Now, method `parseSubject`:

```
private void parseSubject () {
    if (currentTerminal matches 'I')
        accept('I');
    else
    if (currentTerminal matches 'a') {
        accept('a');
        parseNoun();
    } else
    if (currentTerminal matches 'the') {
        accept('the');
        parseNoun();
    } else
        report a syntactic error
}
```

Subject ::=

I

|

a Noun

|

the Noun

This is a little more complicated. According to the production rule, a subject must have one of three forms: '**I**', '**a** Noun', or '**the** Noun'. Method `parseSubject` must decide which form it is, and the only way to decide is to inspect the current terminal. On entry to `parseSubject`, the current terminal should contain the first terminal of the subject. If the current terminal is '**I**', then clearly the subject is of the form '**I**'; if the current terminal is '**a**', then presumably the subject is of the form '**a** Noun'; if the current terminal is '**the**', then presumably the subject is of the form '**the** Noun'; otherwise the subject is ill-formed.

Now method `parseNoun`:

```
private void parseNoun () {
    if (currentTerminal matches 'cat')
        accept('cat');
    else
    if (currentTerminal matches 'mat')
        accept('mat');
    else
    if (currentTerminal matches 'rat')
        accept('rat');
    else
        report a syntactic error
}
```

Noun ::=

cat

|

mat

|

rat

This is straightforward. According to the production rule, a noun must be '**cat**', '**mat**', or '**rat**', and `parseNoun` simply checks the contents of `currentTerminal` to discover which it is. If `currentTerminal` does not contain one of these alternatives then the noun is ill-formed.

Method `parseObject` is analogous to `parseSubject`, and `parseVerb` to `parseNoun`, so we omit the details here. (See Exercise 4.6.)

The parser is initiated using the following method:

```
public void parse () {
    currentTerminal = first input terminal;
    parseSentence();
    check that no terminal follows the sentence
}
```

This parser does not actually construct a syntax tree. But it does (implicitly) determine the input string's phrase structure. For example, `parseNoun` whenever called finds the beginning and end of a phrase of class Noun, and `parseSubject` whenever called finds the beginning and end of a phrase of class Subject. (See Figure 4.5.)

□

In general, the methods of a recursive-descent parser cooperate as follows:

- The variable `currentTerminal` will successively contain each input terminal. All parsing methods have access to this variable.
- On entry to method `parse`*N*, `currentTerminal` is supposed to contain the first terminal of an *N*-phrase. On exit from `parse`*N*, `currentTerminal` is supposed to contain the input terminal immediately following that *N*-phrase.
- On entry to method `accept` with argument *t*, `currentTerminal` is supposed to contain the terminal *t*. On exit from `accept`, `currentTerminal` is supposed to contain the input terminal immediately following *t*.

If the production rules are mutually recursive, then the parsing methods will also be mutually recursive. For this reason (and because the parsing strategy is top-down), the algorithm is called *recursive descent*.

4.3.4 Systematic development of a recursive-descent parser

A recursive-descent parser can be *systematically* developed from a (suitable) context-free grammar, in the following steps:

(1) Express the grammar in EBNF, with a single production rule for each nonterminal symbol, and perform any necessary grammar transformations. In particular, always eliminate left recursion, and left-factorize wherever possible.

(2) Transcribe each EBNF production rule $N ::= X$ to a parsing method `parse`*N*, whose body is determined by *X*.

(3) Make the parser consist of:

- a private variable `currentToken`;
- private parsing methods developed in step (2);

- private auxiliary methods `accept` and `acceptIt` (to be explained later), both of which call the scanner;
- a public `parse` method that calls `parseS` (where *S* is the start symbol of the grammar), having first called the scanner to store the first input token in `currentToken`.

Example 4.12 Recursive-descent parser for Mini-Triangle

Consider the language Mini-Triangle whose BNF grammar was given in Example 1.3. We systematically develop a Mini-Triangle parser as follows.

Step (1) is to express the grammar in EBNF, performing any necessary transformations. Recall production rules (1.2a–b):

```
Command     ::=  single-Command
               |  Command ; single-Command
```

The left recursion here is a BNF device for specifying a sequence of single-commands separated by semicolons. By eliminating the left recursion, we can specify this more directly using the '*' notation of EBNF:

```
Command     ::=  single-Command (; single-Command)*
```

Now recall production rule (1.6):

```
V-name      ::=  Identifier
```

We can simplify the grammar (for parsing purposes) by substituting Identifier for V-name wherever it appears on the right-hand side of a production rule, such as (1.3):

```
single-Command  ::=  Identifier := Expression
                   |  Identifier ( Expression )
                   |  if Expression then single-Command
                        else single-Command
                   |  ...
```

The first two alternatives above can now be left-factorized:[3]

```
single-Command  ::=  Identifier (:= Expression | ( Expression ))
                   |  if Expression then single-Command
                        else single-Command
                   |  ...
```

[3] Distinguish carefully between '(' and ')', which are EBNF grouping parentheses, and the emboldened '**(**' and '**)**', which are terminal symbols of the source language. We will consistently use this typography to distinguish between EBNF symbols and any terminal symbols that happen to resemble them.

These transformations are justified because they will make the grammar more suitable for parsing purposes. After making similar transformations to other parts of the grammar, we obtain the following complete EBNF grammar of Mini-Triangle:

```
Program             ::=  single-Command                                  (4.6)

Command             ::=  single-Command (; single-Command)*              (4.7)

single-Command      ::=  Identifier (:= Expression | ( Expression ))     (4.8)
                       |  if Expression then single-Command
                            else single-Command
                       |  while Expression do single-Command
                       |  let Declaration in single-Command
                       |  begin Command end

Expression          ::=  primary-Expression                              (4.9)
                            (Operator primary-Expression)*

primary-Expression  ::=  Integer-Literal                                 (4.10)
                       |  Identifier
                       |  Operator primary-Expression
                       |  ( Expression )

Declaration         ::=  single-Declaration (; single-Declaration)*      (4.11)

single-Declaration  ::=  const Identifier ~ Expression                   (4.12)
                       |  var Identifier : Type-denoter

Type-denoter        ::=  Identifier                                      (4.13)
```

We have excluded production rules (1.10) through (1.13), which specify the syntax of operators, identifiers, literals, and comments, all in terms of individual characters. This part of the syntax is called the language's *lexicon* (or *microsyntax*). The lexicon is of no concern to the parser, which will view each identifier, literal, and operator as a single token. Instead, the lexicon will later be used to develop the scanner, in Section 4.5.

We shall assume that the scanner returns tokens of class `Token`, defined in Example 4.2. Each token consists of a kind and a spelling. The parser will examine only the kind of each token.

Step (2) is to convert each EBNF production rule to a parsing method. The parsing methods will be as follows:

```
private void parseProgram ();
private void parseCommand ();
private void parseSingleCommand ();
private void parseExpression ();
private void parsePrimaryExpression ();
private void parseDeclaration ();
private void parseSingleDeclaration ();
```

```
private void parseTypeDenoter ();
private void parseIdentifier ();
private void parseIntegerLiteral ();
private void parseOperator ();
```

Here is method `parseSingleDeclaration`:

```
private void parseSingleDeclaration () {
   switch (currentToken.kind) {                    single-Declaration ::=

   case Token.CONST:
      {
         acceptIt();                                   const
         parseIdentifier();                            Identifier
         accept(Token.IS);                             ~
         parseExpression();                            Expression
      }
      break;

   case Token.VAR:
      {
         acceptIt();                                   var
         parseIdentifier();                            Identifier
         accept(Token.COLON);                          :
         parseTypeDenoter();                           Type-denoter
      }
      break;

   default:
      report a syntactic error
   }
}
```

Note the use of the auxiliary method `acceptIt`, which unconditionally fetches the next token from the source program. The following is also correct:

```
case Token.VAR:
   {
      accept(Token.VAR);                             var
      parseIdentifier();                             Identifier
      accept(Token.COLON);                           :
      parseTypeDenoter();                            Type-denoter
   }
   break;
```

Here '`accept(Token.VAR);`' would check that the current token is of kind `Token.VAR`. In this context, however, such a check is redundant.

Now here is method `parseCommand`:


```
private void parseCommand () {                   Command ::=
   parseSingleCommand();                            single-Command
   while (currentToken.kind
         == Token.SEMICOLON)
   {                                                (
      acceptIt();                                    ;
      parseSingleCommand();                          single-Command
   }                                                 )*
}
```

This method illustrates something new. The EBNF notation '(**;** single-Command)*' signifies a sequence of zero or more occurrences of '**;** single-Command'. To parse this we use a while-loop, which is iterated zero or more times. The condition for continuing the iteration is simply that the current token is a semicolon.

Method `parseDeclaration` is similar to `parseCommand`. The remaining methods are as follows:

```
private void parseProgram () {                   Program ::=
   parseSingleCommand();                            single-Command
}

private void parseSingleCommand () {             single-Command ::=
   switch (currentToken.kind) {

   case Token.IDENTIFIER:
      {
         parseIdentifier();                          Identifier
         switch (currentToken.kind) {                (
         case Token.BECOMES:
            {
               acceptIt();                           :=
               parseExpression();                    Expression
            }
            break;

         case Token.LPAREN:                          |
            {
               acceptIt();                           (
               parseExpression();                    Expression
               accept(Token.RPAREN);                 )
            }
            break;

         default:
            report a syntactic error
         }                                           )
      }
      break;
```

```
        case Token.IF:                          |
            {
                acceptIt();                     if
                parseExpression();              Expression
                accept(Token.THEN);             then
                parseSingleCommand();           single-Command
                accept(Token.ELSE);             else
                parseSingleCommand();           single-Command
            }
            break;

        case Token.WHILE:                       |
            {
                acceptIt();                     while
                parseExpression();              Expression
                accept(Token.DO);               do
                parseSingleCommand();           single-Command
            }
            break;

        case Token.LET:                         |
            {
                acceptIt();                     let
                parseDeclaration();             Declaration
                accept(Token.IN);               in
                parseSingleCommand();           single-Command
            }
            break;

        case Token.BEGIN:                       |
            {
                acceptIt();                     begin
                parseCommand();                 Command
                accept(Token.END);              end
            }
            break;

        default:
            report a syntactic error
        }
    }

    private void parseExpression () {           Expression ::=
        parsePrimaryExpression();                   primary-Expression
        while (currentToken.kind                |
                == Token.OPERATOR) {            (
```



```
            parseOperator();                    Operator
            parsePrimaryExpression();           primary-Expression
        }                                       )*
    }

    private void parsePrimaryExpression () {    primary-Expression ::=
        switch (currentToken.kind) {
        case Token.INTLITERAL:
            parseIntegerLiteral();              Integer-Literal
            break;

        case Token.IDENTIFIER:                  |
            parseIdentifier();                  Identifier
            break;

        case Token.OPERATOR:                    |
            {
                parseOperator();                Operator
                parsePrimaryExpression();       primary-Expression
            }
            break;

        case Token.LPAREN:                      |
            {
                acceptIt();                     (
                parseExpression();              Expression
                accept(Token.RPAREN);           )
            }
            break;

        default:
            report a syntactic error
        }

        private void parseTypeDenoter () {      Type-denoter ::=
            parseIdentifier();                      Identifier
        }
```

The nonterminal symbol Identifier corresponds to a single token, so the method `parseIdentifier` is similar to `accept`:

```
    private void parseIdentifier () {
        if (currentToken.kind == Token.IDENTIFIER)
            currentToken = scanner.scan();
        else
            report a syntactic error
    }
```

The methods `parseIntegerLiteral` and `parseOperator` are analogous.[4]

Step (3) is to assemble the complete parser:

```
public class Parser {

    private Token currentToken;

    private void accept (byte expectedKind) {
        if (currentToken.kind == expectedKind)
            currentToken = scanner.scan();
        else
            report a syntactic error
    }

    private void acceptIt () {
        currentToken = scanner.scan();
    }

    ...  // Parsing methods, as above.

    public void parse () {
        currentToken = scanner.scan();
        parseProgram();
        if (currentToken.kind != Token.EOT)
            report a syntactic error
    }
}
```

The parser reads the next input token by calling the scanner. The method call '`scanner.scan()`' constructs the next token from the input and returns it. (This will be explained in Section 4.5.)

Note the following points:

- The parser examines only the kind of the current token, ignoring its spelling.
- After parsing the program, `parse` checks that the token following the program is the end-of-text.
- The parsing methods are mutually recursive (because the production rules are mutually recursive). For example, `parseCommand` calls `parseSingleCommand`, which may call `parseCommand` recursively.

□

[4] Later we shall enhance method `parseIdentifier` to construct an AST terminal node containing the identifier's spelling. It would be wrong to write simply '`accept(Token.IDENTIFIER);`', because this would discard the identifier token, including its spelling. The same point applies to `parseIntegerLiteral`, and `parseOperator`.

Having worked through a complete example, let us now study in general terms how we systematically develop a recursive-descent parser from a suitable grammar. The two main steps are: (1) express the grammar in EBNF, performing any necessary transformations; and (2) convert the EBNF production rules to parsing methods. It will be convenient to examine these steps in reverse order.

Converting EBNF production rules to parsing methods

Consider an EBNF production rule $N ::= X$. We convert this production rule to a parsing method named `parse`*N*. This method's body will be derived from the extended RE *X*:

```
private void parseN () {
    parse X
}
```

Here 'parse *X*' is supposed to parse an *X*-phrase, i.e., a terminal string generated by *X*. (And of course the task of method `parse`*N* is to parse an *N*-phrase.)

Next, we perform stepwise refinement on 'parse *X*', decomposing it according to the structure of *X*. (In the following, *X* and *Y* stand for arbitrary extended REs.)

- We refine 'parse ε' to a dummy statement.
- We refine 'parse *t*' (where *t* is a terminal symbol) to:

  ```
  accept(t);
  ```

  In a situation where the current terminal is already known to be *t*, the following is also correct and more efficient:

  ```
  acceptIt();
  ```

- We refine 'parse *N*' (where *N* is a nonterminal symbol) to a call of the corresponding parsing method:

  ```
  parseN();
  ```

- We refine 'parse *X Y*' to:

  ```
  {
      parse X
      parse Y
  }
  ```

  The reasoning behind this is simple. The input must consist of an *X*-phrase followed by a *Y*-phrase. Since the parser works from left to right, it must parse the *X*-phrase and then parse the *Y*-phrase.

  This refinement rule is easily generalized to 'parse $X_1 \ldots X_n$'.

- We refine 'parse $X \mid Y$' to:

```
switch (currentToken.kind) {
cases in starters[[X]]:
    parse X
    break;
cases in starters[[Y]]:
    parse Y
    break;
default:
    report a syntactic error
}
```

The reasoning behind this is also straightforward. The input must consist of either an *X*-phrase or a *Y*-phrase. The parser must parse one of these, and it must decide immediately which it will be. It should choose 'parse *X*' only if the current token is one that can start an *X*-phrase (since otherwise 'parse *X*' would certainly fail). And likewise it should choose 'parse *Y*' only if the current token is one that can start a *Y*-phrase. We can express these conditions abstractly in terms of the starter sets of *X* and *Y*, and concretely in terms of Java case labels.

The parser will work correctly only if *starters*[[*X*]] and *starters*[[*Y*]] are disjoint. Otherwise the parser could not know whether to parse an *X*-phrase or a *Y*-phrase. In fact, if token *t* is in both *starters*[[*X*]] and *starters*[[*Y*]], the switch-statement will contain two occurrences of '**case** *t*:', and will fail to compile. (See Example 4.15.)

This refinement rule is easily generalized to 'parse $X_1 \mid \ldots \mid X_n$'.

- We refine 'parse *X**' to:

```
while (currentToken.kind is in starters[[X]])
    parse X
```

The reasoning behind this is as follows. The input must consist of zero or more consecutive *X*-phrases. The parser must repeatedly parse *X*-phrases, and it does this by means of a while-loop. Before each iteration, it must decide whether to terminate or to continue parsing *X*-phrases. It should continue only if the current token is one that can start an *X*-phrase (since otherwise 'parse *X*' would certainly fail).

The parser will work correctly only if *starters*[[*X*]] is disjoint from the set of tokens that can follow *X** in this particular context. Suppose that some token *t* is in *starters*[[*X*]] and can also follow *X**. When the current token is *t*, the parser will continue parsing *X*-phrases even when it should terminate. (See Example 4.16.)

The following examples illustrate the stepwise refinement of parsing methods.

Example 4.13 Stepwise refinement of `parseCommand`

Let us follow the stepwise refinement of the method `parseCommand` of Example 4.12, starting from production rule (4.7):

Command ::= single-Command (**;** single-Command)*

We start with the following outline of the method:

```
private void parseCommand () {
    parse single-Command (; single-Command)*
}
```

Now we refine 'parse single-Command (**;** single-Command)*' to:

```
parseSingleCommand();
parse (; single-Command)*
```

Now we refine 'parse (**;** single-Command)*' to:

```
while (currentToken.kind == Token.SEMICOLON)
    parse (; single-Command)
```

since *starters*[[**;** single-Command]] = {**;**}.

Finally we refine 'parse (**;** single-Command)' to:

```
{
    acceptIt();
    parseSingleCommand();
}
```

In this situation we know already that the current token is a semicolon, so '`acceptIt();`' is a correct alternative to '`accept(Token.SEMICOLON);`'. □

Example 4.14 Stepwise refinement of `parseSingleDeclaration`

Let us also follow the stepwise refinement of the method `parseSingleDeclaration` of Example 4.12, starting from production rule (4.11):

single-Declaration ::= **const** Identifier **~** Expression
| **var** Identifier **:** Type-denoter

We start with the following outline of the method:

```
private void parseSingleDeclaration () {
    parse const Identifier ~ Expression | var Identifier : Type-denoter
}
```

Now we refine 'parse **const** ... | **var** ...' to:

```
switch (currentToken.kind) {

case Token.CONST:
    parse const Identifier ~ Expression
    break;
```

```
    case Token.VAR:
        parse var Identifier : Type-denoter
        break;

    default:
        report a syntactic error
}
```

since *starters*[[**const** ...]] = {**const**} and *starters*[[**var** ...]] = {**var**}. Fortunately, these starter sets are disjoint.

Finally, we refine 'parse **const** Identifier ~ Expression' to:

```
{
    acceptIt();
    parseIdentifier();
    accept(Token.IS);
    parseExpression();
}
```

and 'parse **var** Identifier : Type-denoter' similarly, as shown in Example 4.12.

□

In defining how to refine 'parse $X \mid Y$' and 'parse X^*', we stated certain conditions that must be satisfied. These conditions are:

- If the grammar contains $X \mid Y$, *starters*[[X]] and *starters*[[Y]] must be disjoint.
- If the grammar contains X^*, *starters*[[X]] must be disjoint from the set of tokens that can follow X^* in this particular context.

A grammar that satisfies both these conditions is called an ***LL(1) grammar***.

Recursive-descent parsing is suitable *only* for LL(1) grammars.

Not all programming language grammars are LL(1). In practice, however, nearly every programming language grammar can easily be transformed to make it LL(1), without changing the language it generates. Why this should be so is a matter for conjecture, but often a language designer will consciously design the new language's syntax to be suitable for recursive-descent parsing.

The following examples illustrate grammars that are not LL(1). However, simple transformations of these grammars are sufficient to make them LL(1).

Example 4.15 Non-LL(1) grammar for Mini-Triangle

Recall production rules (1.3a–f) in the original grammar of Mini-Triangle:


```
single-Command  ::=  V-name := Expression
                 |   Identifier ( Expression )
                 |   if Expression then single-Command
                         else single-Command
                 |   ...
```

The relevant starter sets are:

starters[[V-name := Expression]] = *starters*[[V-name]] = {Identifier}

starters[[Identifier (Expression)]] = {Identifier}

starters[[**if** Expression **then** ...]] = {**if**}

The first two are *not* disjoint, so the grammar is not LL(1).

What would happen if we tried to develop a parsing method directly from the above production rule? The parsing method would turn out as follows:

```
private void parseSingleCommand () {
    switch (currentToken.kind) {

    case Token.IDENTIFIER: {
            parseVname();
            accept(Token.BECOMES);
            parseExpression();
        }
        break;

    case Token.IDENTIFIER: {
            parseIdentifier();
            accept(Token.LPAREN);
            parseExpression();
            accept(Token.RPAREN)
        }
        break;

    case Token.IF:
        ...

    default:
        ...
    }
}
```

This parser is clearly incorrect, and will not compile due to the duplicate case label.

Fortunately the problematic production rule can easily be transformed, by substitution and left factorization, to solve this particular problem. This was done in Example 4.12.

□

Example 4.16 Non-LL(1) grammar for Algol

Consider the following production rules taken from a grammar of Algol:

Block ::= **begin** Declaration (**;** Declaration)* **;** Command **end**

Declaration ::= **integer** Identifier (**,** Identifier)*

Here *starters*[[**;** Declaration]] = {**;**}, and the set of terminals that can follow '(**;** Declaration)*' in this context is {**;**}. These sets are not disjoint, so the grammar is not LL(1).

If we tried to develop a parsing method directly from the production rule defining Block, we would get:

```
private void parseBlock () {
    accept(Token.BEGIN);
    parseDeclaration();
    while (currentToken.kind == Token.SEMICOLON)
    {
        acceptIt();
        parseDeclaration();
    }
    accept(Token.SEMICOLON);
    parseCommand();
    accept(Token.END);
}
```

This is clearly incorrect. Iteration will continue as long as the current token is a semicolon. But this might be the semicolon that separates the declarations from the command, e.g., the second semicolon in:

```
begin integer i; integer j; i := j+1 end
```

Then `parseBlock` would attempt to parse the command '`i := j+1`' as a declaration.

Fortunately, we can transform the production rule defining Block:

Block ::= **begin** Declaration **;** (Declaration **;**)* Command **end**

This does not affect the generated language, but leads to the following correct parsing method:

```
private void parseBlock () {
    accept(Token.BEGIN);
    parseDeclaration();
    accept(Token.SEMICOLON);
    while (currentToken.kind == Token.INTEGER)
    {
        parseDeclaration();
        accept(Token.SEMICOLON);
    }
    parseCommand();
    accept(Token.END);
}
```


This eliminates the problem, assuming that *starters*[[Declaration **;**]] is disjoint from *starters*[[Command]].

□

The above examples are quite typical. Although the LL(1) condition is quite restrictive, in practice most programming language grammars can be transformed to make them LL(1) and thus suitable for recursive-descent parsing.

Performing grammar transformations

Left factorization is essential in some situations, as illustrated by the following example.

Example 4.17 Left factorization

In Example 4.12, the production rule 'V-name ::= Identifier' was eliminated. The occurrences of V-name on the right-hand sides of (1.3a) and (1.5b) were simply replaced by Identifier, giving:

single-Command ::= Identifier **:=** Expression
| Identifier **(** Expression **)**
| **if** Expression **then** single-Command
else single-Command
| ...

The starter sets are not disjoint:

starters[[Identifier **:=** Expression]] = {Identifier}

starters[[Identifier **(** Expression **)**]] = {Identifier}

However, the substitution created an opportunity for left factorization:

single-Command ::= Identifier (**:=** Expression | **(** Expression **)**)
| **if** Expression **then** single-Command
else single-Command
| ...

This is an improvement, since now the relevant starter sets *are* disjoint:

starters[[**:=** Expression]] = {**:=**}

starters[[**(** Expression **)**]] = {**(**}

□

Left recursion must always be eliminated if the grammar is to be LL(1). The following example shows why.

Example 4.18 Left recursion elimination

Recall production rules (1.2a–b) in the grammar of Mini-Triangle:

```
Command     ::=  single-Command
              |  Command ; single-Command
```

In Example 4.12 we eliminated this left recursion, yielding:

```
Command     ::=  single-Command (; single-Command)*
```

What would happen if we omitted this transformation? First we would compute the relevant starter sets:

starters[[single-Command]] = { Identifier, **if**, **while**, **let**, **begin**}
starters[[Command ; single-Command]] = { Identifier, **if**, **while**, **let**, **begin**}

Then we would write the parsing method like this:

```
private void parseCommand () {
   switch (currentToken.kind) {

   case Token.IDENTIFIER:
   case Token.IF:
   case Token.WHILE:
   case Token.LET:
   case Token.BEGIN:
      parseSingleCommand();
      break;

   case Token.IDENTIFIER:
   case Token.IF:
   case Token.WHILE:
   case Token.LET:
   case Token.BEGIN:  {
         parseCommand();
         accept(Token.SEMICOLON);
         parseSingleCommand();
      }
      break;

   default:
      report a syntactic error
   }
}
```

This method cannot tell which way to go if the current token is an identifier, '`if`', '`while`', '`let`', or '`begin`'. It simply does not have the information required to make a correct decision. (In fact, this method will fail to compile due to the duplicate case labels.)

□

In general, a grammar that exhibits left recursion cannot be LL(1). Any attempt to convert left-recursive production rules directly into parsing methods would result in an incorrect parser. It is easy to see why. Given the left-recursive production rule:

$N ::= X \mid N\,Y$

we find:

$starters[\![N\,Y]\!] = starters[\![N]\!] = starters[\![X]\!] \cup starters[\![N\,Y]\!]$

so $starters[\![X]\!]$ and $starters[\![N\,Y]\!]$ cannot be disjoint.

4.4 Abstract syntax trees

A recursive-descent parser determines the source program's phrase structure *implicitly*, in the sense that it finds the beginning and end of each phrase. In a one-pass compiler, this is quite sufficient for the syntactic analyzer to know when to call the contextual analyzer and code generator. In a multi-pass compiler, however, the syntactic analyzer must construct an *explicit* representation of the source program's phrase structure. Here we shall assume that the representation is to be an AST.

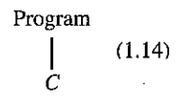
4.4.1 Representation

The following example illustrates how we can define ASTs in Java.

Example 4.19 Abstract syntax trees of Mini-Triangle

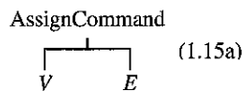
Figure 4.4 shows an example of a Mini-Triangle AST. Below we summarize all possible forms of Mini-Triangle AST, showing how each form relates to one of the production rules of the Mini-Triangle abstract syntax (Example 1.5):

- Program ASTs (*P*):

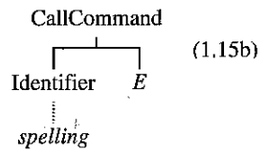


(1.14)

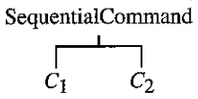
- Command ASTs (*C*):



(1.15a)



(1.15b)



(1.15c)