число, называемое *идентификатором лексического класса (token)* или, короче, *лексическим классом*.

Пример. Рассмотрим оператор языка Pascal *const pi = 3.1416;* Этот оператор состоит из следующих лексем:

- *const* – лексический класс Const_LC
- *pi* – лексический класс Identifier_LC
- = – лексический класс Relation_LC
- *3.1416* – лексический класс Number_LC
- *;* – лексический класс Semicolon_LC

Лексический анализ различных языков программирования

- Особенности некоторых языков могут существенно затруднять лексический анализ:
  - Фиксированный формат программы (Фортран, PL/I, Кобол)
  - Трактовка пробела как незначащего символа (Фортран, Алгол 68)
  - Использование ключевых слов как идентификаторов (PL/I)

Лексический анализ различных языков программирования

Некоторые языки обладают особенностями, существенно затрудняющими лексический анализа. Такие языки, как Фортран и Кобол, требуют размещения конструкций языка в фиксированных позициях входной строки. Такое размещение лексем могло быть очень важным при выснении корректности программы. Например, при переносе строки в Коболе необходимо поставить специальный символ в 6-й колонке, иначе следующая строка будет разобрана неправильно. Основной тенденцией современных языков программирования является свободное размещение текста программы.

От одного языка к другому варьируются правила использования символов языка, в частности, пробелов. В некоторых языках, таких как Алгол 68 и Фортран, пробелы являются значащими только в строковых литералах. Рассмотрим популярный пример, иллюстрирующий потенциальную сложность распознавания лексем в Фортране. В операторе `DO 5 I = 1.25` мы не можем определить, что DO не является ключевым словом до тех пор, пока не встретим десятичную точку.

С другой стороны, в операторе `DO 5 I = 1,25` мы имеем семь лексем: ключевое слово DO, метку 5, идентификатор I, оператор =, константу 1, запятую и константу 25. Причем, до тех пор пока мы не встретим запятую, мы не можем быть уверены в том, что DO – это ключевое слово. Чтобы как-то разрешить эту ситуацию, Fortran 77 позволяет использовать необязательную запятую между меткой и индексом DO оператора. Использование такой запятой позвляет сделать DO оператор понятнее и более читабельным.

В большинстве современных языков программирования ключевые слова являются зарезервированными, т.е. их смысл предопределен и не может быть изменен пользователем. Если ключевые слова не являются зарезервированными, то лексический анализ должен уметь различать ключевые слова и определенные пользователем идентификаторы. Естественно, что это сильно затрудняет лексический анализ; например, в PL/I вполне легален следующий оператор:

```
IF THEN THEN THEN  = ELSE; ELSE ELSE = THEN;
```

При разборе такого оператора необходимо постоянно переключаться с режима "THEN, ELSE как ключевые слова" на трактовку "THEN, ELSE как идентификаторы", и обратно.

Атрибуты лексем

- Атрибуты – это способ хранения дополнительной информации о лексеме (например, конкретное значение константы, имя идентификатора и т.д.)
- Достаточно часто атрибуты представляют собой ссылки в специальные таблицы, например, таблицу представлений, таблицу идентификаторов и т.п.

Атрибуты лексем

Поскольку весьма часты ситуации, в которых более чем одна лексема принадлежит одному лексическому классу, лексический анализатор должен предоставлять дополнительную информацию о том, какая конкретная лексема была выделена. Например, в лексический класс Number_LC попадет и строка 1, и строка 0, однако последующим этапам компилятора (скажем, кодогенератору) было бы полезно знать конкретное значение константы в исходной программе. Такую информацию можно записывать в *атрибуты* лексем; обычно лексема имеет только один атрибут – ссылку в некоторую таблицу с дополнительной информацией. В целях диагностики мы можем также сохранить номера строк начала и конца этой лексемы в исходной программе.

Пример. Рассмотрим следующий фрагмент программы на C#:

```
bool c; int a=1,b=2;
c = a>b>>2;
```

Последний оператор порождает следующую последовательность лексем и их атрибутов:
<Identifier_LC, указатель в таблицу на *c*>
<AssignOP_LC, >
<Identifier_LC, указатель в таблицу на *a*>
<GreaterThanOP_LC, >
<Identifier_LC, указатель в таблицу на *b*>
<RightShiftOP_LC, >
<Number_LC, указатель в таблицу на *2*>

Будем считать, что у нас определен тип, соответствующий указателю в эту таблицу – ReprInd, и тип, служащий для представления позиции в исходном файле – FilePos. В этом случае мы можем полностью определить лексему следующим образом:

```
struct LEXEME {
   ushort LexClass;
   ReprInd ReprTabPtr;
   FilePos beg;
   FilePos end;
}
```

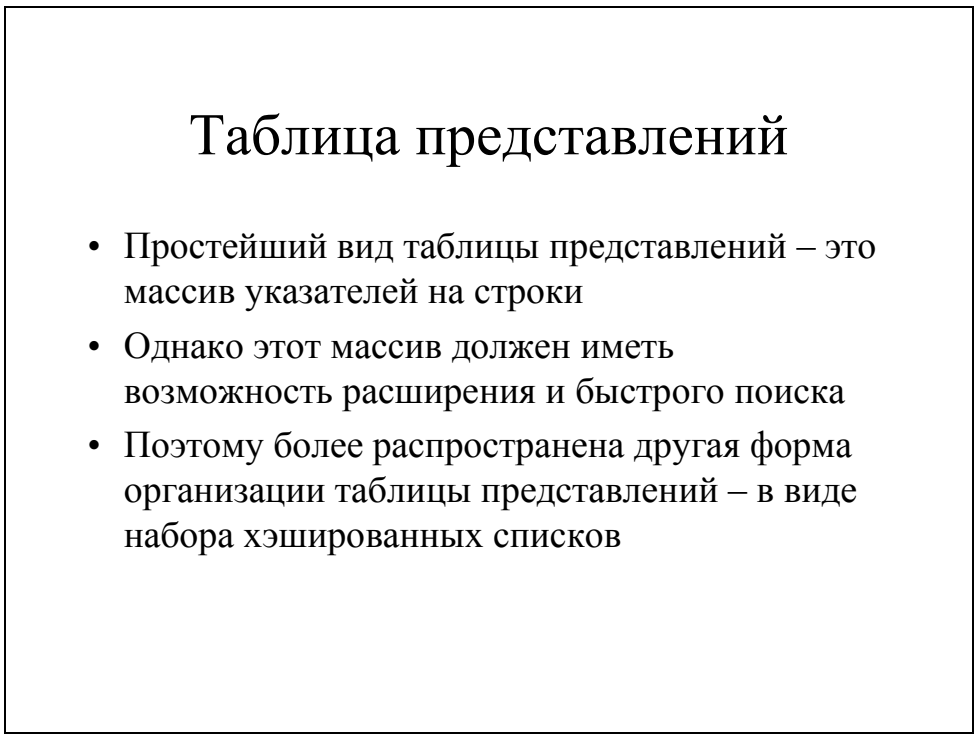


Таблица представлений

В таблице представлений хранится по одному экземпляру всех *внешних представлений* идентификаторов (и, возможно, также для всех констант). Затем идентификаторы заменяются на ссылку в эту таблицу – этот процесс называется *свертыванием*.

Одна из простейших форм организации таблицы – это массив указателей на строки. Однако при таком подходе замедляются два основных процесса, связанных с таблицей представлений: поиск идентификатора в таблице и добавление нового элемента. При этом поиск идентификатора в таблице является, наверное, самой массовой задачей в процессе компиляции, так как выполняется для каждого использующего вхождения идентификатора. Поэтому хотелось бы добиться максимального быстродействия для этой операции.

Поэтому более распространена другая форма организации таблицы представлений – в виде набора хэшированных списков. Для этого выбирается некоторая *хэш-функция* (в русской литературе иногда также называемая функцией расстановки), выдающая по данному идентификатору некоторое число от 0 до $H–1$, где H – константа, называемая длиной оглавления. Затем все идентификаторы с одинаковым хэш-значением связываются в список. Таким образом, для того, чтобы проверить, встречался ли уже новый идентификатор в программе или нет, достаточно сравнить его только с идентификаторами из таблицы представлений, имеющими одинаковое хэш-значение.

Замечено, что большинство использований идентификатора находится недалеко от места его описания, поэтому рекомендуется добавлять новые идентификаторы в начало хэш-списка, а не в конец. Это повышает скорость поиска, а также упрощает поддержку реализации стандартных правил видимости в языках с блочной структурой. Например, перед входом в блок можно запоминать текущее состояние хэш-таблицы, а затем при поиске идентификатора внутри данного блока считать активным идентификатором первую переменную с данным именем, встреченную в хэш-таблице. Затем при выходе из блока необходимо восстанавливать предыдущее состояние хэш-таблицы.

Хэш-функции

- Хэш-функция должна равномерно распределять результаты хэширования и зависить от всех символов идентификатора
- Система классов .NET содержит специальный класс HashTable, с помощью которого легко реализовать функциональность хэш-таблиц
- Для настройки хэш-таблицы на необходимый тип данных нужно реализовать методы GetHashCode и Equals

Хэш-функции

Хэш-функция должна равномерно распределять идентификаторы по таблице представлений. Для этого желательно, чтобы хэш-функция зависела от всех символов идентификатора. Например, для идентификторов длины n удовлетворительным вариантом хэш-функции является следующий: $Hash\ (id) = (id_0 + \ldots + id_{n-1}) / H$, где H – длина оглавления хэш-списка.

Так как хэширование используется в самых разнообразных приложениях, система классов .NET содержит специальный класс HashTable, с помощью которого легко реализовать функциональность хэш-функций и хэш-таблиц. Объекты, хранящиеся в HashTable должны явно переопределять методы GetHashCode и Equals.

При работе с идентификаторами достаточно воспользоваться методом String.GetHashCode, реализующим чувствительную к регистру хэш-функцию для строк, и String.Equals, выдающую true при равенстве строк (а не в случае совпадения указателей на объекты, как по умолчанию реализовано в Object.Equals).

Использование грамматик для лексического анализа

- Лексические свойства языков обычно носят достаточно регулярную структуру, поэтому для задания правил лексического анализа можно использовать (праволинейные) грамматики:

  letter => 'a'..'z' | 'A'..'Z' | '_'
  digit => '0'..'9'
  ident => letter | letter tail
  tail => letter | digit | letter tail | digit tail

- Однако на практике чаще используется эквивалентный праволинейным грамматикам механизм регулярных выражений

Использование грамматик для лексического анализа

Нетрудно заметить, что большинство распознаваемых лексем носят четко заданную структуру и потому возникает естественное желание применить к задаче лексического анализа теорию языков, т.е. описать с помощью какого-либо формализма характер цепочек, принимаемых на вход, а затем автоматически сгенерировать по этому описанию лексический анализатор.

Действительно, легко выписать, например, праволинейную грамматику для распознавания идентификаторов:

letter → *'a'..'z' | 'A'..'Z' | '_'*
digit → *'0'..'9'*
ident → *letter | letter tail*
tail → *letter | digit | letter tail | digit tail*

Как мы уже видели, по такой грамматике можно сгенерировать конечный автомат, который распознавал бы правильно сформированные идентификаторы.

Однако исторически сложилось, что для описания лексических свойств чаще используется другой формализм – регулярные выражения, к рассмотрению которых мы сейчас и перейдем.

Регулярные выражения

- Регулярное выражение над алфавитом A может быть задано с помощью следующих правил:
  - Пустое множество и множество, состоящее только из пустой строки, являются регулярными выражениями
  - Любой одиночный символ *a* из A является регулярным выражением
  - Для любых регулярных выражений P и Q следующие множества также являются регулярными выражениями:
    - PQ (Q следует за P)
    - P|Q (P или Q)
    - P^* (нуль или более экземпляров P)

Регулярные выражения

Определим следующие операции над множествами:

- PQ = {x| x=yz, $x \in P$, $y \in Q$ }
- $P^k = \begin{cases} \{\varepsilon\}, \text{если } k = 0 \\ P^{k-1}P, \text{если } k > 0 \end{cases}$
- $P^* = \bigcup_{i=0}^{\infty} P^i$

Тогда регулярные выражения над алфавитом T и языки, представляемые ими, могут быть определены следующим образом:

- Символ Λ, представляющий пустое множество, является регулярным выражением.
- ε является регулярным выражением и представляет множество *{ε}*.
- Для каждого символа $a \in T$ *a* является регулярным выражением и представляет множество *{a}*.
- Если p и q – регулярные выражения, представляющие множества P и Q, то *(pq)*, *(p+q)* и *(p*)* являются регулярными выражениями, представляющими множества PQ, $P \cup Q$ и P^* соответственно.

Отметим, что в этом определении подразумевается следующая система приоритетов: знак * обладает наивысшим приоритетом, за ним следует символ конкатенации, за которым следует символ |. Приоритеты можно изменять с помощью использования скобок.

Пример. Регулярное выражение *1(0+1)*1+1* представляет множество цепочек, начинающихся и заканчивающихся символом 1.

Упомянем без доказательства, что регулярные выражения эквивалентны праволинейным грамматикам. Таким образом, регулярным выражениям также соответствует естественный класс распознавателей в виде конечных автоматов.

Пример. Рассмотренная выше грамматика для идентификатора может быть записана с помощью следующего регулярного выражения: *Letter (Letter | Digit)**.

Построение лексического анализатора по регулярному выражению

- По регулярным выражениям легко написать лексический анализатор вручную (далее приведен пример разбора идентификатора):

```
if (Char.IsLetter (src [pos]) || src [pos] == '_')
{
   int fst = pos;
   do
      ++ pos;
   while (pos!=src.Length &&
     (Char.IsLetterOrDigit(src[pos]) || src [pos]=='_'));
   string name = src.Substring (fst, pos - fst);
   ...
```

Построение лексического анализатора по регулярному выражению

По имеющемуся регулярному выражению легко написать лексический анализатор вручную. Ниже приведен пример лексического анализа идентификатора, взятый из демонстрационного компилятора Cь.

```
if (Char.IsLetter (src [pos]) || src [pos] == '_')
{
   int fst = pos;
   do
      ++ pos;
   while (pos != src.Length &&
       (Char.IsLetterOrDigit (src [pos]) || src [pos]=='_'));
   string name = src.Substring (fst, pos - fst);
   object tag = keys_table [name];
   if (tag != null)
      return new Token.Single (new Coor (fname, fst, pos), (Token.Tag) tag);
   return new Token.Ident (new Coor (fname, fst, pos), name);
}
```

В этом фрагменте производятся следующие действия:

- анализ первого символа (буква или символ подчеркивания?)
- продвижение вперед по исходной строке, покуда мы встречаем буквы, цифры или символ подчеркивания
- проверка, не является ли разобранный идентификатор ключевым словом?
- если это действительно ключевое слово, то выдается соответствующий лексический класс (ключевое слово, Single), вместе с привязкой к исходному тексту и точным значением ключевого слова
- если это не ключевое слово, то это идентификатор, который и выдается вместе с привязкой к исходному тексту и его именем.

Lex

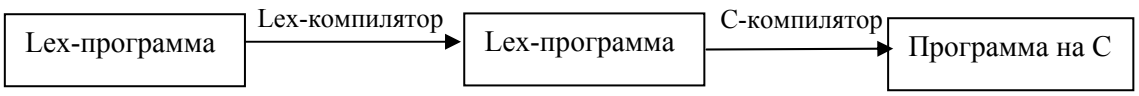
- Существует целый ряд инструментов для создания лексических анализаторов
- Большинство подобных средств основывается на регулярных выражениях
- Общая схема работы Lex заключается в преобразовании Lex-программы (записанной в виде регулярных выражений) в объектную программу на целевом языке

Lex

Существует целый ряд инструментов для создания лексических анализаторов; большинство этих инструментов основывается на регулярных выражениях. Одним из традиционных срсдств подобного рода является Lex, состоящий из Lex-языка и Lex-компилятора. На самом деле запись спецификаций на языке Lex полезна даже тогда, когда Lex компилятор не доступен, поскольку эти спецификации могут быть без особого труда преобразованы в программу вручную. На данный момент, компиляторы Lex существуют на многих платформах и, несомненно, в ближайшее время появятся и на платформе .NET.

Процесс использования Lex'a выглядит следующим образом: спецификации лексического анализатора на языке Lex подготавливаются в виде программы lex.l. Затем этот файл обрабатывается Lex компилятором, в результате чего создается программа на языке программирования. Большинство существующих реализаций генерируют программы на C и потому в дальнейшем рассмотрении средства Lex мы будем подразумевать использование C, хотя с тем же успехом можно было бы использовать и любой другой язык, например, C#.

Сгенерированная программа состоит из табличного представления диаграмм переходов, построенных по регулярным выражениям, и стандартных подпрограмм, которые используют эти таблицы для разбора лексем. Действия, связанные с реакцией на встреченные регулярные выражения, пишутся непосредственно на C и обычно помещаются сразу же за самими правилами. Затем эта программа обрабатывается компилятором C, в результате чего создается объектная программа, которая и является лексическим анализатором.



Структура Lex-программы

- Lex-программа состоит из описаний, правил трансляции и процедур:
  - Описания задают символы, необходимые для работы лексического анализатора
  - Правила трансляции описывают собственно поведение лексического анализатора
  - Процедуры описывают дополнительную функциональность анализатора (например, поведение при разборе нескольких файлов)

Структура Lex-программы

Lex-программа состоит из трех частей: описаний, правил трансляции и процедур. Каждая часть отделяется от следующей строкой, содержащей два символа %%.

Секция описаний включает описания переменных, констант и регулярных определений. Раздел описаний содержит определения макросимволов (метасимволов) в виде:
ИМЯ ВЫРАЖЕНИЕ
Если в последующем тексте в регулярном выражении встречается {ИМЯ}, то оно заменяется на ВЫРАЖЕНИЕ. Если строка описаний начинается с пробелов или заключена в скобки %{ ... }%, то она просто копируется в выходной файл.

Регулярные определения – это последовательность определений вида
$d_1\ r_1$
…
$d_n\ r_n$,
где каждое d_i – некоторое имя, а каждое r_i – регулярное выражение над алфавитом $\Sigma \cup \{d_1,...,d_n\}$.

Правила трансляции – это операторы вида
$p_1\ \{action_1\}$
…
$p_n\ \{action_n\}$
где p_i – регулярное выражение, $action_i$ – фрагмент программы, описывающий, какие действия должен выполнять лексический анализатордля лексемы, определяемой p_i.

Третья секция содержит процедуры, выполняемые при разборе. В частности, здесь описывается функция yywrap(), которая определяет, что делать при достижении автоматом конца входного файла. Ненулевое возвращаемое значение приводит к завершению разбора, нулевое – к продолжению (перед продолжением, естественно, надо открыть какой-нибудь файл как yyin). Вообще говоря, эти процедуры могут быть скомпилированы отдельно.

Способы записи регулярных выражений в Lex-программе

- Существует возможность задания любого символа или класса символов из входного алфавита
- Грамматика для записи регулярных выражений позволяет задавать такие операции, как повторение элемента, конкатенацию, наличие фрагментов в начале или конце строки и т.д.

Способы записи регулярных выражений в Lex-программе

Рассмотрим способы записи регулярных выражений во входном языке Lex'a. Символ из входного алфавита, естественно, представляет регулярное выражение из одного символа. Специальные символы (в том числе +-*?()[]{}|/\^$.<>) записываются после префикса \. Символы и цепочки можно брать в кавычки, например допустимы следующие три способа кодирования символа *a*: *a*, *"a"* и *\a*.

Имеется возможность задания класса символов:

[0-9] или [0123456789] – любая цифра
[A-Za-z] – любая буква
[^0-7] – любая литера, кроме цифр от 0 до 7
. – любая литера, кроме \n

Грамматика для записи регулярных выражений (в порядке убывания приоритета):

<p>* – повторение 0 или более раз
<p>+ – повторение 1 или более раз
<p>? – необязательный фрагмент
<p><p> – конкатенация
<p>{m,n} – повторение от m до n раз
<p>{m} – повторение m раз
<p>{m,} – повторение m или более раз
^<p> – фрагмент в начале строки
<p>$ – фрагмент в конце строки
<p>|<p> – любое из выражений
<p>/<p> – первое выражение, если за ним следует второе
(p) – скобки, используются для группировки

Пример. Регулярное выражение ^[^aeiou]*$ означает любую строку, не содержащую букв *a, e, i, o*.

Пример Lex-программы

- Рассмотрим задачу написания разбора простого языка, описывающего арифметические выражения и условные предложения
- Некоторые правила такого языка:
  - *id => letter (letter + digit)**
  - *num => digit$^+$ (.digit$^+$ + ε) (E('+' + '–' + ε)digit$^+$ + ε)*

Пример Lex-программы

Предположим, что мы пишем анализатор, который должен уметь обрабатывать условные предложения на базе примитивных выражений, состоящих из идентификаторов и чисел. Приведем правила, задающие лексические классы для такого языка:

*id → letter (letter + digit)**

num → digit$^+$ (.digit$^+$ + ε) (E('+' + '–' + ε)digit$^+$ + ε)

Кроме того, будем считать заданной структуры лексических классов и их атрибутов, которые должен порождать наш анализатор (см. след. таблицу):

Регулярное выражение	**Лексический класс (token)**	**Атрибут**
ws	–	–
if	if_LC	–
then	then_LC	–
else	else_LC	–
Id	Identifier_LC	Pointer to ReprTab
num	Number_LC	Pointer to ReprTab
<	relop_LC	LT
<=	relop_LC	LE
=	relop_LC	EQ
<>	relop_LC	NE
>	relop_LC	GT
>=	relop_LC	GE

Напишем соответствующую Lex-программу:

```
%{
        определение констант
%}
%%
/* регулярные  определения */
delim       {\t\n}
ws            {delim}+
letter        {A-Za-z}
digit         {0-9}
id             {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+|-]?{digit}+)?
%%
{ws}          {}
if               {return (if_LC);}
then          {return (then_LC);}
else           {return (else_LC);}
{id}           {yylval = addToReprTab (); return  (Identifier_LC);}
{number}  {yylval = addToReprTab (); return  (Number_LC);}
"<"            {yylval = LT; return (relop_LC);}
...
```

Другие возможности Lex'a

- Еще один пример, не связанный напрямую с задачами компиляции: подсчет числа слов и строк в файле
- Внутри действий в правилах можно использовать специальные функции Lex'a, такие, как yytext (выдает только что разобранную строку) или yyleng (длина этой строки)

Другие возможности Lex'a

Рассмотрим еще один пример – подсчет числа слов и строк в файле:

```
/***************** Раздел определений ********************/

NODELIM     [^" "\t\n]
         /* NODELIM означает любой символ, кроме разделителей слов */
int l, w, c;                          /* Число строк, слов, символов */

%% /******************* Раздел правил ***********************/

                  { l=w=c=0;          /* Инициализация */      }
{NODELIM}+        { w++; c+=yyleng;   /* Слово */              }
\n                { l++;              /* Перевод строки */     }
.                 { c++;              /* Остальные символы */  }

%% /****************** Раздел программ **********************/

int main()
{ yylex(); }

yywrap()
{
      printf( " Lines - %d  Words - %d Chars - %d\n", l, w, c );
      return( 1 );
}
```

Внутри действий в правилах можно использовать некоторые специальные конструкции и функции Lex'a:

yytext	– указатель на отождествленную цепочку символов, оканчивающуюся нулем;
yyleng	– длина этой цепочки
yyless(n)	– вернуть последние n символов цепочки обратно во входной поток;
yymore()	– считать следующие символы в буфер yytext после текущей цепочки
yyunput(c)	– поместить байт с во входной поток
ECHO	– копировать текущую цепочку в yyout
yylval	– еще одно возвращаемое значение

Заглядывание вперед при лексическом анализе

- В некоторых случаях может потребоваться заглядывание вперед для точного определения текущей лексемы
- Для этого используется выражение A/B, которое выдает лексический класс A только в том случае, если за ним следует B
- В качестве примера рассмотрим разбор переменных, начинающихся с DO в Фортране

Заглядывание вперед при лексическом анализе

Отметим, что Lex всегда работает детерминированным образом, так как не содержит возвратов к уже рассмотренным символам и всегда выдает наиболее длинную подходящую строку. Однако иногда для корректного выполнения лексического анализа необходимо производить заглядывание вперед. Например, при лексическом анализе программ на C# после прочтения символа > необходимо прочитать и последующие символы, т.к. лексема может оказаться одной из следующих: >, >=, >>, >>=, >>>, >>>=.

В некоторых случаях, заглядывание вперед еще более критично. Вернемся к рассматривавшемуся выше примеру на Фортране:

```
DO 5 I=1.25
DO 5 I=1,25
```

Поскольку в Фортране пробелы не являются значащими литерами вне комментариев и строк, то предположим, что все пробелы удаляются до начала лексического анализа. Тогда на вход лексического анализатора попадет следующее:

```
DO5I=1.25
DO5I=1,25
```

Для выделения лексем в этой ситуации мы можем использовать выражение вида r_1/r_2, где r_1 и r_2 – произвольные регулярные выражения. С использованием этого мы можем написать Lex спецификацию, которая выделяет ключевое слово DO:

DO/ ({letter} | {digit}) = ({letter} | {digit})*,*

При такой спецификации лексический анализатор будет заглядывать вперед пока не просканирует регулярное выражение, написанное после /. Однако, только литеры *D* и *O* будут выделенной лексемой. После удачного выделения yytext будет указывать на *D* и yyleng=2.

Использование регулярных выражений .NET

- Некоторые задачи лексического анализа можно решать путем использования механизма регулярных выражений, предоставляемого .NET – см. System.Text.RegularExpressions
- Класс Regex реализует поиск с использованием недетерминированного конечного автомата, как в Perl или Python (в Lex и awk применяется детерминированный поиск)
- Regex позволяет делать перебор с возвратами, заглядыванием вперед, поиском подвыражений и т.д.

Использование регулярных выражений .NET

Для решения некоторых локальных задач лексического анализа удобно использовать механизм регулярных выражений, предоставляемый .NET (см. классы Regex, Match в пространстве имен System.Text.RegularExpressions).

Класс Regex реализует максимальный по функциональности механизм регулярных выражений. Этот механизм представляет собой недетерминированный поиск регулярных выражений в строке, основанный на "жадном" алгоритме с возвратами (аналогичные механизмы используются в Perl и Python). При таком подходе входная строка в определенном порядке проверяется на наличие любого варианта заданного регулярного выражения и в качестве результата принимается первое же совпадение. У такого подхода есть некоторые недостатки. Во-первых, в связи с тем, что механизм подразумевает возвраты, теоретически мы можем побывать в одном и том же состоянии многократно; таким образом, в худшем случае алгоритм может отработать за экспоненциальное время. Во-вторых, так как алгоритм останавливается на первой же найденной подходящей подстроке, этот алгоритм потенциально может не найти более длинные совпадения с данным регулярным выражением.

Традиционные детерминированные алгоритмы поиска регулярных выражений в строке, реализованные в Lex и awk, лишены таких недостатков. Тем не менее, на практике недетерминированные алгоритмы вполне удовлетворительны, так как среднее время их работы можно сделать линейным или полиномиальным путем использования простых вариантов или отсечения излишних возвратов, а выразительность недетерминированных алгоритмов значительно выше засчет возможности ссылок на разобранные группы, заглядывания вперед и назад, поиска подвыражений и т.п.

Пример использования механизма регулярных выражений .NET

```
void DumpHrefs(String inputString)
{
  Regex r;
  Match m;
  r = new Regex("href\\s*=\\s*(?:\"(?<1>[^\"]*)\"|(?<1>\\S+))",
     RegexOptions.IgnoreCase|RegexOptions.Compiled);
  for (m = r.Match(inputString); m.Success; m = m.NextMatch())
  {
    Console.WriteLine("Found href " + m.Groups[1] + " at "
    + m.Groups[1].Index);
  }
}
```

Пример использования механизма регулярных выражений .NET

Рассмотрим пример использования регулярных выражений .NET. На слайде приведена функция, позволяющая найти в строке все выражения вида href="…". Для этого используются два объекта – Regex, задающий регулярное выражение для поиска, и Match, позволяющий обработать результаты применения регулярного выражения ко входной строке.

Регулярное выражение *r* описывает следующий шаблон для поиска: строка href, за которой следует произвольное количество пробелов, за которыми следует символ =, затем снова произвольное количество пробелов, за которым следует кавычка. Начиная с кавычки, все последующие символы до закрывающей кавычки записываются в специальную строку под номером один <1>. Эта строка впоследствии выводится на печать для всех найденных вхождений такого шаблона (см. печать m.Groups[1]).

Более подробное описание синтаксиса и возможностей регулярных выражений можно найти в документации к Visual Studio.NET.

К сожалению, механизм регулярных выражений .NET вряд ли применим для решения задач лексического анализа в целом. Этот механизм хорошо подходит для проверки наличия того или иного шаблона в заданной входной строке, но использовать его для записи всех возможных последовательностей входных лексем было бы крайне неудобно. Кроме того, регулярные выражения .NET не дают возможности исполнения сторонних действий в процессе нахождения шаблонов в строке.

Литература к лекции

- А. Ахо, Дж. Ульман "Теория синтаксического анализа, перевода и компиляции", Т.1 "Синтаксический анализ", М.: Мир, 1978
- Р. Хантер "Проектирование и конструирование компиляторов", ФиС, 1984

Литература к лекции

- А. Ахо, Дж. Ульман "Теория синтаксического анализа, перевода и компиляции", Т.1 "Синтаксический анализ", М.: Мир, 1978
- Р. Хантер "Проектирование и конструирование компиляторов", ФиС, 1984