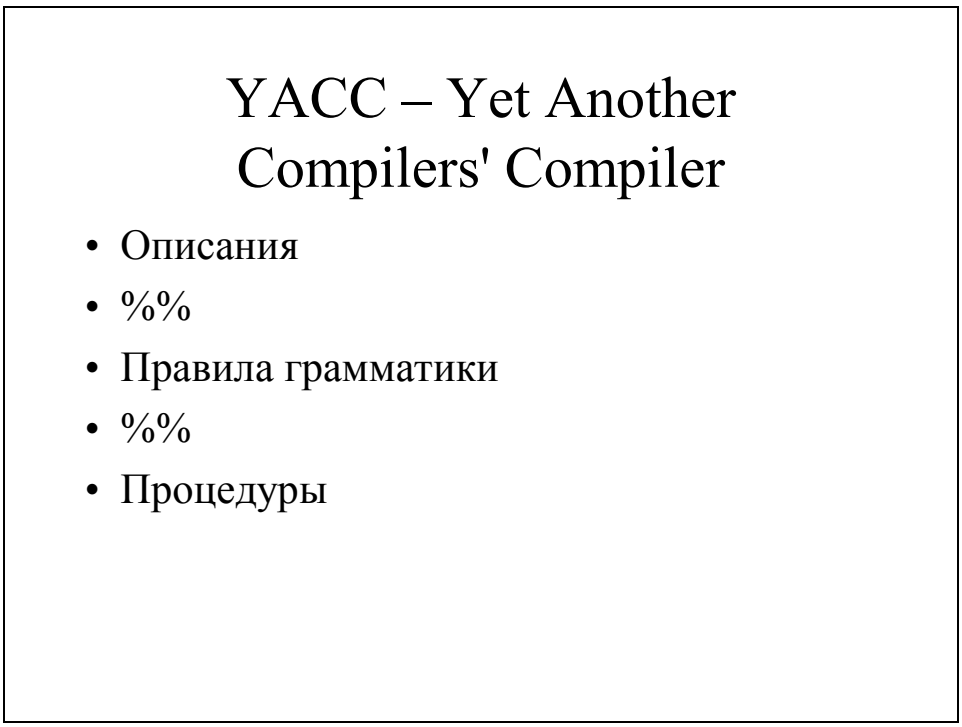
# 8. Грамматики и YACC

- Генератор анализаторов YACC

**Лекция 8. Грамматики и YACC**

В этой лекции рассматриваются следующие вопросы:

- Генератор анализаторов YACC




**Генератор анализаторов YACC**

Генератор синтаксических анализаторов YACC (Yet Another Compilers' Compiler) – это программа, которая строит LALR- анализаторы. Он был разработан в начале 70-х годов прошлого века. Эта программа создана для большинства наиболее распространенных операционных систем, а именно, для UNIX, Windows, OS/2. На самом деле, YACC – это имя генератора в операционной системе UNIX, в остальных операционных системах программа называется PCYACC.

Входом программы является грамматика языка и некоторая дополнительная информация, выход – программа на языке C. Более точно, на вход YACC получает файл со спецификациями (этот файл должен иметь имя с расширением `y`, например, `name.y`). Выходом являются файлы `name.yy.c` (`name.c` для PCYACC'a) и, возможно, `name.yy.h` (`name.h`) и `y.output` (`yy.lrt`). Первый из этих файлов содержит сгенерированную YACC'ом программу анализатора. Второй файл, который создается при задании параметра `-h`, – описания, которые также генерирует YACC. Третий файл содержит протокол, т.е. представление управляющей таблицы анализатора.

Файл name.y должен быть устроен следующим образом:

```
Секция описаний
%%
Секция грамматических правил
%%
Секция  процедур
```

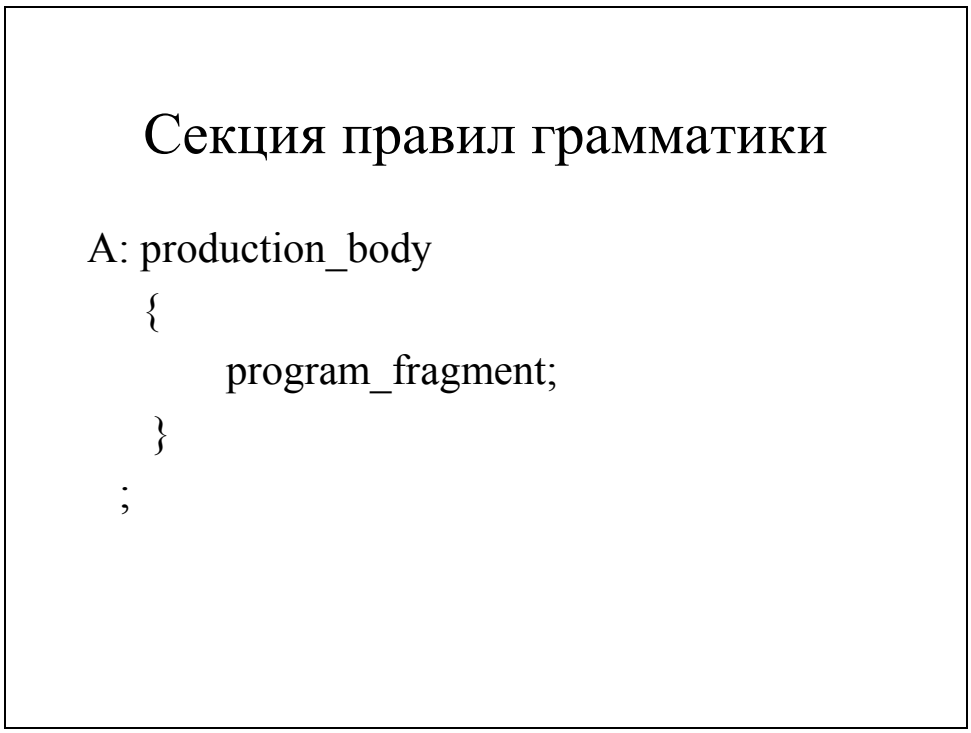
# Секция описаний

- Описания переменных языка С
- Определения типов
- Объявления терминальных символов грамматики
- Объявления нетерминальных символов грамматики
- Определения ассоциативности и приоритетов операций

### Секция описаний

Секция описаний содержит:

- описания переменных языка С, которые используются при описании грамматики. Эти описания заключаются в скобки `%{ … }%`, они будут перенесены в текст результирующей программы без изменения.
  Например,
  ```
  %{
        int myCount;
  }%
  ```
- определения типов, значения которых возвращаются как значения семантик. Эти типы определяются как элементы объединенного типа
  ```
  %union
        {
              type1 id1;
              ...
        }
  ```
- объявления терминальных символов (лексических классов, *tokens*) грамматики в форме `%token` *`lc1 lc2 ...`*
  Например,
  ```
  %token MINUS_LC PLUS_LC TIMES_LC
  %token PLUS_TO_LC TIMES_TO_LC
  ```
  Лексические классы нумеруются либо пользователем, либо самим YACC'ом. В последнем случае лексические классы получают номера, начиная с 257.
- объявления нетерминальных символов грамматики в форме %type <id> name
  Например,
  ```
  %type <id1> conditional_stmt
  ```
- определения ассоциативности и приоритетов операций в форме
  ```
  1. %left op1 op2 ...
  2. %right op3 op4 ...
  3. %nonassoc op5 op6 ...
  ```
  Эти определения должны размещаться в порядке увеличения проиритетов.
  Например,
  ```
  %nonassoc PLUS_TO_LC          /* операция += */
  %left MINUS_LC  PLUS_LC       /* бинарные операции плюс и минус */
  %left TIMES_LC                /* операция умножения */
  ```




**Секция правил грамматики**

Секция грамматических правил состоит из правил, которые записываются следующим образом:

*A: production_body;*

где *A* – имя нетерминала, *production_body* –последовательность нуля или большего количества имен и литералов.

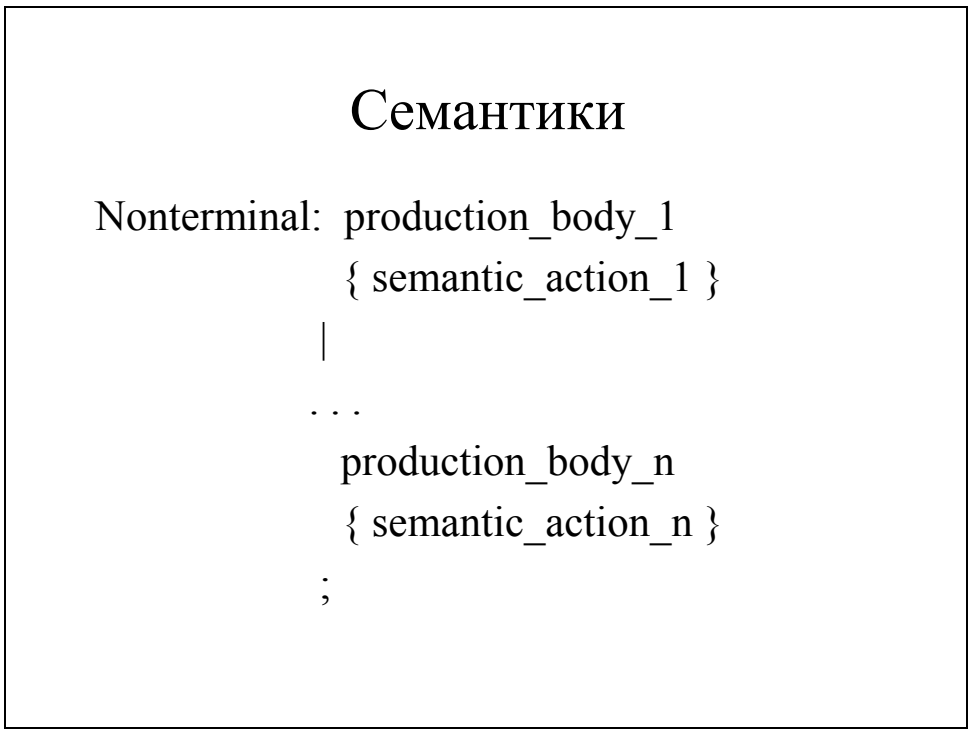
Имена могут быть произвольной длины и содержать буквы, цифры (как обычно, цифра не может быть первой литерой имени), подчеркивания и точки. Литерал состоит из литер, заключенных в апострофы. Как и в языке C, литера обратная косая черта (backslash) используется для задания *управляющей последовательности* (escape sequence).

Если имеется несколько грамматических правил с одинаковой левой частью, то может использовать литера вертикальная черта для объединения всех правил в одно:

```
A:  production_body_1
  | production_body_2
 ;
```

Заметим, что каждое имя, не объявленное как терминал, считается нетерминалом. Каждый нетерминал должен появиться в левой части, хотя бы одного правила. Нетерминал, являющийся левой частью первого правила, по умолчанию считается аксиомой. Вообще говоря, аксиому можно определить в секции объявлений как:

```
%start axiom.
```




**Семантики**

Грамматические правила могут содержать так называемые *семантики* (semantic actions), представляющие собой фрагменты программ на языке C, заключенные в фигурные скобки. Например,

```
lines: lines expr '\n'
           {
             printf ("%s\n", %2);
           }
```

Нетерминал может иметь значение некоторого типа, который указан в объединении, определенном в секции объявлений. Для этого

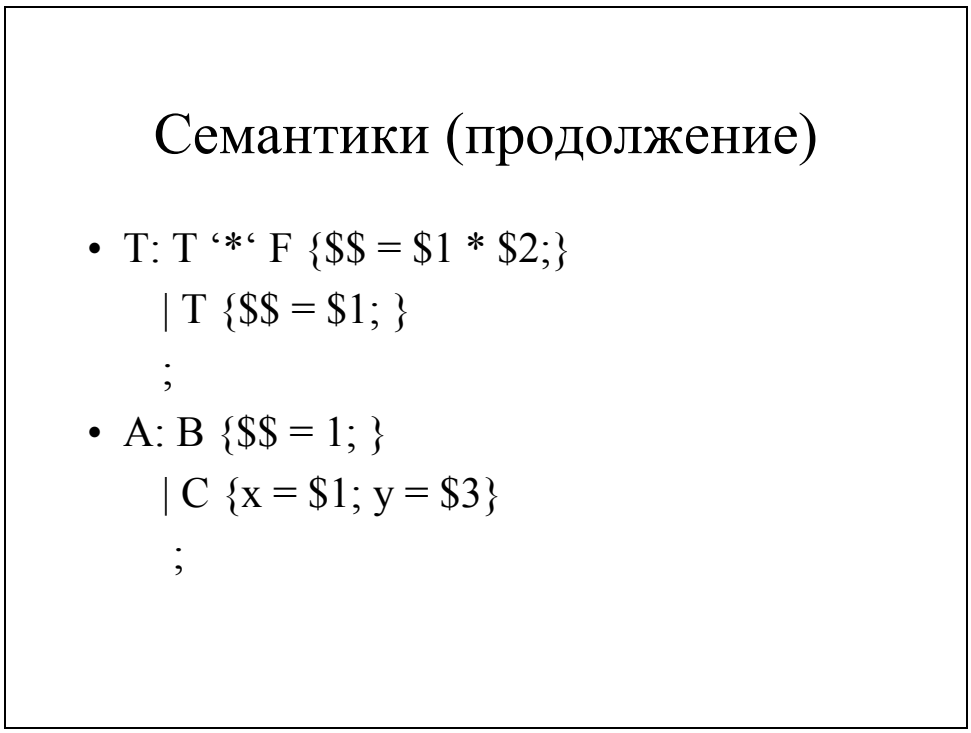
1) нетерминал должен быть объявлен следующим образом:
   `%type <имя-вида> имя-нетерминала`
   Причем, в объединении должен быть элемент *вид имя-вида.* Например,

   ```
   %union
   {
      ...
      unsigned short int myCounter;
      ...
   }
   %type <myCounter> counter
   %%
   ```

2) Семантика правил, левой частью которых является этот нетерминал, должна содержать оператор: `$$ = значение;`

Заметим, что значение может иметь не только нетерминалы, но и терминальные символы. В этом случае терминал должен быть объявлен следующим образом:
`%type <имя-вида> имя-нетерминала`
Естественно, в объединении должен быть элемент *вид имя-вида.* Например,

```
%union {... signed int myValue;...}
%token <myValue> NUMBER_LC
```




**Семантики (продолжение)**

Поскольку нетерминал, может иметь значение, то если такой нетерминал находится в правой части правила, мы можем воспользоваться его значением. Все имена и литералы, содержащиеся в правой части правила нумеруются слева направо, начиная с единицы. Для того, чтобы воспользоваться значением нетерминала следует написать $*номер-нетерминала*, например:

```
T: F { $$=$1; }
   | T*F { $$ = $1*$2; }
   ;
```

Заметим, что по умолчанию значение правила и тем самым значение нетерминала, находящегося в его левой части, - это значение первого элемента в нем. Таким образом, предыдущий пример может быть переписан:

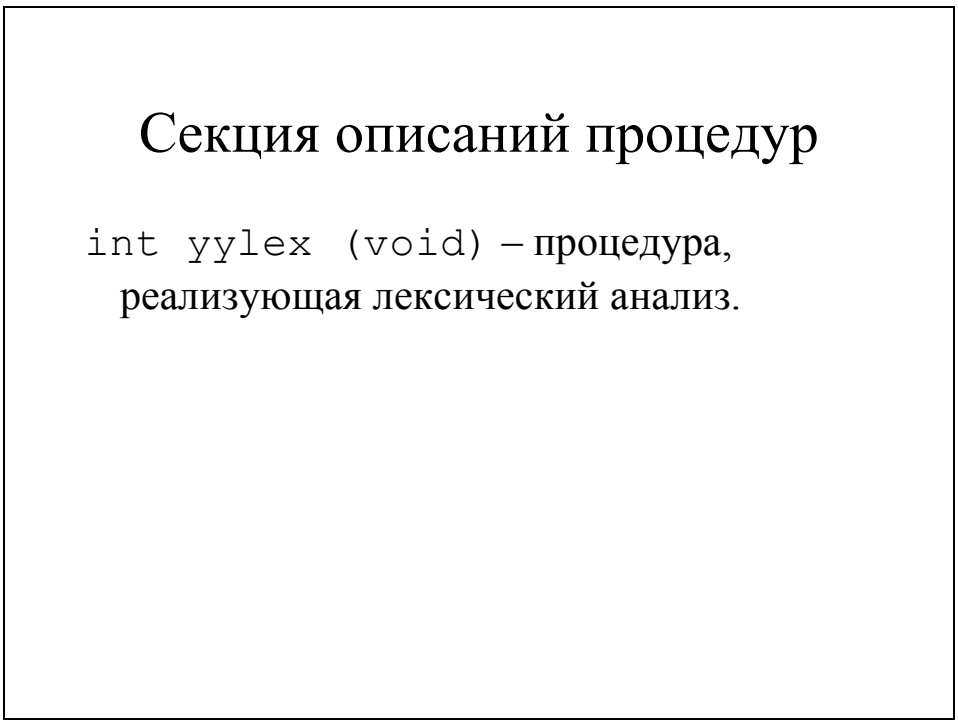
```
T:  F
   |  T*F { $$ = $1*$2; }
   ;
```

На самом деле, семантики могут быть использованы не только в конце правила, но и в середине. Например,

```
A: B { $$ = 1; }
     C { x = $2; y = $3; }
    ;
```

Правда, при этом надо иметь в виду, что семантики, которые не завершают правило, обрабатываются путем введения нового нетерминала и нового правила, сопоставляющего этот нетерминал пустой строке. Т.е. на самом деле приведенное выше правило эквивалентно следующему:

```
$$1:    /* пустая правая часть */  { $$ = 1; }
      ;
A:    B $$1 C { x = $2; y = $3; }
    ;
```




### Секция описаний процедур

Секция описаний процедур содержит процедуры, которые пользователь использует при написании семантических действий. Впрочем, эти процедуры могут быть размещены и в других файлах и откомпилированы отдельно. Таким образом, эта секция необязательна, в отличие от секции описаний и секции грамматических правил.

Пользователь должен предоставить две проедуры:

- процедуру `int yylex (void)`, которая реализует лексический анализ и возвращает лексический класс лексемы
- процедуру `int yyerror (char * s)`, которая вызывается построенным анализатором в случае возникновения ошибки во входной цепочке

YACC создает процедуру `int yyparse (void)`, возвращающую код завершения (0 или 1).

Опишем некоторые параметры программы YACC:
`-Cf` - созданный анализатор будет помещен в файл `f`
`-Df` - будет построен заголовочный файл с именем `f`
`-v` - в файл с именем `yy.lrt` будет выведен протокол, т.е. управляющая таблица анализатора

# Пример

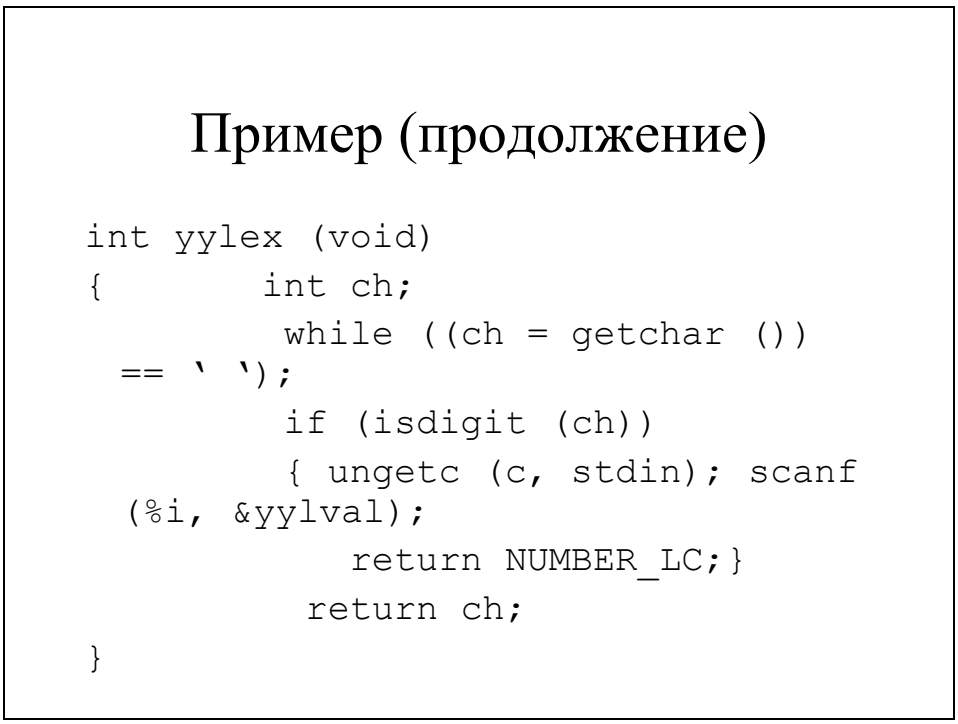
- Разработка программы, вычисляющей значение формулы.

**Пример**

Воспользуемся YACC для реализации калькулятора, т.е. программы, которая вычисляет значение константной арифметической формулы.

Уточним формулировку задачи. Входной поток содержит множество формул, каждая из которых занимает отдельную строчку входного потока. Требуется вычислить значение каждой формулы.

Формула может содержать целые числа, операции +, -, *, / (для вычисления целочисленного частного). Например, для входного потока
(5+3)*7
3+4/2-5/3
будут выведены значения
56
4




**Пример (продолжение)**

Начнем с описания функции `yylex`.

```
int yylex (void)
{
	int ch;
	/* пропускаем пробелы в начале строки */
	while ((ch = getchar ()) == ' ');

	if (isdigit (ch))
	{
		ungetc (ch, stdin);
		scanf (%i, &yylval);
		return NUMBER_LC;
	}

	return ch;
}
```

Функция `yylex` вычисляет пару значений, одно из которых лексический класс, а другое связанный с ним атрибут. Если лексический класс функция `yylex` возвращает в качестве своего значения, то атрибут передается анализатору через присваивание переменной `yylval`. Иначе говоря, то значение, которое присваивается переменной `yylval`, это значение терминального символа `NUMBER`.

# Пример (продолжение)

- Секция определений
- %union {int VALUE; }
- %token <VALUE> NUMBER_LC
- %left ‘+’ ‘-’
- %left ‘*’ ‘/’
- %start lines
- %%

**Пример (продолжение)**

В секции определений определен терминал `NUMBER_LC`, который имеет тип `int`, нетерминал `expression`, также имеющий тип `int`, и правила ассоциативности операций, которые могут быть использованы в формуле. Нетерминал `lines` не должен определяться в этой секции, поскольку он не имеет значения.

```
%union
   {
            int VALUE;
   }

%token  <VALUE> NUMBER_LC

%type <VALUE> expression

%left ‘+’ ‘-‘
%left ‘*’ ‘/’

%start expression                 /* аксиома грамматики */

%%
```

# Пример (продолжение)

- Секция правил грамматики

lines: lines expr ‘\n’ | lines ‘\n’ ! /* empty */ ;
expr: NUMBER_LC
| ‘(‘ expr ‘)’
| expr ‘+’ expr
| expr – expr
| expr * expr
| expr / expr
;

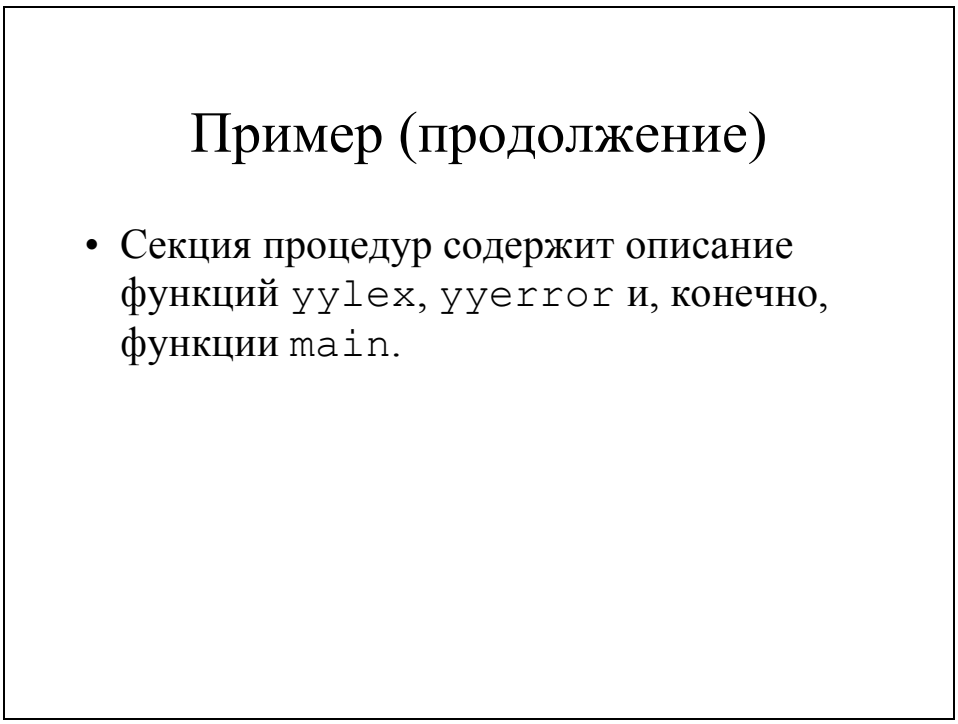
**Пример (продолжение)**

Секция правил грамматики содержит правила для двух нетерминалов `lines` и `expression`. Правила для нетерминала `lines` порождают последовательность строчек входного потока, каждая из которых содержит одну формулу (нетерминал `expression`):

```
lines:   lines expression '\n' { printf ("%I \n", $2); }
       |   lines '\n'
       |   /* empty */
       ;

expression:   NUMBER_LC  { $$ = $1; }
                |    '(' expression ')'          {$$ = $2; }
                |    expression '+'  expression  {$$ = $1+$2; }
                |    expression '-'  expression  {$$ = $1-$2; }
                |    expression '*'  expression  {$$ = $1*$2; }
                |    expression '/'  expression  {$$ = $1/$2; }

%%
```




**Пример (продолжение)**

Секция процедур содержит описание функций `yylex`, `yyerror` и, конечно, функции main. Хотя, как уже было сказано, эта секция может быть опущена, если все необходимые функции содержатся в некотором другом файле, который будет компилироваться отдельно.

Итак, секция процедур для нашего примера может выглядеть следующим образом. Для полноты картины описание функции `yylex` приводится вновь, но на этот раз без комментариев.

```
int yylex (void)
{
      int ch;
      /* пропускаем пробелы в начале строки */
      while ((ch = getchar ()) == ' ');

      if (isdigit (ch))
      {
          ungetc (ch, stdin);
          scanf ("%i", &yylval);
          return NUMBER_LC;
      }

      return ch;
}

yyerror (char *s)
{
      printf ("error: %s", s);
}

main ()
{
     return yyparse ();
}
```

# Пример (продолжение)

Управляющая таблица анализатора.

**Пример (продолжение)**

Для того, чтобы получить управляющую таблицу анализатора достаточно запустить программу YACC с ключом `-v`.

Рассмотрим фрагмент таблицы для состояния 2.

```
+-------------------------     STATE 2     -------------------------+
+ CONFLICTS:
+ RULES:
          lines :  lines expression^\n
     expression :  expression^+ expression
     expression :  expression^- expression
     expression :  expression^* expression
     expression :  expression^/ expression
+ ACTIONS AND GOTOS:
              + : shift & new state 7
              - : shift & new state 8
              * : shift & new state 9
              / : shift & new state 10
             \n : shift & new state 6
                : error
```

В первой строке фрагмента приведено название состояния. Секция `CONFLICTS` перечисляет встреченные конфликты (подробнее о конфликтах – см. в лекции 9). Секция `RULES` перечисляет все правила, задействованные в конфигурациях данного состояния (вместо символа точки, используемого в курсе, ичпаользуется ^). Секция `ACTIONS AND GOTOS` представляет собой столбец управляющей таблицы анализатора, соответствующий 2-му состоянию. Подробнее о составлении управляющей таблицы можно узнать в лекции 7.

# Обработка ошибок

- Ошибки в исходной программе могут быть обнаружены на различных фазах компиляции, промышленные компиляторы производят восстановление после ошибок
- Ошибки, обнаруживаемые на этапе лексического анализа
- Ошибки, обнаруживаемые на этапе синтаксического и видозависимого анализа
- Ошибки на более поздних стадиях работы компилятора

### Обработка ошибок

Каждая фаза компиляции может обнаружить ошибки в транслируемой программе. После обнаружения ошибки фаза должна каким-то образом справиться с возникшей ситуацией. Иными словами, процесс компиляции должен быть продолжен, причем так, чтобы была возможность поиска следующих ошибок в исходной программе. Компилятор, который останавливается после обнаружения первой ошибки, не может быть признан достаточно хорошим. Впрочем, в некоторых ситуациях это вполне приемлемо. Такие ситуации возникают, например, если разрабатывается диалоговый транслятор, который будет использоваться в учебных целях, поскольку начинающему программисту, с одной стороны, вполне достаточно получать информацию об одной ошибке, с другой стороны, получение информации сразу о большом количестве ошибках может его дезориентировать. Одно из основных требований, предъявляемых промышленным трансляторам, заключается в том, чтобы пользователь получил как можно больше корректных ошибок за одну трансляцию. Мы не зря использовали прилагательное «корректные», говоря об ошибках, которые обнаруживает компилятор. Дело в том, что иногда трансляторы выдают информацию о так называемых «наведенных» ошибках. Наведенные ошибки, т.е. такие, которых в программе на самом деле нет, могут возникнуть в результате не совсем корректной работы транслятора после обнаружения какой-нибудь ошибки.

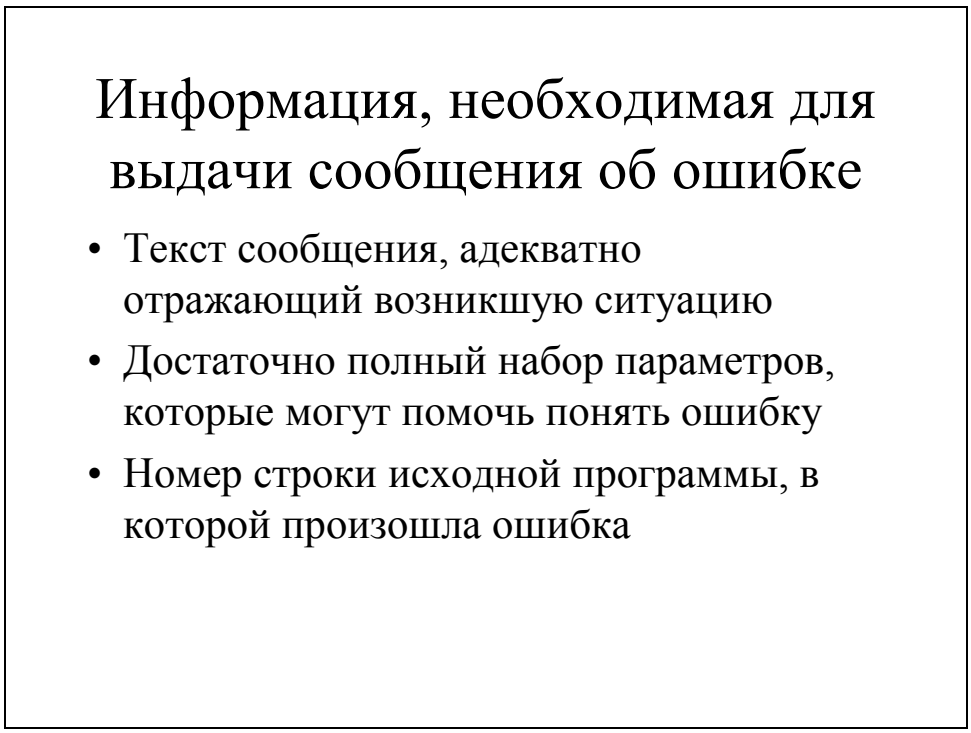
Наибольшая доля ошибок приходится, как правило, на две фазы: синтаксический анализ и фазу контроля типов. Лексический анализатор может обнаружить только те ошибки, которые связаны, например, с использованием неверных литер, или если выделенная лексема не принадлежит ни одному из лексических классов языка. Количество типов ошибок, которые может обнаружить фаза лексического анализа, весьма незначительно, поскольку лексический анализатор «видит» только небольшой, локальный, участок программы. Например, лексический анализатор не сможет обнаружить ошибку в следующем контексте:

```
fi (x == y) { ... }
```

Ошибки, связанные с нарушением синтаксической структуры исходной программы, определяются на фазе синтаксического анализа. Ошибки, возникающие на фазе контроля

типов, связаны с неверным использованием идентификаторов, с некорректной передачей фактических параметров процедурам и т.п.

Обычно, фазы оптимизации и генерации не обнаруживают ошибки, хотя и здесь бывают исключения. Например, представим себе, что в реализуемом языке определено присваивание одной структуры другой по именам полей. Это означает, что если у нас есть две структуры, то присваивание одной структуры другой будет иметь эффект в том случае, если обе из этих структур имеют по крайней мере одну пару одинаковых выделителей полей. При таком определении присваивания структур, компилятор, должен проверить, возможно ли такое присваивание, и если все выделители полей структур различны, должен выдать сообщение об ошибке. Понятно, что ситуация такого рода станет ясна только после контроля типов. Кроме того, поскольку компилятор должен выполнить так называемую «расклейку» присваивания (т.е. преобразовать исходное присваивание в одно или несколько присваиваний соответствующих полей), то такого рода действия можно считать оптимизирующими и, соответственно, сообщение об ошибке в случае, если все имена полей структуры-получателя и структуры-источника различны, будет выдавать фаза оптимизации.




**Информация, необходимая для выдачи сообщения об ошибке**

В однопросмотровом компиляторе все фазы выполняются параллельно. Если возникает ошибка в исходной программе, то текущая позиция лексического анализатора является приемлемой аппроксимацией позиции исходной программы, содержащей ошибку. В таком компиляторе лексический анализатор сохраняет текущую позицию в глобальной переменной. Процедура, предназначенная для выдачи сообщений об ошибках, печатает сообщение об ошибке и значение переменной, содержащей текущую позицию.

Компиляторы, состоящие более чем из одного просмотра, часто выполняют синтаксический анализ и типовой анализ на разных просмотрах. Естественно, это облегчает жизнь в различных аспектах, но существенно усложняет выдачу сообщений о типовых ошибках. Лексический анализатор достигает конца исходной программы раньше, чем начнет выполняться фаза контроля типов. Контроль типов осуществляется во время обхода синтаксического дерева, поэтому невозможно использовать текущую позицию в исходной программе, которую поддерживает лексический анализатор, для выдачи информации об ошибке. Поэтому каждый узел синтаксического дерева должен содержать позицию соответствующей ему конструкции в исходном файле, т.е. структура, определяющая узел дерева, должна содержать поле *pos*, предназначенное для этой цели. Это поле *pos* само является структурой из двух полей: номера строки исходной программы и номера позиции в строке. Понятно, что текущая позиция первоначально определяется лексическим анализатором (оно является одним из полей структуры Lexeme), а затем передается синтаксическому анализатору, который и помещает это значение в поле *pos* узла синтаксического дерева. Для более точного определения позиции в исходном файле каждый узел синтаксического дерева обычно содержит два поля, определяющих положение конструкции, а именно, позицию начала конструкции и позицию ее конца (*beg_pos* и *end_pos* соответственно).

# Локальные стратегии продолжения анализа

- Режим паники
- Восстановление на уровне фраз
- Error-правила

**Локальные стратегии продолжения анализа**

Оказывается, большинство ошибок в программе обнаруживается на фазе синтаксического анализа. Это можно объяснить с одной стороны тем, что многие ошибки являются синтаксическими по своей природе или их проще выявить, когда поток лексем поступает на вход синтаксическому анализатору. С другой стороны, это можно объяснить тем, что наиболее развиты именно методы синтаксического анализа. Вообще говоря, ошибки в программе можно классифицировать следующим образом: 60% составляют пунктуационные ошибки, 20% - ошибки в операторах и операндах, 15% - ошибки в ключевых словах, на все остальные ошибки остается 5%.

Пусть дана контекстно-свободная грамматика и неверная цепочка $s_{err}= t_1t_2...t_{e-1}t_et_{e+1}...t_n$. Мы можем выделить ошибочный символ $t_e$ как первый символ, на котором может быть определена ошибка при сканировании входной цепочки слева направо. Таким образом, подцепочка $t_1t_2...t_{e-1}$ является префиксом некоторой правильной цепочки $t_1t_2...t_{e-1}...$ языка в то время, как не существует правильной цепочки правильной цепочки $t_1t_2...t_{e-1}t_e$ …, содержащей неверный символ.

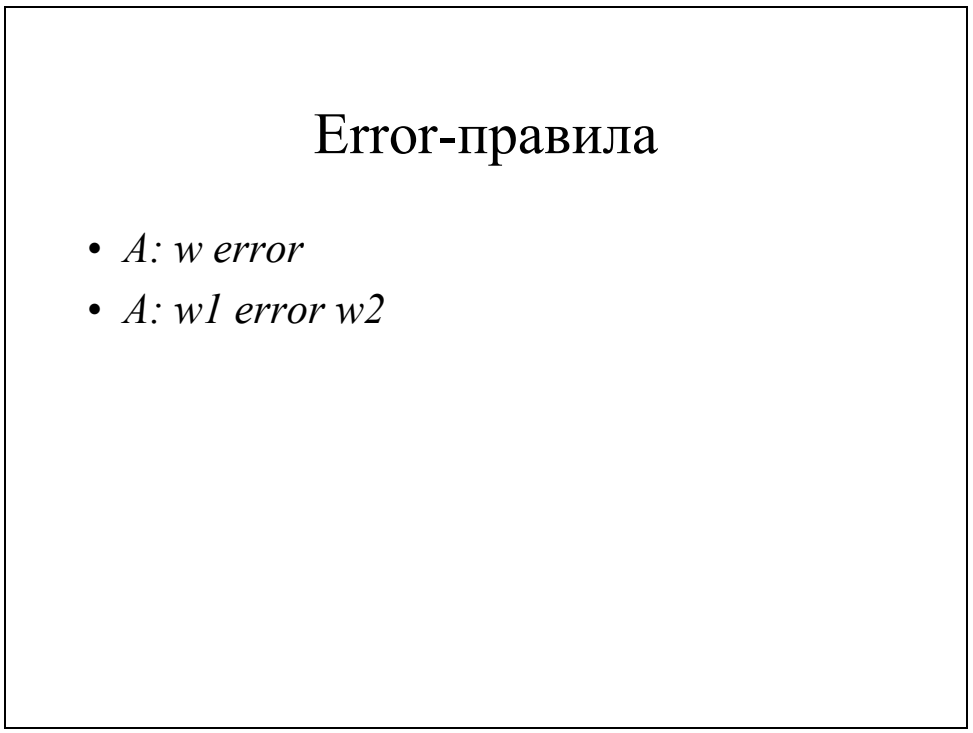
В случае ошибки лексический анализатор, сгенерированный YACC'ом, вызывает функцию yyerror, которая должна быть описана пользователем, и полностью завершает обработку. Это означает, что вы можете обнаружить только одну ошибку.

Когда обнаружена ошибка, редко бывает достаточно остановить всю обработку при обнаружении ошибки; более полезно продолжить сканирование входных данных для нахождения дальнейших синтаксических ошибок. Методы восстановления после синтаксической ошибки разделяются на локальные и глобальные. Локальные методы сводятся к изменению только цепочки $t_et_{e+1}...t_n$, тогда как глобальные методы позволяют изменять символы, расположенные до ошибочного символа. Локальные методы меньше влияют на среду анализатора, поскольку при их использовании не приходится отменять решения уже принятые анализатором, например, не требуется перестраивать синтаксическое дерево.

Имеются различные стратегии продолжения анализа после нахождения синтаксической ошибки:

- *Panic mode*. Это простейший для реализации метод и он может быть использован для большинства синтаксических анализаторов. При обнаружении ошибки анализатор пропускает лексемы до тех пор, пока не будет обнаружена одна из синхронизирующих лексем. Под синхронизирующими лексемами обычно понимаются ограничители, например, такие, как точка с запятой или end, роль которых в исходной программе ясна. Конечно, разработчики компилятора могут выбрать и другие лексемы в качестве синхронизирующих. Понятно, что при такой стратегии может быть пропущено большое количество лексем без анализа их правильности. Этот метод привлекает своей простотой, хотя в отличии от других методов он не может гарантировать, что компилятор не попадет, например, в бесконечный цикл. Наиболее эффективен этот метод может быть в таких случаях, когда несколько ошибок в одном операторе встречаются редко. На самом деле, это почти всегда так. По статистике 80% неверных операторов имеют только одну ошибку, 13% - две.

- *Phrase level*. При обнаружении ошибки анализатор может попытаться подправить еще не обработанную часть входной цепочки. Например, он может заменить префикс оставшейся части входной цепочки на некоторую строку, которая позволит ему продолжить анализ. Типичные локальные коррекции – это замена точки с запятой на запятую, удаление ошибочной точки с запятой или вставка отсутствующей. Выбор локальных исправлений полностью ложится на разработчика компилятора. Главный недостаток этого метода заключается в том, что на самом деле ошибка может находиться до того, как она будет обнаружена. Обычно возможно более одного изменения неверной цепочки и требуется сделать выбор между ними. Один из методов принятия решения об изменении неверной цепочки называется методом минимального расстояния или минимальной цены восстановления. Этот метод связывает с каждым символом стоимость его удаления из неверной строки или его добавления к неверной строке. Затем выбирается символ и операция над ним с минимальной стоимостью. Методы восстановления, связанные с системами генерации синтаксических анализаторов, требуют взаимодействия с пользователем генератора. Некоторые из них полностью автоматические, иные требуют, чтобы пользователь определил стоимости вставки/удаления.




**Error-правила**

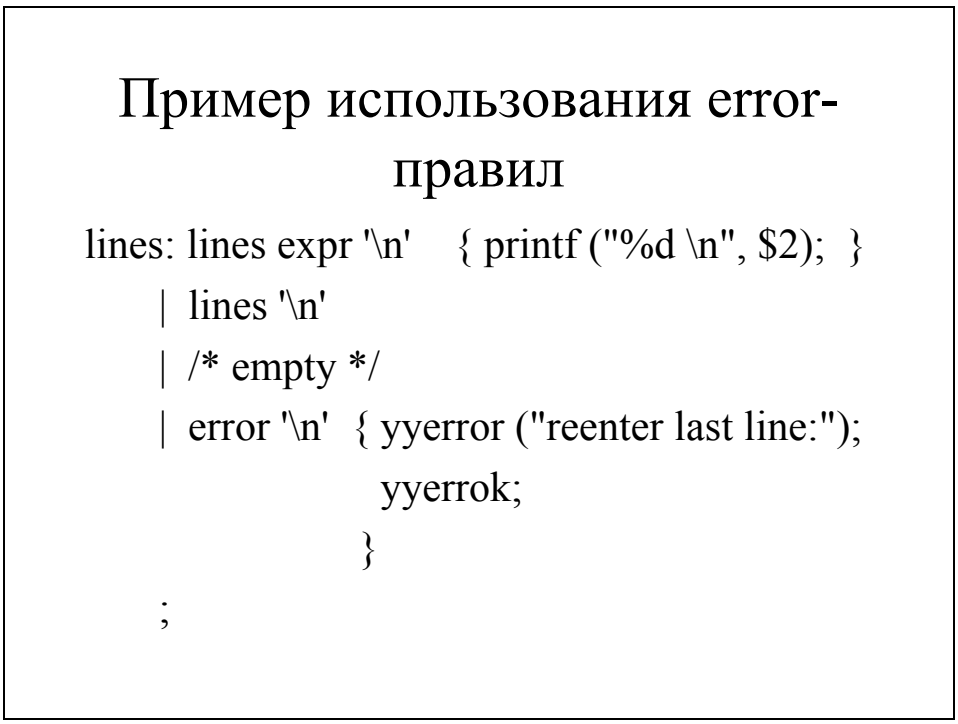
Если мы в состоянии понять, в каких ситуациях могут встретиться ошибки, то мы можем добавить к грамматике языка правила, которые будут использоваться в случае ошибки. Эти правила называются “error productions”. В частности, добавлять такие правила позволяет YACC.

Error-правила в YACC’е имеет один из следующих видов:

- `A:` $w$ `error`
- `A:` $w_1$ `error` $w_2$

Имя `error` зарезервировано для обработки ошибок. Это имя может использоваться в грамматических правилах; в сущности, это имя сообщает о месте, где ожидаются ошибки, и может происходить восстановление. YACC обрабатывает эти правила как обычные правила. Однако, когда анализатор сгенерированный YACC’ом, встречает ошибку, он обрабатывает состояние специальным образом. Из магазина извлекаются символы до тех пор, пока не будет найдено такое состояние, которое лежит как можно ближе к вершине магазина и под которым находится элемент вида: $A: w_1 \text{ ^ } error\ w_2$. Затем анализатор переносит фиктивный лексический класс *error* на стек, как будто этот терминальный символ находился во входной цепочке.

- Если $w_2$ - пусто, то свертка к $A$ выполняется незамедлительно и исполняется семантика, связанная с правилом $A: w\ error$. Затем анализатор сбрасывает символы входной цепочки до тех пор, пока он не отыщет символ, с которым нормальная обработка может быть продолжена.
- Если $w_2$ - непусто, то YACC пропускает первые символы входной цепочки, пока не будет найдена, которая может быть свернута в $w_2$. Затем анализатор сворачивает $A: w_1\ error\ w_2$ в нетерминал $A$ и восстанавливает нормальную обработку. Например, правило *stmt: error ‘;’* указывает анализатору, что он должен пропустить все литеры до ближайшей точки с запятой.




**Пример использования error-правил**

Ниже приведен пример использования error-правил в рассмотренной ранее программе, вычисляющей арифметическое выражение. Теперь, если поданная на вход строка не распознана как выражение, выведется сообщение с предложением ввести последнюю строку заново.

```
%union {
              int myValue;
       }

/*  Terminals */
%token <myValue> Number_LC

%left '+' '-'
%left '*' '/'
%right UNARYMINUS

/*  Nonterminals  */
%type <myValue> expr

%start lines
%%

/* Grammar rules */
lines: lines expr '\n'    { printf ("%d \n", $2);  }
       |  lines '\n'
       |  /* empty */
       |  error '\n'      { yyerror ("reenter last line:");  yyerrok;  }
       ;

expr:     Number_LC      { $$ = $1;     }
       |  expr '*' expr  { $$ = $1*$3;  }
       |  expr '/' expr  { $$ = $1/$3;  }
       |  expr '+' expr  { $$ = $1+$3;  }
       |  expr '-' expr  { $$ = $1-$3;  }
       |  '-' expr %prec UNARYMINUS  { $$ = -$2; }
       ;
```

# Литература к лекции

- А. Ахо, Р. Сети, Дж. Ульман. «Компиляторы: принципы, технологии и инструменты», М.: «Вильямс», 2001. 768 стр.
- D. Grune, G. H. J. Jacobs “Parsing Techniques – A Practical Guide”, Ellis Horwood, 1990. 320 pp.

**Литература к лекции**

- А. Ахо, Р. Сети, Дж. Ульман. «Компиляторы: принципы, технологии и инструменты», М.: «Вильямс», 2001. 768 стр.
- D. Grune, G. H. J. Jacobs “Parsing Techniques – A Practical Guide”, Ellis Horwood, 1990. 320 pp.