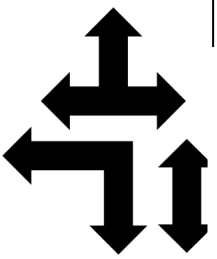
ГЛАВА 4



Разбор и вычисление выражений

Перед программистом нередко возникает задача вычисления арифметических или иных выражений, не известных на этапе компиляции программы. Готовых средств для этого в Delphi нет. В Интернете можно найти компоненты и законченные примеры вычисления выражений, но нередко требуется создать что-то свое. В этой главе мы рассмотрим способы программного разбора и вычисления арифметических выражений. Кроме самих примеров будут изложены основы теории синтаксического анализа, с помощью которой эти примеры написаны. Эти сведения не только помогут лучше понять приведенные примеры, но и позволят легко написать код для синтаксического разбора любых выражений, которые подчиняются некоторому формальному описанию синтаксиса.

Синтаксический анализатор мы будем создавать поэтапно, переходя от простых примеров к сложным. Сначала научимся распознавать вещественное число и напишем простейший калькулятор, который умеет выполнять четыре действия арифметики над числами без учета приоритета операций. Затем наша программа научится учитывать приоритет этих операций, а чуть позже — использовать скобки для изменения этого приоритета. Далее калькулятор обретет способность работать с переменными, вычислять функции и возводить в степень, т. е. станет вполне полноценным. И на последнем этапе мы добавим лексический анализатор — средство, которое формализует разбор выражений со сложной грамматикой и тем самым существенно облегчает написание синтаксических анализаторов.

4.1. Синтаксис и семантика

Прежде чем двигаться дальше, введем базовые определения. *Языком* мы будем называть множество строк (в большинстве случаев это будет бесконеч-

ное множество). Каждое *выражение* (в некоторых источниках вместо "выражение" употребляются термины "предложение" или "утверждение") может принадлежать или не принадлежать языку. Например, определим язык так: любая строка произвольной длины, состоящая из нулей и единиц. Тогда выражения "000101001" и "1111" принадлежат языку, а выражения "5x" и "R@8" — нет.

Синтаксисом называется набор правил, которые позволяют сделать заключение о том, принадлежит ли заданное выражение языку или нет.

С практической точки зрения наиболее интересны те языки, выражения которых не только подчиняются каким-либо синтаксическим правилам, но и несут смысловую нагрузку. Например, выражения языка Delphi — программы — приводят к выполнению компьютером тех или иных действий. В данном случае семантика языка Delphi — это правила, определяющие, к выполнению каких именно действий приведет то или иное выражение. В более общем смысле *семантика языка* — это описание смысла языковых выражений.

Другими словами, синтаксические правила позволяют понять, допустимо ли в выражении, принадлежащем заданному языку, появление в указанной позиции данного символа, а семантические — что означает появление этого символа в данной позиции.

Чтобы подчеркнуть разницу между синтаксисом и семантикой, рассмотрим такой оператор присваивания в Delphi: `X := Y + Z;`. С точки зрения синтаксиса это правильное выражение, т. к. требования синтаксиса заключаются в том, чтобы слева от знака присваивания стоял корректный идентификатор, справа — корректное выражение. Очевидно, что эти правила выполнены. Но с точки зрения семантики это выражение может быть ошибочным, если, например, один из встречающихся в нем идентификаторов не объявлен, или их типы несовместимы, или же идентификатор `X` объявлен как константа. Таким образом, синтаксически правильное выражение не всегда является семантически верным. Примером подобного арифметического выражения может служить "0 / 0" — два корректных числа, между которыми стоит допустимый знак операции, т. е. синтаксически все верно. Однако смысла такое выражение не имеет, т. к. данная операция неприменима к указанным операндам.

Таким образом, синтаксический анализ арифметических выражений — это всего лишь выяснение, корректно ли выражение. Мы же говорили о вычислении выражений, а это уже относится к семантике, т. е., строго говоря, мы здесь будем заниматься не только синтаксическим, но и семантическим анализом. С точки зрения теории синтаксический и семантический анализ разделены, т. е. анализировать семантику можно начинать "с нуля" после того, как анализ синтаксиса закончен. Но на практике легче объединить эти два про-

цесса в один, чтобы пользоваться результатами синтаксического разбора при семантическом анализе. Из-за этого, как мы увидим в дальнейшем, иногда приходится вводить сложные синтаксические правила, которые в итоге описывают тот же язык, что и более простые, чтобы упростить семантический анализ.

На примере выражения `X := Y + Z;` мы могли наблюдать интересную особенность: для заключения о синтаксической корректности или некорректности отдельной части выражения языка нам достаточно видеть только эту часть, в то время как для выяснения ее семантической корректности необходимо знать "предысторию", т. е. то, что было в выражении раньше. Это объясняется следующим образом: существуют формальные способы описания синтаксиса, позволяющие выделить отдельные синтаксические конструкции. В принципе, язык может использовать другие синтаксические правила, не позволяющие однозначно выделить отдельные конструкции и, соответственно, сделать вывод о допустимости вырванной из контекста строки (примером такого языка является FORTRAN, особенно его ранние версии), но на практике такой синтаксис неудобен, поэтому при разработке языков конструкции стараются все-таки выделять. Это облегчает как чтение программы, так и создание трансляторов языка.

Что касается семантики, то формальные правила ее описания отсутствуют. Поэтому семантика описывается словами, или же язык использует интуитивно понятную семантику. Например, арифметическое выражение "2+2" выглядит очень понятно в силу того, что мы к нему привыкли, хотя с точки зрения математики объяснить, что такое число и что такое операция сложения двух чисел, не так-то просто.

Кроме синтаксического и семантического анализа существует еще и лексический анализ — разделение выражения на отдельные лексемы. *Лексемами* называются последовательности символов языка, которые имеют смысл только как единое целое. Например, выражение "2+3" не относится к лексемам, т. к. его части — "2", "3" и "+" — имеют значение и вне выражения, а смысл всего выражения будет суперпозицией значений этих частей. А вот идентификатор `TForm` является лексемой, т. к. его невозможно разделить на имеющие смысл части. Таким образом, лексема — это синтаксическая единица самого нижнего уровня. Описание лексических правил может быть обособлено от синтаксических, и тогда сначала лексический анализатор выделяет из выражения все лексемы, а потом синтаксический анализатор проверяет правильность выражения, составленного из этих лексем. Попутно лексический анализатор может удалять из выражения комментарии, лишние разделители и т. п.

Для разбора простого синтаксиса нет нужды проводить отдельный лексический анализ, лексемы выделяются непосредственно при синтаксическом ана-

лизе. Поэтому большинство примеров, приведенных далее, будет обходиться без лексического анализатора.

4.2. Формальное описание синтаксиса

Существует несколько различных (но, тем не менее, эквивалентных) способов описания синтаксиса. Мы здесь познакомимся только с самой употребляемой из них — *расширенной формой Бэкуса — Наура*. Эта форма была предложена Джоном Бэкусом и немного модифицирована Питером Науром, который использовал ее для описания синтаксиса языка Алгол. (Примечательно, что практически идентичная форма была независимо изобретена Ноамом Хомски для описания синтаксиса естественных языков.) В русскоязычной литературе форму Бэкуса — Наура обычно обозначают аббревиатурой *БНФ* (Бэкуса — Наура Форма). Несколько неестественный для русского языка порядок слов принят, чтобы сохранилось сходство с английской аббревиатурой BNF (Backus-Naur Form). Со временем в БНФ были добавлены новые правила описания синтаксиса, и эта форма получила название *РБНФ* — расширенная БНФ (далее для краткости мы не будем делать различия между БНФ и РБНФ). Совокупность правил, записанных в виде БНФ (или другим формализованным способом), называется *грамматикой языка*.

Основные понятия БНФ — терминальные и нетерминальные символы. *Терминальные символы* — это отдельные символы или их последовательности, являющиеся с точки зрения синтаксиса неразрывным целым, не сводимым к другим символам. Другими словами, терминальные символы — это лексемы. Терминальные символы могут состоять из одного или нескольких символов в обычном понимании этого слова. Примером терминальных символов, состоящих из нескольких символов, могут служить зарезервированные слова языка Паскаль и символы операций >=, <= и <>. Чтобы отличать терминальные символы от служебных символов БНФ, мы будем заключать их в одинарные кавычки.

Нетерминальный символ — это некоторая абстракция, которая по определенным правилам сводится к комбинации терминальных и/или других нетерминальных символов. Правила должны быть такими, чтобы существовала возможность выведения из них выражения, полностью состоящего из терминальных символов, за конечное число шагов, хотя рекурсивные определения терминальных символов друг через друга или через самих себя допускаются. Нетерминальные символы имеют имена, которые принято обрамлять угловыми скобками: `<Operator>`.

Операция `::=` означает определение нетерминального символа. Слева от этого знака ставится нетерминальный символ, смысл которого надо определить,

справа — комбинация символов, которой соответствует данный нетерминальный символ. Примером может служить следующее определение:

```
<Separator> ::= '.'
```

В данном примере мы определили нетерминальный символ <Separator>, который можем использовать в дальнейшем, например, при описании синтаксиса записи вещественного числа. Если мы затем захотим поменять разделитель с точки на запятую, нам достаточно будет переопределить смысл символа <Separator>, а не менять определения всех остальных символов, где встречается этот разделитель.

В более сложных случаях нетерминальному символу ставится в соответствие не один символ, а их цепочка, в которую могут входить как терминальные, так и нетерминальные символы. Примером такого определения может служить описание синтаксиса оператора присваивания в Delphi:

```
<Assignment> ::= <Var> ':=' <Expression>
```

При записи синтаксиса в БНФ часто сначала дают определение абстракции самого верхнего уровня, описывающей все выражение в целом, и только потом — определения абстракций нижнего уровня, которые необходимы при ее определении, т. е. порядок определения абстракций может отличаться от принятого в языках программирования определения идентификаторов, согласно которому идентификатор должен быть сначала описан, и лишь затем использован. В частности, в данном примере символы <Var> (переменная) и <Expression> (выражение) могут быть определены после определения <Assignment>.

Операция | в БНФ означает "или" — показывает одну из двух альтернатив. Например, если под нетерминальным символом <Sign> может подразумевать знак "+" или "–", его определение будет выглядеть следующим образом:

```
<Sign> ::= '+' | '-'
```

Если альтернатив больше, чем две, они записываются в ряд, разделенные символом |, например:

```
<Digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Здесь мы определили нетерминальный символ <Digit> (цифра), под которым можем понимать один из символов диапазона '0'..'9'.

Операция | подразумевает, что все, что стоит слева от этого знака, является альтернативой того, что стоит справа (до конца определения или до следующего символа |). Если в качестве альтернативы выступает только часть определения, то чтобы обособить эту часть, ее заключают в круглые скобки, например:

```
<for> ::= 'for' <Var> ':=' <Expression>
  ('to' | 'downto') <Expression> 'do' <Operator>
```

Здесь с помощью БНФ описан синтаксис оператора `for` языка Delphi.

В квадратные скобки заключается необязательная часть определения, как присутствие, так и отсутствие которой допускается синтаксисом, например:

```
<if> ::= 'if' <Condition> 'then' <Operator> ['else' <Operator>]
```

Здесь дано определение условного оператора `if` языка Delphi. Квадратные скобки указывают на необязательность части `else`.

Примечание

Строго говоря, определения операторов `if` и `for` в Delphi сложнее, чем те, которые мы здесь привели. Это связано с тем, что `<if>` и `<for>` — это альтернативы символа `<Operator>`. Поэтому может возникнуть конструкция типа `if Condition1 then if Condition2 then Operator1 else Operator2`. Из нашего определения невозможно сделать вывод о том, к какому из двух `if` в данном случае относится `else`. В языках программирования принято, что `else` относится к последнему из `if`, который еще не имеет `else`. Чтобы описать это правило, требуется более сложный синтаксис, чем мы здесь привели. Однако этот вопрос выходит за рамки данной книги и более подробно рассмотрен в [5].

Фигурные скобки означают повторение того, что в них стоит, ноль или более раз. Например, целое число без знака записывается повторением несколько раз цифр, т. е. соответствующий нетерминальный символ можно определить так:

```
<Unsigned> ::= {<Digit>}
```

Это простое определение не совсем верно, т. к. фигурные скобки указывают на повторение ноль или большее число раз, т. е. пустая строка также будет соответствовать нашему определению `<Unsigned>`. Чтобы этого не происходило, исправим наше определение:

```
<Unsigned> ::= <Digit> {<Digit>}
```

Теперь синтаксическое правило, определяемое символом `<Unsigned>`, требует, чтобы выражение состояло из одной или более цифр.

В некоторых случаях после закрывающей фигурной скобки ставят символ "+" в верхнем индексе, чтобы показать, что содержимое скобок должно повторяться не менее одного раза. Например, следующее определение `<Unsigned>` эквивалентно предыдущему:

```
<Unsigned> ::= {<Digit>}+
```

Однако это обозначение не является общепризнанным, поэтому мы не будем им пользоваться.

Этим исчерпывается набор правил БНФ. Далее мы будем использовать эти правила для описания различных синтаксических конструкций. При этом мы увидим, что, несмотря на простоту, БНФ позволяет описывать очень сложные конструкции, и это описание просто для понимания.

4.3. Синтаксис вещественного числа

Попытаемся описать синтаксис вещественного числа с помощью БНФ. Сначала опишем этот синтаксис словами: "Перед числом может стоять знак — плюс или минус. Затем идет одна или несколько цифр. Потом может следовать точка, после которой будет еще одна или несколько цифр. Затем может быть указан показатель степени "E" (большое или малое), после которого может стоять знак плюс или минус, а затем должна быть одна или несколько цифр". Указанные правила описывают синтаксис записи вещественных чисел, принятый в Delphi. Согласно им, правильными вещественными числами считаются, например, выражения "10", "0.1", "+4", "–3.2", "8.26e–5" и т. п. Такие выражения, как, например, ".6" и "–.5", этим правилам не удовлетворяют, т. к. перед десятичной точкой должна стоять хотя бы одна цифра. В некоторых языках программирования такая запись допускается, но Delphi требует обязательного наличия целой части.

Теперь переведем эти правила на язык БНФ (листинг 4.1).

Листинг 4.1. Синтаксис вещественного числа

```
<Number> ::= [<Sign>] <Digit> {<Digit>}
  [<Separator> <Digit> {<Digit>}]
  [<Exponent> [<Sign>] <Digit> {<Digit>}]
<Digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<Sign> ::= '+' | '-'
<Separator> ::= '.'
<Exponent> ::= 'E' | 'e'
```

На основе этих правил можно написать функцию `IsNumber`, которая в качестве параметра принимает строку и возвращает `True`, если эта строка удовлетворяет правилам записи числа, и `False`, если не удовлетворяет (листинг 4.2).

Листинг 4.2. Функция для определения соответствия строки синтаксису вещественного числа

```
// Проверка символа на соответствие <Digit>
function IsDigit(Ch: Char): Boolean;
begin
  Result := Ch in ['0'..'9'];
```

```
end;

// Проверка символа на соответствие <Sign>
function IsSign(Ch: Char): Boolean;
begin
  Result := (Ch = '+') or (Ch = '-');
end;

// Проверка символа на соответствие <Separator>
function IsSeparator(Ch: Char): Boolean;
begin
  Result := Ch= '.';
end;

// Проверка символа на соответствие <Exponent>
function IsExponent(Ch: Char): Boolean;
begin
  Result := (Ch = 'E') or (Ch = 'e');
end;

function IsNumber(const S: string): Boolean;
var
  P: Integer; // Номер символа выражения, который сейчас проверяется
begin
  Result := False;
  // Проверка, что выражение содержит хотя бы один символ — пустая строка
  // не является числом
  if Length(S) = 0 then
    Exit;
  // Начинаем проверку с первого символа
  P := 1;
  // Если первый символ — <Sign>, переходим к следующему
  if IsSign(S[P]) then
    Inc(P);
  // Проверяем, что в данной позиции стоит хотя бы одна цифра
  if (P > Length(S)) or not IsDigit(S[P]) then
    Exit;
  // Переходим к следующей позиции, пока не достигнем конца строки
  // или не встретим не цифру
  repeat
    Inc(P);
  until (P > Length(S)) or not IsDigit(S[P]);
  // Если достигли конца строки, выражение корректно — число,
```

```
  // не имеющее дробной части и экспоненты
  if P > Length(S) then
  begin
    Result := True;
    Exit;
  end;
  // Если следующий символ — <Separator>, проверяем, что после него
  // стоит хотя бы одна цифра
  if IsSeparator(S[P]) then
  begin
    Inc(P);
    if (P > Length(S)) or not IsDigit(S[P]) then
      Exit;
    repeat
      Inc(P);
    until (P > Length(S)) or not IsDigit(S[P]);
    // Если достигли конца строки, выражение корректно — число
    // без экспоненты
    if P > Length(S) then
    begin
      Result := True;
      Exit;
    end;
  end;
  // Если следующий символ — <Exponent>, проверяем, что после него
  // стоит все то, что требуется правилами
  if IsExponent(S[P]) then
  begin
    Inc(P);
    if P > Length(S) then
      Exit;
    if IsSign(S[P]) then
      Inc(P);
    if (P > Length(S)) or not IsDigit(S[P]) then
      Exit;
    repeat
      Inc(P);
    until (P > Length(S)) or not IsDigit(S[P]);
    if P > Length(S) then
    begin
      Result := True;
      Exit;
    end;
```

```
  end;
  // Если выполнение дошло до этого места, значит, в выражении остались
  // еще какие-то символы. Так как никакие дополнительные символы
  // синтаксисом не предусмотрены, такое выражение не считается
  // корректным числом.
end;
```

Для каждого нетерминального символа мы ввели отдельную функцию, разбор начинается с символа самого верхнего уровня — `<Number>` — и следует правилам, записанным для этого символа. Такой способ синтаксического анализа называется *левосторонним рекурсивным нисходящим анализом*. Левосторонним потому, что символы в выражении перебираются слева направо, нисходящим — потому, что сначала анализируются символы верхнего уровня, а потом — символы нижнего. Рекурсивность метода на данном примере не видна, т. к. наша грамматика не содержит рекурсивных определений, но мы с этим столкнемся в последующих примерах.

Пример использования функции `IsNumber` содержится на компакт-диске и называется IsNumberSample.

В заключение рассмотрим альтернативный способ записи грамматики вещественного числа — графический (такой способ называется синтаксическим графом, или рельсовой диаграммой). Это направленный граф, узлами которого являются терминальные (круги) и нетерминальные (прямоугольники) символы. Двигаться от одного узла к другому можно только по линиям в направлениях, указанных стрелками. В таком графе достаточно легко разобраться, а по возможностям описания синтаксиса он эквивалентен БНФ. На рис. 4.1 показана запись синтаксиса вещественного числа с помощью рельсовой диаграммы.

В качестве самостоятельного упражнения рекомендуем нарисовать с помощью рельсовой диаграммы грамматику символа "Цифра", используемого на рис. 4.1.

4.4. Простой калькулятор

Теперь у нас уже достаточно знаний, чтобы создать простейший калькулятор, т. е. функцию, которая будет на входе принимать выражение, а на выходе, если это выражение корректно, возвращать результат его вычисления. Для начала ограничимся простым калькулятором, который умеет работать только с числовыми константами и знает только четыре действия арифметики. Изменение порядка вычисления операторов с помощью скобок также оставим на потом.

Таким образом, наш калькулятор будет распознавать и вычислять цепочки чисел, между которыми стоят знаки операции, которые над этими числами

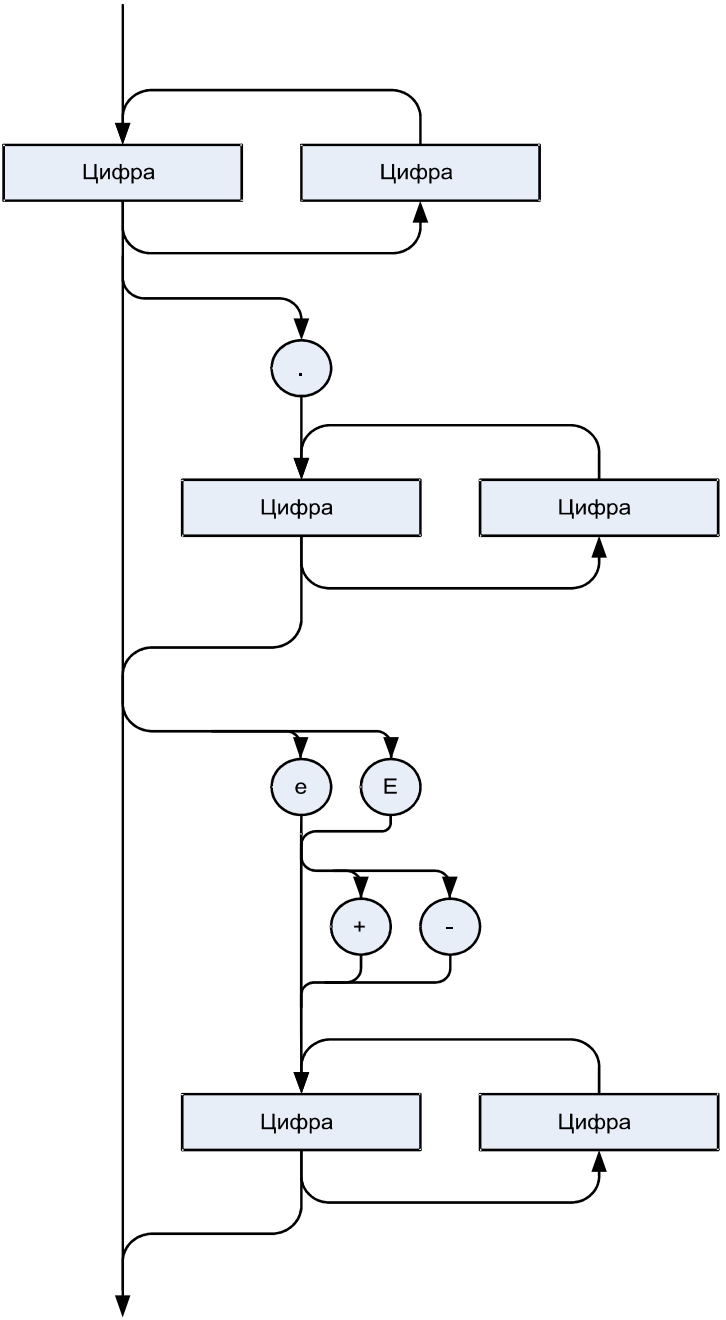


Рис. 4.1. Диаграмма синтаксиса вещественного числа

выполняются. В вырожденном случае выражение может состоять из одного числа и, соответственно, не содержать ни одного знака операции. Опишем эти правила с помощью БНФ и ранее определенного символа `<Number>`.

```
<Expr> ::= <Number> {<Operation> <Number>}
<Operation> ::= '+' | '-' | '*' | '/'
```

Примечание

В нашей грамматике не предусмотрено, что между оператором и его операндами может находиться пробел, т. е. выражение "2 + 2", в отличие от "2+2", не удовлетворяет данной грамматике. В отсутствие лексического анализатора игнорирование пробелов и прочих разделителей (переводов строки, комментариев) является трудоемкой рутинной операцией, поэтому во всех примерах без лексического анализатора мы будем требовать, чтобы выражения не содержали пробелов.

Для написания калькулятора нам понадобятся две новых функции — `IsOperator`, которая проверяет, является ли следующий символ оператором, и `Expr`, которая получает на входе строку, анализирует ее в соответствии с указанными правилами и вычисляет результат. Кроме того, функция `IsNumber` сама по себе нам тоже больше не нужна — мы создадим на ее основе функцию `Number`, которая получает на входе строку и номер позиции, начиная с которой в этой строке должно быть расположено число, проверяет, так ли это, и возвращает это число. Кроме того, функция `Number` должна перемещать указатель на следующий после числа символ строки, чтобы функция `Expr`, вызвавшая `Number`, могла узнать, с какого символа продолжать анализ. Если последовательность символов не является корректным числом, функция `Number` возбуждает исключение `ESyntaxError`, определенное специально для указания на ошибку в записи выражения.

Сама по себе задача преобразования строки в вещественное число достаточно сложна, и чтобы не отвлекаться на ее решение, мы воспользуемся функцией `StrToFloat` из модуля `SysUtils`. Когда функция `Number` выделит из строки последовательность символов, являющуюся числом, эта последовательность передается функции `StrToFloat`, и преобразованием занимается она. Здесь следует учесть два момента. Во-первых, в нашей грамматике разделителем целой и дробной части является точка, а `StrToFloat` использует системные настройки, т. е. разделителем может быть и запятая. Чтобы обойти эту проблему, слегка изменим синтаксис и будем сравнивать аргумент функции `IsSeparator` не с символом ".", а с `DecimalSeparator` (таким образом, наш калькулятор тоже станет чувствителен к системным настройкам). Во-вторых, не всякое выражение, соответствующее нашей грамматике, будет допустимым числом с точки зрения `StrToFloat`, т. к. эта функция учитывает диапазон типа `Extended`. Например, синтаксически верное выражение "2e5000" даст ис-

ключение EConvertError, т. к. данное число выходит за пределы этого диапазона. Но пока мы остаемся в рамках типа Extended, мы вынуждены мириться с этим.

Новые функции приведены в листинге 4.3.

Листинг 4.3. Реализация простейшего калькулятора

```
// Выделение из строки подстроки, соответствующей
// определению <Number>, и вычисление этого числа
// S — строка, из которой выделяется подстрока
// P — номер позиции в строке, с которой должно
// начинаться число. После завершения работы функции
// этот параметр содержит номер первого после числа
// символа
function Number(const S: string; var P: Integer): Extended;
var
  InitPos: Integer;
begin
  // InitPos нам понадобится для выделения подстроки,
  // которая будет передана в StrToFloat
  InitPos := P;
  if (P <= Length(S)) and IsSign(S[P]) then
    Inc(P);
  if (P > Length(S)) or not IsDigit(S[P]) then
    raise ESyntaxError.Create(
      'Ожидается цифра в позиции ' + IntToStr(P));
  repeat
    Inc(P);
  until (P > Length(S)) or not IsDigit(S[P]);
  if (P <= Length(S)) and IsSeparator(S[P]) then
  begin
    Inc(P);
    if (P > Length(S)) or not IsDigit(S[P]) then
      raise ESyntaxError.Create(
        'Ожидается цифра в позиции ' + IntToStr(P));
    repeat
      Inc(P);
    until (P > Length(S)) or not IsDigit(S[P]);
  end;
  if (P <= Length(S)) and IsExponent(S[P]) then
  begin
    Inc(P);
```

```
    if P > Length(S) then
      raise ESyntaxError.Create('Неожиданный конец строки');
    if IsSign(S[P]) then
      Inc(P);
    if (P > Length(S)) or not IsDigit(S[P]) then
      raise ESyntaxError.Create(
        'Ожидается цифра в позиции ' + IntToStr(P));
    repeat
      Inc(P);
    until (P > Length(S)) or not IsDigit(S[P]);
  end;
  Result := StrToFloat(Copy(S, InitPos, P - InitPos));
end;

// Проверка символа на соответствие <Operator>
function IsOperator(Ch: Char): Boolean;
begin
  Result := Ch in ['+', '-', '*', '/'];
end;

// Проверка строки на соответствие <Expr>
// и вычисление выражения
function Expr(const S: string): Extended;
var
  P: Integer;
  OpSymb: Char;
begin
  P := 1;
  Result := Number(S, P);
  while (P <= Length(S)) and IsOperator(S[P]) do
  begin
    OpSymb := S[P];
    Inc(P);
    case OpSymb of
      '+': Result := Result + Number(S, P);
      '-': Result := Result - Number(S, P);
      '*': Result := Result * Number(S, P);
      '/': Result := Result / Number(S, P);
    end;
  end;
  if P <= Length(S) then
   raise ESyntaxError.Create(
     'Некорректный символ в позиции ' + IntToStr(P));
```

```
end;
```

Код приведен практически без комментариев, т. к. он очень простой, и все моменты, заслуживающие упоминания, мы уже разобрали в тексте. На прилагаемом компакт-диске находится программа SimpleCalcSample, которая демонстрирует работу нашего калькулятора. Калькулятор выполняет действия над числами слева направо, без учета приоритета операций, т. е. вычисление выражения "2+2*2" даст 8.

Грамматика выражения является простой для разбора, т. к. разбор выражения идет слева направо, и для соотнесения очередной части строки с тем или иным нетерминальным символом на любом этапе анализа достаточно знать только следующий символ. Такие грамматики называются *LR(1)-грамматиками* (в более общем случае требуется не один символ, а одна лексема). Класс этих грамматик был исследован Кнутом.

Грамматика Паскаля не относится к классу LR(1)-грамматик из-за уже упоминавшейся проблемы отнесения `else` к тому или иному `if`. Чтобы решить эту проблему, приходится вводить два нетерминальных символа — завершенной формы оператора `if` (с `else`) и незавершенной (без `else`). Таким образом, встретив в тексте программы лексему `if`, синтаксический анализатор не может сразу отнести ее к одному из этих символов, пока не продвинется вперед и не натолкнется на наличие или отсутствие `else`. А поскольку оператор `if` может быть оператором в циклах `for`, `while` или в операторе `with`, для них также приходится вводить завершенную и незавершенную форму. Именно из-за этой проблемы Вирт (разработчик Паскаля) отказался от идеи составного оператора и модифицировал синтаксис в своем новом языке Оберон таким образом, чтобы проблема `else` не возникала.

Другое достоинство нашей простой грамматики — ее однозначность. Любая синтаксически верная строка не допускает неоднозначной трактовки. Неоднозначность могла бы возникнуть, например, если бы какая-то операция обозначалась символом "." (точка). Тогда было бы непонятно, должно ли выражение "1.5" трактоваться как число "одна целая пять десятых" или как выполнение операции над числами 1 и 5. Этот пример выглядит несколько надуманным, но неоднозначные грамматики, тем не менее, иногда встречаются на практике. Например, если запятая служит для отделения дробной части числа от целой и для разделения значений в списке параметров функций, то выражение `f(1,5)` может, с одной стороны, трактоваться как вызов функции `f` с одним аргументом 1.5, а с другой — как вызов ее с двумя аргументами 1 и 5. Правила решения неоднозначных ситуаций не описываются в виде БНФ, их приходится объяснять "на словах", что затрудняет разбор соответствующих выражений. Другой пример неоднозначной грамматики — грамматика

языков C/C++. В них оператор инкремента, записывающийся как "++", имеет две формы записи — префиксную (перед увеличиваемой переменной) и постфиксную (после переменной). Кроме того, этот оператор возвращает значение, поэтому его можно использовать в выражениях. Синтаксически допустимо, например, выражение `a+++b`, но грамматика не дает ответа, следует ли это трактовать как `(a++)+b` или как `a+(++b)`. Кроме того, т. к. существует операция "унарный плюс", возможно и третье толкование — `a+(+(+b))`.

4.5. Учет приоритета операторов

Следующим нашим шагом станет модификация калькулятора таким образом, чтобы он учитывал приоритет операций, т. е. чтобы умножение и деление выполнялись раньше сложения и умножения.

Для примера рассмотрим выражение "2*4+3*8/6". Наш синтаксис должен как-то отразить то, что аргументами операции сложения в данном случае являются не числа 4 и 3, а "2*4" и "3*8/6". В общем случае это означает, что выражение — это последовательность из одного или нескольких слагаемых, между которыми стоят знаки "+" или "–". А слагаемые — это, в свою очередь, последовательности из одного или нескольких чисел, разделенных знаками "*" и "/". А теперь запишем то же самое на языке БНФ (листинг 4.4).

Листинг 4.4. Грамматика выражения с учетом приоритета операций

```
<Expr> ::= <Term> {<Operator1> <Term>}
<Term> ::= <Number> {<Operator2> <Number>}
<Operator1> ::= '+' | '-'
<Operator2> ::= '*' | '/'
```

Примечание

Определение символа `<Operator1>` совпадает с определением введенного ранее символа `<Sign>`. Но использовать `<Sign>` в определении `<Expr>` было бы неправильно, т. к., в принципе, в выражении могут существовать и другие операции, имеющие тот же приоритет (как, например, операции арифметического или и арифметического исключающего или в Delphi), и тогда определение `<Operator1>` будет расширено. Но это не должно затронуть определение символа `<Number>`, в которое входит `<Sign>`.

Чтобы приспособить калькулятор к новым правилам, нужно заменить функцию `Operator` на `Operator1` и `Operator2`, добавить функцию `Term` (слагаемое) и внести изменения в `Expr`. Функция `Number` остается без изменения. Обновленная часть калькулятора выглядит следующим образом (листинг 4.5).

Листинг 4.5. Реализация калькулятора с учетом приоритета операций

```
// Проверка символа на соответствие <Operator1>
function IsOperator1(Ch: Char): Boolean;
begin
  Result := Ch in ['+', '-'];
end;

// Проверка символа на соответствие <Operator2>
function IsOperator2(Ch: Char): Boolean;
begin
  Result := Ch in ['*', '/'];
end;

// Выделение подстроки, соответствующей <Term>,
// и ее вычисление
function Term(const S: string; var P: Integer): Extended;
var
  OpSymb: Char;
begin
  Result := Number(S,P);
  while (P <= Length(S)) and IsOperator2(S[P]) do
  begin
    OpSymb := S[P];
    Inc(P);
    case OpSymb of
      '*': Result := Result * Number(S, P);
      '/': Result := Result / Number(S, P);
    end;
  end;
end;

// Проверка строки на соответствие <Expr>
// и вычисление выражения
function Expr(const S: string): Extended;
var
  P: Integer;
  OpSymb: Char;
begin
  P := 1;
  Result := Term(S, P);
  while (P <= Length(S)) and IsOperator1(S[P]) do
  begin
    OpSymb := S[P];
```

```
    Inc(P);
    case OpSymb of
      '+': Result := Result + Term(S, P);
      '-': Result := Result - Term(S, P);
    end;
  end;
  if P <= Length(S) then
    raise ESyntaxError.Create(
      'Некорректный символ в позиции ' + IntToStr(P));
end;
```

Если вы разобрались с предыдущими примерами, приведенный здесь код будет вам понятен. Некоторых комментариев требует только функция `Term`. Она выделяет, начиная с заданного символа, ту часть строки, которая соответствует определению `<Term>`. Вызвавшая ее функция `Expr` должна продолжить разбор выражения со следующего за этой подстрокой символа, поэтому функция `Term`, как и `Number`, имеет параметр-переменную `P`, которая на входе содержит номер первого символа слагаемого, а на выходе — номер первого после этого слагаемого символа.

Пример калькулятора, учитывающего приоритет операций, находится на компакт-диске под именем PrecedenceCalcSample. Поэкспериментировав с ним, легко убедиться, что теперь вычисление "2+2*2" дает правильное значение 6.

В заключение заметим, что язык, определяемый такой грамматикой, полностью совпадает с языком, определяемым грамматикой из предыдущего примера, т. е. любое выражение, принадлежащее первому языку, принадлежит и второму, и наоборот. Усложнение синтаксиса, которое мы здесь ввели, требуется именно для отражения семантики выражений, а не для расширения самого языка.

4.6. Выражения со скобками

Порядок выполнения операций в выражении может меняться с помощью скобок. Внутри них должно находиться выражение, которое, будучи выделенным в отдельную строку, само по себе отвечает требованиям синтаксиса к выражению в целом.

Выражение, заключенное в скобки, допустимо везде, где допускается появление отдельного числа (из этого, в частности, следует, что допускаются вложенные скобки). Таким образом, мы должны расширить нашу грамматику так, чтобы аргументом операций сложения и умножения могли служить не только числа, но и выражения, заключенные в скобки. Это автоматически

позволит использовать такие выражения и в качестве слагаемых, потому что слагаемое — это последовательность из одного или нескольких множителей, разделенных знаками умножения и деления.

На языке БНФ все сказанное иллюстрирует листинг 4.6.

Листинг 4.6. Грамматика выражения со скобками (первое приближение)

```
<Expr> ::= <Term> {<Operation1> <Term>}
<Term> ::= <Factor> {<Operation2> <Factor>}
<Factor> ::= <Number> | '('<Expr>')'
```

В этих определениях появилась рекурсия, т. к. в определении <Expr> используется (через <Term>) символ <Factor>, а в определении <Factor> — <Term>. Соответственно, подобная грамматика будет реализовываться рекурсивными функциями.

Наша грамматика не учитывает, что перед скобками может стоять знак унарной операции "+" или "–", хотя общепринятые правила записи выражений вполне допускают выражения типа `3*-(2+4)`. Поэтому, прежде чем приступить к созданию нового калькулятора, введем правила, допускающие такой синтаксис. Можно было бы модифицировать определение <Factor> таким образом:

```
<Factor> ::= <Number> | [Sign] '(' <Expr> ')'
```

Однако такой подход страдает отсутствием общности. В дальнейшем мы усложним наш синтаксис, введя другие типы множителей (функции, переменные). Перед каждым из них, в принципе, может стоять знак унарной операции, поэтому логичнее определить синтаксис таким образом, чтобы унарная операция допускалась вообще перед любым множителем. В этом случае можно будет слегка упростить определение <Number>, т. к. знак "+" или "–" в начале числа можно будет трактовать не как часть числа, а как унарный оператор, стоящий перед множителем, представленным в виде числовой константы.

С учетом этого новая грамматика запишется следующим образом (листинг 4.7).

Листинг 4.7. Окончательный вариант грамматики выражения со скобками

```
<Expr> ::= <Term> {<Operation1> <Term>}
<Term> ::= <Factor> {<Operation2> <Factor>}
<Factor> ::= <UnaryOp> <Factor> | <Number> | '('<Expr>')'
<Number> ::= <Digit> {<Digit>} [<Separator> <Digit> {<Digit>}]
  [<Exponent> [<Sign>] <Digit> {<Digit>}]
```

```
<UnaryOp> ::= '+' | '-'
```

Здесь опущены определения некоторых вспомогательных символов, которые не изменились.

Мы видим, что грамматика стала "более рекурсивной", т. е. в определении символа <Factor> используется он сам. Соответственно, функция Factor будет вызывать саму себя.

Символ <UnaryOp>, определение которого совпадает с определениями <Operator1> и <Sign>, мы делаем независимым нетерминальным символом по тем же причинам, что и ранее: в принципе, синтаксис может допускать унарные операции (как, например, not в Delphi), которые не являются ни знаками, ни допустимыми бинарными операциями.

Побочным эффектом нашей грамматики стало то, что, например, -5 воспринимается как множитель, а потому перед ним допустимо поставить унарный оператор, т. е. выражение --5 также является корректным множителем и трактуется как -(-5). А перед --5, в свою очередь, можно поставить еще один унарный оператор. И так — до бесконечности. Это может показаться не совсем правильным, но, тем не менее, такая грамматика широко распространена. Легко, например, убедиться, что компилятор Delphi считает допустимым выражение 2+-+-2, трактуя его как 2+(-(+(-2))).

Листинг 4.8 иллюстрирует реализацию данной грамматики.

Листинг 4.8. Реализация калькулятора со скобками

```
// Так как грамматика рекурсивна, функция Expr
// должна быть объявлена заранее
function Expr(const S: string; var P: Integer): Extended;
  forward;

// Выделение подстроки, соответствующей <Factor>,
// и ее вычисление
function Factor(const S: string; var P: Integer): Extended;
begin
  if P > Length(S) then
    raise ESyntaxError.Create('Неожиданный конец строки');
  // По первому символу подстроки определяем,
  // какой это множитель
  case S[P] of
    '+': // унарный "+"
      begin
        Inc(P);
```

```
        Result := Factor(S, P);
      end;
    '-': // унарный "-"
      begin
        Inc(P);
        Result := -Factor(S, P);
      end;
    '(': // выражение в скобках
      begin
        Inc(P);
        Result := Expr(S, P);
        // Проверяем, что скобка закрыта
        if (P > Length(S)) or (S[P] <> ')') then
          raise ESyntaxError.Create(
            'Ожидается ")" в позиции ' + IntToStr(P));
        Inc(P);
      end;
    '0'..'9': // Числовая константа
      Result := Number(S, P);
    else
      raise ESyntaxError.Create(
        'Некорректный символ в позиции ' + IntToStr(P));
  end;
end;

// Выделение подстроки, соответствующей <Term>,
// и ее вычисление
function Term(const S: string; var P: Integer): Extended;
var
  OpSymb: Char;
begin
  Result := Factor(S, P);
  while (P <= Length(S)) and IsOperator2(S[P]) do
  begin
    OpSymb := S[P];
    Inc(P);
    case OpSymb of
      '*': Result := Result * Factor(S, P);
      '/': Result := Result / Factor(S, P);
    end;
  end;
end;
```

```
// Выделение подстроки, соответствующей <Expr>,
// и ее вычисление
function Expr(const S: string; var P: Integer): Extended;
var
  OpSymb: Char;
begin
  Result := Term(S, P);
  while (P <= Length(S)) and IsOperator1(S[P]) do
  begin
    OpSymb := S[P];
    Inc(P);
    case OpSymb of
      '+': Result := Result + Term(S, P);
      '-': Result := Result - Term(S, P);
    end;
  end;
end;

// Вычисление выражения
function Calculate(const S: string): Extended;
var
  P: Integer;
begin
  P := 1;
  Result := Expr(S, P);
  if P <= Length(S) then
    raise ESyntaxError.Create(
      'Некорректный символ в позиции ' + IntToStr(P));
end;
```

По сравнению с предыдущим примером функция `Term` осталась такой же с точностью до замены вызовов `Number` на новую функцию `Factor`. Функция `Factor` выделяет подстроку, отвечающую отдельному множителю. Множители, напомним, могут быть трех типов: число, выражение в скобках, множитель с унарным оператором. Различить их можно по первому символу подстроки. Функция `Factor` распознает тип множителя и вызывает соответствующую функцию для его вычисления.

Функция `Expr` теперь может применяться не только к выражению в целом, но и к отдельной подстроке. Поэтому она, как и все остальные функции, теперь имеет параметр-переменную `P`, через который передается начало и конец этой подстроки. Из функции убрана проверка того, что в результате ее ис-

пользования строка проанализирована полностью, т. к. теперь допустим анализ части строки.

Функция `Expr` в своем новом виде стала не очень удобной для конечного пользователя, поэтому была описана еще одна функция — `Calculate`. Это вспомогательная функция, которая избавляет пользователя от вникания в детали "внутренней кухни" калькулятора, т. е. использования переменной `P` и проверки того, что строка проанализирована до конца.

Пример калькулятора со скобками записан на компакт-диске под названием BracketsCalcSample. Анализируя его код, можно заметить, что по сравнению с предыдущим примером незначительно изменена функция `Number` — из нее в соответствии с новой грамматикой убрана проверка знака в начале выражения.