ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР.

Лабораторная работа 1.

Общие особенности языков программирования и трансляторов.

Цель работы: изучить особенности трансляторов языков программирования.

Краткие сведения.

Языки программирования достаточно сильно отличаются друг от друга по назначению, структуре, семантической сложности, методам реализации. Это накладывает свои специфические особенности на разработку конкретных трансляторов.

Языки программирования являются инструментами для решения задач в разных предметных областях, что определяет специфику их организации и различия по назначению. В качестве примера можно привести такие языки как Фортран, ориентированный на научные расчеты, С, предназначенный для системного программирования, Пролог, эффективно описывающий задачи логического вывода, Лисп, используемый для рекурсивной обработки списков. Эти примеры можно продолжить. Каждая из предметных областей предъявляет свои требования к организации самого языка. Поэтому можно отметить разнообразие форм представления операторов и выражений, различие в наборе базовых операций, снижение эффективности программирования при решении задач, не связанных с предметной областью. Языковые различия отражаются и в структуре трансляторов. Лисп и Пролог чаще всего выполняются в режиме интерпретации из-за того, что используют динамическое формирование типов данных в ходе вычислений. Для трансляторов с языка Фортран характерна агрессивная оптимизация результирующего машинного кода, которая становится возможной благодаря относительно простой семантике конструкций языка - в частности, благодаря отсутствию механизмов альтернативного именования переменных через указатели или ссылки. Наличие же указателей в языке С предъявляет специфические требования к динамическому распределению памяти.

Структура языка характеризует иерархические отношения между его понятиями, которые описываются синтаксическими правилами. Языки программирования могут сильно отличаться друг от друга по организации отдельных понятий и по отношениям между ними. Язык программирования PL/1 допускает произвольное вложение процедур и функций, тогда как в С все функции должны находиться на внешнем уровне вложенности. Язык С++ допускает описание переменных в любой точке программы перед первым ее использованием, а в Паскале переменные должны быть определены в специальной области описания. Еще дальше в этом вопросе идет PL/1, который допускает описание переменной после ее использования. Или описание можно вообще опустить и руководствоваться правилами, принятыми по умолчанию. В зависимости от принятого решения, транслятор может анализировать программу за один или несколько проходов, что влияет на скорость трансляции.

Семантика языков программирования изменяется в очень широких пределах. Они отличаются не только по особенностям реализации отдельных операций, но и по парадигмам программирования, определяющим принципиальные различия в методах разработки программ. Специфика реализации операций может касаться как структуры обрабатываемых данных, так и правил обработки одних и тех же типов данных. Такие языки, как PL/1 и APL поддерживают выполнение матричных и векторных операций. Большинство же языков работают в основном со скалярами, предоставляя для обработки массивов процедуры и функции, написанные программистами. Но даже при выполнении операции сложения двух целых чисел такие языки, как С и Паскаль могут вести себя по-разному.

Наряду с традиционным процедурным программированием, называемым также императивным, существуют такие парадигмы как функциональное программирование, логическое программирование и объектно-ориентированное программирование. Надеюсь, что в этом ряду займет свое место и предложенная мною процедурно-параметрическая парадигма программирования [Легалов2000] . Структура понятий и объектов языков сильно зависит от избранной парадигмы, что также влияет на реализацию транслятора.

Даже один и тот же язык может быть реализован несколькими способами. Это связано с тем, что теория формальных грамматик допускает различные методы разбора одних и тех же предложений. В соответствии с этим трансляторы разными способами могут получать один и тот же результат (объектную программу) по первоначальному исходному тексту.

Вместе с тем, все языки программирования обладают рядом общих характеристик и параметров. Эта общность определяет и схожие для всех языков принципы организации трансляторов.

1. Языки программирования предназначены для облегчения программирования. Поэтому их операторы и структуры данных более мощные, чем в машинных языках.
2. Для повышения наглядности программ вместо числовых кодов используются символические или графические представления конструкций языка, более удобные для их восприятия человеком.
3. Для любого языка определяется:

- Множество символов, которые можно использовать для записи правильных программ (алфавит), основные элементы.

- Множество правильных программ (синтаксис).
- "Смысл" каждой правильной программы (семантика).

Независимо от специфики языка любой транслятор можно считать функциональным преобразователем F, обеспечивающим однозначное отображение X в Y, где X - программа на исходном языке, Y - программа на выходном языке. Поэтому сам процесс трансляции формально можно представить достаточно просто и понятно:

Y = F(X)

Формально каждая правильная программа X - это цепочка символов из некоторого алфавита A, преобразуемая в соответствующую ей цепочку Y, составленную из символов алфавита B.

Язык программирования, как и любая сложная система, определяется через иерархию понятий, задающую взаимосвязи между его элементами. Эти понятия связаны между собой в соответствии с синтаксическими правилами. Каждая из программ, построенная по этим правилам, имеет соответствующую иерархическую структуру.

В связи с этим для всех языков и их программ можно дополнительно выделить следующие общие черты: каждый язык должен содержать правила, позволяющие порождать программы, соответствующие этому языку или распознавать соответствие между написанными программами и заданным языком.

Связь структуры программы с языком программирования называется ***синтаксическим отображением***.

В качестве примера рассмотрим зависимость между иерархической структурой и цепочкой символов, определяющей следующее арифметическое выражение:

a + (b + c) * d

В большинстве языков программирования данное выражение определяет иерархию программных объектов, которую можно отобразить в виде дерева (рис. 1.1.):

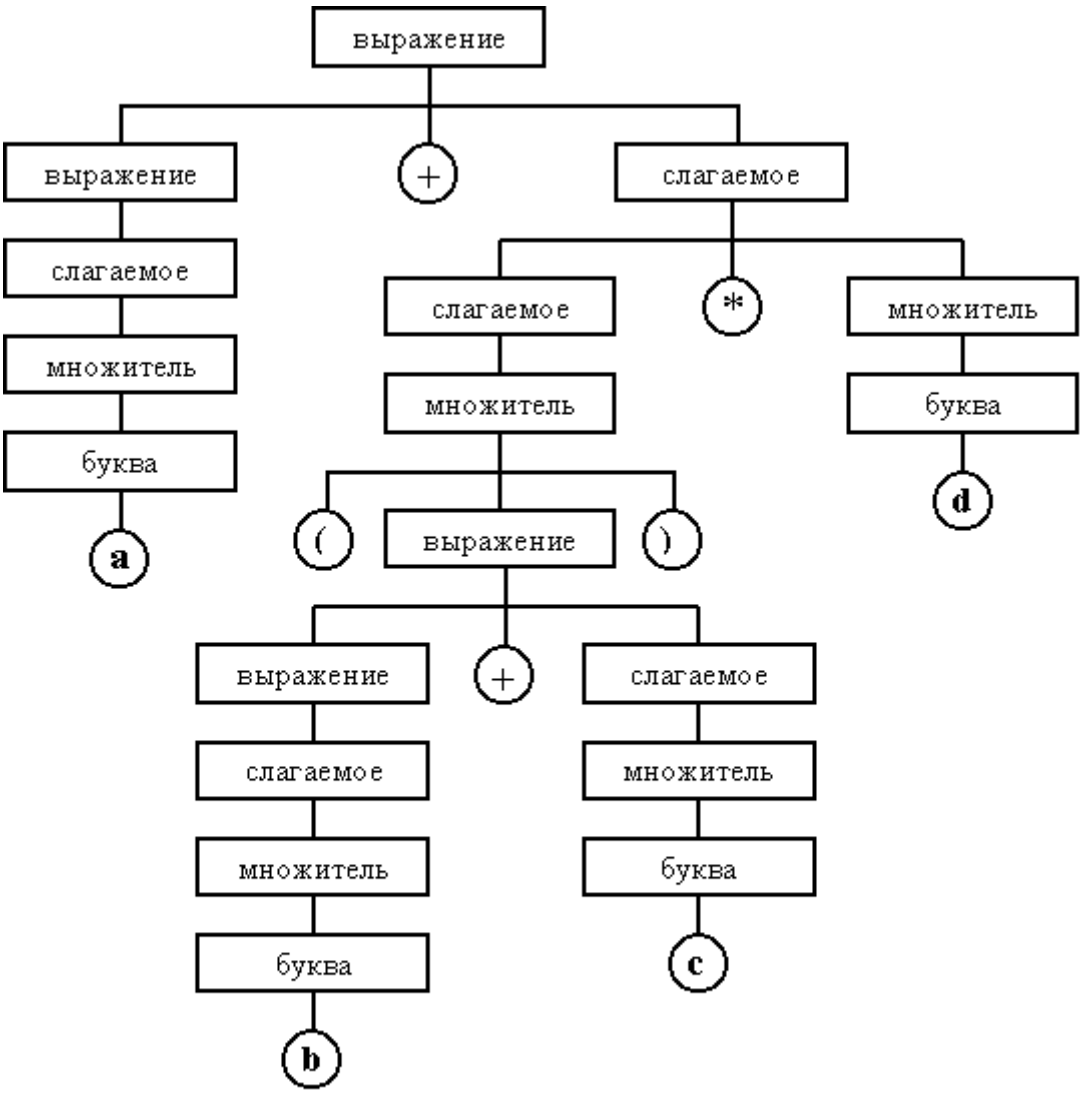

Рис. 1.1. Синтаксическая структура выражения a + (b + c) * d,
построенная по первому описанию

В кружках представлены символы, используемые в качестве элементарных конструкций, а в прямоугольниках задаются составные понятия, имеющие иерархическую и, возможно, рекурсивную структуру. Эта иерархия определяется с помощь синтаксических правил, записанных на специальном языке, который называется метаязыком (подробнее метаязыки будут рассмотрены при изучении формальных грамматик):

<выражение> ::= <слагаемое> | <выражение> + <слагаемое>

<слагаемое> ::= <множитель> | <слагаемое> * <множитель>

<множитель> ::= <буква> | (<выражение>)

<буква> ::= a | b | c | d | i | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

Примечание. Знак "::=" читается как "это есть". Вертикальная черта "|" читается как "или".

Если правила будут записаны иначе, то изменится и иерархическая структура. В качестве примера можно привести следующие способ записи правил:

<выражение> ::= <операнд> | <выражение> + < операнд > | <выражение> * < операнд >

<операнд> ::= <буква> | (<выражение>)

<буква> ::= a | b | c | d | i | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

В результате синтаксического разбора того же арифметического выражения будет построена иерархическая структура, представленная на рис. 1.2.

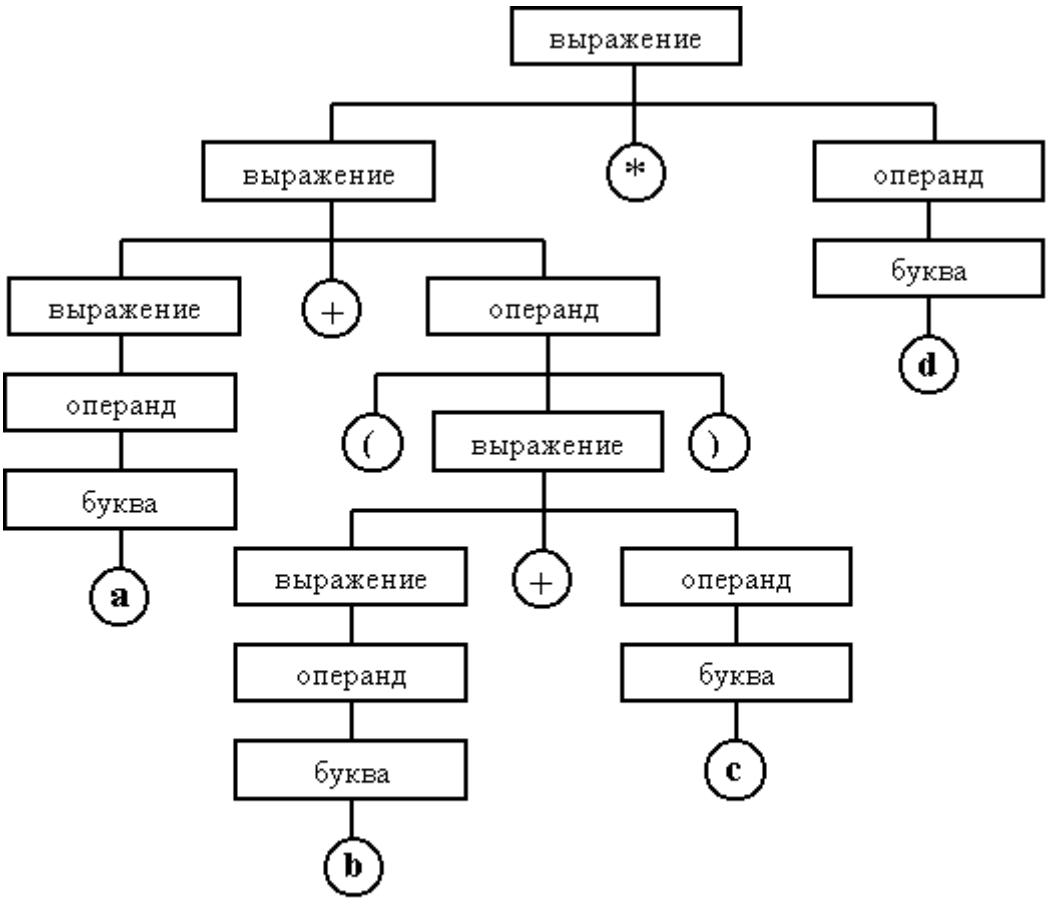

Рис. 1.2. Синтаксическая структура выражения a + (b + c) * d
построенная по второму описанию

Следует отметить, что иерархическая структура в общем случае может быть никоим образом не связана с семантикой выражения. И в том и другом случае приоритет выполнения операций может быть реализован в соответствии с общепринятыми правилами, когда умножение предшествует сложению (или наоборот, все операции могут иметь одинаковый приоритет при любом наборе правил). Однако первая структура явно поддерживает дальнейшую реализацию общепринятого приоритета, тогда как вторая больше подходит для реализации операций с одинаковым приоритетом и их выполнению справа налево.

Процесс нахождения синтаксической структуры заданной программы называется ***синтаксическим разбором***.

Синтаксическая структура, правильная для одного языка, может быть ошибочной для другого. Например, в языке Форт приведенной выражение не будет распознано. Однако для этого языка корректным будет являться постфиксное выражение:

a b c + d * +

Его синтаксическая структура описывается правилами:

<выражение> ::= <буква> | <операнд> <операнд> <операция>

< операнд > ::= < буква > | < выражение >

< операция > ::= + | *

<буква> ::= a | b | c | d | i | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

Иерархическое дерево, определяющее синтаксическую структуру, представлено на рис. 1.3.

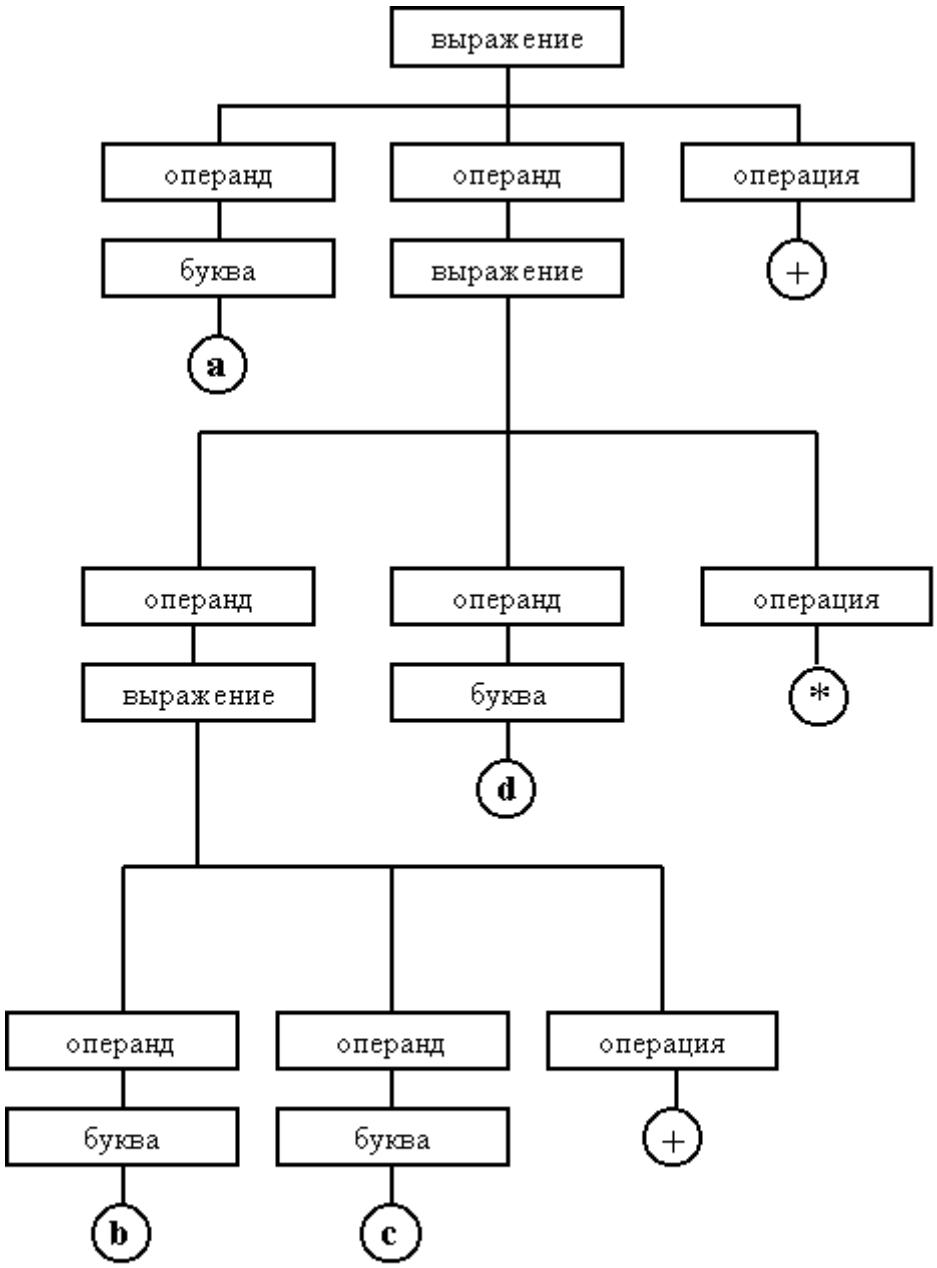

Рис. 1.3. Синтаксическая структура выражения a b c + d * +, построенная по постфиксным правилам

Другой характерной особенностью всех языков является их семантика. Она определяет смысл операций языка, корректность операндов. Цепочки, имеющие одинаковую синтаксическую структуру в различных языках программирования, могут различаться по семантике (что, например, наблюдается в C++, Pascal, Basic для приведенного выше фрагмента арифметического выражения).

Знание семантики языка позволяет отделить ее от его синтаксиса и использовать для преобразования в другой язык (осуществить генерацию кода).

Описание семантики и распознавание ее корректности обычно является самой трудоемкой и объемной частью транслятора, так как необходимо осуществить перебор и анализ множества вариантов допустимых комбинаций операций и операндов.

Обобщенная структура транслятора

Общие свойства и закономерности присущи как различным языкам программирования, так и трансляторам с этих языков. В них протекают схожие процессы преобразования исходного текста. Не смотря на то, что взаимодействие этих процессов может быть организовано различным путем, можно выделить функции, выполнение которых приводит к одинаковым результатам. Назовем такие функции фазами процесса трансляции.

Учитывая схожесть компилятора и интерпретатора, рассмотрим фазы, существующие в компиляторе. В нем выделяются:

1. Фаза лексического анализа.
2. Фаза синтаксического анализа, состоящая из:
   - распознавания синтаксической структуры;
   - семантического разбора, в процессе которого осуществляется работа с таблицами, порождение промежуточного семантического представления или объектной модели языка.
3. Фаза генерации кода, осуществляющая:
   - семантический анализ компонент промежуточного представления или объектной модели языка;
   - перевод промежуточного представления или объектной модели в объектный код.

Наряду с основными фазами процесса трансляции возможны также дополнительные фазы:

2а. Фаза исследования и оптимизации промежуточного представления, состоящая из:

2а.1. анализа корректности промежуточного представления;

2а.2. оптимизации промежуточного представления.

3а. Фаза оптимизации объектного кода.

Интерпретатор отличается тем, что фаза генерации кода обычно заменяется фазой эмуляции элементов промежуточного представления или объектной модели языка. Кроме того, в интерпретаторе обычно не проводится оптимизация промежуточного представления, а сразу же осуществляется его эмуляция.

Кроме этого можно выделить единый для всех фаз процесс анализа и исправление ошибок, существующих в обрабатываемом исходном тексте программы.

Обобщенная структура компилятора, учитывающая существующие в нем фазы, представлена на рис. 1.4.

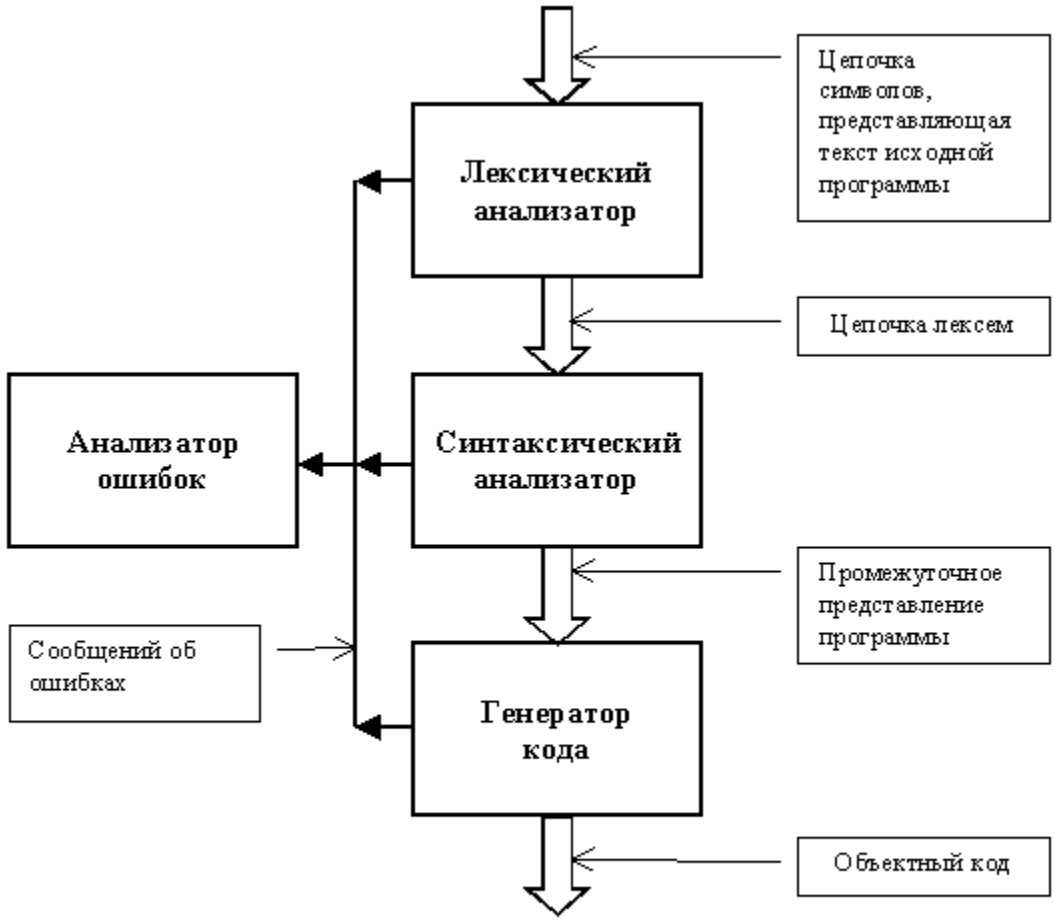

Рис. 1.4. Обобщенная структура компилятора.

Он состоит из лексического анализатора, синтаксического анализатора, генератора кода, анализатора ошибок. В интерпретаторе вместо генератора кода используется эмулятор (рис. 1.5), в который, кроме

элементов промежуточного представления, передаются исходные данные. На выход эмулятора выдается результат вычислений.

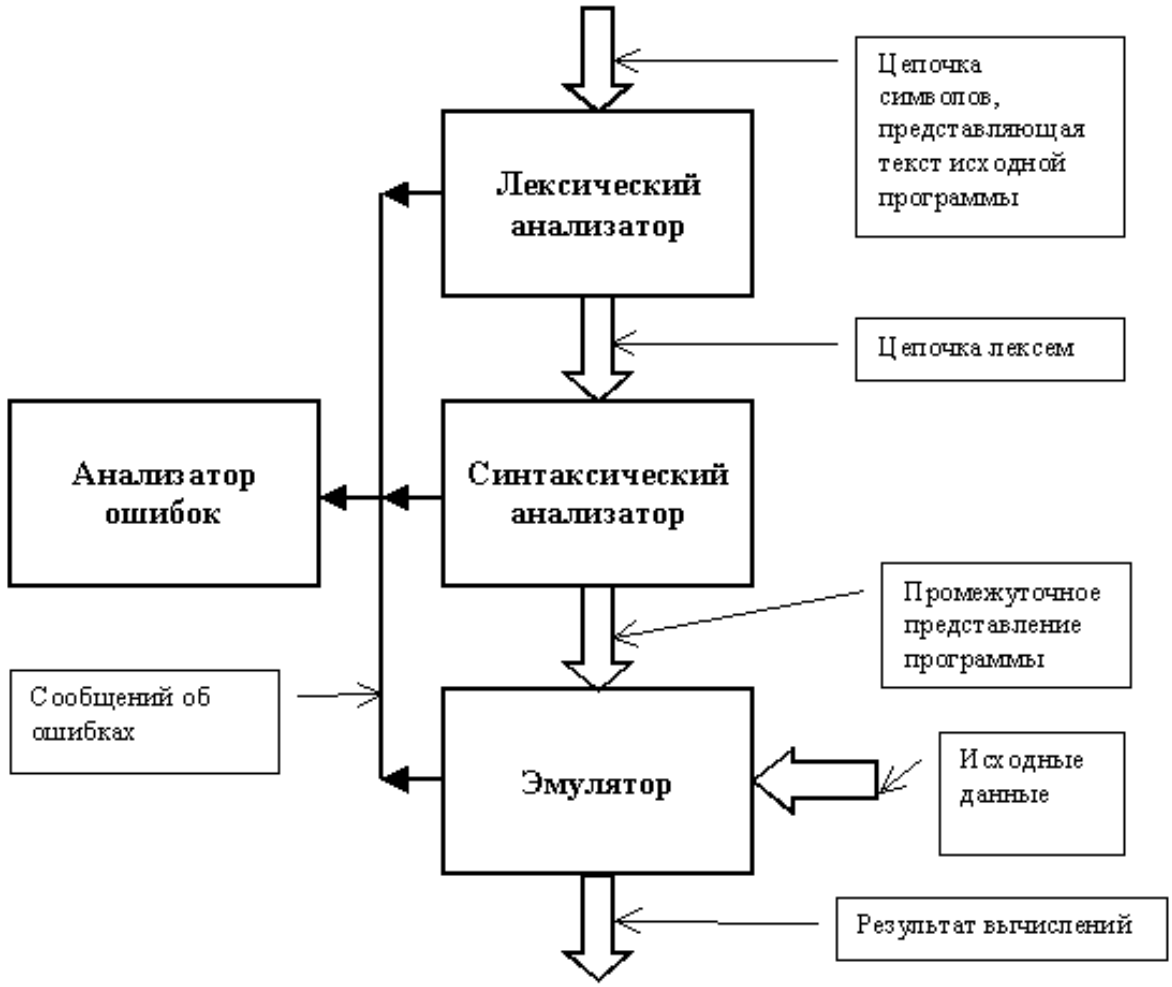

Рис. 1.5. Обобщенная структура интерпретатора.

Лексический анализатор (известен также как сканер) осуществляет чтение входной цепочки символов и их группировку в элементарные конструкции, называемые лексемами. Каждая лексема имеет класс и значение. Обычно претендентами на роль лексем выступают элементарные конструкции языка, например, идентификатор, действительное число, комментарий. Полученные лексемы передаются синтаксическому анализатору. Сканер не является обязательной частью транслятора. Однако, он позволяет повысить эффективность процесса трансляции. Подробнее лексический анализ рассмотрен в теме: "Организация лексического анализа".

Синтаксический анализатор осуществляет разбор исходной программы, используя поступающие лексемы, построение синтаксической структуры программы и семантический анализ с формированием объектной модели языка. Объектная модель представляет синтаксическую структуру, дополненную семантическими связями между существующими понятиями. Этими связями могут быть:

- ссылки на переменные, типы данных и имена процедур, размещаемые в таблицах имен;
- связи, определяющие последовательность выполнения команд;
- связи, определяющие вложенность элементов объектной модели языка и другие.

Таким образом, синтаксический анализатор является достаточно сложным блоком транслятора. Поэтому его можно разбить на следующие составляющие:

- распознаватель;
- блок семантического анализа;
- объектную модель, или промежуточное представление, состоящие из таблицы имен и синтаксической структуры.

Обобщенная структура синтаксического анализатора приведена на рис. 1.6.

Рис. 1.6. Обобщенная схема синтаксического анализатора

Распознаватель получает цепочку лексем и на ее основе осуществляет разбор в соответствии с используемыми правилами. Лексемы, при успешном разборе правил, передаются семантическому анализатору, который строит таблицу имен и фиксирует фрагменты синтаксической структуры. Кроме этого, между таблицей имен и синтаксической структурой фиксируются дополнительные семантические связи. В результате формируется объектная модель программы, освобожденная от привязки к синтаксису языка программирования. Достаточно часто вместо синтаксической структуры, полностью копирующей иерархию объектов языка, создается ее упрощенный аналог, который называется промежуточным представлением.

Анализатор ошибок получает информацию об ошибках, возникающих в различных блоках транслятора. Используя полученную информацию, он формирует сообщения пользователю. Кроме этого, данный блок может попытаться исправить ошибку, чтобы продолжить разбор дальше. На него также возлагаются действия, связанные с корректным завершением программы в случае, когда дальнейшую трансляцию продолжать невозможно.

Генератор кода строит код объектной машины на основе анализа объектной модели или промежуточного представления. Построение кода сопровождается дополнительным семантическим анализом, связанным с необходимостью преобразования обобщенных команд в коды конкретной вычислительной машины. На этапе такого анализа окончательно определяется возможность преобразования, и выбираются эффективные варианты. Сама генерация кода является перекодировкой одних команд в другие.

Пример выполнения задания.

Построить синтаксический анализатор для понятия <идентификатор>:

<Идентификатор>:: = <буква> | <Идентификатор> (<цифра> <буква>).

Листинг программы на языке Turbo Pascal:

```
program identification;
const mlen = 20;
    var ch: char;
    id: array [1..mlen] of char;
    idlen, i: integer;
```

```
procedure idd;
begin
        if ch in [‘A’..’Z’,’a’..’z’,’0’..’9’] then
            begin
                idlen:=idlen+1;
                id[idlen]:=ch;
                read(ch);
                idd
            end;
end;
procedure ident;
begin
        if ch in [‘A’..’Z’,’a’..’z’] then
            begin
                idlen:=1;
                id[1]:=ch;
                read(ch);
                idd
            end;
end;
begin
        read(ch);
        idlen:=0;
        ident;
        write(‘Идентификатор:’);
        for i:=1 to idlen do write (id[i]);
        readln;
end.
```

Варианты заданий.

1. Построить синтаксический анализатор для понятия <простое_выражение>:
<простое_выражение>:: = <простой_идентификатор> | (<простое_выражение> <знак_операции> <простое_выражение>).
2. Проверить, является ли вводимая последовательность символов <вещ_числом>:
<вещ_число>:: = <целое_число>.<целое_число> | <целое_число>.<целое_число_без_знака>E<целое_число> | <целое_число>E<целое_число>
3. Написать программу, которая по заданному простому логическому выражению вычисляет его значение:
<простое_логическое>::= true | false | <идентификатор> | not <простое_логическое> | (<простое_логическое> <знак_операции> <простое_логическое>).
4. Написать программу, которая по заданному правильному простому логическому выражению вычисляет значение этого выражения. Считать, что значения переменных хранятся в массиве value: array [‘a’...’z’] of Boolean.
5. Проверить, является ли вводимая последовательность символов <константным_выражением>:
<константное_выражение>:: = <целое_без_знака> | (<целое_без_знака> <знак_операции> <константное_выражение>).
6. Написать программу, которая по заданному правильному константному выражению либо вычисляет значение этого выражения, либо печатает сообщение об ошибке, если происходит переполнение в результате вычислений, то есть если значение выражения не принадлежит заранее известному диапазону допустимых значений.
7. Проверить, является ли вводимая последовательность символов <списком_параметров>:
<список_параметров>:: = <параметр> [,<параметр>].
<параметр>:: = <имя> | <буква> | <буква>.
8. Проверить, соответствует ли вводимая последовательность символов понятию <скобки>:
<скобки>:: = <круглые> | <квадратные>.
9. Проверить, соответствует ли вводимая последовательность символов понятию <список_списков>:
<список_списков>:: = <список> [;<список>].
10. Проверить, соответствует ли вводимая последовательность символов понятию <текст>:
<текст>:: = <элемент> | <элемент> <текст>.
11. Проверить, соответствует ли вводимая последовательность символов понятию <скобки>:
<скобки>:: = <круглые> | <квадратные>.

<круглые>:: = A | ((<круглые>) [<квадратные>]).
12. Проверить, соответствует ли вводимая последовательность символов понятию <скобки>:
<скобки>:: = <круглые> | <квадратные>.
<квадратные>:: = B | ((<квадратные>) [<круглые>]).
13. Проверить, соответствует ли вводимая последовательность символов понятию <список_списков>:
<список_списков>:: = <список> [;<список>].
<список>:: = <элемент> [,<элемент>].
14. Проверить, является ли вводимая последовательность символов <константным_выражением>:
<константное_выражение>:: = <целое_без_знака> | (<целое_без_знака> <знак_операции> <константное_выражение>).
<целое_без_знака>:: = <цифра> [<цифра>].
15. Проверить, является ли вводимая последовательность символов <константным_выражением>:
<константное_выражение>:: = <целое_без_знака> | (<целое_без_знака> <знак_операции> <константное_выражение>).
<знак_операции>:: = + | - | *.
16. Проверить, является ли вводимая последовательность символов <константным_выражением>:
<константное_выражение>:: = <целое_без_знака> | (<целое_без_знака> <знак_операции> <константное_выражение>).
<цифра>:: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.