A tutorial guide to `rdp` for new users

Adrian Johnstone Elizabeth Scott

Technical Report

CSD-TR-97-24

December 20, 1997

Royal Holloway
University of London

Department of Computer Science
Egham, Surrey TW20 0EX, England

Abstract

rdp is a system for implementing language processors. It takes as input an EBNF-like specification of a language together with a specification, written in C, of behaviour which should result when fragments of the language are recognised. rdp produces as output a program written in C, which parses fragments of the specified language and carries out the specified corresponding actions. Thus rdp can produce, for example, compilers (the actions specify the corresponding target code), interpreters (the actions evaluate the input fragments) and pretty printers (the actions reformat the input fragments).

The version of rdp described here is version 1.50 dated 16 August 1997.

Please send bug reports and copies of modifications to the authors at the address on the title page or electronically to `A.Johnstone@rhbnc.ac.uk`.

Contents

Chapter 1

An overview of translation

Computer programs are often written in a so-called 'high level' language such as C or FORTRAN. Most human programmers find high level languages easier to use than the 'low level' machine oriented languages. However, in order for a machine to execute a program it must be translated from the high level language in which it is written to the native language of that machine. A *compiler* is a program which takes as input a program written in one language and produces as output an equivalent program written in another language.

Computer languages are very simple compared to the languages employed in everyday human communication. This makes the task of writing a compiler less intimidating— at present computer programs that translate from one *human* language to another are rather unsatisfactory because the subtle rules that underpin human languages are not completely understood, and so mis-translations are common.

Although computer languages are designed to be simple to understand and translate, real computer languages still present significant problems. Sometimes, especially with very old languages such as FORTRAN and COBOL, the difficulties in translation arise from the imperfect understanding that the early language designers had of the translation process. More modern languages, such as Pascal and Ada are to a large extent designed to be easy to translate. The discovery that it was possible to design a language which could be translated in linear time (that is the translation time is proportional to the length of the text to be translated) and yet still appear readable to humans was an important result of early work on the theory of programming language syntax.

Other problems are not so easily circumvented. It turns out, for instance, that the ability to directly modify machine addresses provided by the C language's pointer arithmetic operations makes it very difficult for a 'smart' translator to produce efficient translations for conventional computers, and the same facilities create even more serious problems when attempting to produce code that will run on a parallel computer. This kind of difficulty arises from a fundamental design decision taken at the time the language is first specified, and cannot easily be undone.

Subdividing the translation problem

Computer language translation is traditionally viewed as a process with two main parts: the *front end* conversion of a high level language text written in a language such as C, Pascal or Ada into an *intermediate form*, and the *back end* conversion of the intermediate form into the native language of a computer. This approach is useful because it turns out that the challenges encountered in the design of a front end differ fundamentally from the problems posed by back end code generation and separating out the problems makes it easier to think about the overall task.

The language to be translated forms the input to the front end and is called the *source* language. The output of the back end is called the *target* language. In the special (but very common) case of a translator that outputs machine code for a particular computer, the target language is usually called the *object code*.

Interpreters, compilers and in-between

Sometimes the subdivision of the translation problem into front and back ends is explicit in the translator program, but not always. An *interpreter* is a special kind of language translator that executes actions as it translates. Most operating system command shells are of this form: each command is executed as it is encountered. In such a system there is no readily discernible back end or intermediate form although it can still be useful to think of the program as performing front and back end tasks. The macro languages found in most word processors, along with simple programming languages such as BASIC are most often implemented as interpreters.

The intermediate form must provide enough generality to cope with the various source and target languages. Fortunately, front end processors for different languages sometimes display striking similarities. For instance, at a very crude level the variable declaration constructs in C and Pascal are quite similar. Their use of IF-THEN-ELSE selection is almost identical. It is perfectly possible to design an intermediate form that can cope with both C- and Pascal-like structures.

Using this organisation, a compiler for a given language can be moved to a new computer architecture by writing a new back end to take account of the differing instruction sets. More rarely, a new programming language syntax can be quickly implemented on a given architecture by building a new front end and using an existing back end. This saving in engineering effort can be very important in commercial compiler systems, even though it may require an intermediate form that is more complex than that required for a single source/target language pair.

Automated front end production

Many of the theoretical issues surrounding front end translation were solved during the 1960's and 1970's, and it is possible to reduce most of the implementation effort for a new front end to a clerical exercise that may itself be turned into a computer program. *Compiler-compilers* are programs that take the description of a programming language, and output the source code of a program that will recognise, and possibly act upon, phrases written in that language. The availability of such tools has fed back into programming language design. It is very hard to use such tools to generate translators for languages such as FORTRAN, but more recent languages are usually designed in such a way as to facilitate the use of compiler-compilers. The description of the programming language to be input to a compiler-compiler is usually given in a variant of the *generative grammar* formalism which was introduced in the 1950's by Chomsky. The formalism was first applied to the specification of programming languages by John Backus and Peter Naur and in recognition of this the notation used is often called Backus-Naur Form (or BNF). In this guide we shall give an introduction to BNF and the particular version, IBNF, accepted by our compiler-compiler. A full discussion of BNF can be found in standard texts such as [ASU86] or [AU72], and further discussion of our particular IBNF can be found in the associated user manual [JS97b].

Back end design

Code generation, the primary task of the back end, is much less well understood than front end translation. The basic task is the selection of machine code sequences that correctly represent the meaning of the source language phrases. In general we will want to generate code which executes either as quickly as possible, or requires as little space as possible, or both (these two aims may or may not conflict). So far no single unifying theoretical model has appeared, and many compilers use a 'bag of tricks' in the back end that is hard to systematise. As a result, books on compiler design often focus mainly on the front end where the problems are more tractable and the tools more useful.

`rdp` is a program which takes an IBNF specification of a language and, provided the specification has certain properties, generates a compiler which translates from the specified language in to C. `rdp` can be used to generate both compilers, interpreters and simple parsers for languages. In this tutorial document we give a low level introduction to parser generation using `rdp`. The associated case study manual [JS97a] discusses larger examples in which `rdp` is used to generate a compiler for a small language. Full details of the facilities available in `rdp` can be found in the associated users' manual [JS97b].

Chapter 2

Basic parsing issues

`rdp`-generated parsers use a *recursive descent* parsing technique with *one symbol of lookahead*. In order for such parsers to work correctly the specification (grammar) which is input to `rdp` must have certain properties. A full description of these properties will be given in Chapter 4. In this chapter we review enough of the theory of grammars, language specification, and parsing to understand the use of `rdp` at a basic level. We also describe, in a step-by-step fashion, the way in which `rdp` can be used to generate a parser for a specified language, using a language whose elements are arithmetic expressions as an example. At the end of the chapter we describe the built in `rdp` scanner, which is copied into all `rdp`-generated parsers.

We assume that you have already got and unpacked the `rdp` software pack, and that you have built the standard modules (for example by typing `make`). Your main `rdp` directory should contain subdirectories `rdp_doc` and `rdp_supp`. For instructions on how to get and install the `rdp` software pack see Appendix A at the end of this manual.

Throughout this guide we shall illustrate our discussion with example grammars. The grammars which are given titles in the text are included in the `rdp` distribution pack in the subdirectory `tut_exs` so you can use `rdp` to generate the corresponding parsers.

2.1 Specifying a language

It is standard practice to use formal grammars to specify languages. For example,

```
S ::= S '+' S | S '*' S | E .
E ::= 'a' | 'b' .
```

is a set of grammar rules which generates a language of sums and products, for example, `a+b*a+a` or `a`.

A grammar consists of a set $\mathbf{N}$ of *non-terminals*, a set $\mathbf{T}$ of *terminals*, and a set $\mathcal{P}$ of *grammar rules* of the form

$$\mathtt{A} \ \mathtt{::=} \ \alpha_1 \ | \ \alpha_2 \ | \ \dots \ | \ \alpha_n$$

where A is an element of $\mathbf{N}$ and each α_i is a string of elements from $\mathbf{N}$ and $\mathbf{T}$. One of the non-terminals, S say, is singled out and called the *start symbol.*

In the above grammar, the non-terminals are `S`, `E`, the terminals are `+`,`*`,`a`,`b`, and the start symbol is `S`.

How does a grammar specify a language?

We *derive* one string from another by replacing a non-terminal with a string from the right hand side of its grammar rule. So if we have a rule

$$A ::= \ldots \mid \gamma \mid \ldots .$$

we can replace A by γ. We use the symbol $\Rightarrow$ for a derivation, and we write $\alpha A\beta \Rightarrow \alpha\gamma\beta$.

If μ and τ are strings, we say that τ *can be derived from* μ, and we write $\mu \stackrel{*}{\Rightarrow} \tau$, if there is a sequence

$$\mu \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \tau.$$

For the example above we have

$$\begin{array}{lll}
\mathtt{S} & \Rightarrow & \mathtt{S + S} \\
 & \Rightarrow & \mathtt{E + S} \\
 & \Rightarrow & \mathtt{a + S} \\
 & \Rightarrow & \mathtt{a + S + S} \\
 & \Rightarrow & \mathtt{a + S * S + S} \\
 & \Rightarrow & \mathtt{a + b * S + S} \\
 & \Rightarrow & \mathtt{a + b * E + S} \\
 & \Rightarrow & \mathtt{a + b * a + S} \\
 & \Rightarrow & \mathtt{a + b * a + E} \\
 & \Rightarrow & \mathtt{a + b * a + b}
\end{array}$$

and so $\mathtt{S} \stackrel{*}{\Rightarrow} \mathtt{a} + \mathtt{b} * \mathtt{a} + \mathtt{b}$.

The *language specified by a grammar* is the set of strings of terminals which can be derived from its start symbol. We say that $u \in \mathbf{T}^*$ is a *sentence* if $S \stackrel{*}{\Rightarrow} u$. (Here $\mathbf{T}^*$ denotes the set of strings of elements of $\mathbf{T}$ and includes the empty string ϵ, so if $\mathbf{T}=\{a, b, +\}$ then
$\mathbf{T}^* = \{\epsilon, a, b, +, aa, ab, a+, ba, bb, b+, +a, +b, ++, aaa, aab, \ldots\}$.)

The language generated by the grammar above is the set of all sums and products of a's and b's.

2.2 Formal grammars and `rdp`

Only the grammar rules are actually input to **`rdp`**. The following conventions are used by **`rdp`** to deduce the remaining aspects of the grammar: the left hand side of the first grammar rule is the start symbol, the terminal symbols are enclosed in single quotes, and each grammar rule is terminated by a full stop. The above example is an example input for `rdp` (although it should be noted that `rdp` will not automatically generate a parser for this example because it is left recursive — see below).

Given a 'suitable' input grammar (we shall explain what suitable means in this context in Chapter 4), `rdp` generates a *parser* which takes as input a string u of terminals and either reports success, if u is a sentence in the language of the grammar, or issues an error message. (If the input grammar is not suitable `rdp` issues a diagnostic identifying the aspect(s) of the grammar with which it cannot cope.)

In the rest of this chapter we shall describe how to get `rdp` to generate a parser from a simple grammar. It is not possible to use `rdp` to generate a correct parser for the expression grammar given above, because this grammar is not 'suitable' in the sense that we have just discussed. So we shall use the following grammar (which specifies the same language as the original) as an example.

```
(** expr1.bnf **)

S ::= E Y.
Y ::= ['+' S].
E ::= T X.
X ::= ['*' E].
T ::= 'a' | 'b'.
```

We need to note here that `rdp` does not directly accept 'epsilon' rules (i.e. rules of the form $A ::= \epsilon$, where ϵ denotes the empty string). The notation `[`α`]` is used to represent $\alpha|\epsilon$. So, for example,

`Y ::= ['+' S].` corresponds to `Y ::= '+' S |` ϵ.

2.3 Building and running a parser

In this section we shall use the grammar `expr1.bnf` to describe the basic procedure for getting `rdp` to generate a parser.

To build and run parsers using `rdp` we must first compile the support library modules. The exact command to do this depends on the C compiler that you are using. If, for instance your compiler was called `CC` then the following commands would compile the modules:

```
CC -c -Irdp_supp rdp_supp/arg.c
CC -c -Irdp_supp rdp_supp/graph.c
CC -c -Irdp_supp rdp_supp/memalloc.c
CC -c -Irdp_supp rdp_supp/scan.c
CC -c -Irdp_supp rdp_supp/scanner.c
CC -c -Irdp_supp rdp_supp/set.c
CC -c -Irdp_supp rdp_supp/symbol.c
CC -c -Irdp_supp rdp_supp/textio.c
```

The `-c` flag here tells the C compiler to just run as far as producing an object file and to not attempt to link the module into an executable program. The `-Irdp_supp` flag tells the compiler to look for include files in the `rdp_supp` subdirectory.

In general you will have to replace the `CC` with the name of your compiler, possibly along with some special flags. The following table gives some examples: please refer to your compiler documentation if the combination of operating system and compiler you use is not listed here.

If you are using...	then replace `CC` with ...
GNU C on Unix	`gcc`
GNU C++ on Unix	`g++`
Borland C V5.0 on MS-DOS	`bcc`
Borland C++ V5.0 on MS-DOS	`bcc -P`
Borland C V5.0 on Windows-95	`bcc32`
Borland C++ V5.0 on Windows-95	`bcc32 -P`

Once the support library has been built, we can edit a `.bnf` file, process it using `rdp` and then compile the resulting parser before linking with the support library modules and testing the parser against a string file.

To build and run a parser for a language with gcc running under Unix and using the Emacs editor we might use the following commands,

```
emacs expr1.bnf
rdp -oexpr1 expr1
gcc -Irdp_supp -c expr1.c
gcc expr1.o arg.o graph.o memalloc.o scan.o scanner.o set.o
    symbol.o textio.o
emacs expr1.str
expr -v expr1.str
```

To build and run a parser for a language with Borland C++ 5.0 running under Windows-95 and using the standard DOS editor we use the following commands,

```
edit expr1.bnf
rdp -oexpr1 expr1
bcc32 -P -Irdp_supp -c expr1.c
bcc32 expr1.obj arg.obj graph.obj memalloc.obj scan.obj
      scanner.obj set.obj symbol.obj textio.obj
edit expr1.str
expr -v expr1.str
```

We now look at these commands line-by-line. We can type the above grammar into a file using our editor (either `emacs` or `edit` here). If we call the file `expr1.bnf` say, then we can input it to `rdp` as follows.

```
rdp -oexpr1 expr1
```

This causes **rdp** to produce a parser, called `expr1.c` and written in C, for the language specified by the grammar. The flag `-o` directs **rdp** to call the file it produces `expr1.c`; the default name for the output, which will be used if you leave out the `-o` flag, is `rdparser.c`. Note, all the examples discussed in this manual are included in the **rdp** distribution in the subdirectory `examples\rdp_tut`. If you wish to use these directly you will need to give the path

```
rdp -oexpr1 examples\rdp_tut\expr1
```

We then compile this file using a C compiler to produce an object file `expr1.obj`. The following command assumes that the compiler is Borland C++:

```
bcc32 -P -Irdp_supp -c expr1.c
```

The flag `-I` instructs the compiler to look in the subdirectory `rdp_supp` for the additional header files that it needs and the `-c` flag tells the compiler to only produce an object file and not to invoke the linker at this stage.

The final parser needs to use the various support routines which are provided with the **rdp** package. Some of these support modules are discussed later in this manual, and all the support modules are fully documented in the accompanying support manual [JS97c]. Here we shall just generate an executable parser by linking in the appropriate support modules, without explaining their functions.

```
bcc32 expr1.obj arg.obj graph.obj memalloc.obj scan.obj
scanner.obj set.obj symbol.obj textio.obj
```

This produces an executable version of the parser, called `expr1.exe` on DOS and Windows systems or just plain `expr1` on Unix systems.

We can then create a test file, `expr1.str` say, that contains a string for the generated parser to check, and run it through the parser using the command

```
expr1 -v expr1.str
```

The `-v` flag runs the generated parser in verbose mode so that it gives information about the execution. For example, if `expr1.str` is

```
a + b * a + b
```

the following should be produced as a result of the above command:

```
rdparser
Generated on Apr 14 1997 9:56:55 and compiled on Apr 14 1997 at 9:32:21

******: 0.008 CPU seconds used
```

If we give the generated parser a string which is not in the language it issues a suitable error message. For example, on input `a b` we get

```
rdparser
Generated on Apr 14 1997 9:56:55 and compiled on Apr 14 1997 at 9:32:21

     1: Error 1 (expr.str) Scanned 'b' whilst expecting one of EOF, '*', '+'
     1: a b
     1: --1
******: Fatal - error detected in source file
```

and on input **a + + b** we get

```
rdparser
Generated on Apr 14 1997 9:56:55 and compiled on Apr 14 1997 at 9:32:21

     1: Error 1 (expr.str) Scanned '+' whilst expecting one of 'a', 'b'
     1: a + + b
     1: ----1
******: Fatal - error detected in source file
```

In both cases the generated parser issues an error message which prints out the section of the input string which has caused the trouble, indicates where in the string the parse has failed, and issues a list of symbols that would have been legitimate at the point of the error.

2.4 Makefiles

The **rdp** distribution pack contains a makefile which you can use to build and run the parsers. It is called **makefile** and is in the main **rdp** directory.

The makefile contains options for running **rdp** under Unix, DOS, SunOS and Windows-95, using gcc, acc 2.0, Borland C 3.1, Borland C++ 5.0, and Microsoft 'C' 7.0. All that is necessary to use a particular configuration is to remove the commenting '#' from the appropriate section. For example, to run gcc under Unix uncomment the commands

```
# Configuration for gcc on Unix. Also works for g++ if you set CC = g++
 CC       = gcc
 OBJ      = .o
 EXE      =
 DIFF     = diff -s
 RM       = rm
 CP       = cp
 SUPP_DIR = ./rdp_supp/
 CFLAGS = -I$(SUPP_DIR) -Wmissing-prototypes -Wstrict-prototypes
                           -fno-common -Wall -ansi -pedantic -g
 LINK     = $(CC) -o ./
 MATHS    = -lm
 HERE     = ./
 OBJ_ONLY = -c
# End of gcc on Unix configuration
```

There is a section in the makefile which allows it to be used to build a parser for any suitable input grammar. The grammar should be typed into a file with a **.bnf** extension, **myfile.bnf** say. This file is then used by setting **GRAMMAR=myfile** as part of the command line instructions to **make** (see below). The makefile runs **rdp** on the input file, compiles the corresponding C file, links it with the appropriate support files, and finally runs the executable parser on a file **myfile.str** containing a test string. Typing

```
make GRAMMAR=examples\rdp_tut\expr1 parser
```

under DOS generates the following:

```
MAKE Version 4.0  Copyright (c) 1987, 1996 Borland International
       rdp examples\rdp_tut\expr1
       bcc32 -Irdp_supp\ -A -c -P -w -c rdparser.c
Borland C++ 5.0 for Win32 Copyright (c) 1993,1996 Borland International
rdparser.c:
       bcc32 -erdparser.exe rdparser.obj arg.obj graph.obj memalloc.obj
                   scan.obj scanner.obj set.obj symbol.obj textio.obj
Borland C++ 5.0 for Win32 Copyright (c) 1993,1996 Borland International
Turbo Link Version 1.6.72.0 Copyright (c) 1993,1996 Borland International
       rdparser -v -Vparser.vcg -l examples\rdp_tut\expr1.str
rdparser
Generated on Dec 28 1997 9:08:55 and compiled on Dec 27 1997 at 8:41:23
******:
     1: a + b * a + b
******: 0 errors and 0 warnings
******: 0.038 CPU seconds used
```

rdp has built a parser for **expr1.bnf** and run it on the file **expr1.str**, which in this case happened to contain the string **a + b * a + b**.

2.5 rdp parsers and recursive descent parsing

rdp generates parsers which use a *recursive descent* technique. The goal of a parser is to construct a derivation of a given input string. In order to use **rdp** it is necessary to have an understanding of the recursive descent technique which its parsers use. In this section we outline the basic ideas of top-down one-symbol-lookahead parsing in order to motivate our choice of examples. Recursive descent is considered in more detail in Chapter 4.

2.5.1 Left-most derivations

rdp-generated parsers use a *top down* approach; that is, they start with the start symbol and attempt to construct a derivation step-by-step from the left. The parsers also use a *left-most depth-first* approach; that is, at each step in the constructed derivation the left-most non-terminal in the string is expanded.

Consider the grammar **expr1.bnf**

```
S ::= E Y.
Y ::= ['+' S] .
E ::= T X.
X ::= ['*' E] .
T ::= 'a' | 'b'.
```

As it uses a top down depth first approach, an **rdp**-generated parser for the above grammar would construct the following derivation of **a + b * a**:

$$\mathtt{S} \Rightarrow \mathtt{E\ Y} \Rightarrow \mathtt{T\ X\ Y} \Rightarrow \mathtt{a\ X\ Y} \Rightarrow \mathtt{a\ Y} \Rightarrow \mathtt{a + S} \Rightarrow \mathtt{a + E\ Y}$$

$\Rightarrow$ a + T X Y $\Rightarrow$ a + b X Y $\Rightarrow$ a + b * E Y $\Rightarrow$ a + b * T X Y
$\Rightarrow$ a + b * a X Y $\Rightarrow$ a + b * a Y $\Rightarrow$ a + b * a

but would not construct the following (legitimate) derivation of `a`:

S $\Rightarrow$ EY $\Rightarrow$ E $\Rightarrow$ TX $\Rightarrow$ T $\Rightarrow$ a

2.5.2 Selecting an alternate

When there is more than one alternate in a grammar rule recursive descent parsers need an algorithm for deciding which of the alternates to choose. For example, in the first derivation above at the point

S $\Rightarrow$ E Y $\Rightarrow$ T X Y

it was necessary to decide whether to replace `T` by `a` or `b` at the next step. This decision was made by looking at the *current input symbol*. This will be discussed formally below but it may help the reader at this point to consider the following informal discussion. As the parse begins the first symbol of the string to be parsed is read from the input buffer. This is the current input symbol. If the parse is to succeed, eventually this symbol must appear at the beginning (left hand end) of a string generated during the derivation. When this happens the current input symbol has been *matched*, and the next symbol is read from the input buffer, becoming the current input symbol. Eventually this must be matched to the second symbol in a string generated during the derivation, and so on. This reading and matching process carries on until the last symbol from the input buffer is matched to the last symbol of a string generated by the derivation. At this point the parse has succeeded. If this point cannot be reached then the parse has failed.

The value of the current input symbol is the only information the parser has for use in selecting the alternate to be inserted at the next derivation step. So if parser success is to be guaranteed, this information must be sufficient to decide between the alternates. For this reason, `rdp` will not generate a parser from a grammar in which two or more alternates from the same grammar rule have the same first symbol. For example, we could not use an `rdp` parser generated from the grammar

```
S ::= E Y.
Y ::= ['+' S] .
E ::= T X.
X ::= ['*' E] .
T ::= 'a' 'a' | 'a' 'b'.
```

To see why this grammar is unsatisfactory consider the string `ab + aa`. At the point

S $\Rightarrow$ E Y $\Rightarrow$ T X Y

in an attempted parse of this string it is not possible to decide, just by looking at the current input symbol `a`, which of the alternates `aa` and `ab` to use to replace `T`.

In the original grammar,

```
S ::= S '+' S | S '*' S | E .
E ::= 'a' | 'b' .
```

two of the alternates in the rule for `S` have `S` as their first symbol. This is one reason why this grammar cannot be used with `rdp` and a modified version `expr1.bnf` was used as an example in the previous sections.

A complete description of grammars which admit `rdp`-generated parsers is given in Chapter 4.

2.6 The `rdp` scanner

So far we have not discussed how `rdp`-generated parsers match characters in an input file to terminals in the grammar. To use `rdp` effectively it is necessary to know a little about the initial phase of compilation usually called *lexical analysis* or *scanning*.

2.6.1 `rdp` scanner tokens

A parser usually considers sentences in a language at token, or 'word', level. It is presented with streams of tokens which have to be structured into sentences. In reality, a sentence is presented as a stream of characters, or 'letters', and these characters must first be grouped together into words. This is usually the job of the scanner in a compiler.

Tokens are not quite the same thing as words. A token often corresponds to a set of words. For example, we describe the set of integers using the token `INTEGER` and the set of C style identifiers using the token `ID`. Sometimes tokens do correspond exactly to words. For example, the token `'if'` corresponds just to the string which forms the keyword `if`.

So a token is the name of a set of strings of characters; this set is called the *pattern* of the token. We say that a string of characters *matches* a token if it belongs to the pattern of that token. A string which matches a token is called a *lexeme* of the token. For example, a token `op` may correspond to the less-than, greater-than, less-or-equals, and greater-or-equals strings. In this case, the pattern of `op` is the set

$$\{\ <,\ >,\ <=,\ >=\ \}$$

the string `<=` matches `op`, the string `<<` doesn't match `op`, and `<=` is a lexeme of `op`.

`rdp` has a scanner which it uses to process its input files. This scanner is automatically built into any parser that `rdp` generates, so `rdp`-generated parsers come with a hard wired scanner giving the user access to certain standard patterns.

Simple tokens

Any string of characters between single quotes is treated by the scanner as a token which matches exactly that string. So `'a'` matches the sequence `a` and `'while'` matches the sequence `while`, and the singleton set {`a`} is the pattern of `'a'` and {`while`} is the pattern of `'while'`.

The tokens INTEGER and REAL

The token `INTEGER` matches C style integers such as 145 and 0xFE, and `REAL` matches any C style real number, such as 1.45 and 1.45e3. We can use the token `INTEGER` to modify the expression grammar given in Section 2.3 to generate all strings of integer expressions, sums and products of integers.

```
(** expr2.bnf **)

S ::= E Y.
Y ::= ['+' S].
E ::= T X.
X ::= ['*' E].
T ::= INTEGER.
```

The STRING() tokens

Most languages allow string literals — alphanumeric strings enclosed between, for example, single or double quotes. For example, the C command

```
printf("string") ;
```

causes the characters `string` to be printed.

`rdp` allows you to use a paramaterisable token `STRING('`*delimiter character*`')` which matches all sequences of letters and underscores enclosed by the *delimiter character*. The symbol which is to delimit the strings must itself be quoted. For example,

```
rule ::= STRING('"') .
```

matches strings between double quotes. So

```
"this is a string"    "string1"  "another_string"
```

are all lexemes of the token `STRING('"')`.

To distinguish a single quote from the token quotes we use the `\` character. So

```
rule ::= STRING('\'') .
```

matches strings between single quotes, e.g.

```
'this is a string'    'string1'  'another_string'
```

What if we want to include the delimiting character in the actual string? Two consecutive delimiters represent the delimiter symbol itself. So, for example,

```
'this string''s delimiter is '''
```

matches the token `STRING('\'')` and the Pascal statement

```
writeln('this string''s delimiter is ''') ;
```

would cause

```
this string's delimiter is '
```

to be printed on the screen.

It is possible that a language designer would like to use an escape symbol to access non-printing characters. This is fully documented in the user manual [JS97b], here we just show its use for printing a delimiter symbol. Strings of the form

```
"this string is delimited by \" and has a special escape symbol"
```

match the **rdp** token `STRING_ESC('"' '\\')`.

We can now add strings and a print facility to the expression grammar which allow the user to print messages and the results of calculations.

```
(** expr3.bnf **)

S ::= S1 | 'print' '(' [String][S1] ')' .
S1 ::= E Y .
Y ::= ['+' S1].
E ::= T X.
X ::= ['*' E].
T ::= INTEGER.
String ::= STRING_ESC('"' '\\') .
```

A parser for this grammar accepts the input

```
print("the result is " 10*8+5)
```

2.6.2 Defining a language which permits comments

Most programming languages allow the programmer to include comments in the code. These comments are designed to help a human reader of the code to understand it but are not part of the instructions to the computer. Such comments should be ignored by the translator, and hence need to be filtered out at some point.

In the traditional model of compilation the scanner removes any comments from the input before it is passed to the parser. The **rdp** scanner can recognise and remove comments and this facility is also available in **rdp**-generated parsers. The user must specify the form that comments in their language will take. This is done by including the appropriate `COMMENT` primitive in the input grammar.

For example, we can add C style comments to the language of integer expressions described in Section 2.6.1. The parser generated for the grammar

```
(** expr4.bnf **)

S ::= E Y .
Y ::= ['+' S].
E ::= T X.
X ::= ['*' E].
T ::= INTEGER.
comment ::= COMMENT('/*' '*/') .
```

accepts the following input:

```
/* Test input for expression grammar */
/*this string should be accepted*/
3 + 5 * 8/* multiplication will be done first */ + 10
/* this string should fail */
10 7
```

The above input will generate a parser error in the expression parser, and to see if the first part is correct it would be useful to be able to comment out the second part.

In C comments may not nest, but it is possible to define languages with nesting comments using grammars which will be accepted by **rdp**. For example, an **rdp**-generated parser for the grammar

```
(** expr5.bnf **)

S ::= E Y .
Y ::= ['+' S].
E ::= T X.
X ::= ['*' E].
T ::= INTEGER.

comment ::= COMMENT_NEST('(*' '*)') .
```

accepts the following input:

```
(* Test input for expression grammar *)
(*this string should be accepted*)
3 + 5 * 8(* multiplication will be done first *) + 10
(* (* this string should fail *)
10 7 *)
```

Chapter 3

Extended BNF

`rdp` grammars can have a more flexible type of grammar rule than standard BNF allows. One of the major limitations of standard BNF is that it is only possible to write out finitely many alternates. It is common practice in many areas of computer science to use regular expressions as a concise notation for sets of strings. This allows certain infinite sets to be specified, and saves tedious writing out of large finite sets.

For example, the grammar

```
S ::= 'a' { '+' 'a' } .
```

defines the language of sums of a's,

$$a, \quad a+a, \quad a+a+a, \quad a+a+a+a, \ldots$$

and

```
S ::= ('a' | 'b' | 'g')( 'c' | 'd' | 'h') .
```

defines the language

$$ac,\ ad,\ ah,\ bc,\ bd,\ bh,\ gc,\ gd,\ gh$$

In the first case we have specified that a string in the language can contain 0 or arbitrarily many + symbols. To give an equivalent specification in standard BNF we would need to add extra grammar rules, for example

```
S ::= 'a' X .
X ::= ['+' 'a' X] .
```

In the second case, to give an equivalent specification in BNF we would either need to add additional rules or extra alternates

```
S ::= 'a''c' | 'a''d' | 'a''h' | 'b''c' | 'b''d' | 'b''h'
    | 'g''c' | 'g''d' | 'g''h' .
```

(Note: the first grammar can actually be written more concisely using `rdp`'s iterator expression, see below.)

It is usual to allow the set of alternates on the right hand side of a grammar rule to specified using a regular expression and in this case we describe the notation as being *extended BNF* (EBNF). The `rdp` grammar rules can contain the common forms of regular expression, we have already seen the use of `[]` to denote one-or-zero copies of the enclosed string, and zero-or-many copies can be denoted using {}. `rdp` also supports a more general form of regular expression which allows upper and lower limits on the number of repeats of strings. In this chapter we shall describe the full extended BNF that `rdp` grammars can use. We begin with the standard constructs, and then describe `rdp`'s generalised expressions.

3.1 Standard EBNF

Strings and symbols

Any singly quoted string is a regular expression, it represents the set which just contains itself. Thus we can write grammar rules of the form

```
rule ::= 'fred' .
```

The singly quoted strings are the tokens of the grammar.

Any string of alphanumeric characters is a regular expression. Thus we can write grammar rules of the form

```
rule ::= name1 .
```

These are the non-terminals of the grammar.

Concatenation

If r and s are regular expressions then so is rs, and the elements of the set rs are strings obtained by concatenating a string from r and a string from s. So, for example, we can write grammar rules of the form

```
rule ::= 'a' name1 'b1'
```

Alternation

If r and s are regular expressions then so is $r|s$, and the elements of the set $r|s$ are strings in r together with the strings in s. So, for example, we can write grammar rules of the form

```
rule ::= 'a' name1 | B | 'b1' .
```

The concatenation operation has higher priority than the alternate operation. So

`'a' 'b' | B 'c'` is the set {`'a''b', B'c'`}
not the set {`'a''b''c', 'a'B'c'`}.

Parentheses

If r is a regular expression then so is (r), and the elements of the set (r) are exactly the elements of r. Parentheses have higher priority than all the other operations and their role is to allow other priorities to be overridden. So, for example,

```
rule ::= 'a' ( 'b' | B ) 'c' .
```

is equivalent to

```
rule ::= 'a' 'b' 'c' | 'a' B 'c' .
```

and to

```
rule ::= 'a' rule_1 'c' .
rule_1 ::= 'b' | B .
```

Zero or one

We use square brackets to indicate 'one or none'. If r is a regular expression then so is $[r]$, and the elements of the set $[r]$ are the elements of r together with the empty string. So, we can write grammar rules of the form

```
rule ::= [expr] .
```

which is equivalent to the rule

```
rule ::= expr | ϵ .
```

Zero or many

We use braces to indicate 'zero or many'. If r is a regular expression then so is $\{r\}$, and the elements of the set $\{r\}$ are the strings formed by concatenating zero or more strings from r together. So, for example, we can write rules of the form

```
rule ::= {'a'}'b' .
```

which is equivalent to the (infinite) rule

```
rule ::= 'b' | 'a''b' | 'a''a''b' | 'a''a''a''b' | ...
```

We can write more complicated grammar rules, for example

```
rule ::= ('a' | 'c'){'a' | 'c'} .
```

which is equivalent to the (infinite) rule

```
rule ::= 'a' | 'c' | 'a''a' | 'a''c' | 'c''a' | 'c''c'
         | 'a''a''a' | 'a''a''c' | 'a''c''a' | 'a''c''c'
         | 'c''a''a' ...
```

One or many

We use angle brackets to indicate 'one or many'. If r is a regular expression then so is $< r >$, and the elements of the set $< r >$ are the strings formed by concatenating one or more strings from r together. So, for example, we can write rules of the form

```
rule ::= <'a'> .
```

which is equivalent to the rule

```
rule ::= 'a' | 'a''a' | 'a''a''a' | 'a''a''a''a' ...
```

3.2 `rdp`'s IBNF

We call the particular extended form of BNF which **rdp** uses an *iterator BNF* or IBNF. In this section we describe the additional construct `@` which distinguishes IBNF from standard EBNF.

Sometimes we need to specify strings of certain lengths. For example, we may want to specify the strings of up to eight a's and b's. We could do this by writing out all the strings in the standard BNF style, but there are 511 of them! This language can be specified in **rdp** input grammars using the rule

```
rule ::= ('a' | 'b')0@8# .
```

Here, the symbol `#` is being used to denote the absence of a token. It is necessary because the iterator operator can also be used to specify a separator between symbols. Integer arithmetic expressions are sequences of integers with +'s between them, for example $17 + 80 + 9 + 27$. This can't be expressed simply using braces `{}` because the first integer in the list doesn't have a preceeding +, i.e. we can't just write `{'+' INTEGER}`. (Of course, we can write `INTEGER{'+' INTEGER}` but if the are semantic actions associated with the calls to `INTEGER` then these will need to be repeated.) The **rdp** grammar rule

```
rule ::= (INTEGER)2@8'+' .
```

describes the integer sums which have between two and eight operands.

Formal definition of the `@` operator

Each of the last four regular expressions in the EBNF description in section 3.1 is a special case of a regular expression which is based on the parameterised operator `@`. We define *iterator expressions* as follows:

1. Any regular expression is an iterator expression
2. If r is an iterator expressions then so is $(r)_l@_h\,'\mathrm{x}'$, where l and h are integers and $'x'$ is a token or the special symbol `#`.

The elements of the set $(r)_l@_h\,'\mathrm{x}'$ are strings which are the concatenation of between l and h strings from r, alternated with x if it is not `#`. (In the case where $h = 0$ there must be at least l strings from r but there is no upper limit on this number.)

Examples and correspondences

The following rule allows us to declare function prototypes which contain arbitraily long comma delimited lists of parameters

```
rule ::= ID '(' (ID)0@0',' ')' .
```

This rule matches things like

`My_Function(var1, var2, var3)` or `print()`

Note: For **rdp** the upper and lower limits are written next to the @ operator, rather than subscripted.

The following is a list of the correspondences between IBNF and standard EBNF constructs.

```
rule ::= (r)0@0#.  corresponds to   rule ::= { r }.
rule ::= (r)1@0#.  corresponds to   rule ::= < r >.
rule ::= (r)0@1#.  corresponds to   rule ::= [ r ].
rule ::= (r)1@1#.  corresponds to   rule ::= ( r ).
```

We are now in a position to give an `rdp` grammar for the language of integer expressions, including - and /. This forms part of the `mini` grammar which will be discussed in later chapters.

```
(** expr6.bnf **)

e1 ::= e2 {'+' e2 | '-' e2 } .
e2 ::= e3 {'*' e3 | '/' e3 } .
e3 ::= '+' e4 | '-' e4 |  e4 .
e4 ::= ID | INTEGER | '(' e1 ')' .
```

An `rdp`-generated parser for this grammar accepts grammar strings of the form, for example,

```
10 * ( -4 + 6 * (-5)) / (sum - 27).
```

3.3 Derivations in IBNF

In the previous section we have described the strings generated by **rdp**'s iterator constructs. However, to understand what **rdp** is actually doing, to make full sense of the error messages it produces, and to use semantic actions (see Chapter 6) it is necessary to understand the IBNF constructs in terms of the derivations they produce.

Basically, recursive descent parsers call a routine, or function, at each step in the derivation being constructed. Each step corresponds to the replacement of a non-terminal by one of the alternates from its grammar rule. Thus there is effectively a routine for each alternate of each rule. **rdp** makes an internal subrule for each alternate. For example, **rdp** represents the grammar

```
rule ::= 'a' '+' rule | 'b' .
```

internally as

```
rule ::= rdp_rule_0 | rdp_rule_1 .
rdp_rule_0 ::= 'a' '+' rule .
rdp_rule_1 ::= 'b' .
```

If an alternate contains an iterator construct and something else then **rdp** takes the iterator construct out and makes a new rule for it. For example, **rdp** represents the grammar

```
rule ::= 'a' ('b')0@4',' .
```

internally as

```
rule ::= 'a' rdp_rule_0 .
rdp_rule_0 ::= ('b')0@4',' .
```

Recall that parantheses are special case of the iterator operator. Effectively, if if you write

```
rule ::= ( r ) .
```

rdp automatically inserts an iterator construct and translates the rule to

```
rule ::= ( r )1@1# .
```

So including parantheses in a rule will cause **rdp** to generate an new corresponding internal subrule. For example, **rdp** represents the grammar

```
rule ::= 'a' ('b' | 'c') 'd' .
```

internally as

```
rule ::= 'a' rdp_rule_2 'd' .
rdp_rule_2 ::= rdp_rule_1 | rdp_rule_0 .
rdp_rule_1 ::= 'b' .
rdp_rule_0 ::= 'c' .
```

rdp treates single iterator constructs as though they were rules with an aternate for each possible string. In other words

```
rule ::= ('a')1@6'+' .
```

behaves in the same way as

```
rule ::= 'a' | 'a''+''a' | ...
             | 'a''+''a''+''a''+''a''+''a''+''a' .
```

Recursion issues

The language $\{a, aa, aaa, aaaa, \ldots\}$ can be described in three obvious ways in **rdp**'s IBNF, using right recursion, using left recursion and using the iterator construct:

```
rule ::= 'a' [ rule ] .
rule ::= [ rule ] 'a' .
rule ::= ('a')1@0#
```

(In fact **rdp** will not generate a parser from the second grammar because it is not left factored, see Chapter 4, but we shall use it here to illustrate the constrast in approaches.) It would be possible to implement the iterator operator recusively so it behaved like the first rule. In this case a derivation of **aaa** would have the form

rule $\Rightarrow$ **a rule** $\Rightarrow$ **a a rule** $\Rightarrow$ **a a a rule** $\Rightarrow$ **a a a**

In the case of the left recursive representation, a derivation of **aaa** would have the form

rule $\Rightarrow$ **rule a** $\Rightarrow$ **rule a a** $\Rightarrow$ **rule a a a** $\Rightarrow$ **a a a**

However, iterator constructs, as their name implies, are actually implemented in **rdp** using iteration so

```
rule ::= ('a')1@0#
```

behaves as though it were the (infinite) rule

```
rule ::= 'a' | 'a''a' | 'a''a''a' | ...
```

As a result, a derivation of **aaa** in the iterator grammar has the form

rule $\Rightarrow$ **a a a**

These issues are important when using semantic actions in the grammar, see Chapter 6, because they determine when in the parse the actions are executed.

Chapter 4

Restrictions on `rdp` grammars

As we discussed in Chapter 2, **rdp** generates parsers which use a *recursive descent* technique. This means that there are restrictions on the grammars which admit **rdp**-generated parsers. If **rdp** is presented with a grammar which breaks these restrictions then it will issue a diagnostic message explaining the nature of the problem and indicating the place in the grammar where it occurs. In order to understand these messages and to use **rdp** effectively, it is necessary to understand the conditions which input grammars must satisfy. In this chapter we consider these conditions in detail. We shall use the grammar **expr1.bnf**

```
S ::= E Y.
Y ::= ['+' S].
E ::= T X.
X ::= ['*' E].
T ::= 'a' | 'b'.
```

introduced in Chapter 2, to illustrate the discussion.

4.1 Deterministic choice on alternates

At each step in the parsing process, the parser replaces a non-terminal with a string from the right hand side of that non-terminal's grammar rule. When there is more than one alternate in a grammar rule the parser needs an algorithm for deciding which of the alternates to choose. This decision is made by looking at the *current input symbol.* In order to see how this is done we need to consider the parsing process in more detail.

Strings which can appear in derivations that begin with the start symbol are called *sentential forms.* So in the following S, α_1, ..., α_n are all sentential forms:

$$S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n.$$

By the *current sentential form* at a stage in a parser execution we shall mean the sentential form which was constructed at the previous step. So α_n is the current sentential form at the stage at which a parser has constructed the above derivation steps.

Suppose that the input string $xyxyz$ is being parsed. The parser is generating a left-most derivation of the string, thus in the first steps of the process the parser is replacing the first symbol of the current sentential form and is attempting to construct a sentential form which looks like $x\delta$. Suppose that the parser has constructed the steps

$$S \Rightarrow \alpha \Rightarrow \beta \Rightarrow \gamma$$

that γ, the current sentential form, begins with the non-terminal X and that X has associated grammar rule

$$X ::= \ \beta_1 \mid \beta_2 \mid \beta_3 \mid \beta_4 \ .$$

At the next step the parser should only replace X with β_1 if there is some derivation $\beta_1 \stackrel{*}{\Rightarrow} x\beta_1'$ or if $\beta_1 \stackrel{*}{\Rightarrow} \epsilon$. In any other case it will be impossible to complete the derivation if β_1 is chosen to replace X.

For example, suppose that we are using the expression grammar **expr1.bnf** to parser the string **a+b** and that the construction has reached the stage

$$S \Rightarrow EY \Rightarrow TXY$$

If at the next step **T** were to be replaced by **b**, giving

$$S \Rightarrow EY \Rightarrow TXY \Rightarrow bXY$$

it would be impossible to complete the derivation and generate **a+b**.

4.2 FIRST sets

The above discussion highlights the need to know which terminals can appear at the beginning of something derivable from a given string γ. Such terminals belong to the so-called FIRST set of γ. Formally we define FIRST(γ) to be the set of terminals which can begin a string derivable from γ, together with ϵ if $\gamma \stackrel{*}{\Rightarrow} \epsilon$.

For the example **expr1.bnf** at the beginning of the chapter we have

$\text{FIRST}(a) = \{a\}$
$\text{FIRST}(T) = \{a, b\} = \text{FIRST}(E) = \text{FIRST}(S)$
$\text{FIRST}(X) = \{*, \epsilon\}$
$\text{FIRST}(EY) = \{a, b\} = \text{FIRST}(TX)$.

The general description of FIRST sets is:

$$\text{FIRST}_{\mathbf{T}}(\gamma) = \{\ t \in \mathbf{T} \mid \gamma \stackrel{*}{\Rightarrow} t\gamma' \}$$

$$\text{FIRST}(\gamma) = \begin{cases} \text{FIRST}_{\mathbf{T}}(\gamma) \cup \{\epsilon\}, & \text{if } \gamma \stackrel{*}{\Rightarrow} \epsilon, \\ \text{FIRST}_{\mathbf{T}}(\gamma), & \text{otherwise.} \end{cases}$$

Notice that if t is a terminal then

$$\text{FIRST}(t) = \{t\} = \text{FIRST}(t\gamma)$$

for any string γ.

rdp-generated parsers maintain a list of the FIRST sets, and their contents, for each alternate in each grammar rule of the grammar. The parsers use these sets to decide with which alternate to replace a given non-terminal.

4.3 Parsing with FIRST sets

As the parse proceeds, the first few symbols of the current sentential form will correspond exactly to an initial portion of the input string. The parser reads in the input string symbol-by-symbol, starting from the left of the string. When the first symbol in the input string appears at the front of the current sentential form the parser reads in the next symbol from the input. At each stage in the parsing process the parser tries to replace the left-most non-terminal in the current sentential form with an alternate that has the current input symbol in its FIRST set. For example, suppose that the input string is $xyxyz$, that the parser has so far constructed the derivation

$$S \Rightarrow \alpha_1 \Rightarrow \ \ldots \ \Rightarrow \alpha_m \Rightarrow xyxX\gamma,$$

and that X has associated grammar rule

$$X ::= \ \beta_1 \mid \beta_2 \mid \beta_3 \mid \beta_4 \ .$$

The current input symbol is y and so the parser needs to replace X by which ever alternate has y in its FIRST set.

If no alternate has this property, and if X does not derive ϵ, then the derivation cannot be completed and the parse has failed. If more than one alternate has this property then the parser cannot decide how to proceed.

Thus it is necessary for grammars which are to have **rdp**-generated parsers to have the *disjoint* FIRST *set property.* In fact, grammars that are input to **rdp** need to have a slightly stronger property than this, which we will describe in the next three sections.

4.4 The problem with ϵ rules

We have seen in the previous section that **rdp** requires each grammar rule in the input grammar to have alternates with disjoint FIRST sets. That is, if

$$X ::= \ \alpha \mid \beta \ .$$

then the intersection of the FIRST sets must be empty, i.e. $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$ (here $\emptyset$ denotes the emptyset, the set $\{\}$ which has no elements). However, if the grammar contains ϵ rules (grammar rules of the form $A ::= \ldots \mid \ \epsilon \ \mid \ldots$) then this property may not be enough to allow the parser to determine which alternate to use.

Consider the grammar

```
S ::= 'b' A 'a' .
A ::= ['a'] .
```

If an **rdp**-generated parser for this grammar is given input **ba** and has constructed the steps

$$\texttt{S} \Rightarrow \texttt{bAa}$$

then the current input symbol is a and since $A \Rightarrow a$ the parser may take this as the next step, giving

$$S \Rightarrow bAa \Rightarrow baa$$

which cannot be extended to generate ba. However, if the parser performed the step $A \Rightarrow \epsilon$ then we would get a successful parse

$$S \Rightarrow bAa \Rightarrow ba$$

Why not make choosing the ϵ step the default? If the parser is given input baa then choosing the ϵ alternate would result in failure. The problem is that the parser cannot decide whether to use ϵ just by looking at the current input symbol a.

We can see the general problem by considering again the example, from the previous section, in which the input string is $xyxyz$, the parser has so far constructed the derivation

$$S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_m \Rightarrow xyxX\gamma,$$

and X has associated grammar rule

$$X ::= \beta_1 \mid \beta_2 \mid \beta_3 \mid \beta_4 \; .$$

Suppose also that $\beta_4 \stackrel{*}{\Rightarrow} \epsilon$. We could replace X with β_4 at the next step in the derivation and hope to complete the derivation from the string γ. If it is also the case that one of the other alternates, β_1 say, has y in its FIRST set then we can't decide whether to use β_1 or β_4 to replace X.

If β_4 were used in the next step of the derivation and the parse were to be successful then we would have to have $y \in \text{FIRST}(\gamma)$ and hence $\gamma \stackrel{*}{\Rightarrow} y\gamma'$ for some string γ'. This would mean that

$$S \stackrel{*}{\Rightarrow} xyxX\gamma \stackrel{*}{\Rightarrow} xyxXy\gamma'$$

and hence that y can follow X in some sentential form. Thus we are led to consider the so-called FOLLOW sets.

4.5 FOLLOW sets

The convention for recursive descent parsers is that the ϵ generating rule, β_4 in the above example, will only be chosen if there is no alternate that has an appropriate FIRST set. This will be a correct strategy if none of the elements in these FIRST sets can also *follow* X in a sentential form. This is what is required for the above example because the existence of a successful derivation using β_4 would mean that y could follow X, and hence the non-overlap between FIRST sets and elements that can follow X would mean that no alternate had been chosen on application of the FIRST set criterion.

Thus we are interested in the set of terminals which can follow X in some sentential form.

In the expression grammar **expr1.bnf** we have

$$\mathtt{S} \stackrel{*}{\Rightarrow} \mathtt{E} + \mathtt{S}, \qquad \mathtt{S} \stackrel{*}{\Rightarrow} \mathtt{T} * \mathtt{EY}, \qquad \mathtt{S} \stackrel{*}{\Rightarrow} \mathtt{T} + \mathtt{S},$$

in fact

$\text{FOLLOW}_{\mathbf{T}}(\mathtt{E}) = \{+\}$
$\text{FOLLOW}_{\mathbf{T}}(\mathtt{T}) = \{*, +\}$

Formally, for any non-terminal A in any grammar we define

$$\text{FOLLOW}_{\mathbf{T}}(A) = \{t \in \mathbf{T} \mid S \stackrel{*}{\Rightarrow} \alpha A t \beta\}.$$

If A can occur at the end of a sentential form then the FOLLOW set of A also contains a special end-of-file symbol **EOF**. Then we have

$$\text{FOLLOW}(A) = \begin{cases} \text{FOLLOW}_{\mathbf{T}}(A) \cup \{\mathtt{EOF}\}, & \text{if } S \stackrel{*}{\Rightarrow} \gamma A, \\ \text{FOLLOW}_{\mathbf{T}}(A), & \text{otherwise.} \end{cases}$$

So for the expression grammar **expr1.bnf** we have

$\text{FOLLOW}(\mathtt{X}) = \{+, \mathtt{EOF}\}$

4.6 LL(1) grammars

Grammars which have the properties that

- no two distinct alternates in one grammar rule have common elements in their FIRST sets,
- and that if a non-terminal X derives ϵ then the FIRST sets of the alternates in its grammar rule must be disjoint from the FOLLOW set of X

are called LL(1) grammars. Formally, a grammar is LL(1) if

1. if X ::= ...| α |...| β |... t hen $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
2. if X ::= ...| α |... and $X \stackrel{*}{\Rightarrow} \epsilon$ then $\text{FIRST}(\alpha) \cap \text{FOLLOW}(X) = \epsilon$.

rdp requires its input grammars to be LL(1) (although there is one special case involving the iterator construct in **rdp**'s IBNF). If the input grammar is not LL(1) then **rdp** issues an error message detailing which rule(s) is causing the problem.

4.7 Overriding the LL(1) restrictions

There are some constructs in some programming languages which cannot be expressed in an LL(1) grammar. A classic example is the **if..then..else** statement and its variants. For example, using the following grammar

```
(** ifelse1.bnf **)

S ::= 'if' B 'then' S X | 'STOP' | 'SKIP' .
X ::= ['else' S] .
B ::= 'true' | 'false' .
```

there are two distinct derivations of the string

if true then if false then STOP else SKIP

When the parser has constructed the following portion of a derivation

S $\Rightarrow$**if B then SX** $\Rightarrow$**if true then SX** $\Rightarrow$**if true then if B then S X X**

there is no way to decide whether to replace the first X by ϵ or by **'else' S**.

If the above grammar is input to **rdp** the following message is generated:

```
******: Error - LL(1) violation - rule
 X ::= X .
 contains null but first and follow sets both include: 'else'
******: Error - LL(1) violation - rule
 rdp_X_1 ::= [ 'else' S ] .
 contains null but first and follow sets both include: 'else'
******: Error - LL(1) violation - rule
 rdp_X_2 ::= [ 'else' S ] .
 contains null but first and follow sets both include: 'else'
******: Fatal - Run aborted without creating output files
 - rerun with -F to override
```

From the point of view of parsing, the choice is irrelevant because either will result in a successful completion of the derivation. (Of course, the user needs to know which choice will be made so that appropriate semantics can be inserted. The **rdp** default actions are explained in detail in Chapter 7 of [JS97b].)

We can force **rdp** to create a parser by running it with the flag **-F**. There is a target **parserf** in the makefile which calls **rdp** with the **-F** flag. Typing

```
make GRAMMAR=examples\rdp_tut\ifelse1 parserf
```

excutes the command **rdp -F -oifelse1 examples\rdp_tut\ifelse1** and generates the following message:

```
******: Error - LL(1) violation - rule
 X ::= X .
 contains null but first and follow sets both include: 'else'
******: Error - LL(1) violation - rule
 rdp_X_1 ::= [ 'else' S ] .
 contains null but first and follow sets both include: 'else'
******: Error - LL(1) violation - rule
 rdp_X_2 ::= [ 'else' S ] .
 contains null but first and follow sets both include: 'else'
******: Warning - Grammar is not LL(1) but -F set: writing files
******: 3 errors and 1 warning
Borland C++ 5.0 for Win32 Copyright (c) 1993,1996 Borland International
```

```
rdparser.c:
bcc32 -erdparser.exe rdparser.obj arg.obj graph.obj memalloc.obj
              scan.obj scanner.obj set.obj symbol.obj textio.obj
Borland C++ 5.0 for Win32 Copyright (c) 1993,1996 Borland International
Turbo Link Version 1.6.72.0 Copyright (c)1993,1996 Borland International
rdparser -v -Vrdparser.vcg -l examples\rdp_tut\ifelse1.str

rdparser
Generated on Dec 6 1997 8:55:27 and compiled on Dec 4 1997 at 9:22:37
******:
     1: if true then if false then STOP else SKIP
******: 0 errors and 0 warnings
******: 0.000 CPU seconds used
```

Notice that this time instead of issuing a fatal error message, **rdp** has issued a warning and generated the files. (Since the **-F** flag overrides **rdp**'s safety checks it should be used with caution.)

4.8 Inspecting the FIRST and FOLLOW sets

rdp allows you to look at the FIRST and FOLLOW sets that it has created for a particular grammar. This is useful if the grammar is not LL(1) because it is possible to see where the offending sets overlap. It also means that **rdp** can be used as a tool for constructing FIRST and FOLLOW sets for any given grammar. This is particularly useful for FOLLOW sets whose construction by hand is quite error prone.

Using the flag **-e** causes **rdp** to print out in BNF format the grammar rules that it is using internally, to detail the FIRST set for each alternate and the **STOP** set (the FOLLOW set together with **EOF**) for each non-terminal. Typing

rdp -e examples\rdp_tut\expr1

results in the following output. (Here, the non-terminals are shown as functions for reasons which are explained in Chapter 6.)

```
E(void):void ::= rdp_E_0() .
First set is {'a', 'b'}
Stop set is {EOF, '+'}
Production is called 2 times

S(void):void ::= rdp_S_0() .
First set is {'a', 'b'}
Stop set is {EOF}
Production is called 2 times

T(void):void ::= rdp_T_0() | rdp_T_1() .
First set is {'a', 'b'}
Stop set is {EOF, '*', '+'}
Production is called once

X(void):void ::= rdp_X_2() .
First set is {(NULL) '*'}
```

```
Stop set is {EOF, '+'}
Production is called once

Y(void):void ::= rdp_Y_2() .
First set is {(NULL) '+'}
Stop set is {EOF}
Production is called once

rdp_E_0(void):void ::= T() X() .
First set is {'a', 'b'}
Stop set is {EOF, '+'}
Production is called once

rdp_S_0(void):void ::= E() Y() .
First set is {'a', 'b'}
Stop set is {EOF}
Production is called once

rdp_T_0(void):void ::= RDP_T_a() .
First set is {'a'}
Stop set is {EOF, '*', '+'}
Production is called once

rdp_T_1(void):void ::= RDP_T_b() .
First set is {'b'}
Stop set is {EOF, '*', '+'}
Production is called once

rdp_X_0(void):void ::= RDP_T_17 /* * */() E() .
First set is {'*'}
Stop set is {EOF, '+'}
Production is called once

rdp_X_1(void):void ::= [ rdp_X_0() ].
First set is {(NULL) '*'}
Stop set is {EOF, '+'}
Production is called once

rdp_X_2(void):void ::= rdp_X_1() .
First set is {(NULL) '*'}
Stop set is {EOF, '+'}
Production is called once

rdp_Y_0(void):void ::= RDP_T_18 /* + */() S() .
First set is {'+'}
Stop set is {EOF}
Production is called once

rdp_Y_1(void):void ::= [ rdp_Y_0() ].
First set is {(NULL) '+'}
Stop set is {EOF}
Production is called once
```

```
rdp_Y_2(void):void ::= rdp_Y_1() .
First set is {(NULL) '+'}
Stop set is {EOF}
Production is called once
```

Chapter 5

The mini grammar

In this chapter we give an `rdp` IBNF definition `mini1.bnf` of a small language. The language allows variable declarations and assignment of integer arithmetic expressions to those variables. Addition, subtraction, multiplication and division are all left associative, and multiplication and division have higher priority than addition and subtraction. There are also unary plus, `+`, and minus, `-`, signs, and an exponentiation operator, `^`, which is right associative. It allows branching *via* an `if` statement, and variable assignment and declaration. The C-style `=` sign is used for assignment. There is a print statement which can print sequences of strings and values of expressions. Strings are delimited by double quotes (`"`), and comments are enclosed in Pascal-style brackets `(* *)` and can be nested.

```
(** mini1.bnf **)

program   ::= {([var_dec  | statement ]) ';' }.
var_dec   ::= 'int' ( ID [ '=' e1 ] )@','.
statement ::=  ID '=' e0
             | 'if' e0 'then' statement [ 'else' statement ]
             | 'print' '(' ( e0 | String )@',' ')'.
e0 ::=  e1 ['>' e1 | '<' e1 | '>=' e1 |'<=' e1 | '==' e1 | '!=' e1].
e1 ::= e2 {'+' e2 | '-' e2 } .
e2 ::= e3 {'*' e3 | '/' e3 } .
e3 ::= '+' e4 | '-' e4 | e4 .
e4 ::= e5 ['^' e1] .
e5 ::= ID | INTEGER | '(' e1 ')' .
comment ::= COMMENT_NEST('(*' '*)').
String ::= STRING_ESC('"' '\\') .
```

The `'if'...` alternate in the grammar rule for `statement` is inherently ambiguous. Thus the above grammar is not LL(1). When the grammar is input to `rdp` it will issue an error message and terminate. If `rdp` is run with the flag `-F` this will force `rdp` to produce a parser from the above grammar.

```
rdp -F examples\rdp_tut\mini1

******: Error - LL(1) violation - rule
 rdp_statement_2 ::= [ 'else' statement ] .
 contains null but first and follow sets both include: 'else'
******: Warning - Grammar is not LL(1) but -F set: writing files
******: 1 error and 1 warning
```

The generated parser will resolve the ambiguity in 'if' statements by using 'longest match'. The effect of this is that an 'else' clause will be assumed to match the nearest 'then' to the left. In other words

```
if a<b then if c==d then c=1 else d=1 ;
```

will be parsed as

$$if\ a < b\ then\ (if\ c == d\ then\ c = 1\ else\ d = 1)\ ;$$

not as

$$if\ a < b\ then\ (if\ c == d\ then\ c = 1)\ else\ d = 1\ ;$$

Chapter 6

Semantic actions

So far we have only described how to use **rdp** to generate a parser for a language. Parsers allow us to test whether a given string is in the language, but what we actually want is to execute some behaviour when a sentence is recognised. For example, a compiler, on recognising a sentence, constructs an equivalent sentence (one with the same meaning) in some specified target language. An interpreter, on recognising a sentence, executes it. A pretty printer, on recognising a sentence, reformats it and then prints it out in what is usually a more readable form.

rdp can be used to generate parsers which do useful work of these types. The basic approach is that *semantic actions* are inserted in the grammar rules; these actions are executed by the generated parser when the rule is used.

Attributes are used to pass information between rules and can be used within semantic actions. Two kinds of attributes are supported by **rdp**: *inherited attributes* which are passed as parameters into rules and *synthesized attributes* that act like return values from rules.

In this chapter we describe how various aspects of these mechanisms work for **rdp**-generated parsers. A full description of **rdp** semantic actions can be found in [JS97b]. In this guide we just illustrate the basic ideas and point out some of the things to be wary of.

We begin with a little discussion of the function call based approach which is used to implement **rdp**-generated parsers. We then consider a particular example of semantic action use, consider the iterator construct in more detail and give an introduction to the use of inherited attributes.

6.1 The function based implementation of `rdp`-generated parsers

The input for **rdp** is a language specification, **grammar.bnf** say, written in IBNF. **rdp** writes a C program, **grammar.c** say, based on the IBNF it is given. This program contains a function for each non-terminal in the grammar. The code for each function depends on the right hand side of the grammar rule for the non-terminal. In a simple **rdp** parser the functions take no input and return nothing.

Suppose that the input grammar is

```
(** functn1.bnf **)

S ::= INTEGER E .
E ::= '+' E | '-' E | INTEGER .
```

The rdp-generated file functn1.c will contain two functions void S() and void E() and a function scan_() which reads the next input symbol and stores it in a global variable current_input_symbol [1]. The function S() can be thought of as beginning with a call to a scanner function scan_test(INTEGER) which tests the current input symbol to see if it is an integer. If it is an integer then this symbol has been correctly parsed in which case scan_() is called to read in the next input symbol and then the function E() is called to continue the parse.

```
void S(void){
    scan_test( INTEGER );
    scan_();
    E();
}
```

The function E() contains a branch statement with a branch for each of the three alternates in its grammar rule. The current input symbol is tested against the FIRST set of each alternate and then the corresponding branch is executed. In this case scan_test('+') is called and if it returns with a match to the current input symbol, the next input symbol is read and then E() is called again. If the match is not found then scan_test('-') is called. If this matches then the next input symbol is read and E() is called, otherwise scan_test(INTEGER) is called. If this matches then the next input symbol is read, no other action is required. If this does not match then the input string is not in the language and a suitable error message is issued.

The complete parser simply calls scan_() to read in the first input symbol and then calls the function for the start symbol, in this case S().

When rdp generates a C file from the above grammar it provides a lot of other code to, for example, set up a symbol table and issue error messages. The following is a severely filleted version of the file functn1.c created using the command

```
rdp -ofunctn1 examples\rdp_tut\functn1
```

[1] This global variable is actually a record text_scan_data with a field which is a union one of whose entries holds the lexeme of the input symbol. To simplify this exposition we shall think of this entry as a global variable which we shall call current_input_symbol .)

```
/*****************************************************************
*
* Parser generated by RDP on Nov 02 1997 09:33:36 from functn1.bnf
*
*****************************************************************/
...
/* Parser forward declarations and macros */
static void E(void);
void S(void);

/* Parser functions */
static void E(void){
  if (scan_test('+')) {
    scan_();
    E(); }
  else
    if (scan_test('-')) {
      scan_();
      E(); }
    else
      if (scan_test( INTEGER )) { scan_(); }
      else { /* report error in input */ }
}

void S(void){
  scan_test( INTEGER );
  scan_();
  E();
}

int main( ... ) {
  ...
  scan_();
  S();         /* call parser at top level */
  ...
}
/* End of functn1.c */
```

6.2 Semantic actions – an example

The following grammar can be input to `rdp`:

```
(** functn2.bnf **)

S:integer ::= INTEGER:val1 [* result = val1 ; *]
              ('+' S:val2 [* result = val1 + val2 ; *] | ';' )
```

The underlying grammar rule is

```
S ::= INTEGER ( '+' S | ';' ) .
```

which generates sums of integers terminated by a semi-colon. The additional details are attribute names (following colons :) and actions (between brackets

[* *]). We shall see below that the attributes cause the corresponding parser functions to return values rather than void results, so we have **integer S()**, and the actions are inserted verbatim into the code for S(). We now describe these effects in detail.

The parser functions associated with each non-terminal in the grammar can be made to return a value. The type of this value is indicated after a colon on the left hand side of the corresponding grammar rule. So, the declaration **:integer** which appears on the left hand side of the rule causes the function S() to return an (unsigned) integer which is defined as a C **unsigned long**. The identifier which holds the value to be returned is always called **result**. Thus the function has the form

```
integer S(void){
  integer result;
  scan_test( INTEGER );
  scan_();
  if (scan_test('+')) {
    scan_() ;
    val2 = S(); }
  else
  ...
  return result ;
}
```

The grammar writer will often want to give the returned value from a routine a local name, so that it can be used. This is done by putting a colon and then the chosen local name after the symbol which generated the call to the routine. For example, the declaration S():val2 in **functn2.bnf** instructs the parser to write the value returned by the call to S() to a variable called val2. The value of a token such as INTEGER may also be given a local name, again by putting a colon and then the chosen local name after the token. For example, INTEGER:val1 causes the value of the INTEGER just scanned, which was written to **current_input_symbol** by **scan_()**, to be copied to val1. Thus the function S() can be thought of as having the form

```
integer S(void) {
  integer result;
  long int val1;
  integer val2;

  scan_test(INTEGER);
  val1 = current_input_symbol;
  scan_();
  if (scan_test('+')) {
     scan_();
     val2 = S();
  }
  else
```

```
    ...
    return result;
}
```

The statements which appear between the brackets `[*...*]` in **functn2.bnf** are fragments of C code. As the parser constructs a derivation it executes the fragments, as it meets them. This is done by inserting the code fragments verbatim in the parser function at the place where they are encountered. For example, the action after `INTEGER` is inserted before the call to `scan_test(+)` and the second action is inserted after the call to `S()`. Thus we can think of `S()` as having the form

```
integer S(void) {
  integer result;
  long int val1;
  integer val2;

  val1 = current_input_symbol;
  scan_test(INTEGER);
  scan_();
  result = val1 ;

  if (scan_test('+')) {
     scan_();
     val2 = S();
     result = val1 + val2 ; }
  else
     if (scan_test(';')) { scan_(); }
     else /* error report */

  return result;
}
```

To gain experience of these ideas the you might like to add an extra grammar rule and associated semantic action which prints out the value of an expression.

```
(** functn3.bnf **)

S ::= E:val [* printf("%i\n", val); *] .
E:integer ::= INTEGER:val1 [* result = val1; *]
               ('+' E:val2 [* result = val1 + val2; *] | ';' ) .
```

Running the generated parser on the string `2+3+6;` effectively causes the following code to be executed (the indentation indicates the nesting level of the parser function producing the output)

```
val1 = 2
result = val1
```

```
        /* code from subcall to E() */
        val1 = 3
        result = val1
            /* code from second subcall to E() */
            val1 = 6
            result = val1
        val2 = result     /* val2 == 6 */
        result = val1 + val2
    val2 = result         /* val2 == 9 */
    result = val1 + val2
    return result         /* return 11 */
```

The following output should be produced:

```
Generated on May 01 1997 17:15:40 and compiled on Apr 30 1997 at 13:02:55
******:
     1: 2 + 3 + 6;
11
******: 0 errors and 0 warnings
******: 0.020 CPU seconds used
```

6.3 Semantic actions in empty grammar rules

The following grammar generates strings which are sums of integers that are not terminated by a semi-colon.

```
(** arith1.bnf **)

E:integer ::= INTEGER:val1 [* result = val1 *]
                [ '+' E:val2 [* result = val1 + val2; *] ] .
```

This corresponds to a parser function which is essentially of the form

```
integer E(void) {
  integer result;
  long int val1;
  integer val2;

  scan_test(INTEGER);
  val1 = current_input_symbol;
  scan_();
  result = val1;
  if (scan_test('+')) {
     scan_();
     val2 = E();
     result = val1 + val2;
  }
  return result;
}
```

The semantic actions inserted inside IBNF square brackets [] are only executed if the non-epsilon part of the bracket is executed. In other words, the above grammar is treated like

```
E:integer ::= INTEGER:val1 [* result = val1 *]
               ('+' E:val2 [* result = val1 + val2; *] | epsilon).
```

not like

```
E:integer ::= INTEGER:val1 [* result = val1 *]
                ('+' E:val2 [* result = val1 + val2; *]
                     | epsilon [* result = val1 + val2; *] ) .
```

Often we need to execute a semantic action when an empty rule is used, for example to initialise a variable. Semantic actions which are to be executed on application of an empty rule (so called default rules) are appended to the square brackets using a colon. For example, the grammar

```
(** arith2.bnf **)

E:integer ::= INTEGER:val1
              ['+' E:val2 [*result=val1+val2;*] ]:[*result=val1;*]
```

behaves like

```
E:integer ::= INTEGER:val1
               ('+' E:val2 [* result = val1 + val2; *]
                     | epsilon [* result = val1; *] ) .
```

and corresponds to a parser function which is essentially of the form

```
integer E(void) {
  integer result;
  long int val1;
  integer val2;

  scan_test(INTEGER);
  val1 = current_input_symbol;
  scan_();
  if (scan_test('+')) {
     scan_();
     val2 = E();
     result = val1 + val2;  }
  else {
       /* default action processing */
       result = val1 ;  }
  return result;
}
```

6.4 Semantic actions and the iterator construct

Iterators are implemented as 'while loops'. At the beginning of each execution of the loop the left hand side of the iterator operator is executed as though it were an alternate of a grammar rule. The next input symbol is then compared to the delimiter (right hand side) of the iterator operator. If there is a match then the while loop is executed again.

For example, the rule

```
(** iter1.bnf **)

E ::= ('a' | 'b' ) 0@0 ',' .
```

can be thought of as corresponding to a parser function of the form

```
void E(void) {
   while (1) {
     if (scan_test('a')) { scan_(); }
     else {
        if (scan_test('b')) { scan_(); }
        else /* error */ }
     if (current_input_symbol !=  ',' ) break;
     scan_();
   }
}
```

which accepts, for example, `a,a,b,a` and `a`.

Part of the point of an iterator construct is that it does not involve sub-function calls, thus semantic actions are executed immediately after the corresponding token is parsed, i.e. 'on the way down', rather than when the function call is complete, i.e. 'on the way back up'.

Semantic actions can be placed in the left hand argument of the iterator and after the delimiter using a colon. The latter action is executed only if the iterator does not consume any input symbols, i.e. if the low count is 0 and it matches the empty string. (This is the same feature as `[...]:[*...*]` described in the previous section.)

In the following example semantic actions are being used to count the numbers of a's, b's and delimiter ,'s in a given input string. For example, on input `a,b,b,b,a,a,a,a,a,b,a` we get

```
******:
     1: a, b, b, b, a, a, a, a, a, b, a
7, 4, 10
******:
```

The number of delimiters should be one less than the sum of the numbers of a's and b's, except in the case of the empty string when all the numbers should be 0. Thus a separate action is executed in this case.

```
(** iter2.bnf **)

E ::= [*int left=0, right=0, delim=-1;*]
        ('a'[*left++; delim++;*] | 'b'[*right++; delim++;*])
           0@0 ',':[* delim=0;*]
      [* printf("%i, %i, %i\n", left, right, delim);*] .
```

This can be thought of as corresponding to a parser function of the form

```
void E(void) {
    int left=0, right=0, delim=-1;
    if (scan_test('a') | scan_test('b')) {
      while (1)  {
        if (scan_test('a')) {
          scan_();
          left++; delim++; }
        else
          if (scan_test('b')) {
            scan_();
            right++; delim++; }
          else /* error */
        if (current_input_symbol != , ) break;
        scan_();
      } }
    else { delim=0; }

    printf("%i, %i, %i\n", left, right, delim);
}
```

6.5 Left associative operators

Recall the grammar `arith2.bnf`

```
E:integer ::= INTEGER:val1
              ['+' E:val2 [*result=val1+val2;*] ]:[*result=val1;*]
```

If we run the parser generated from this grammar on the string `2 + 3 + 6` the sum will effectively be calculated in a right associative manner, i.e. `2 + (3 + 6)`. This is acceptable since addition is associative and the result is the same in either case. However, if we used the same approach to specify subtraction we would get counter-intuitive outcomes. Running **rdp** with the grammar

```
(** arith3.bnf **)

S ::= E:val [* printf("%i\n", val) ; *] .
E:integer ::= INTEGER:val1
              ['-' E:val2 [*result=val1-val2;*] ]:[*result=val1;*] .
```

and then running the resulting parser on the string `2 - 3 - 6` produces

```
Generated on May 3 1997 7:01:20 and compiled on Apr 30 1997 at 8:02:55
******:
     1: 2 - 3 - 6
5
******: 0 errors and 0 warnings
******: 0.058 CPU seconds used
```

The result is `2-(3-6) = 5` rather than the expected `(2-3)-6 = -7`.

We consider three ways of specifying a grammar so that operators such as `'-'` are left associative.

One is to use inherited attributes, and will be discussed in the next section.

Another is to use a left recursive definition. We could begin with the grammar

```
S ::= E .
E ::= [ E '-' ] INTEGER.
```

then annotate it to give

```
S ::= E .
E:integer ::= [* int flag = 1; *] [ E:valB '-']:[* flag = 0; *]
              INTEGER:valA [* if(flag){result = valB - valA;}
                                  else {result=valA;}; *].
```

A correct parser based on this grammar would give subtraction left associative semantics but because the grammar is left recursive **rdp** cannot generate a correct parser from it.

A third way of enforcing left associativity is to use the iterator construct. An **rdp**-generated parser from the grammar

```
(** arith4.bnf **)

S ::= E:val [* printf("%i\n", val) ; *] .
E:integer ::= INTEGER:result
              {'-' INTEGER:val2 [*result=result-val2;*] } .
```

(which has underlying form `E ::= INTEGER { - INTEGER }.`) effectively executes the following steps on input `2 - 3 - 6`, giving the required evaluation.

```
current_input_symbol = 2 ;
result = current_input_symbol ; /* result == 2 */
  current_input_symbol = 3 ;
  val2 = current_input_symbol ;
  result = result - val2 ;      /* result == 2-3 */
    current_input_symbol = 6 ;
    val2 = current_input_symbol ;
    result = result - val2 ;    /* result == (2-3)-6 */
```

6.6 Expression semantics in mini

We can use the techniques discussed in the previous section to add semantic actions to the grammar rules which define expressions in mini.

```
(** miniexp.bnf **)
USES("mexp_aux.h")

S ::= e1:val [* printf("%i\n", val); *] .
e1:integer ::= e2:result {'+' e2:val [* result = result + val; *]
                         | '-' e2:val [* result = result - val; *] }.
e2:integer ::= e3:result
              {'*' e3:val [* result = result * val; *]
               | '/' e3:val [* if(val==0)
                {text_message(TEXT_FATAL,"divide by zero attempted\n");}
                else {result = result / val;}; *] } .
e3:integer ::= '+' e4:result | '-' e4:val [*result = -val;*] | e4:result .
e4:integer ::= e5:result ['^' e4:val
            [*result = (integer) pow((double) result, (double) val );*] ].
e5:integer ::= ID | INTEGER:result | '(' e1:result ')' .
```

The exponent operator ^ is implemented using the C maths library function `pow()`. The `rdp` *directive* `USES(`*file*`)` tells `rdp` to include the contents of *file* in the generated parser. In our case the file `mexp_aux.h` contains the command to include the maths library.

```
#include <math.h>
```

The file `mexp_aux.h` can also be used to declare global variables which can then be used in the semantic actions.

6.7 Inherited attribute definition

Recall the grammar

```
(** arith3.bnf **)

S ::= E:val [* printf("%i\n", val) ; *] .
E:integer ::= INTEGER:val1
             ['-' E:val2 [*result=val1-val2;*] ]:[*result=val1;*].
```

from the previous section. This generates sequences of differences of integers, but it calculates the result using right associativity.

To get left associativity using the right recursive 'subtraction' grammar

```
S ::= E .
E ::= INTEGER [ - E ].
```

we need to add semantic actions in such a way that `val1`, the value of the first `INTEGER`, is passed into the function called for the following `E`.

We can pass parameters into function calls by inserting them in parentheses after the appropriate non-terminal. For example, consider the following annotation of the right recursive 'subtraction' grammar

```
(** arith5.bnf **)

S:integer ::= INTEGER:val E(val):result [*printf("%i\n",result);*].
E(lhs:integer):integer ::= ['-' INTEGER:val [* val = lhs - val; *]
                            E(val):result ]:[* result = lhs; *].
```

An **rdp**-generated parser for this grammar would effectively execute the following steps on input `2 - 3 - 6`

```
val = 2 ;
   lhs = val;          /* lhs == 2 */
   val = 3 ;
   val = lhs - val;
      lhs = val ;      /* lhs == 2-3 */
      val = 6;
      val = lhs - val;
         lhs = val;    /* lhs = (2-3)-6 */
         result = lhs; /* result = (2-3)-6 */
```

Running the **rdp**-generated parser for **arith5.bnf** on the input `2 - 3 - 6` should produce the following:

```
Generated on May 18 1997 9:19:07 and compiled on May 12 1997 at 9:15:30
******:
     1: 2 - 3 - 6
-7
******: 0 errors and 0 warnings
******: 0.034 CPU seconds used
```

Thus we see that **rdp** rules can have parameters passed into them. Each **rdp** rule name may be followed by a parenthesised list of **identifier:type** pairs which are instantiated into the parser rule as value parameters, so that

```
inherited_rule( x : integer y: real) ::= 'a' 'b'.
```

maps to

```
integer inherited_rule(integer x, real y) { ... }
```

6.7.1 Semantic actions for IF statements

A common use of inherited attributes is to pass information into a rule that will be used to switch semantic actions off and on.

The `if` statement in the `mini` grammar has two subclauses, one that should be executed if a specified conditional is true and another that should be executed if the conditional is false. We achieve this by passing a parameter in to all statements and ensuring that the semantic actions associated with the statement are only executed if the parameter is true.

```
(** mini2.bnf **)
USES("mexp_aux.h")

program   ::= {([var_dec | statement(1) ]) ';' }.
XSvar_dec   ::= 'int' ( ID [ '=' e1 ] )@','.

statement(flag:integer) ::=
          ID '=' e1 [*if(flag){/* assignment will go here*/;};*]
          | 'if' e0:cnd 'then' [* cnd = cnd && flag;*] statement(cnd)
                     [ 'else' [*cnd =!cnd&&flag;*] statement(cnd) ]
          | 'print''(' ( e0:val [* if(flag){printf("%i\n", val);};*]
                         | String:str
                              [*if(flag){printf("%s\n", str);};*]
                        )@',' ')'.

e0:integer ::=  e1:result
                [  '>' e1:val  [*result = result > val;*]
                 | '<' e1:val  [*result = result < val;*]
                 | '>=' e1:val [*result = result >= val;*]
                 | '<=' e1:val [*result = result <= val;*]
                 | '==' e1:val [*result = result == val;*]
                 | '!=' e1:val [*result = result != val;*]
                ].

e1:integer ::= e2:result {'+' e2:val [* result += val; *]
                           | '-' e2:val [* result -= val; *] } .

e2:integer ::= e3:result
               {'*' e3:val [* result *= val; *]
                | '/' e3:val [* if(val==0)
                {text_message(TEXT_FATAL,"divide by zero attempted\n");}
                else {result = result / val;}; *] } .

e3:integer ::= '+' e4:result | '-' e4:val [*result = -val;*] | e4:result.

e4:integer ::= e5:result ['^' e4:val
           [*result = (integer) pow((double) result, (double) val );*] ].

e5:integer ::= ID | INTEGER:result | '(' e1:result ')' .

comment ::= COMMENT_NEST('(*' '*)').
String:char* ::= STRING_ESC('"' '\\'):result .
```

In the grammar rule for **program** the call to **statement** is passed a constant value 1 because its associated semantic actions should always be executed. The actions associated with the assignment and print alternates of the **statement** grammar rule will be executed if the parent statement is called with a 'true' flag. (Some of the actions have not actually been written because we need to use a symbol table which will be discussed in Chapter 7.) The actions associated with the first sub-statement in the 'if' alternate will be executed if both the governing condition and the flag in the parent statement are true.

If we run an **rdp**-generated parser for the above grammar on input

```
if 1>2 then print(1, "true") else print(2, "false") ;
```

we get output of the form

```
******:
2
false
     1: if 1>2 then print(1, "true") else print(2, "false") ;
******: 0 errors and 0 warnings
******: 0.145 CPU seconds used
```

Chapter 7

Symbol tables in `rdp`

We have already mentioned that **rdp**-generated parsers deal with tokens, and that there is a built-in scanner which groups the input stream into token lexemes. The parser must keep track of the lexemes which match each token. For example, the parser only needs to know that the token it is currently dealing with is `ID`, but in the final output code we need to restore the actual identifier originally given. So this information must be stored somewhere. Also, at various stages in the input program an identifier will have a specific associated value, and usually an associated type. This information is held by an **rdp**-generated parser in a *symbol table.*

rdp has a built-in symbol table building library. The user can write parsers which use symbol tables by including calls to the **rdp** symbol table library functions. In this chapter we shall give a basic guide to using this library.

7.1 Hash coded symbol tables

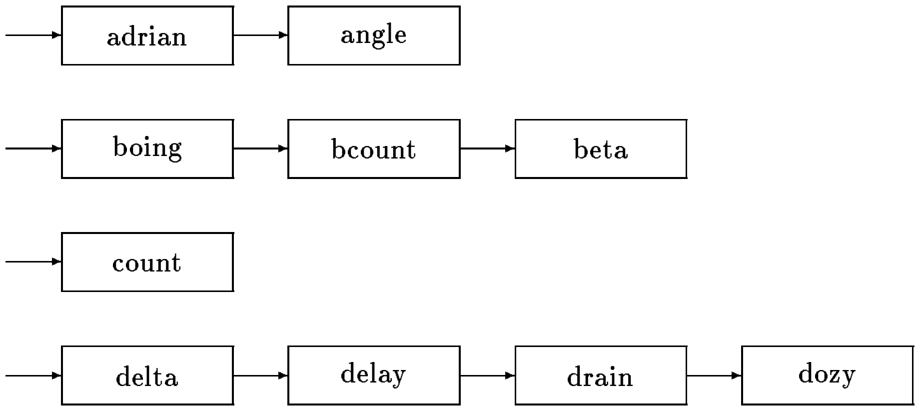
The symbol table is declared by the user in the BNF file which defines their language. The following is an example of a declaration of a symbol table which could be used in a parser for the mini language.

```
SYMBOL_TABLE(mini 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char* id; integer i; *]
            )
```

The first parameter, in this case `mini`, is the name of the symbol table. A novice **rdp** user can just use the above incantation, putting in the name they require, but in order to have some understanding of the different components of the definition it is necessary to have an elementary understanding of hash tables. (More detailed information on **rdp** symbol tables can be found in [JS97b].)

Symbol tables need to be of arbitrary size, since we do not know in advance how many identifiers we will encounter during a particular parse. The **rdp**-generated symbol tables are based on linked lists, organised to make looking up a value reasonably efficient. Rather than a single list, a symbol table actually has several lists, called *buckets*. So, for example, instead of having one list of length about 100, we may have 10 lists each of length about 10 and, provided we know which list to search, looking up an entry could be a factor of 10 quicker.

This is the principle of a *hash table*. The bucket in which a particular entry should be stored is calculated from that entry by a *hash function*. The idea is that the hash function should assign approximately the same number of entries to each of the buckets.



Perhaps the simplest hash function for a string is to add together the ASCII values for all of the characters in the string, and then take the modulus of the result with the number of sub-lists available. It turns out that this function works best if there are a prime number of sublists. An even better result is achieved if another number, coprime with the number of lists is factored in at each addition.

The **rdp** symbol table library contains a hash function of this type, called **symbol_hash_string**. To use it in an **rdp**-generated parser just declare it in the symbol table definition, as above. The two numbers 101 and 31 in the definition are the primes that the hash function is to use.

Every record in the symbol table has a key field which is used to access that record. **symbol_compare_string** is a function which is used by **rdp**'s symbol table library to compare an input string with these keys when accessing records.

At the end of the symbol table definition, between the `[* *]` brackets, are the data fields which contain the actual information held for each entry in the symbol table. The mini symbol table holds the identifiers of a mini program, and these all have type **integer**. So there are two data fields; the first holds the lexeme of the token and the other contains its (integer) value.

7.2 Assignment

The construct `ID = e1` in the mini grammar is intended to assign the value of the expression `e1` to the identifier which is the particular lexeme of `ID`. When an assignment is carried out the new value is placed in the appropriate field in the symbol table. This is done by using semantic actions in the rule which defines identifier declaration.

```
statement ::= ID:name '=' e1:val [* mini_cast(
                 symbol_lookup_key(mini, &name, NULL))->i = val; *]
```

The call to ID returns the lexeme which matched that instance of ID, and the symbol table is 'keyed' on this value. The rdp symbol library function symbol_lookup_key() looks up the entry in the symbol table which is keyed on name. In this entry, the field i holds the value of the identifier and the semantic action above assigns the value of e1, returned in a variable called val, to the field i.

When an identifier is to be assigned the value of another identifier, or when an expression involves an identifier,

```
fred_copy = fred ;
total = sub_total + 15;
```

then the values of these identifiers need to be extracted from the symbol table. This is also done using symbol_lookup_key().

```
e5:integer ::= ID:name [* result =  mini_cast(
                    symbol_lookup_key(mini, &name, NULL))->i; *]
```

The return type of a function such as symbol_lookup_key() depends in part on the user-defined structure of the entries in the symbol table, and thus is not fixed. To cope with this symbol_lookup_key() actually returns a void pointer, which is then *cast* to the appropriate type by a function *table*_cast(). This function is constructed automatically by rdp. The reader who is not confident in dealing with C-style void pointers need not worry about it. Just encase calls to functions such as symbol_lookup_key() in a call to *table*_cast() and everything will be dealt with automatically.

7.3 Identifier declaration

The construct

```
var_dec   ::= 'int' ID:name [ '=' e1:val ] .
```

in the mini grammar allows identifiers to be declared. The effect of a declaration is intended to be that an entry in the symbol table is created for that identifier. There is an option to assign a value to the identifier at the same time as it is declared. These effects can be achieved using the symbol table library function symbol_insert_key().

```
var_dec   ::= 'int' ID:name [ '=' e1:val ] [* mini_cast(
                symbol_insert_key(mini, &name, sizeof(char*),
                                   sizeof(mini_data)))->i = val;
                *].
```

The function `symbol_insert_key()` reserves enough space for both the key which will be used to access the particular entry and for the actual data which will be stored. These values depend on the data types specified by the user in their input grammar. In the case of `mini` we have specified that the key will be a string (the lexeme recognised by the scanner) and that the data will contain that string and an integer value. The size of the key is the third parameter of `symbol_lookup_key()`, and the size of the entry is the fourth parameter. The data fields enclosed between `[* *]` brackets in a declaration of a symbol table, *table* say, is referred to as *table*`_data` by `rdp`. Thus `sizeof(mini_data)` is the value required as the fourth parameter in our example.

7.4 Using undeclared variables

The symbol table can be used to resolve context sensitivities. In `mini` we intend that an identifier cannot be used before it is declared. However, to exclude something of the form

```
fred = 3 ;
int fred ;
```

from a language usually requires a context sensitive grammar. So instead we allow such constructs but then issue an error message when an attempt is made to execute semantic actions on such input. We use the fact that `symbol_lookup_key()` returns `NULL` if it doesn't find a particular entry in the symbol table.

```
statement ::= ID:name '=' e1:val
    [* if (symbol_lookup_key(mini, &name, NULL) == NULL)
        text_message(TEXT_ERROR,
               "Undeclared variable '%s'\n", name);
       else { mini_cast(
               symbol_lookup_key(mini, &name, NULL))->i = val;}
    *]
```

Chapter 8

A mini interpreter

We are now in a position to give a full decorated grammar, `mini_itp.bnf`, for the mini language. Running this grammar through `rdp` generates a parser which acts as an interpreter for programs written in the mini language.

```
(**   mini_itp.bnf   **)

USES("mexp_aux.h")

SYMBOL_TABLE(mini 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char* id; integer i; *]
            )

program ::= {([var_dec | statement(1) ]) ';' }.

var_dec ::= 'int' ( ID:name [ '=' e1:val ]
             [* mini_cast(symbol_insert_key(mini, &name, sizeof(char*),
                                            sizeof(mini_data)))->i = val;
                *]
              )@','.

statement(flag:integer) ::=
       ID:name '=' e1:val
       [* if(flag)
           if (symbol_lookup_key(mini, &name, NULL) == NULL)
             text_message(TEXT_ERROR, "Undeclared variable '%s'\n", name);
           else {
             mini_cast(symbol_lookup_key(mini, &name, NULL))->i = val; }
       *]

       | 'if' e0:cnd 'then' [* cnd = cnd && flag;*] statement(cnd)
                     [ 'else' [*cnd =!cnd&&flag;*] statement(cnd) ]
       | 'print''(' ( e0:val [* if(flag){printf("%i\n", val);};*]
                        | String:str
                             [*if(flag){printf("%s\n", str);};*]
                       )@',' ')'.
```

```
e0:integer ::=  e1:result
                [  '>' e1:val  [*result = result > val;*]
                 | '<' e1:val  [*result = result < val;*]
                 | '>=' e1:val [*result = result >= val;*]
                 | '<=' e1:val [*result = result <= val;*]
                 | '==' e1:val [*result = result == val;*]
                 | '!=' e1:val [*result = result != val;*]
                ].

e1:integer ::= e2:result {'+' e2:val [* result += val; *]
                           | '-' e2:val [* result -= val; *] } .

e2:integer ::= e3:result
               {'*' e3:val [* result *= val; *]
                | '/' e3:val [* if(val==0)
                {text_message(TEXT_FATAL,"divide by zero attempted\n");}
                else {result = result / val;}; *] } .

e3:integer ::= '+' e4:result | '-' e4:val [*result = -val;*] | e4:result.

e4:integer ::= e5:result ['^' e4:val
          [*result = (integer) pow((double) result, (double) val );*] ].

e5:integer ::= ID:name
          [* if (symbol_lookup_key(mini, &name, NULL) == NULL)
                text_message(TEXT_ERROR,
                         "Undeclared variable '%s'\n", name);
             else { result = mini_cast(
                         symbol_lookup_key(mini, &name, NULL))->i; }
          *]
          | INTEGER:result | '(' e1:result ')' .

comment ::= COMMENT_NEST('(*' '*)').
String:char* ::= STRING_ESC('"' '\\'):result .
```

If we input the above grammar to rdp

```
rdp -F examples\rdp_tut\mini_itp
bcc32 -P -Irdp_supp -c rdparser.c
bcc32 -erdparser.exe rdparser.obj arg.obj graph.obj memalloc.obj
            scan.obj scanner.obj set.obj symbol.obj textio.obj
rdparser examples\rdp_tut\mini_itp.str
```

running the resultant parser on the input

```
(****** mini_itp.str ******)

int fred = 1 ;
print("value of fred = ", fred) ;
int me = fred + 1 ;
print("value of me = ", me) ;
me = fred * me + 6/3*5 ;
```

```
print("value of me = ", me) ;
undefined = 4 * me ;
fred = undefined + 4 ;
```

then the following output is printed on the screen.

```
value of fred =
1
value of me =
2
value of me =
12
     9: Error (examples\rdp_tut\mini_itp.str) Undeclared variable 'undefined'
    10: Error (examples\rdp_tut\mini_itp.str) Undeclared variable 'undefined'
******: Fatal - errors detected in source file
```

The parser has evaluated each statement, printing out error messages when undeclared variables are used – as specified in the semantic actions in the grammar.

In this sense the program produced by **rdp** is an *interpreter*; no executable code from the mini input statements remains when the interpreter has finished running. In the associated case study document [JS97a] the mini grammar is extended and given different semantic actions so that **rdp** generates a compiler from the grammar rather than an interpreter.

Appendix A

Acquiring and installing rdp

rdp may be fetched using anonymous ftp to ftp.dcs.rhbnc.ac.uk. If you are a Unix user download pub/rdp/rdpx_y.tar or if you are an MS-DOS user download pub/rdp/rdpx_y.zip. In each case x_y should be the highest number in the directory. You can also access the rdp distribution *via* the rdp Web page at http://www.dcs.rhbnc.ac.uk/research/languages/rdp.shmtl. If all else fails, try mailing directly to A.Johnstone@rhbnc.ac.uk and a tape or disk will be sent to you.

A.1 Installation

1. Unpack the distribution kit. You should have the files listed in Table A.1.

2. The makefile can be used with many different operating systems and compilers.

   Edit it to make sure that it is configured for your needs by uncommenting one of the blocks of macro definitions at the top of the file.

3. To build everything, go to the directory containing the makefile and type make. The default target in the makefile builds rdp, the mini_syn syntax analyser, the minicalc interpreter, the minicond interpreter, the miniloop compiler, the minitree compiler an assembler called mvmasm and its accompanying simulator mvmsim, a parser for the Pascal language and a pretty printer for ANSI-C. The tools are run on various test files. None of these should generate any errors, except for LL(1) errors caused by the mini and Pascal if statements and warnings from rdp about unused comment() rules, which are normal.

   make then builds rdp1, a machine generated version of rdp. rdp1 is then used to reproduce itself, creating a file called rdp2. The two machine generated versions are compared with each other to make sure that the bootstrap has been successful. Finally the machine generated versions are deleted.

4. If you type make clean all the object files and the machine generated rdp versions will be deleted, leaving the distribution files plus the new

`00readme.1_5`	An overview of `rdp`
`makefile`	Main `rdp` makefile
`minicalc.bnf`	`rdp` specification for the `minicalc interpreter`
`minicond.bnf`	`rdp` specification for the `minicond interpreter`
`miniloop.bnf`	`rdp` specification for the `miniloop compiler`
`minitree.bnf`	`rdp` specification for the `minitree compiler`
`mini_syn.bnf`	`rdp` specification for the `mini syntax checker`
`ml_aux.c`	`miniloop` auxiliary file
`ml_aux.h`	`miniloop` auxiliary header file
`mt_aux.c`	`minitree` auxiliary file
`mt_aux.h`	`minitree` auxiliary header file
`mvmasm.bnf`	`rdp` specification of the `mvmasm` assembler
`mvmsim.c`	source code for the `mvmsim` simulator
`mvm_aux.c`	auxiliary file for `mvmasm`
`mvm_aux.h`	auxiliary header file for `mvmasm`
`mvm_def.h`	op-code definitions for MVM
`pascal.bnf`	`rdp` specification for Pascal
`pretty_c.bnf`	`rdp` specification for the ANSI-C pretty printer
`pr_c_aux.c`	auxiliary file for `pretty_c`
`pr_c_aux.h`	auxiliary header file for `pretty_c`
`rdp.bnf`	`rdp` specification for `rdp` itself
`rdp.c`	`rdp` main source file generated from `rdp.bnf`
`rdp.exe`	32-bit `rdp` executable for Win-32 (`.zip` file only)
`rdp.h`	`rdp` main header file generated from `rdp.bnf`
`rdp_aux.c`	`rdp` auxiliary file
`rdp_aux.h`	`rdp` auxiliary header file
`rdp_gram.c`	grammar checking routines for `rdp`
`rdp_gram.h`	grammar checking routines header for `rdp`
`rdp_prnt.c`	parser printing routines for `rdp`
`rdp_prnt.h`	parser printing routines header for `rdp`
`test.c`	ANSI-C pretty printer test source file
`test.pas`	Pascal test source file
`testcalc.m`	`minicalc` test source file
`testcond.m`	`minicond` test source file
`testloop.m`	`miniloop` test source file
`testtree.m`	`minitree` test source file
`rdp_doc\rdp_case.dvi`	case study TeX dvi file
`rdp_doc\rdp_case.ps`	case study Postscript source
`rdp_doc\rdp_supp.dvi`	support manual TeX dvi file
`rdp_doc\rdp_supp.ps`	support manual Postscript source
`rdp_doc\rdp_tut.dvi`	tutorial manual TeX dvi file
`rdp_doc\rdp_tut.ps`	tutorial manual Postscript source
`rdp_doc\rdp_user.dvi`	user manual TeX dvi file
`rdp_doc\rdp_user.ps`	user manual Postscript source
`rdp_supp\arg.c`	argument handling routines
`rdp_supp\arg.h`	argument handling header
`rdp_supp\graph.c`	graph handling routines
`rdp_supp\graph.h`	graph handling header
`rdp_supp\memalloc.c`	memory management routines
`rdp_supp\memalloc.h`	memory management header
`rdp_supp\scan.c`	scanner support routines
`rdp_supp\scan.h`	scanner support header
`rdp_supp\scanner.c`	the `rdp` scanner
`rdp_supp\set.c`	set handling routines
`rdp_supp\set.h`	set handling header
`rdp_supp\symbol.c`	symbol handling routines
`rdp_supp\symbol.h`	symbol handling header
`rdp_supp\textio.c`	text buffer handling routines
`rdp_supp\textio.h`	text buffer handling header
`examples\...`	examples from manuals

Table A.1 Distribution file list

executables. If you type `make veryclean` then the directory is cleaned and the executables are also deleted.

A.2 Build log

The output of a successful makefile build on MS-DOS is shown below. Note the warning messages from `rdp` on some commands: these are quite normal.

```
        cc -Irdp_supp\ -c rdp.c
rdp.c:
        cc -Irdp_supp\ -c rdp_aux.c
rdp_aux.c:
        cc -Irdp_supp\ -c rdp_gram.c
rdp_gram.c:
        cc -Irdp_supp\ -c rdp_prnt.c
rdp_prnt.c:
        cc -Irdp_supp\ -c rdp_supp\arg.c
rdp_supp\arg.c:
        cc -Irdp_supp\ -c rdp_supp\graph.c
rdp_supp\graph.c:
        cc -Irdp_supp\ -c rdp_supp\memalloc.c
rdp_supp\memalloc.c:
        cc -Irdp_supp\ -c rdp_supp\scan.c
rdp_supp\scan.c:
        cc -Irdp_supp\ -c rdp_supp\scanner.c
rdp_supp\scanner.c:
        cc -Irdp_supp\ -c rdp_supp\set.c
rdp_supp\set.c:
        cc -Irdp_supp\ -c rdp_supp\symbol.c
rdp_supp\symbol.c:
        cc -Irdp_supp\ -c rdp_supp\textio.c
rdp_supp\textio.c:
        cc -erdp.exe rdp.obj rdp_*.obj arg.obj graph.obj memalloc.obj
           scan.obj scanner.obj set.obj symbol.obj textio.obj
        rdp -F -omini_syn mini_syn
        cc -Irdp_supp\ -c mini_syn.c
mini_syn.c:
        cc -emini_syn.exe mini_syn.obj arg.obj graph.obj memalloc.obj
           scan.obj scanner.obj set.obj symbol.obj textio.obj
        mini_syn testcalc
        rdp -F -ominicalc minicalc
        cc -Irdp_supp\ -c minicalc.c
minicalc.c:
        cc -eminicalc.exe minicalc.obj arg.obj graph.obj memalloc.obj
           scan.obj scanner.obj set.obj symbol.obj textio.obj
        minicalc testcalc
a is 7
b is 14, -b is -14
7 cubed is 343
        rdp -F -ominicond minicond
******: Error - LL(1) violation - rule
 rdp_statement_2 ::= [ 'else' _and_not statement ] .
```

```
 contains null but first and follow sets both include: 'else'
******: Warning - Grammar is not LL(1) but -F set: writing files
******: 1 error and 1 warning
        cc -Irdp_supp\ -c minicond.c
minicond.c:
        cc -eminicond.exe minicond.obj arg.obj graph.obj memalloc.obj
           scan.obj scanner.obj set.obj symbol.obj textio.obj
        minicond testcond
a is 7
b is 14, -b is -14
7 cubed is 343
z equals a
z does not equal a
        rdp -F -ominiloop miniloop
******: Error - LL(1) violation - rule
 rdp_statement_2 ::= [ 'else' statement ] .
 contains null but first and follow sets both include: 'else'
******: Warning - Grammar is not LL(1) but -F set: writing files
******: 1 error and 1 warning
        cc -Irdp_supp\ -c miniloop.c
miniloop.c:
        cc -Irdp_supp\ -c ml_aux.c
ml_aux.c:
        cc -eminiloop.exe miniloop.obj ml_aux.obj arg.obj graph.obj memalloc.obj
           scan.obj scanner.obj set.obj symbol.obj textio.obj
        rdp -F -omvmasm mvmasm
        cc -Irdp_supp\ -c mvmasm.c
mvmasm.c:
        cc -Irdp_supp\ -c mvm_aux.c
mvm_aux.c:
        cc -emvmasm.exe mvmasm.obj mvm_aux.obj arg.obj graph.obj memalloc.obj
           scan.obj scanner.obj set.obj symbol.obj textio.obj
        cc -Irdp_supp\ -c mvmsim.c
mvmsim.c:
        cc -emvmsim.exe mvmsim.obj arg.obj graph.obj memalloc.obj
           scan.obj scanner.obj set.obj symbol.obj textio.obj
        miniloop -otestloop.mvm testloop
        mvmasm -otestloop.sim testloop
******: Transfer address 00001000
        mvmsim testloop.sim
a is 7
b is 14, -b is -14
7 cubed is 343
z equals a
z does not equal a
a is 3
a is 2
a is 1
 -- Halted --
        rdp -F -ominitree minitree
******: Error - LL(1) violation - rule
 rdp_statement_2 ::= [ 'else' statement ] .
 contains null but first and follow sets both include: 'else'
```

```
******: Warning - Grammar is not LL(1) but -F set: writing files
******: 1 error and 1 warning
        cc -Irdp_supp\ -c minitree.c
minitree.c:
        cc -Irdp_supp\ -c mt_aux.c
mt_aux.c:
        cc -eminitree.exe minitree.obj m*_aux.obj arg.obj graph.obj memalloc.obj
           scan.obj scanner.obj set.obj symbol.obj textio.obj
        minitree -otesttree.mvm testtree
        mvmasm -otesttree.sim testtree
******: Transfer address 00001000
        mvmsim testtree.sim
a is 7
b is 14, -b is -14
7 cubed is 343
z equals a
z does not equal a
a is 3
a is 2
a is 1
 -- Halted --
        rdp -opascal -F pascal
******: Error - LL(1) violation - rule
 rdp_statement_9 ::= [ 'else' statement ] .
 contains null but first and follow sets both include: 'else'
******: Warning - Grammar is not LL(1) but -F set: writing files
******: 1 error and 1 warning
        cc -Irdp_supp\ -c pascal.c
pascal.c:
        cc -epascal.exe pascal.obj arg.obj graph.obj memalloc.obj
           scan.obj scanner.obj set.obj symbol.obj textio.obj
        pascal test
        rdp -opretty_c pretty_c
        cc -Irdp_supp\ -c pretty_c.c
pretty_c.c:
        cc -Irdp_supp\ -c pr_c_aux.c
pr_c_aux.c:
        cc -epretty_c.exe pretty_c.obj pr_c_aux.obj arg.obj graph.obj memalloc.obj
           scan.obj scanner.obj set.obj symbol.obj textio.obj
        pretty_c test
test.c,2133,12267,5.75
        fc test.c test.bak
Comparing files test.c and test.bak
FC: no differences encountered

        del test.bak
        rdp -F -ordp1 rdp
        cc -Irdp_supp\ -c rdp1.c
rdp1.c:
        cc -erdp1.exe rdp1.obj rdp_*.obj arg.obj graph.obj memalloc.obj
           scan.obj scanner.obj set.obj symbol.obj textio.obj
        copy rdp1.c rdp2.c
        rdp1 -F -ordp1 rdp
```

```
        fc rdp1.c rdp2.c
Comparing files rdp1.c and rdp2.c
****** rdp1.c
*
* Parser generated by RDP on Dec 20 1997 21:05:05 from rdp.bnf
*
****** rdp2.c
*
* Parser generated by RDP on Dec 20 1997 21:05:02 from rdp.bnf
*
******
```

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles techniques and tools.* Addison-Wesley, 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1 — Parsing of *Series in Automatic Computation.* Prentice-Hall Inc., 1972.

[JS97a] Adrian Johnstone and Elizabeth Scott. Designing and implementing language translators with `rdp` – a case study. Technical Report TR-97-27, Royal Holloway, University of London, Computer Science Department, December 1997.

[JS97b] Adrian Johnstone and Elizabeth Scott. `rdp` - a recursive descent compiler compiler. user manual for version 1.5. Technical Report TR-97-25, Royal Holloway, University of London, Computer Science Department, December 1997.

[JS97c] Adrian Johnstone and Elizabeth Scott. `rdp_supp` – support routines for the `rdp` compiler compiler version 1.5. Technical Report TR-97-26, Royal Holloway, University of London, Computer Science Department, December 1997.