Chapter 2: Grammars

Aarne Ranta

Slides for the book "Implementing Programming Languages. An Introduction to Compilers and Interpreters", College Publications, 2012.

Grammars

Hands-on introduction to BNFC (the BNF Converter)

Step by step grammar writing

Lexer and parser generation

Testing grammars

Everything needed for solving Assignment 1 - a parser for a fragment of C++

Defining a language

A **grammar** is a systems of rules for a language.

Used for teaching languages at school:

- how words are formed (e.g. the plural of *baby* is *babies*)
- how words are combined to sentences (e.g. word order)

Used in **linguistics** to *describe* languages.

- remains an open problem: "all grammars leak"

Used in compilers to *define* languages.

- grammars are complete by definition

Example: a grammar of arithmetic expressions

```
EAdd. Exp  ::= Exp  "+" Exp1 ;
ESub. Exp  ::= Exp  "-" Exp1 ;
EMul. Exp1 ::= Exp1 "*" Exp2 ;
EDiv. Exp1 ::= Exp1 "/" Exp2 ;
EInt. Exp2 ::= Integer ;

coercions Exp 2 ;
```

`Calc.cf`, **a labelled BNF grammar for integer arithmetic.**

In words: expressions (`Exp`) are built with the operators `+`, `-`, `*`, and `/`, ultimately from integers.

Digit suffixes (`Exp1`, `Exp2`) and `coercions` to be explained later.

BNF = Backus Naur Form

(Named after John Backus and Peter Naur's work in the 1950-60's)

Routinely used for the specification of programming languages, appearing in language manuals.

The parser can be automatically derived from a BNF grammar.

The BNFC tool derives a parser and some other components.

Using BNFC

Install BNFC from the home page (Linux, Mac OS, Windows)

Then type, in a Unix-style shell,

```
bnfc
```

to get a usage message.

Type

```
bnfc -m Calc.cf
```

to process the file \texttt{Calc.cf}.

The system will respond by generating a bunch of files:

```
writing file AbsCalc.hs    # abstract syntax
writing file LexCalc.x     # lexer
writing file ParCalc.y     # parser
writing file DocCalc.tex   # language document
writing file SkelCalc.hs   # syntax-directed translation skeleton
writing file PrintCalc.hs  # pretty-printer
writing file TestCalc.hs   # top-level test program
writing file ErrM.hs       # monad for error handling
writing file Makefile      # Makefile
```

These files are different components of a compiler.

Most of them are Haskell (`.hs`) files, but you can also say e.g.

```
bnfc -m -java Calc.cf
```

to generate the components for Java.

Running BNFC for Haskell

One of the generated files is a `Makefile`, which specifies the commands for compiling the compiler.

So now type

```
make
```

which succeeds if you have the Haskell tools GHC, Happy, Alex.

The process terminates with the message

```
Linking TestCalc ...
```

`TestCalc` is a program for testing the parser defined by `Calc.cf`.

Testing the parser

`TestCalc` reads Unix standard input:

```
echo "5 + 6 * 7" | ./TestCalc
```

Response:

```
Parse Successful!

[Abstract Syntax]
EAdd (EInt 5) (EMul (EInt 6) (EInt 7))

[Linearized tree]
5 + 6 * 7
```

The **abstract syntax tree** is the result of parsing.

The **linearization** is the string obtained from the tree by using the grammar in the opposite direction.

This linearization can be different from the input string, for instance, if the input has unnecessary parentheses.

Reading input from a file

With the standard input method:

```
./TestCalc < FILE_with_an_expression
```

With a file name argument:

```
./TestCalc FILE_with_an_expression
```

Running BNFC for Java

If you use Java rather:

```
bnfc -m -java Calc.cf
```

More files are generated:

```
Calc/Absyn/Exp.java          # abstract syntax
Calc/Absyn/EAdd.java
Calc/Absyn/ESub.java
Calc/Absyn/EMul.java
Calc/Absyn/EDiv.java
Calc/Absyn/EInt.java
Calc/PrettyPrinter.java      # pretty-printer
Calc/VisitSkel.java          # syntax-directed translation skeleton
Calc/ComposVisitor.java      # utilities for syntax-dir. transl
Calc/AbstractVisitor.java
Calc/FoldVisitor.java
Calc/AllVisitor.java
Calc/Test.java               # top-level test file
Calc/Yylex                   # lexer
Calc/Calc.cup                # parser
Calc.tex                     # language document
Makefile                     # Makefile
```

Compiling the Java files

The `Makefile` works exactly like before:

```
make
```

You need Javac, Cup, and JLex instead of the Haskell tools.

A common problem:

```
java  JLex.Main Calc/Yylex
Exception in thread "main" java.lang.NoClassDefFoundError: JLex/Main
make: *** [Calc/Yylex.java] Error 1
```

Fixing it:

```
export CLASSPATH=.:/usr/local/java/Cup:/usr/local/java
```

Runnins the Java parser

```
echo "5 + 6 * 7" | java Calc/Test

Parse Successful!

[Abstract Syntax]
(EAdd (EInt 5) (EMul (EInt 6) (EInt 7)))

[Linearized Tree]
5 + 6 * 7
```

A summary of BNFC

We can use a BNF grammar to generate several compiler components:

- lexer, parser, linearizer, abstract syntax, test program

The components can be generated in different languages from the same BNF source:

- C, C++, C#, Haskell, Java, OCaml

Rules and categories

A BNFC source file is a set of **rules**

Most rules have the format

Label . *Category* ::= *Production* ;

The *Label* and *Category* are **identifiers** (without quotes).

The *Production* is a sequence of two kinds of items:

- identifiers, called **nonterminals**
- **string literals** (strings in double quotes), called **terminals**

The semantics of a BNF rule

Label . *Category* ::= *Production* ;

A **tree** of type *Category* can be built with *Label* as the topmost node, from any sequence specified by the production, whose nonterminals give the subtrees of the tree built.

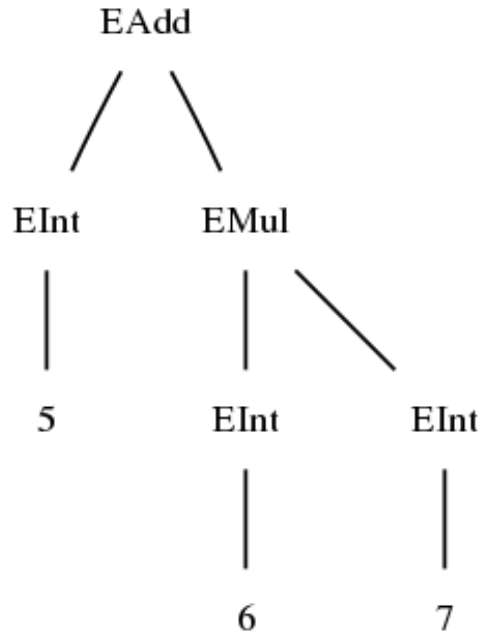
The **type** of the trees is a **categories** of the grammar, e.g. expression, statement, program,. . .

Tree labels are the **constructors** of those categories, i.e. the nodes of abstract syntax trees.

The tree for 5 + 6 * 7

In linear notation (as in Haskell), as well as graphically

```
EAdd (EInt 5) (EMul (EInt 6) (EInt 7))
```



Precedence levels

Why the is the `EMul` below the `EAdd` node? Why doesn't `5 + 6 * 7` give

```
EMul (EAdd (EInt 5) (EInt 6)) (EInt 7)
```

Answer: multiplication expressions have a **higher precedence**.

In BNFC, **precedence levels** are the digits attached to category symbols:

- `Exp1` has precedence level 1,
- `Exp2` has precedence level 2, etc.
- `Exp` is a shorthand for `Exp0`

The rule

```
EAdd. Exp ::= Exp "+" Exp1 ;
```

can be read:

> `EAdd` forms an expression of level 0 from an expression of level 0 on the left of + and of level 1 on the right.

The rule

```
EMul. Exp1 ::= Exp1 "*" Exp2 ;
```

can be read

> `EMul` form an expression of level 1 from an expression of level 1 on the left of * and of level 2 on the right.

The semantics of precedence

All precedence variants of a nonterminal denote the same type in the abstract syntax.

- Thus `2`, `2 + 2`, and `2 * 2` are all of type `Exp`.

An expression of higher level can always be used on lower levels as well.

- Thus `2 + 3` is correct: integer literals have level 2, but are here used on level 0 on the left and on level 1 on the right.

An expression of any level can be lifted to the highest level by putting it in parentheses.

- Thus `(5 + 6)` is an expression of level 2.

The coercions macro

If the highest precedence level is specified, BNFC can generate a bunch of rule.

Example:

```
coercions Exp 2
```

generates the "ordinary" BNF rules

```
_. Exp0 ::= Exp1 ;
_. Exp1 ::= Exp2 ;
_. Exp2 ::= "(" Exp0 ")" ;
```

The underscore _ is a **dummy label**, which indicates that no constructor is added.

Abstract and concrete syntax

Abstract syntax trees are the hub of a modern compiler

- the target of the parser
- the place where most compilation phases happen, e.g. type checking and code generation

Abstract syntax is purely about structure:

- what are the immediate parts of this expression, and the parts of those parts?

Abstract syntax ignores the questions

- what do the parts look like
- what is the order of the parts (to some extent)

Example: from an abstract syntax point of view, all of the following expressions are the same:

`2 + 3`	Java, C (infix)
`(+ 2 3)`	Lisp (prefix)
`(2 3 +)`	postfix

`bipush 2`
`bipush 3`
`iadd`
	JVM (postfix)

the sum of 2 and 3	English (prefix/mixfix)
2:n ja 3:n summa	Finnish (postfix/mixfix)

The simplest possible compiler

1. Parse the source language expression, e.g. `2 + 3`.
2. Obtain an abstract syntax tree, `EAdd (EInt 2) (EInt 3)`.
3. Linearize the tree to another format, `bipush 2 bipush 3 iadd`.

Not always so simple, though: the tree may have to be converted to another tree before code generation

- add type annotations
- optimize

Abstract and concrete syntax

A BNF grammar simultaneously specifies **concrete syntax**:

- what the expression parts look like
- what order they appear in
- precedences

BNFC rule

```
EAdd. Exp0 ::= Exp0 "+" Exp1
```

Its purely abstract syntax part ("skeleton")

```
EAdd. Exp  ::= Exp      Exp
```

which hides the actual symbol used for addition (and thereby the place where it appears). It also hides the precedence levels, since they don't imply any differences in the abstract syntax trees.

From concrete to abstract syntax

1. Remove all terminals.
2. Remove all precedence numbers.
3. Remove all `coercions` rules.

From `Calc.cf`, we obtain

```
EAdd. Exp ::= Exp Exp ;
ESub. Exp ::= Exp Exp ;
EMul. Exp ::= Exp Exp ;
EDiv. Exp ::= Exp Exp ;
EInt. Exp ::= Integer ;
```

From abstract to concrete syntax

1. Add any terminals
2. Define precedences in any way you want

From the `Calc.cf` skeleton, you can obtain a JVM grammar

```
EAdd. Exp ::= Exp Exp "iadd" ;
ESub. Exp ::= Exp Exp "isub" ;
EMul. Exp ::= Exp Exp "imul" ;
EDiv. Exp ::= Exp Exp "idiv" ;
EInt. Exp ::= "bipush" Integer ;
```

Abstract trees and parse trees

Abstract syntax trees (AST's)

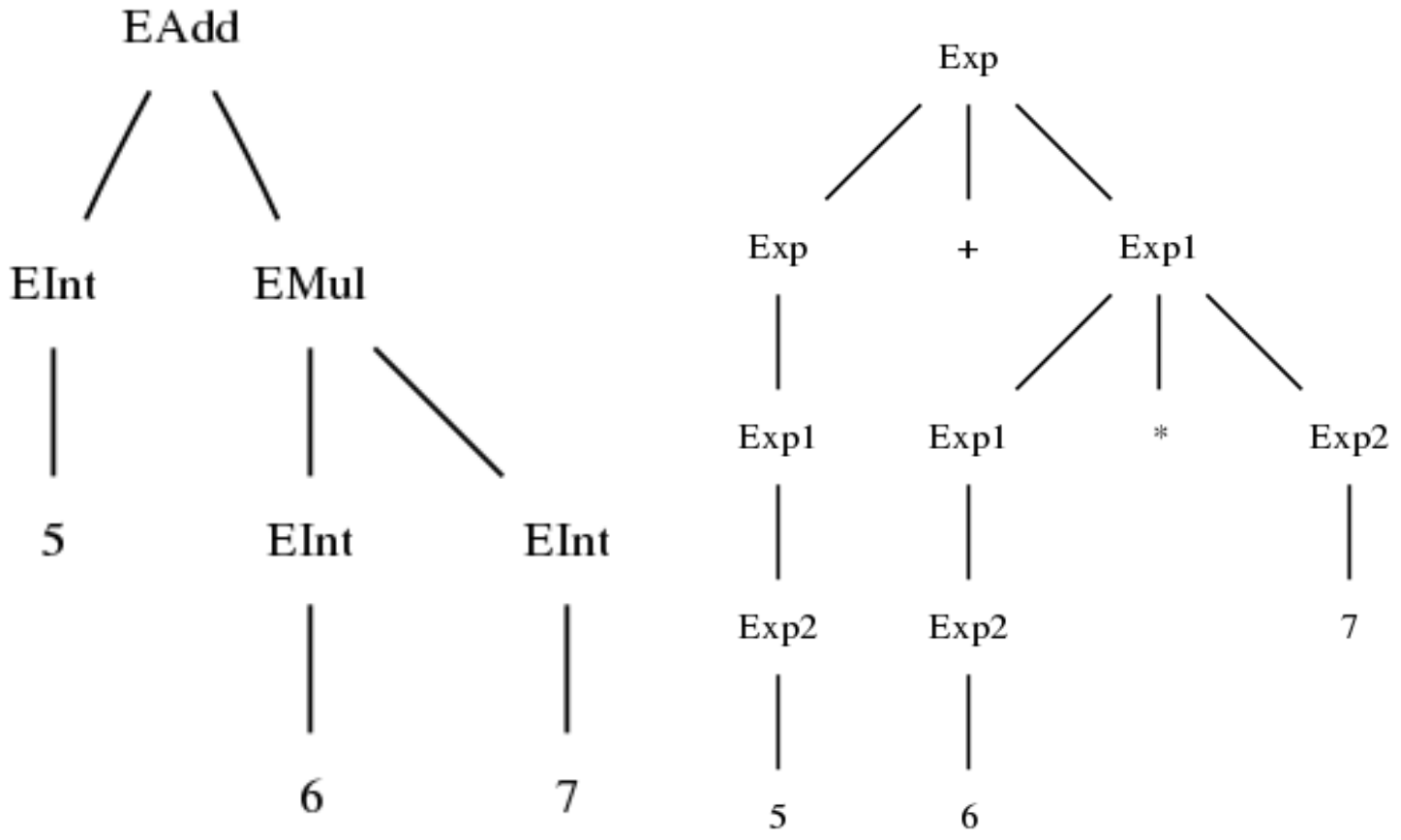
- the nodes are constructors (i.e. labels of BNF rules).
- the leaves are constructors

Concrete syntax trees a.k.a. **parse trees**

- the nodes are category symbols (i.e. nonterminals)
- the leaves are tokens (i.e. terminals)

A parse tree is an accurate encoding of all BNF rules applied, and hence it shows all coercions between precedence levels and all tokens in the input string.

Abstract and concrete trees for 5 + 6 * 7



Abstract syntax in Haskell

The BNF grammar is converted to a system of **algebraic datatypes**.

For `Calc.cf`, we obtain

```
data Exp =
   EAdd Exp Exp
 | ESub Exp Exp
 | EMul Exp Exp
 | EDiv Exp Exp
 | EInt Integer
```

Syntax-directed translation

Structural recursion on abstract syntax trees.

In Haskell, defined by **pattern matching**.

Example: a calculator, i.e. an interpreter for the language `Calc`:

```
module Interpreter where

import AbsCalc

eval :: Exp -> Integer
eval x = case x of
  EAdd exp1 exp2  -> eval exp1 + eval exp2
  ESub exp1 exp2  -> eval exp1 - eval exp2
  EMul exp1 exp2  -> eval exp1 * eval exp2
  EDiv exp1 exp2  -> eval exp1 `div` eval exp2
  EInt n  -> n
```

Turning the parser into an interpreter

Change the generated file `TestCalc.hs`: instead of showing the syntax tree, show the value from interpretation:

```
module Main where

import LexCalc
import ParCalc
import AbsCalc
import Interpreter
import ErrM

main = do
  interact calc
  putStrLn ""

calc s = let Ok e = pExp (myLexer s)
         in show (interpret e)
```

Using the calculator

Put your `Main` module in a file named `Calculator.hs`.

Compile it with GHC as follows:

```
ghc --make Calculator.hs
```

Run it on command-line input:

```
echo "1 + 2 * 3" | ./Calculator
7
```

A compiler in a nutshell

1. Write a grammar and convert it into parser and other modules.
2. Write some code that manipulates syntax trees in a desired way.
3. Let the main file output the results of syntax tree manipulation.

Now, let's do the same thing in Java.

Abstract syntax in Java

Java has no notation for algebraic datatypes. Use **classes** instead:

- For each category, an abstract base class.
- For each constructor of the category, a class extending the base class.

In the case of `Calc.cf`, we have

```
Calc/Absyn/EAdd.java
Calc/Absyn/EDiv.java
Calc/Absyn/EInt.java
Calc/Absyn/EMul.java
Calc/Absyn/ESub.java
Calc/Absyn/Exp.java
```

What class looks like

```
public abstract class Exp implements java.io.Serializable {
}
public class EAdd extends Exp {
  public final Exp exp_1, exp_2;
  public EAdd(Exp p1, Exp p2) { exp_1 = p1; exp_2 = p2; }
}
public class EAdd extends Exp {
  public final Exp exp_1, exp_2;
  public ESub(Exp p1, Exp p2) { exp_1 = p1; exp_2 = p2; }
}
public class EAdd extends Exp {
  public final Exp exp_1, exp_2;
  public EMul(Exp p1, Exp p2) { exp_1 = p1; exp_2 = p2; }
}
public class EAdd extends Exp {
  public final Exp exp_1, exp_2;
  public EDiv(Exp p1, Exp p2) { exp_1 = p1; exp_2 = p2; }
}
public class EInt extends Exp {
  public final Integer integer_;
  public EInt(Integer p1) { integer_ = p1; }
}
```

Tree manipulation in Java

Java doesn't have pattern matching.

But there are three ways out:

1. Add an interpreter method to each class.
2. Use a separate **visitor interface** to implement tree traversal in a general way.
3. Use the `instanceof` type comparison operator (generated by BNFC with `-java1.4` option, but not recommended by Java purists).

We choose the first method here.

The visitor method is the best choice in more advanced applications, later.

The interpreter in Java

Take the classes generated in `Calc/Absyn/` by BNFC, and add an `eval` method to each of them:

```
public abstract class Exp {
  public abstract Integer eval() ;
}
public class EAdd extends Exp {
  public final Exp exp_1, exp_2;
  public EAdd(Exp p1, Exp p2) { exp_1 = p1; exp_2 = p2; }
  public Integer eval() {return exp_1.eval() + exp_2.eval() ;}
}
public class ESub extends Exp {
  public final Exp exp_1, exp_2;
  public ESub(Exp p1, Exp p2) { exp_1 = p1; exp_2 = p2; }
  public Integer eval() {return exp_1.eval() - exp_2.eval() ;}
}
public class EMul extends Exp {
  public final Exp exp_1, exp_2;
  public EMul(Exp p1, Exp p2) { exp_1 = p1; exp_2 = p2; }
  public Integer eval() {return exp_1.eval() * exp_2.eval() ;}
}
public class EDiv extends Exp {
  public final Exp exp_1, exp_2;
```

```
  public EDiv(Exp p1, Exp p2) { exp_1 = p1; exp_2 = p2; }
  public Integer eval() {return exp_1.eval() / exp_2.eval() ;}
}
public class EInt extends Exp {
  public final Integer integer_;
  public EInt(Integer p1) { integer_ = p1; }
  public Integer eval() {return integer_ ;}
}
```

The top-level calculator class

Change the file `Calc/Test.java` into a calculator:

```
public class Calculator {
  public static void main(String args[]) throws Exception
  {
    Yylex l  = new Yylex(System.in) ;
    parser p = new parser(l) ;
    Calc.Absyn.Exp parse_tree = p.pExp() ;
    System.out.println(parse_tree.eval());
  }
}
```

(omitting error handling for simplicity.)

Using the Java calculator

Compile it:

```
javac Calc/Calculator.java
```

Run it:

```
echo "1 + 2 * 3" | java Calc/Calculator
7
```

Run it on file input:

```
java Calc/Calculator < ex1.calc
9102
```

The file `ex1.calc` contains the text

```
(123 + 47 - 6) * 222 / 4
```

List categories

Lists of various kinds are used everywhere in grammars.

Standard BNF: a pair of rules (”Nil” for the empty list, and ”Cons”) for adding an element.

Example: list of definitions (`Def`):

```
NilDef.  ListDef ::= ;
ConsDef. ListDef ::= Def ListDef ;
```

Terminators

A **terminator** is a token that appear after every item of a list.

For instance, definitions might have semicolons (;) as terminators:

```
NilDef.  ListDef ::= ;
ConsDef. ListDef ::= Def ";" ListDef ;
```

This common pattern is generated by the macro

```
terminator Def ";" ;
```

If no terminator is used, we can give an "empty terminator",

```
terminator Def "" ;
```

This generates the pair of rules on the previous slide.

Separators

A **separators** is a token that appears between list items, just not after the last one.

Example: the list of arguments of a function in C has a comma (`,`) as separator. We write

```
separator Exp "," ;
```

This shorthand expands to a set of rules for the category `ListExp`.

The rule for function calls can now be written

```
ECall. Exp ::= Ident "(" ListExp ")" ;
```

The list category notation

Instead of `ListExp`, BNFC programmers can use the bracket notation for lists, `[Exp]`.

Example: an alternative formulation of the `ECall` rule is

```
ECall. Exp ::= Ident "(" [Exp] ")" ;
```

The notation is borrowed from Haskell. But it translates to all target languages of BNFC:

- in Haskell, `[C]` is represented as `[C]`
- in Java, `[C]` is represented as `java.util.LinkedList<C>`

Nonempty lists

Lists that have at least one element

```
terminator nonempty Def "" ;
separator nonempty Ident "," ;
```

Their internal representations are still lists in the host language, but the parser never returns an empty list

Specifying the lexer

Simplest way: just use **predefined token types** as categories:

- `Integer`, **integer literals**: sequence of digits, e.g. `123445425425436`;
- `Double`, **float literals**: two sequences of digits with a decimal point in between, possibly with an exponent after, e.g. `7.098e-7`;
- `String`, **string literals**: any characters between double quotes, e.g. `"hello world"`, with a backslash (\) escaping a quote and a backslash;
- `Char`, **character literals**: any character between single quotes, e.g. `'x'` and `'7'`;
- `Ident`, **identifiers**: a letter (`A..Za..z`) followed by letters, digits, and characters `_` or `'`, e.g. `r2_out'`

User-defined tokan types

By **regular expressions**.

For instance, a type upper-case identifiers:

```
token UIdent (upper (letter | digit | '_')*) ;
```

Regular expressions available in BNFC

name	**notation**	**explanation**
symbol	'a'	the character `a`
sequence	*A B*	*A* followed by *B*
union	*A* | *B*	*A* or *B*
closure	*A*`*`	any number of *A*'s (possibly 0)
empty	`eps`	the empty string
character	`char`	any character (ASCII 0..255)
letter	`letter`	any letter (`A..Za..z`)
upper-case letter	`upper`	any upper-case letter (`A..Z`)
lower-case letter	`lower`	any lower-case letter (`a..z`)
digit	`digit`	any digit (`0..9`)
option	*A*?	optional *A*
difference	*A* - *B*	*A* which is not *B*

Position tokens

To remember the **position of a token**.

Useful in error messages at later compilation phases.

Just add the keyword `position` to a `token` definition:

```
position token CIdent (letter | (letter | digit | '_')*) ;
```

Position token types are represented with pairs of integers indicating the line and the column in the input:

```
newtype CIdent = CIdent (String, (Int,Int))
```

Comments

Comments are parts of source code that the compiler ignores.

BNFC permits the definition of two kinds of comments:

- single-line comments, which run from a start token till the end of the line;
- arbitrary-length comments, which run from a start token till a closing token.

Example: comments in C are defined as follows:

```
comment "//" ;
comment "/*" "*/" ;
```

Since comments are resolved by the lexer, they are processed by using a finite automaton. Therefore nested comments are not possible. A more thorough explanation of this will be given in next chapter.

Working out a grammar

We conclude this section by working out a grammar for a small C-like programming language. This language is the same as targeted by the Assignments 2 to 4 at the end of this book, called **CPP**.

We go through the language constructs top-down, i.e. from the largest to the smallest, and build the appropriate rules at each stage.

The reader is advised to copy all the rules of this section into a file and try this out in BNFC, with various programs as input. The grammar is also available on the book web page.

- *A program is a sequence of definitions.*

This suggests the following BNFC rules:

```
PDefs.   Program ::= [Def] ;

terminator Def "" ;
```

- *A program may contain comments, which are ignored by the parser. Comments can start with the token // and extend to the end of the line. They can also start with /* and extend to the next */.*

This means C-like comments, which are specified as follows:

```
comment "//" ;
comment "/*" "*/" ;
```

- *A function definition has a type, a name, an argument list, and a body. An argument list is a comma-separated list of argument declarations enclosed in parentheses* (*and*). *A function body is a list of statements enclosed in curly brackets* { *and* } . *For example:*

```
int foo(double x, int y)
{
  return y + 9 ;
}
```

We decide to specify all parts of a function definition in one rule, in addition to which we specify the form of argument and statement lists:

```
DFun.       Def    ::= Type Id "(" [Arg] ")" "{" [Stm] "}" ;
separator   Arg "," ;
terminator  Stm "" ;
```

- *An argument declaration has a type and an identifier.*

```
ADecl.   Arg    ::= Type Id ;
```

- *Any expression followed by a semicolon* `;` *can be used as a statement.*

```
SExp.    Stm    ::= Exp ";" ;
```

- *Any declaration followed by a semicolon* `;` *can be used as a statement. Declarations have one of the following formats:*
  - *a type and one variable (as in function parameter lists),*

    ```
    int i ;
    ```
  - *a type and many variables,*

    ```
    int i, j ;
    ```
  - *a type and one initialized variable,*

    ```
    int i = 6 ;
    ```

Now, we could reuse the function argument declarations `Arg` as one kind of statements. But we choose the simpler solution of restating the rule for one-variable declarations.

```
SDecl.   Stm    ::= Type Id ";" ;
SDecls.  Stm    ::= Type Id "," [Id] ";" ;
SInit.   Stm    ::= Type Id "=" Exp ";" ;
```

- *Statements also include*
  - *Statements returning an expression,*

    ```
    return i + 9 ;
    ```

  - *While loops, with an expression in parentheses followed by a statement,*

    ```
    while (i < 10) ++i ;
    ```

  - *Conditionals:* `if` *with an expression in parentheses followed by a statement,* `else`*, and another statement,*

    ```
    if (x > 0) return x ; else return y ;
    ```

  - *Blocks: any list of statements (including the empty list) between curly brackets. For instance,*

    ```
    {
      int i = 2 ;
      {
      }
      i++ ;
    }
    ```

The statement specifications give rise to the following BNF rules:

```
SReturn. Stm    ::= "return" Exp ";" ;
SWhile.  Stm    ::= "while" "(" Exp ")" Stm ;
SBlock.  Stm    ::= "{" [Stm] "}" ;
SIfElse. Stm    ::= "if" "(" Exp ")" Stm "else" Stm ;
```

- *Expressions are specified with the following table that gives their precedence levels. Infix operators are assumed to be left-associative, except assignments, which are right-associative. The arguments in a function call can be expressions of any level. Otherwise, the subexpressions are assumed to be one precedence level above the main expression.*

level	**expression forms**	**explanation**
15	literal	literal (integer, float, string, boolean)
15	identifier	variable
15	`f(e,...,e)`	function call
14	`v++`, `v--`	post-increment, post-decrement
13	`++v`, `--v`	pre-increment, pre-decrement
13	`-e`	numeric negation
12	`e*e`, `e/e`	multiplication, division
11	`e+e`, `e-e`	addition, subtraction
9	`e<e`, `e>e`, `e>=e`, `e<=e`	comparison
8	`e==e`, `e!=e`	(in)equality
4	`e&&e`	conjunction
3	`e||e`	disjunction
2	`v=e`	assignment

The table is straightforward to translate to a set of BNFC rules. On the level of literals, integers and floats ("doubles") are provided by BNFC, whereas the boolean literals `true` and `false` are defined by special rules.

```
EInt.    Exp15  ::= Integer ;
EDouble. Exp15  ::= Double ;
EString. Exp15  ::= String ;
ETrue.   Exp15  ::= "true" ;
EFalse.  Exp15  ::= "false" ;
EId.     Exp15  ::= Id ;

ECall.   Exp15  ::= Id "(" [Exp] ")" ;

EPIncr.  Exp14  ::= Exp15 "++" ;
EPDecr.  Exp14  ::= Exp15 "--" ;

EIncr.   Exp13  ::= "++" Exp14 ;
```

```
EDecr.   Exp13  ::= "--" Exp14 ;
ENeg.    Exp13  ::= "-" Exp14 ;

EMul.    Exp12  ::= Exp12 "*"  Exp13 ;
EDiv.    Exp12  ::= Exp12 "/"  Exp13 ;
EAdd.    Exp11  ::= Exp11 "+"  Exp12 ;
ESub.    Exp11  ::= Exp11 "-"  Exp12 ;
ELt.     Exp9   ::= Exp9  "<"  Exp10 ;
EGt.     Exp9   ::= Exp9  ">"  Exp10 ;
ELEq.    Exp9   ::= Exp9  "<=" Exp10 ;
EGEq.    Exp9   ::= Exp9  ">=" Exp10 ;
EEq.     Exp8   ::= Exp8  "==" Exp9 ;
ENEq.    Exp8   ::= Exp8  "!=" Exp9 ;
EAnd.    Exp4   ::= Exp4  "&&" Exp5 ;
EOr.     Exp3   ::= Exp3  "||" Exp4 ;
EAss.    Exp2   ::= Exp3 "=" Exp2 ;
```

Finally, we need a `coercions` rule to specify the highest precedence level, and a rule to form function argument lists.

```
coercions Exp 15 ;
separator Exp "," ;
```

- *The available types are* `bool`, `double`, `int`, `string`, and `void`.

```
Tbool.   Type ::= "bool" ;
Tdouble. Type ::= "double" ;
Tint.    Type ::= "int" ;
Tstring. Type ::= "string" ;
Tvoid.   Type ::= "void" ;
```

- *An identifier is a letter followed by a list of letters, digits, and underscores.*

Here we cannot use the built-in `Ident` type of BNFC, because apostrophes (') are not permitted! But we can define our identifiers easily by a regular expression:

```
token Id (letter (letter | digit | '_')*) ;
```

Alternatively, we could write

```
position token Id (letter (letter | digit | '_')*) ;
```

to remember the source code positions of identifiers.