Введение

Компиляторы — это важнейшая область исследований, связанных с программным обеспечением, и одновременно — одна из главных составляющих инструментария разработчиков программ. Без компиляторов программистам пришлось бы писать программы непосредственно в машинных кодах, единственно понятных компьютеру. Компиляторы же позволяют создавать программы на языках, понятных и удобных для человека, а затем переводят (транслируют) их в машинные коды.

Цели этой книги: рассмотреть типовую структуру компилятора, а именно представить компилятор как совокупность логически взаимосвязанных модулей; определить взаимодействие между этими модулями и изучить принципы их построения; и наконец, используя метод пошаговой детализации, описать основные функции отдельных модулей.

Чтобы продемонстрировать на практике создание собственного компилятора, в книге описываются методы компиляции Паскаль-программ средствами языка С. Это позволит читателям, знакомым с наиболее распространенными языками программирования Паскаль и С, самим научиться писать компиляторы. Кроме того, для понимания материала книги читатель должен знать способы представления различных информационных структур (данных) в памяти компьютера, а также основные алгоритмы работы с ними.

В книге излагается один из возможных методов создания Паскаль-компилятора. Другие, альтернативные варианты здесь не рассматриваются.

Материал, изложенный в книге, используется автором в течение нескольких лет при чтении курса лекций и проведении практических занятий на механико-математическом факультете Пермского государственного университета. В рамках этого курса каждый студент должен написать, отладить и выполнить тестирование компилятора на примере некоторого фрагмента на языке Паскаль. Только в этом случае действительно становится понятным, как работает компилятор. Как показывает практика, знания и навыки, полученные в результате изучения этого курса, позволяют создать компилятор для своего собственного языка программирования.

Глава 1

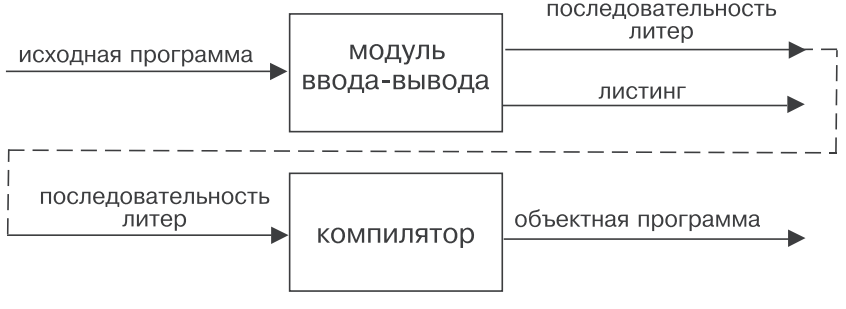
Структура компилятора

Компилятор — это программа, которая переводит программу на языке высокого уровня в эквивалентную программу на другом (объектном) языке. Обычно компилятор также выдает листинг, содержащий текст исходной программы и сообщения обо всех обнаруженных ошибках.

Разработка программного обеспечения (ПО) подразумевает **модульность** и хорошую **структурированность** программ. Учитывая это, представим компилятор как совокупность логически взаимосвязанных модулей, определим взаимодействие между ними и, используя **метод пошаговой детализации**, опишем основные функции отдельных модулей на языке С.

Следуя определению, изобразим компилятор в виде схемы:

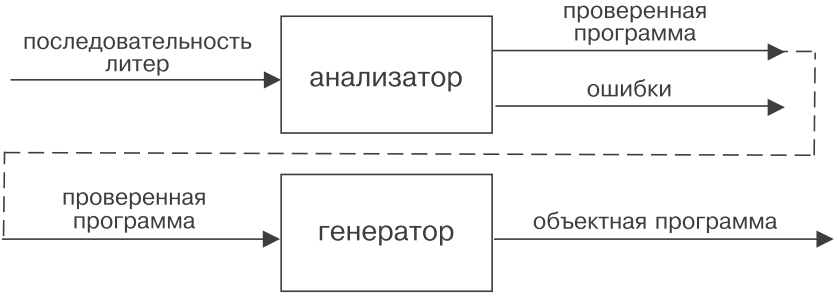
При вводе исходной программы и получении листинга мы имеем дело с конкретными устройствами ввода-вывода (клавиатура, дисплей, магнитные диски). Чтобы легко адаптировать компилятор к различным внешним устройствам конкретной машины, отделим все действия по вводу-выводу данных от собственно процесса компиляции:



Работа компилятора включает в себя два основных этапа:

1) **анализ** — определение правильности исходной программы и формирование (в случае необходимости) сообщений об ошибках;
2) **синтез** — генерация объектной программы; этот этап выполняется для программ, не содержащих ошибок.

Таким образом, собственно компилятор разбивается на составляющие модули:



Одно из достоинств компилятора заключается в возможности генерировать объектную программу для компьютеров с различной архитектурой и различных операционных систем. Однако сам компилятор представляет собой машинно-зависимую программу, так как результат его работы определяется архитектурой конкретного компьютера, а именно представлением данных, кодами операций, форматами машинных команд, способами адресации и т. д. Поэтому уже на ранних стадиях проектирования компилятора в нем выделяют машинно-зависимые и машинно-независимые части. В этом случае работа при переносе компилятора с одной машины на другую существенно облегчается. При этом **анализатор представляет собой машинно-независимую часть компилятора**, а **генератор**, который отображает машинно-независимое промежуточное представление исходной программы на реальную ЭВМ и должен переписываться для каждой новой машины, — это **машинно-зависимая часть компилятора**.

При проверке правильности программы используется полное описание *синтаксиса языка программирования*. Для задания синтаксиса широко применяются **формальные правила** — формы Бэкуса—Наура, а также синтаксические диаграммы.

Например, описание раздела объявления переменных в виде форм Бэкуса—Наура выглядит так:

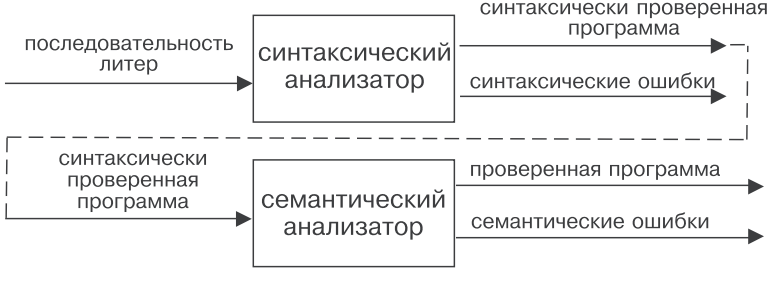
```
<раздел переменных> ::=
   var <описание однотипных переменных> ;
    {<описание однотипных переменных> ; }| < пусто>
<описание однотипных переменных> ::=
                          <имя> { , <имя>} : <тип>
```

В соответствии с этим описанием, следующий фрагмент программы на Паскале:

```
var   name1, name1, name2 : integer;
   name1, name2, name2: real;
```

не содержит ошибок. Причина заключается в том, что, с точки зрения форм Бэкуса—Наура, конкретное представление имени не имеет значения. Поэтому дополнительно к формальным правилам синтаксис языков программирования должен описываться *неформально* — с помощью естественного языка. В нашем примере это неформальное правило формулируется так: «в любой области действия без внутренних по отношению к ней областей никакой идентификатор не может быть описан более одного раза».

Учитывая особенности описания синтаксиса языков программирования, разделим анализатор на два модуля:

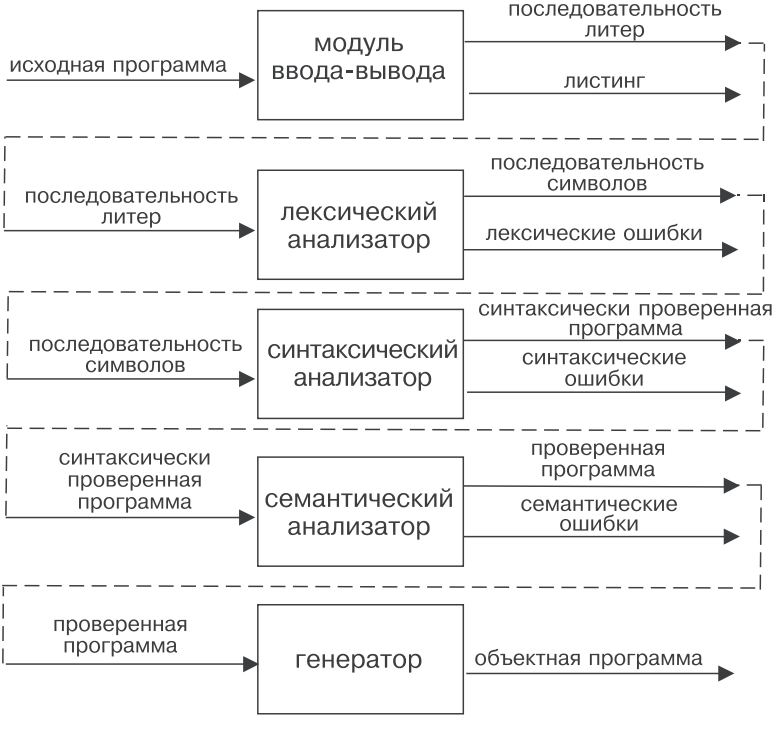


Синтаксический анализатор проверяет, удовлетворяет ли программа формальным правилам. Назначение же **семантического анализатора** состоит в том, чтобы выяснить, не нарушены ли неформальные правила описания языка.

Дальнейшее разбиение на модули обычно выполняется внутри синтаксического анализатора. Первый модуль **(сканер)** просматривает текст (последовательность литер) исходной программы и строит **символы (лексемы)** — идентификаторы, ключевые слова, разделители, числа. Фактически сканер осуществляет простой *лексический анализ* исходной программы, поэтому его называют **лексическим анализатором.**

Второй модуль **(синтаксический анализатор)** выполняет *синтаксический анализ* последовательности символов. На этом этапе символы рассматриваются как *неделимые*, и их представление как последовательности литер несущественно.

Итак, в результате мы получили следующую структуру компилятора:



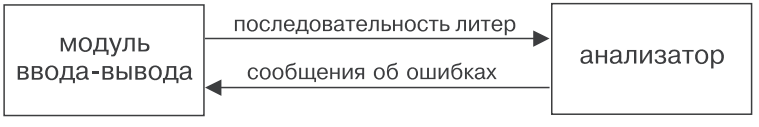
Все указанные на схеме *фазы компиляции* могут работать последовательно или параллельно, но с определенной взаимной синхронизацией. Если какая-нибудь фаза требует чтения исходного текста или результата его трансляции на некоторый внутренний язык, то это обычно называется *проходом*. В некоторых случаях программа полностью компилируется за один проход; при этом обычное ограничение для *однопроходных компиляторов* заключается в том, что все идентификаторы должны быть описаны до их использования. В ряде случаев необходимо иметь несколько проходов, и критерии, определяющие выбор количества проходов, могут быть самыми разнообразными.

Глава 2

Модуль ввода-вывода

2.1. Взаимодействие между модулем ввода-вывода и анализатором

Модуль ввода-вывода считывает последовательность литер исходной программы с внешнего устройства и передает их анализатору. **Анализатор** проверяет, удовлетворяет ли эта последовательность литер правилам описания языка, и формирует (в случае необходимости) сообщения об ошибках. Такое взаимодействие между модулем ввода-вывода и анализатором можно представить в виде схемы:



Будем считать, что в результате очередного обращения к модулю ввода-вывода анализатор получает текущую литеру в переменной:

```
char ch;
```

Чтобы напечатать сообщение об ошибке, анализатор должен передать модулю ввода-вывода причину и местоположение ошибки. Так как она может встретиться в любом месте исходной программы, анализатору необходимо знать *координаты всех литер во входном потоке*. Поэтому модуль ввода-вывода должен формировать номер строки и номер позиции в строке для каждой литеры:

```
struct textposition positionnow;
```

Определим структуру **textposition**:

```
struct textposition
  {
    unsigned linenumber; /*номер строки */
    unsigned charnumber; /*номер позиции
                           в строке */
  };
```

Анализатор выявляет максимально возможное количество ошибок за один просмотр исходной программы. Информация об ошибках заносится в *таблицу ошибок*, каждый элемент которой обязательно содержит код и местоположение ошибки. Анализатор запоминает информацию об ошибках в таблице ошибок в результате обращения к функции:

```
error ( unsigned errorcode ,    /* код ошибки */
        textposition position   /* местоположение
                                   ошибки */ )
```

Модуль ввода-вывода использует содержимое этой таблицы для печати сообщений об ошибках при формировании листинга.

2.2. Программирование модуля ввода-вывода

Структура модуля ввода-вывода (функция **nextch**) может быть представлена следующим образом:

```
nextch( )
 { if ( текущая литера является последней
        литерой строки )
    { напечатать текущую строку;
      if ( в текущей строке обнаружены ошибки )
         напечатать соответствующие сообщения;
      прочитать следующую строку;
    }
   установить в качестве текущей
   следующую литеру и запомнить ее координаты;
 }
```

Будем считать, что максимальная длина строки исходной программы определяется константой **MAXLINE**. Так как исходная программа считывается построчно, буфер ввода-вывода опишем как массив:

```
char line [MAXLINE]
```

Длина строки, считываемой с внешнего запоминающего устройства, может быть меньше размера буфера, поэтому введем переменную:

```
short LastInLine;
```

значение которой — это количество литер в текущей строке.

[. . .]

```
 10           k: i := i * k;
**01**         ^ ошибка код 100
******  использование имени не соответствует описанию
  11         'b' : i := i + 1;
  12          i : k := k + 2;
**02**         ^ ошибка код 100
******  использование имени не соответствует описанию
 13           b: i := i - k;
**03**         ^ ошибка код 147
******  тип метки не совпадает с типом выбирающего выражения
 14           c: i := ( i + k ) * 2
**04**         ^ ошибка код 147
******  тип метки не совпадает с типом выбирающего выражения
 15       end;
 16       writeln( i, k )
 17   end.

Компиляция окончена: ошибок - 4 !
```

Коротко о главном

1. Модуль ввода-вывода выполняет следующие действия:
   - считывает последовательность литер исходной программы и передает их анализатору;
   - формирует листинг.
2. Анализатор запоминает информацию об ошибках в таблице ошибок. Модуль ввода-вывода использует содержимое этой таблицы для печати сообщений об ошибках.
3. Сообщение об ошибке содержит:
   - порядковый номер ошибки;
   - код ошибки;
   - пояснение в текстовом виде.

Задания

1. Опишите функцию **nextch** — модуль ввода-вывода.

(*Примечание*: из-за отсутствия анализатора, формирующего таблицу ошибок, создайте таблицу ошибок вручную.)

2. Разработайте набор тестов для тестирования модуля ввода-вывода.
3. Выполните тестирование модуля ввода-вывода.