# rdp – a recursive descent compiler compiler

# User manual for version 1.5

Adrian Johnstone Elizabeth Scott

Technical Report

CSD−TR−97−25

December 20, 1997

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

## Abstract

rdp is a system for implementing language processors. Compilers, assemblers and interpreters may all be specified in the rdp source language (an extended Backus-Naur Form with support for inherited and synthesized attributes which may be accessed from within C-language semantic actions). These specifications may then be processed by the rdp command to produce a program written in ANSI C which can be compiled by any ANSI standard C compiler. It is possible to use rdp write, for example, compilers (by using the semantic actions to specify the corresponding target code), interpreters (by using the semantic actions to evaluate input fragments) and pretty printers (by using the actions to reformat the input fragments).

This report describes the rdp source language, command switches and error messages. Serious usage of rdp-generated parsers requires an understanding of the support library rdp_supp which is documented in a companion report [JS97b]. A third, tutorial, report assumes no knowledge of parsing, grammars or language design and shows how to use rdp to develop a small calculator-like language [JS97c]. The emphasis in the tutorial guide is on learning to use the basic rdp features and command line options. A large case study is documented in [JS97a] which extends the language described in the tutorial guide with details of a syntax checker, an interpreter and a compiler along with an assembler and simulator for a synthetic architecture which is used as the compiler target machine.

The rdp source code is public domain and has been successfully built using Borland C++ version 3.1 and Microsoft C++ version 7 on MS-DOS, Borland C++ version 5.1 on Windows-95, GNU gcc and g++ running on OSF/1, Ultrix, MS-DOS, Linux and SunOS, and Sun's own acc running on Solaris. Users have also reported straightforward ports to the Amiga, Macintosh and Archimedes systems.



This document is © Adrian Johnstone and Elizabeth Scott 1997.

Permission is given to freely distribute this document electronically and on paper. You may not change this document or incorporate parts of it in other documents: it must be distributed intact.

The rdp system itself is © Adrian Johnstone but may be freely copied and modified on condition that details of the modifications are sent to the copyright holder with permission to include such modifications in future versions and to discuss them (with acknowledgement) in future publications.



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


**1 Introduction** **1**

1.1 Specifying languages to `rdp` 1
1.2 Acceptable languages 2
1.3 Support modules 3
1.4 System flow 3

**2 IBNF – the `rdp` source language** **7**

2.1 Grammars and BNF 7
2.2 Layout and comments 9
2.3 Identifiers 9
2.4 Grammar rule format 10

**3 IBNF extended forms** **11**

3.1 Sequences 11
3.2 Alternate productions 11
3.3 Recursion 12
3.4 Do-first 12
3.5 Zero-or-one occurrences (optional sub-productions) 12
3.6 Zero-or-many occurrences (Kleene closure) 13
3.7 One-or-many occurrences (positive closure) 13
3.8 The iterator operator `@` 13
3.9 Using iterators to implement lists 14

**4 Scanner elements** **15**

4.1 Introduction 15
Keyword tokens 16
`ID` token 16
`INTEGER` token 16
`REAL` token 17
`STRING(`*open*`)` token 17
`STRING_ESC (`*open escape*`)` token 17
`CHAR(`*open*`)` token 17
`CHAR_ESC (`*open escape*`)` token 18
`EOLN` token 18
4.2 Describing comments to the scanner 18

**5 Attributes and semantic actions** **21**
5.1 An introductory example 21
5.2 Synthesized attribute definition 22
5.3 Synthesized attribute types for scanner primitives 23
5.4 Using synthesized attributes 23
5.5 Inherited attribute definition 24
5.6 Semantic actions 24
5.7 Error checking of semantic actions 25
5.8 Default actions for iterators that match the empty string 25
5.9 Semantic rules 25
5.10 Semantic actions in multi-pass parsers 26

**6 Directives** **29**
6.1 Flow control directives 29
`INCLUDE("`*filename*`")` 29
6.2 Parser setup directives 29
`USES("`*filename*`")` 29
`TITLE("`*string*`")` 30
`SUFFIX("`*string*`")` 30
`PARSER(`*start*`)` 30
`PRE_PARSE([*` *action* `*])` 30
`POST_PARSE([*` *action* `*])` 30
`OUTPUT_FILE("`*file*`")` 30
`PASSES(`*count*`)` 30
`SYMBOL_TABLE(`*name size prime compare hash print* `[*` *data* `*])` 31
6.3 Command line argument definition directives 32
`ARG_BOOLEAN(`*key identifier key_string*`)` 32
`ARG_NUMERIC(`*key identifier key_string*`)` 32
`ARG_STRING(`*key identifier key_string*`)` 33
`ARG_BLANK(`*key_string*`)` 33
6.4 Command line default directives 33
`TAB_WIDTH(`*count*`)` 33
`TEXT_SIZE(`*count*`)` 33
`MAX_ERRORS(`*count*`)` 33
`MAX_WARNINGS(`*count*`)` 34
6.5 Scanner directives 34
`CASE_INSENSITIVE` 34
`SHOW_SKIPS` 34
6.6 Tree generation directives 34
`TREE([*` *data* `*])` 34
`EPSILON_TREE([*` *data* `*])` 34
`ANNOTATED_EPSILON_TREE([*` *data* `*])` 35

**7 Running `rdp`** **37**
7.1 `rdp` filename parameters 37
7.2 `rdp` option parameters 37
`-e` write out expanded IBNF 37

-E add rule name to error messages 38
-f filter mode 38
-F force creation of output files 38
-l make a listing 38
-ofilename write output to filename 39
-p make parser only 39
-R add rule entry and exit messages 39
-s echo each scanner symbol as it is read 39
-S print summary symbol table statistics 39
-tn tab expansion width 39
-Tn text buffer size in bytes for scanner 40
-v set verbose mode 40
-V dump derivation tree in VCG format 40
7.3 Options understood by rdp-generated parsers 41

**8 rdp global variables 43**
8.1 Monitoring parser status at runtime 43
rdp_error_return 43
rdp_outputfilename 43
rdp_pass 43
rdp_sourcefilename 43
rdp_tree 43
rdp_verbose 44
8.2 Defining the message stream 44
8.3 Adding reserved words to the dangerous identifier list 44

**9 Derivation tree construction and visualisation 45**
9.1 Derivation trees 46
9.1.1 A larger example 47
9.2 Tree generation directives 51
TREE([* *data* *]) 51
EPSILON_TREE([* *data* *]) 51
ANNOTATED_EPSILON_TREE([* *data* *]) 51
9.3 Using VCG to visualise derivation trees 51

**10 Tree manipulation 55**
10.1 Normal tree construction 55
10.2 Modifying tree construction with promotion operators 56
10.2.1 Promote underneath parent 56
10.2.2 Promote on top of parent 56
10.2.3 Promote above parent 56
10.2.4 Insert here (no promotion) 57
10.3 Valid contexts for promotion operators 57
10.4 A complete example 58
10.4.1 Removing syntactic sugar 58
10.4.2 Making operators parent nodes to their operands 58
10.4.3 Handling left associative operators 61

**11 Error and informational messages** **63**
11.1 Fatal errors 64
11.2 Errors 65
11.3 Warnings 69
11.4 Informational messages 69

**12 Understanding and debugging rdp-generated parsers** **73**
12.1 The header file 73
12.2 The **rdp** scanner 73
12.2.1 The token enumeration 74
12.2.2 Interaction between the scanner and the parser 75
12.3 The main file 76
12.3.1 Implementing parser functions 76
12.4 Selecting alternate productions 79
12.5 Parsing iterators 80
12.6 Debugging **rdp**-generated parsers 81
12.7 Errors reported by **rdp** when parsing a specification 81
12.8 LL(1) errors reported by **rdp** during the analysis phase 82
12.9 Refining a grammar 83
12.10Debugging semantic actions 84

**A Acquiring and installing rdp** **87**
A.1 Installation 87
A.2 Build log 89

# List of Figures

1.1 rdp design flow 4

2.1 rdp syntax 8

9.1 A simple derivation tree 47
9.2 Derivation tree for expression grammar 49
9.3 A modified derivation tree 50
9.4 Effect of the TREE directive 51
9.5 Effect of the EPSILON_TREE directive 52
9.6 Effect of the ANNOTATED_EPSILON_TREE directive 52

10.1 Simple expression: full tree 59
10.2 Simple expression: result of adding promotion operators 60
10.3 Right associative operator tree 61
10.4 Left associative operator tree 62

12.1 Extracts from an rdp-generated header file 74
12.2 Extracts from an rdp-generated parser main file 77

# List of Tables


2.1 Reserved identifier prefixes 10

3.1 Iterator:bracket correspondences 14

4.1 `rdp` character and string escape sequences 18

5.1 Scanner primitive attribute types 23

7.1 Standard command line options 40

A.1 Distribution file list 88

# Chapter 1

# Introduction

rdp is a system for implementing language processors. Compilers, assemblers and interpreters may all be specified in the rdp source language (an extended Backus-Naur Form featuring *iterators* along with support for inherited and synthesized attributes which may be accessed from within C-language semantic actions). These specifications may then be processed by the rdp command to produce a program written in ANSI C which can be compiled by any ANSI standard C compiler. It is possible to use rdp write, for example, compilers (by using the semantic actions to specify the corresponding target code), interpreters (by using the semantic actions to evaluate input fragments) and pretty printers (by using the actions to reformat the input fragments).

This report describes the rdp source language, command switches and error messages. Serious usage of rdp-generated parsers requires an understanding of the support library rdp_supp which is documented in a companion report [JS97b]. A third, tutorial, report assumes no knowledge of parsing, grammars or language design and shows how to use rdp to develop a small calculator-like language [JS97c]. The emphasis in the tutorial guide is on learning to use the basic rdp features and command line options. A large case study is documented in [JS97a] which extends the language described in the tutorial guide with details of a syntax checker, an interpreter and a compiler along with an assembler and simulator for a synthetic architecture which is used as the compiler target machine.

The rdp source code is public domain and has been successfully built using Borland C++ version 3.1 and Microsoft C++ version 7 on MS-DOS, Borland C++ version 5.1 on Windows-95, GNU gcc and g++ running on OSF/1, Ultrix, MS-DOS, Linux and SunOS, and Sun's own acc running on Solaris. Users have also reported straightforward ports to the Amiga, Macintosh and Archimedes systems.

## 1.1 Specifying languages to rdp

Parser generator tools like rdp usually work from specifications written using variant of the *generative grammar* formalism which was introduced in the 1950's by Chomsky. The formalism was first applied to the development of program-

ming languages by John Backus and Peter Naur in the design of Algol-60 and the notation used is called Backus-Naur Form (or BNF) [Bac60] in commemoration of that pioneering work. For technical reasons, programming languages and other synthetic computer languages rarely use the full power of generative grammars but instead are based on a restricted kind of generative grammar called a *context free* grammar.

BNF is sufficient to describe context free grammars, but most real tools add extensions to the basic notion which might be thought of as shorthands for commonly occurring BNF idioms. These *extended* BNF's come in several varieties, but the basic idea is to combine the notion of *regular expressions* with the core BNF notation in such a way as to provide compact ways of specifying repetition within the strings of a language. This allows language specifications to be smaller, and may allow the parser generator to easily exploit structure within the rules so as to improve efficiency.

The particular extended BNF used by `rdp` is called *Iterator Backus-Naur Form* or IBNF. An iterator is a generalisation of the kinds of regular expressions found in other forms of extended BNF. In addition to supporting the traditional BNF extensions, iterators allow *list-like* structures within languages to be conveniently specified.

You can find an introduction to the basic notions of generative grammars, BNF and IBNF in the first chapters of the `rdp` tutorial manual [JS97c]. A full description of the capabilities of IBNF is given below.

## 1.2 Acceptable languages

`rdp` generates parsers that work using recursive descent. It requires grammars to be LL(1) (or something very close to LL(1)), that is they must be unambiguously parsable using a single token of lookahead and there must be no left recursive rules. This is not a significant constraint for modern languages like Ada, Pascal and C which were to some extent designed with a view to easy parsing. On the other hand, some languages are just too difficult — you should probably forget all about using `rdp` to write a FORTRAN parser, for instance.

The main advantages of recursive descent parsing are

- fast (linear time) parsing,
- the availability of standardised error recovery mechanisms and
- the straightforward one-to-one relationship between the code in the parser and the rules in the grammar specification. This makes it straightforward to debug the grammar, because a C language debugger may be used to step through the parser functions and, equivalently, to step through the grammar rules.

If the language you present to `rdp` is not LL(1), `rdp` issues detailed diagnostics explaining which tokens and rules are giving the trouble. It is possible to write algorithms that translate certain non-LL(1) grammars automatically to

LL(1) form, but rdp does not attempt to perform any such transformations because that might produce an obscure parser that was no longer directly related to the input grammar.

rdp is itself a language processor, and the rdp source language has a grammar that is almost LL(1). In fact rdp is 'written in itself'—an early version of the system was hand written and later versions developed from a grammar description written in rdp's own source language. This process (developing new versions of a tool by writing each new version in the language acceptable to the old version of the tool) is called *bootstrapping* and is a common technique in compiler development.

## 1.3 Support modules

rdp-generated parsers use a set of general purpose support modules collectively known as rdp_supp. There are seven parts to rdp_supp:

- a hash coded symbol table handler which allows multiple tables to be managed with arbitrary user data fields (symbol.c),
- a set handler which supports dynamically resizable sets (set.c),
- a graph manager which allows arbitrary directed graphs to be constructed and manipulated, with a facility to output any graph in a form that may be read and visualised by the VCG [San95] tool on Windows and Unix/X-windows systems (graph.c),
- a memory manager which wraps fatal error handling around the standard ANSI C heap allocation routines (memalloc.c),
- a text handler which provides line buffering and string management without imposing arbitrary limits on input line length (textio.c),
- a command line argument parsing package that allows Unix style options to be implemented in a standardised way (arg.c),
- scanner support routines for handling tokens in recursive descent parsers (scan.c and scanner.c).

Writing effective language processors in rdp requires a detailed understanding of these modules. The rdp_supp routines are documented in [JS97b].

## 1.4 System flow

The steps involved in producing a new language processor for a mythical new language myth using the rdp parser generator are

1. Create a file myth.bnf containing an Iterator Backus-Naur Form (IBNF) specification of the myth language. Decorate the grammar with attributes and C-language fragments describing semantic actions. By convention, large semantic routines are kept in a file called the *auxiliary* file with a name like myth_aux.c.

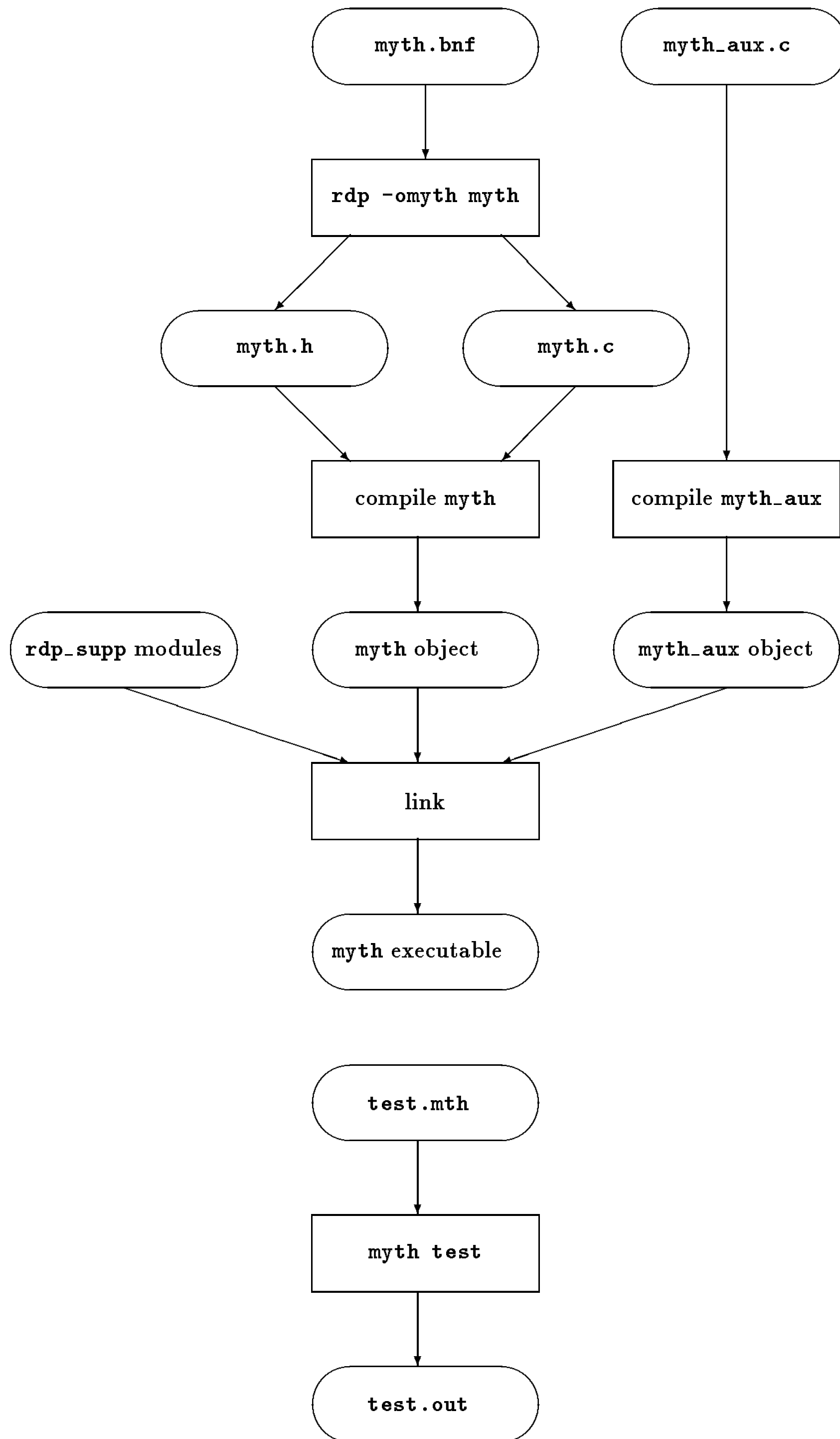


**Figure 1.1** rdp design flow

2. Process **myth.bnf** with **rdp** to produce **myth.c**, the **rdp**-generated parser for our language **myth**.

3. Compile **myth.c** and **myth_aux.c** with an ANSI C compiler.

4. Link the **myth** object file with the **rdp_supp** modules and any other required semantic routines.

5. Run the resulting executable on **test.mth**, a test program for the **myth** language.

This process is illustrated in Figure 1.1.

A complete sequence of commands to generate and build an **rdp** based parser using Unix commands might be

```
edit myth.bnf          Create BNF file
edit myth_aux.c        Create auxiliary file (optional)
rdp -omyth myth        Use rdp to generate the parser
cc -c myth.c           Compile the generated C parser
cc -c myth_aux.c       Compile the auxiliary file
cc -omyth myth.o myth_aux.o arg.o graph.o memalloc.o scan.o
scanner.o set.o symbol.o textio.o  Link object files
edit test.mth          Create a test file contaning a myth program
myth test.mth          Run the executable parser on the test file
```

The following chapters include reference documentation on **rdp** command line parameters, a full description of the **rdp** iterator BNF source language, a discussion of the use of derivation trees with the VCG tool, and a summary of all **rdp** error messages. Any serious use of **rdp** will require familiarity with the support library which is described in a companion report [JS97b]. Extensive tutorial documentation, suitable for both novice and expert users, will be found in the tutorial guide [JS97c] and in the case study documented in [JS97a].

# Chapter 2

# IBNF – the `rdp` source language

This chapter introduces the basic syntax of Iterator Backus-Naur Form (IBNF) which is the language in which **rdp** specifications are written. Figure 2.1 shows the syntax of the **rdp** source language written in IBNF. An **rdp** source file is a collection of rules and directives. Directives are used to parameterise the grammar, often by setting default values for command line switches. Rules are IBNF grammar rules describing the syntax of the target language. Below we describe basic BNF. In the next chapter we describe the extended forms which are accepted by **rdp**.

## 2.1 Grammars and BNF

It is standard practice to use formal grammars to specify computer languages. We give a very brief summary of the notation here. You can find more detail in the tutorial manual [JS97c] and in standard texts on compiler design.

A grammar consists of a set **N** of *non-terminals*, a set **T** of *tokens*, and a set $\mathcal{P}$ of *grammar rules*. Non-terminals are written as strings which must start with an alphabetic character or an underscore, and may then continue with alphabetic, numeric or underscore characters. Tokens are written as singly quoted strings.

Each grammar rule is of the form

**rule_name** `::=` *rule expression.*

where **rule_name** is a non-terminal and the *rule expression* is a collection of sequences of grammar symbols. These sequences are called *alternate productions* of the grammar rule.

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

In basic BNF the rule expression is described by writing out the sequences and separating them using a vertical bar (`|`). IBNF also allows certain sets of

```
unit ::= { rule | dir}.

dir ::= 'ARG_BOOLEAN' '(' ID ( ID | code ) String ')'  |
        'ARG_NUMERIC' '(' ID ( ID | code ) String ')'  |
        'ARG_STRING'  '(' ID ( ID | code ) String ')'  |
        'ARG_BLANK'   '(' String ')'                  |
        'CASE_INSENSITIVE'  |
        'INCLUDE' '(' code ')' |
        'OUTPUT_FILE' '(' String  ')' |
        'MAX_ERRORS' '(' INTEGER  ')' |
        'MAX_WARNINGS' '(' INTEGER  ')' |
        'PARSER' '(' ID  ')' |
        'PASSES' '(' INTEGER  ')' |
        'POST_PARSE' '(' code  ')' |
        'PRE_PARSE' '(' code  ')' |
        'SUFFIX' '(' String  ')' |
        'SHOW_SKIPS'  |
        'SYMBOL_TABLE' '(' ID INTEGER INTEGER ID ID ID code ')' |
        'TITLE' '(' String  ')' |
        'TAB_WIDTH' '(' INTEGER  ')' |
        'TEXT_SIZE' '(' INTEGER  ')' |
        'USES' '(' String ')'.


rule ::= ID ['(' { ID [':' ID {'*'} ] } ')' ] [':' ID {'*'} ] [ '!' ] '::=' prod '.' .

prod ::= ( seq )@'|' .

seq ::= < (item_ret [':' ID] | item_inl) [ '!' ] > .

item_ret ::= ID '(' { ( INTEGER | REAL | String | ID ) } ')' |
             token  |
     'CHAR' '(' token ')'  |
     'CHAR_ESC' '(' token token ')'  |
     'STRING' '(' token ')'  |
     'STRING_ESC' '(' token token ')'  |
     'COMMENT' '(' token token ')'  |
     'COMMENT_VISIBLE' '(' token token ')'  |
     'COMMENT_NEST' '(' token token ')'  |
     'COMMENT_NEST_VISIBLE' '(' token token ')'  |
     'COMMENT_LINE' '(' token ')'  |
     'COMMENT_LINE_VISIBLE' '(' token ')'.


item_inl ::= code [ '@' INTEGER  ] |
     '('prod')' [ '@' [ INTEGER [ '..' INTEGER ] ] ( token | '#' ) ] |
     '{'prod'}' (* Kleene closure *) |
     '['prod']' (* Optional *) |
     '<'prod'>' (* Positive closure *).

token   ::= STRING_ESC('\'' '\\').
String  ::= STRING_ESC('"' '\\').
code    ::= COMMENT_VISIBLE('[*' '*]').
comment ::= COMMENT('(*' '*)').
```

**Figure 2.1** rdp syntax

alternates to be represented using regular expression like structures. We shall discuss the details of these IBNF expressions in Chapter 3.

We *derive* one sequence of grammar symbols from another by replacing a non-terminal (rule name) with a sequence from the right hand side of its grammar rule. For example, given the grammar

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

we can derive `'a''b'` from `'a'start` by replacing `start` with `'b'`. We write

$$\texttt{'a' start} \Rightarrow \texttt{'a' 'b'}$$

We can perform a series of derivations one after the other:

$$\texttt{start} \Rightarrow \texttt{'a' start} \Rightarrow \texttt{'a' 'a' start} \Rightarrow \texttt{'a' 'a' 'b'}$$

In this case we write

$$\texttt{start} \stackrel{*}{\Rightarrow} \texttt{'a' 'a' 'b'}.$$

The *language generated by the grammar* is the set of sequences of tokens which can be derived from the start symbol.

In the rest of this chapter and in Chapter 3 we shall describe the detailed capabilities of IBNF.

## 2.2 Layout and comments

The `rdp` IBNF source language is *free-format*, that is whitespace and newlines can be used anywhere between lexemes to provide a neat layout.

It is possible to insert comments into an `rdp` IBNF source file. IBNF comments are delimited by `(*    *)` brackets and may appear in any position that white space is legal. Comments may be nested: the maximum nesting level is limited only by available memory. For example,

```
S ::= S '+' S | (* Sum *)
      S '*' S | (* Product *)
      E .       (* Constant terms *)

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

## 2.3 Identifiers

Tokens are singly quoted strings and rule names are identifiers in the `rdp` source language. User defined `rdp` identifiers must start with an alphabetic character or an underscore (_) and can contain only alphanumeric characters or underscores. In general, `rdp` identifiers must obey the rules for ANSI C identifiers.

Internally, `rdp` uses many identifiers and it would be catastrophic if a user defined, say, a rule name that clashed with an internal library routine's name. To stop this happening `rdp` reserves several prefixes which may not be used

| arg_ |
|---|
| graph_ |
| mem_ |
| rdp_ |
| scan_ |
| set_ |
| symbol_ |
| text_ |

**Table 2.1** Reserved identifier prefixes

to start an identifier. The list of reserved prefixes is shown in Table 2.1: they correspond to the function name prefixes used in the support library.

As well as checking for reserved prefixes, `rdp` also checks user defined identifiers against a list of C keywords and library names to ensure that clashes do not occur at compile time: defining a production called `printf` for instance would cause the library `printf()` routine to become invisible with very confusing results. You can add names to this list by modifying the definition of `RDP_RESERVED_WORDS` in file `rdp_supp/rdp_aux.h`.

`rdp` itself sets no limit on the length of identifiers subject to there still being room left in the text buffer, but note that many C compilers only recognise the first 32 characters of an identifier as significant and some linkers only recognise the first eight characters. In some contexts `rdp` can generate identifiers that are extensions of a user-defined identifier, so it might be wise to keep your identifiers to less than 20 characters in length. `rdp` is case sensitive, so `Adrian` is a different identifier to `adrian`.

## 2.4 Grammar rule format

Each grammar rule must only be defined once, in other words there can only be one rule for each rule name.

Any non-terminal in a sequence on the right hand side of a rule must also appear on the left hand side of some rule, i.e. must be a rule name. However, forward references are allowed (that is, rule names may appear on the right hand side of a rule before they appear on the left hand side). Such forward references are resolved using a two pass parser.

Every rule must terminate with a full stop (period). Neither empty rules nor empty alternate productions are allowed: there are special constructions for describing rules that expand to the empty string and these are described in Chapter 3.

# Chapter 3

# IBNF extended forms

In this chapter we shall describe the full generality of grammar rules which can be written in `rdp`-IBNF (the `rdp` source language).

## 3.1 Sequences

A grammar rule might have a single sequence on its left hand side. A sequence is simply the concatenation of the tokens and rule names which make up that sequence:

```
seq1 ::= seq2 seq2 'z'.
seq2 ::= 'a' 'b' 'c'.
```

These two productions define a small language that will be recognised by the corresponding `rdp`-generated parser. The complete language comprises the following set which contains just one string:

{ `abcabcz` }

## 3.2 Alternate productions

Alternative sequences on the right hand side of a rule are written out separated by vertical bars (|) which we call the *alternation operator*. Alternation has lower priority than sequencing, so the rule

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

describes the sequences {`ab, cd`}, *not* the sequences {`abd, acd`}.

The alternation operator can be used to separate any type of rule expression. For example, we can write

```
rule ::= 'a' {rule 'b'} | [ 'c' ].
```

(The constructs {} and `[]` are described below.)

For a grammar to be acceptable to `rdp`, no two alternates in the same rule can generate sequences that begin with the same token. For example, if

```
rule_name ::= ... | alternate1 | ... | alternate2 | ...  .
```

then we cannot have

$$\texttt{alternate1} \stackrel{*}{\Rightarrow} \texttt{'a' string1}, \quad \texttt{alternate2} \stackrel{*}{\Rightarrow} \texttt{'a' string2}$$

for any token `'a'`. This is because `rdp`-generated parsers may only look one token ahead, and must be able to make a deterministic choice of alternates on this basis. `rdp` will issue an LL(1) error if it cannot disambiguate alternate productions.

Hence,

```
seq1 ::= 'a' 'b' | 'a' 'c'.
```

is not legal, but

```
seq2 ::= 'a' seq3.
seq3 ::=  'b' | 'c'.
```

specifies the same language and is acceptable to `rdp`.

## 3.3 Recursion

`rdp` allows most directly recursive and indirectly recursive rules, but *left recursive* rules of either type generate LL(1) errors. This is because a left recursive rule will generate a similarly left recursive set of function calls, which will never terminate, so

```
rec1 ::= 'a' rec1 | 'b'.
```

is legal, but

```
rec1 ::= rec1 'a' | 'b'.
```

is not.

## 3.4 Do-first

Parentheses `(...)` may be used to override the relative priority of alternation and sequencing, so the language {`abd`, `acd`} may be described with

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

In fact any the contents of any `(...)`, `[...]`, `{...}` or `<...>` bracket pair is evaluated immediately, that is all bracket pairs have maximum priority.

## 3.5 Zero-or-one occurrences (optional sub-productions)

Optional parts of a grammar are enclosed in square brackets `[...]`. The set of tokens that may appear first in an optional phrase must not include any tokens that can appear immediately after an optional phrase. This is because `rdp`-generated parsers may only look one token ahead. If there is a token that could be both the first in the optional phrase *and* be the first token of the phrase after the optional phrase then the parser will not know which rule to follow. `rdp` will issue an error in this case.

## 3.6 Zero-or-many occurrences (Kleene closure)

Iteration may be directly represented (without using recursion) by curly braces `{...}` which is a shorthand for 'zero or many' occurrences of the iterator body. The set of tokens that may appear first in a string derived from the repeat construct must not include any tokens that can appear immediately after that repeat construct. This is because `rdp`-generated parsers may only look one token ahead. If there is a token that could be both the first token in a string derived from the repeat construct *and* be the first token of a string derived from the sequence following that repeat construct then the parser will not know which rule to follow. `rdp` will issue an error in this case.

## 3.7 One-or-many occurrences (positive closure)

Angle brackets <...> form a shorthand for 'one or many' occurrences of the iterator body. The set of tokens that may appear first in a string derived from the repeat construct must not include any tokens that can appear immediately after that repeat construct. This is because `rdp`-generated parsers may only look one token ahead. If there is a token that could be both the first token in a string derived from the repeat construct *and* be the first token of a string derived from the sequence following that repeat construct then the parser will not know which rule to follow. `rdp` will issue an error in this case.

## 3.8 The iterator operator `@`

`rdp` provides a generalised iterator operator which subsumes the standard extended BNF brackets described above. The construction

```
( 'body' ) 2 @ 4 'separator'
```

matches the following strings

```
body separator body
body separator body separator body
body separator body separator body separator body
```

that is, between two and four instances of `body` separated by the token `separator`. The general form of the iterator is

( *valid subproduction* ) *lo* @ *hi token*

This specifies that the `rdp`-generated parser should match the body represented by *valid subproduction* between *lo* and *high* times interspersing each instance with one instance of the separating *token*. A *hi* value of zero means 'without limit', that is the iteration will continue arbitrarily.

Either, or both, of *hi* and *lo* may be absent in which case they default to zero. The separating *token* may be set to the special token `#` which means 'nothing' or the empty string (sometimes represented by $\epsilon$). In this case no separating token is looked for.

```
( ... )  →  ( ... ) 1@1 #
< ... >  →  ( ... ) 1@0 #
[ ... ]  →  ( ... ) 0@1 #
{ ... }  →  ( ... ) 0@0 #
```

**Table 3.1** Iterator:bracket correspondences

The IBNF brackets described in the previous sections are in fact just shorthands for special cases of the iterator construct. The correspondences are shown in Table 3.1. None of them carries a separating token, and all of them have lower bounds of zero or one and upper bounds of one or zero (*without limit*).

## 3.9 Using iterators to implement lists

Delimited lists are common in high level languages. Consider, for instance, a function call in C:

```
func(param1, param2, param3)
```

In general, parameter lists are comma-delimited lists of identifiers. If we have an `rdp` rule `ID` which matches a C-style identifier, one way of writing an `rdp` specification of a function call is:

```
func_call ::= ID '(' param_list ')'.
param_list ::= [ ID param_tail ].
param_tail ::=  [ ',' ID param_tail ].
```

This uses recursion to match an arbitrary number of parameters. We can use the `{ ... }` iterator brackets and give a more compact description:

```
func_call ::= ID '(' param_list ')'.
param_list ::= [ ID {',' ID } ].
```

Here the recursion has been replaced by iteration.

Using the iterator operator with the optional delimiter token we can further compact this to

```
func_call ::= ID '(' param_list ')'.
param_list ::= [ ID @ ',' ].
```

or just

```
func_call ::= ID '(' [ ID @ ',' ] ')'.
```

# Chapter 4

# Scanner elements

## 4.1 Introduction

Under the traditional model of compilation parsers operate on the grammar of a language, and a separate *scanner* or *lexical analyser* reads the input characters and groups them into tokens which are then passed on to the parser for processing.

It is possible to build the scanning phase into the parser by specifying the language down to character level in the grammar. In practice this is both tedious and difficult because the grammar specification required may not satisfy the LL(1) requirements. Another traditional role of the scanner is to remove comments from the input file. It is also possible to remove comments at parser level by adding appropriate rules to the grammar, but this is both tedious and inefficient.

`rdp` does not use a scanner generator, relying instead on a hard wired scanner when it reads an IBNF input file. This same hard wired scanner is automatically attached to the front end of `rdp`-generated parsers and the user can access this scanner by putting certain built in token names and parser directives into their IBNF grammars. In addition, the inclusion of the scanner allows the generated parsers to efficiently perform buffering and error reporting on their source files.

Although the scanner is essentially pre-defined, aspects of it are paramaterisable, and it can be made to handle the constructs in most high level languages. For example, it is possible to define a language in which strings are enclosed in single quotes (Pascal style) and to define a language in which strings are enclosed in double quotes (C style). This is because, as we shall see below, the hard wired token which matches strings takes a parameter which is the string delimiter.

The scanner also has primitives which allow commenting styles to be specified in a language. These primitives are paramaterisable to allow, for example, Pascal like comments, which are enclosed in braces, and C++ like comments, which begin `//` and terminate at the end of the line.

We now give a summary of the types of hard wired tokens available in the scanner, and the tokens and strings (lexemes) that they match, along with the family of comment definition primitives.

- simple character sequences such as `'fred'` are tokens and they match the corresponding lexeme or string (*fred* in this case),
- the token `ID` matches C-style identifiers such as `adr123` and `_temp`,
- the token `INTEGER` matches C-style integer literals such as `145` and `0xFE` (a hexadecimal integer),
- the token `REAL` matches C-style real literals such as `1.45`, `1.`, `1.45e3` and `1.45E-02`,
- the token `STRING('\'')` matches Pascal style strings where two adjacent quotes are read as a quote mark in the string, as in `'Adrian''s book'`,
- the token `STRING_ESC('"' '\\')` matches C-style strings where as *escape character* is used to introduce quote marks and other special characters into the string, as in `"\"Good\" she said"`,
- the token `EOLN` matches the end of line marker,
- a variety of comment styles are supported including both nestable and non-nestable comment brackets along with comments that start with a token and terminate at the end of line.

### Keyword tokens

A keyword token is any character string delimited by single quotes, such as `'>='` `'while'` and `'++'`. This token matches just the string itself, so `'while'` matches the string of letters *while*. Just as in a C string, the backslash character can be used as an escape character so that the token `'` is represented as `'\''`. Empty tokens (`''`) and tokens containing non-printing characters such as `'long int'` and `'bad_code\8'` are illegal.

In the following sections we describe hard wired tokens which can be used in an input grammar for `rdp`.

### `ID` token

The token `ID` matches any string comprising an alphabetic character (`a..z` and `A..Z`) or an underscore (`_`) followed by any number of alphabetic, numeric or underscore characters, such as `temp`, `temp123` or `_temp_123`.

### `INTEGER` token

The token `INTEGER` matches any valid C-style integer such as `145` and `0xFE` (a hexadecimal integer). In an extension to standard C-style integers, you may insert underscore characters within a numeric literal so as to improve readability: hence `999_999_999` is valid. The scanner builds a `long unsigned` value from the digits.

### `REAL` token

The token `REAL` matches any valid C-style real literal, such as `1.45`, `1.`, `1.45e3` and `1.45E-02`. In an extension to standard C-style integers, you may insert underscore characters within a numeric literal so as to improve readability: hence `999_999.999` is valid. The scanner builds a `double` value from the digits.

### `STRING(`*open*`)` token

Simple strings are specified with a single token which marks both the opening and closing quote. Two consecutive quotes are used to represent an embedded quote character.

If there exists a rule like

```
string1 ::= STRING('\'').
```

then the scanner will look for Pascal-style strings delimited by single quotes, such as `'adrian'` which returns *adrian* and `'adrian''s book'` which returns *adrian's book*. Similarly

```
string2 ::= STRING('|').
```

accepts strings such as `|adrian|` and `|the symbol || is used for alternation|` which returns *the symbol | is used for alternation*.

### `STRING_ESC(`*open escape*`)` token

There is no way to directly represent control characters with a simple `STRING` primitive. `rdp` *escaped strings* support the full range of ANSI standard C escape sequences although trigraph sequences are not available. The specification of the string includes an opening token and the escape token, so that strings in C can be recognised with

```
stringc ::= STRING_ESC('"' '\\').
```

Table 4.1 shows the special escape sequences. In addition, any combination $\backslash y$, where $y$ is a character not shown in Table 4.1, is replaced by the character $y$[1]. For octal numbers there must be exactly 1, 2 or 3 digits. For hex numbers any sequence of valid hexadecimal digits will be accepted regardless of length. Both upper and lower case hex digits will be accepted.

### `CHAR(`*open*`)` token

Single characters are specified with one token which marks both the opening and closing quote. Two consecutive quotes are used to represent an embedded quote character.

If there exists a rule like

```
string1 ::= CHAR('\'').
```

then the scanner will look for Pascal-style character literals delimited by single quotes, such as `'a'` which returns the single character *a*.

[1] Note that in ANSI standard C the result of such an escape sequence is undefined.

| Escape sequence | Replacement character | Name |
|---|---|---|
| `\a` | BEL | alert |
| `\b` | BS | backspace |
| `\f` | FF | form feed |
| `\n` | NL | newline |
| `\r` | CR | carriage return |
| `\t` | HT | horizontal tab |
| `\v` | VT | vertical tab |
| `\\` | \ | backslash |
| `\"` | " | double quote |
| `\`*ooo* | | character with octal code *ooo* |
| `\x`*hh* | | character with hex code *hh* |

**Table 4.1** `rdp` character and string escape sequences

### `CHAR_ESC` (*open escape*) **token**

There is no way to directly represent control characters within a simple `CHAR` primitive. `rdp` *escaped character literals* support the full range of ANSI standard C escape sequences although trigraph sequences are not available. The specification of the literal includes an opening token and the escape token so that, for instance, C-language character literals can be recognised with

```
stringc ::= CHAR_ESC('\'' '\\').
```

Table 4.1 shows the special escape sequences. Any combination \\$y$, where $y$ is a character not shown in Table 4.1, is replaced by the character $y$[2]. For octal numbers there must be exactly 1, 2 or 3 digits. For hex numbers any sequence of valid hexadecimal digits will be accepted regardless of length. Both upper and lower case hex digits will be accepted.

### `EOLN` **token**

`EOLN` matches a newline marker. If you do not use the `EOLN` primitive anywhere in your grammar, then newlines are suppressed and treated as whitespace by the scanner.

## 4.2 Describing comments to the scanner

Most programming languages include a commenting facility. If `rdp` is to generate a parser for a language which has a commenting facility then the commenting style required must be specified in the language grammar. In this section

[2]Note that in ANSI standard C the result of such an escape sequence is undefined.

we shall describe the primitives which allow the built in scanner which is included in an `rdp`-generated parser to detect and remove comments. However, to motivate the definitions of these primitives, we first give a brief discussion on various commenting conventions.

Specifying comments in grammars is a rather tricky area. The usual practice is to allow comments wherever whitespace is legal in the language, and to suppress comments in the scanner so that they are not visible in the phrase level grammar. If comments are to be visible in the phrase level grammar then there either there must be a call to a comment rule after every token in the phrase level grammar or else comments must be restricted to certain contexts. This latter option was tried early on in the development of free-format languages (for instance in Algol-60) but was soon found to introduce inconvenient and unnecessary restrictions on program layout.

There are several varieties of commenting conventions in use. The most common in block structured languages is to use opening and closing comment brackets such as `{...}` or the alternate form `(*...*)` in Pascal, or `/*...*/` in C. In both these languages, comments are not nestable, that is

```
/* This is a C comment
   /* with a nested comment inside it */
  which is illegal */
```

is illegal in C. The first `*/` will be taken as closing off the first `/*`, and the second `*/` will generate an error. However, there are languages which do allow nested comments, and some C compilers (such as Borland C++ 3.1) allow nested comments to be switched on in C.

A third style of comment specification is to use a token to introduce the comment which terminates at the line end. This form is the standard in non-free format languages such as assemblers and FORTRAN, and has made a belated comeback in free format languages such as Ada and C++.

There is some dispute as to which style is best, and some languages offer more than one. The argument for nested comments is that they allow sections of code to be easily 'commented out', that is removed from a compilation. The argument against is that it is easy to overlook a comment bracket and not realise that a block has actually been commented out. Commenting out by prepending, say, `--` to each line as in Ada certainly makes disabled code stand out and any good editor will allow simple macros to be written which add and delete comment prefixes to a block of code.

A final complication with comments is that under some circumstances they need to be visible to the phrase level grammar. This is particularly so for languages that support *pragmas* which are special comments usually used to switch compiler features on and off, replicating the functionality of command line arguments.

`rdp` provides a family of six comment primitives, which may be intermixed. If you use more than one comment primitive then of course they must all have different opening tokens so that the parser can tell them apart purely on the basis of their opening tokens. The full set is

| | |
|---|---|
| `COMMENT(OPEN CLOSE)` | Everything between `OPEN` and first `CLOSE`. |
| `COMMENT_VISIBLE(OPEN CLOSE)` | Everything between `OPEN` and first `CLOSE`. |
| `COMMENT_NEST(OPEN CLOSE)` | Everything between `OPEN` and matching `CLOSE`. |
| `COMMENT_NEST_VISIBLE(OPEN CLOSE)` | Everything between `OPEN` and matching `CLOSE`. |
| `COMMENT_LINE(OPEN)` | Everything between `OPEN` and the line end. |
| `COMMENT_LINE_VISIBLE(OPEN)` | Everything between `OPEN` and the line end. |

The `..._VISIBLE` primitives return the body of the comment to the phrase level grammar and so may be used for pragmas. The normal primitives discard the comments in the scanner, treating them identically to white space.

Comment close tokens can only be a maximum of two characters long, and conform to the usual token rules, that is empty tokens and tokens containing white space are not allowed.

# Chapter 5

# Attributes and semantic actions

A basic `rdp`-generated parser acts as a *syntax checker* for the specified language, which is a useful but rather limited function. By including embedded *semantic actions* within a parser specification we can force the running parser to execute particular functions as it recognises portions of the input text string. In general these semantic actions will need to be able to interact with the parsing process proper. A calculator, for instance, will need to be able to parse numbers and operators and then execute the appropriate semantic actions. In detail, the action to add two numbers together will need to know not just that a number has been parsed, but what its value was. This information is transmitted into semantic actions by using *synthesized attributes* which act a little like the return values of a function in a conventional programming language.

Occasionally, the semantic actions may need to influence the future behaviour of the parser, and so `rdp` also supports the use of *inherited attributes* which act a little like parameters to the rules that make up a language specification.

This chapter provides a very brief overview of the action and attribute definition features of `rdp`. A much more extensive discussion of the design of language processors using these features will be found in the `rdp` tutorial manual [JS97c] and example case study [JS97a].

## 5.1 An introductory example

Here is a very simple example grammar that can be input to `rdp`. We shall examine the runtime behaviour of the parser generated by `rdp` from this grammar.

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

Imagine that the generated parser is asked to parse the string `2 + 4 * 5`. As the parse proceeds, the parser calls the functions associated with each grammar symbol it encounters. The *return type declarations* of the form `:integer`

which appear on the left hand sides of the rules cause the functions for **start** and **expr** to return a value of type **integer**. The identifier which holds the value to be returned is always called **result**.

The parser begins by calling the function for the start symbol, **start**. This then calls the scanner routine for INTEGER, which will return the value of the integer recognised, in this case the value 2. The *synthesized attribute declaration* :val1 which appears after the INTEGER scanner directive instructs the parser to write the value returned by the call to the INTEGER scanner primitive into a local variable called val1.

The parser then recognises + and then calls the function corresponding to grammar rule **expr**. This function parses the phrase 4*5, writing the values 4 and 5 to the local variables val1 and val2 respectively. The last section of the **expr** routine then executes the semantic action which is enclosed between [* and *] brackets. The effect of this is to write the value 20 to the predefined identifier **result**, and this value is then returned as the **expr** routine is exited.

The synthesized attribute declaration :val2 which is appended to the symbol **expr** in the rule for **start** instructs the parser to write the returned value from **expr** to the identifier val2, so in this case 20 is assigned to val2. The semantic action at the end of the start rule is then executed so that the final return value, held in **result**, is 22.

We now give a fuller description of attribute and semantic action use in **rdp**-generated parsers.

## 5.2 Synthesized attribute definition

Each **rdp** rule and token can optionally return a single attribute. A rule that does *not* return an attribute is implemented in the C code for the generated parser as a void function with the same name as the rule, so

```
simple_rule ::= 'a' 'b'.
```

maps to

```
void simple_rule(void)
{
  ...
}
```

If the rule name is followed by a colon and a data type, then a function returning that type is declared along with a local variable called **result**, also of the same type as the function, which is used to hold the return value.

```
attributed_rule: integer ::= 'a' 'b'.
```

maps to

```
integer attributed_rule(void)
{
  integer result;
```

| Primitive | Return type | Return value |
|---|---|---|
| ID | string | characters making up the identifier |
| INTEGER | unsigned_integer | unsigned integer value of literal constant |
| REAL | real | floating point value of literal constant |
| STRING | string | characters making up the string |
| STRING_ESC | string | characters making up the string |
| COMMENT_VISIBLE | string | characters making up the comment |
| COMMENT_NEST_VISIBLE | string | characters making up the comment nest |
| COMMENT_LINE_VISIBLE | string | characters making up the comment |

**Table 5.1** Scanner primitive attribute types

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

The return value in **result** can be loaded using semantic actions, or directly from the return value of a rule call: the following are both valid ways of getting a value into **result**.

```
rule1: integer ::= INTEGER:value [* result = value; *].
rule2: integer ::= INTEGER:result.
```

Note that the data type **integer** is defined to be a synonym for **long int** in the parameter file **scan.h** held in the **rdp_supp** subdirectory.

The data type can consist of a *single* identifier followed by one or more stars (to indicate indirection). If you need a rule to return a complex datatype, such as a struct, then use a **typedef** to define a synonym for it.

Multiple attributes can be returned from a rule by packing them into a **struct** or **array**. You will need to implement the code to do this yourself.

## 5.3 Synthesized attribute types for scanner primitives

The scanner primitives have built in attribute types which may be used to retrieve, for instance, the string associated with an identifier or the numeric value of a **REAL** or **INTEGER**. The full set of primitives, their return types and return values is summarised in Table 5.1.

## 5.4 Using synthesized attributes

Synthesized attribute values are created by appending a colon and a name to a symbol on the right hand side of a grammar rule. **rdp** declares a local variable with the same name, and of the type specified by the return type of the generated parser function for that symbol. After the generated parser has called the function for a symbol, the return value is loaded into the local variable, making it available to subsequent semantic actions. Synthesized attributes cease to exist when a parser function is exited.

```
rule ::= ID:name attributed_rule:value.
```

where `attributed_rule` has been defined as returning an integer attribute, maps to

```
void rule(void)
{
  string name;
  integer value;
  ...
}
```

Values will be loaded into the variables `name` and `value` after the corresponding phrases have been parsed.

If a rule that returns a result is called without an attribute name being declared then the result is simply thrown away.

## 5.5 Inherited attribute definition

`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):integer ::= 'a' 'b'.
```

maps to

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

The most common use of inherited attributes is to pass information into a rule that will be used to enable semantic actions. In the `rdp` case study [JS97a], an interpreter for an enhanced version of the `mini` language is used to illustrate the application of inherited attributes to the implementation of an IF ... THEN ... ELSE statement.

In general, parser rules can have both inherited attributes and return a single synthesized attribute.

## 5.6 Semantic actions

C-code fragments may be added to `rdp`-generated parsers by enclosing them in `[*... *]` brackets in the IBNF specification. These brackets do not nest, and no escape sequence is needed. The contents of each bracket pair is simply copied directly to the parser without any intervening spaces. If for some reason you want to get the string `something*]else` into the output you can write it as

```
[*something**] [*]else*]
```

The usual purpose of semantic actions is to manipulate the values of attributes passed into the rule by other rules or scanner tokens.

## 5.7 Error checking of semantic actions

No syntax checking of code fragments occurs when `rdp` generates a parser since `rdp` is not itself a C compiler. As a result, any syntactic or logical errors that you introduce into the generated parser will not be detected until it is compiled or run. If you have a problem and are not clear whether it is the semantics or the grammar that is causing it then try running `rdp` with the `-p` option, which suppresses semantic action insertion, to check whether your grammar correctly parses a test file.

## 5.8 Default actions for iterators that match the empty string

An iterator with a lower bound of zero (which of course includes the `{ }` and `[ ]` shorthands) may match against nothing, or to put it another way the body of the iterator may not be entered. In such cases, it is useful to be able to specify a semantic action that acts as a *default*, that is, an action that is executed only when the body is not entered. `rdp`-IBNF allows such a default action to be appended to an iterator or bracket with a lower bound of zero by inserting a colon.

Consider the grammar

```
start ::= 'a' { 'b' } 'c'.
```

which generates the language comprising zero or more instances of `b` bracketed by `a` and `c`. The following grammar prints out a message when presented with the input string `ac`.

```
start ::= 'a' { 'b' }:[* printf("No b's in string"); *] 'c'.
```

The construct `{ }:[*` *default action* `*]` defines a default action that is only executed when the body of the iterator is not entered, i.e. the `{ }` brackets are matched against the empty string $\epsilon$.

Defaults can also be attached to full iterators as in:

```
start ::= ID 0@8 ',':[* printf("No identifiers seen"); *] .
```

## 5.9 Semantic rules

Semantic actions are often quite large pieces of code, and they can obscure the flow of the grammar by overwhelming the IBNF. In addition, it is confusing to read a single specification that contains code operating on different levels — in this case the high level IBNF and the low level C syntax.

`rdp` allows you to parcel up large semantic actions into rules of their own, whose right hand sides contain only semantic actions, which are inserted *inline* into the appropriate function in the generated parser. This allows semantic actions to be described away from the actual instantiation point with no loss of efficiency.

A *semantic rule* is a special form of grammar rule that contains only semantic actions.

Consider this grammar fragment:

```
statement ::= ID:name '=' e1:value
                 [* symbol_lookup_id(name)->data.i = value; *] |
               'print' '(' ( e1:value [* printf("%i",value); *] |
                             string:str [* printf("%s", str+1); *]
                           )@','
                       ')'.
```

Using semantic rules this may be reworked as

```
statement ::= ID:name '=' e1:value _1 |
               'print' '(' ( e1:value _2 | string:str _3 )@',' ')'.

_1 ::= [* symbol_lookup_id(name)->data.i = value; *].
_2 ::= [* printf("%i",value); *].
_3 ::= [* printf("%s", str+1); *].
```

The version incorporating the semantic rules splits the semantics out from the syntax definition making the grammar rather more readable.

## 5.10 Semantic actions in multi-pass parsers

By default, **rdp**-generated parsers make a single pass over the input text, executing semantic actions on the fly. Many programming languages are designed to be parsed in this way: in C and Pascal for instance identifiers must be declared before use to allow single pass translation.

Some translation tasks are hard to accomplish in a single pass. **rdp** itself, for instance, is a translator for a language that does *not* require identifiers to be declared before use. In fact **rdp** makes two passes over a **.bnf** file: on the first pass all of the rule names are collected together and any rules that have been declared more than once are reported. On the second pass any references to undeclared rules can be detected. Other examples of translators that typically use more than one pass are assemblers and other low level languages that allow identifiers to be defined and used in arbitrary order.

**rdp** allows multiple pass parsers to be created easily. If a `PASSES(`*n*`)` declaration is inserted into a **.bnf** file then **rdp** will make *n* passes over the text (see section 6.1). Usually, semantic actions in multi-pass translators are designed to be executed on particular passes. Within a multi-pass parser, the global variable **rdp_pass** holds the current pass number (see section 8.1) which may be used to filter actions. A semantic action such as

```
[* if (rdp_pass == 2) printf("Executing actions on second pass"); *]
```

will only generate output on pass two. It would be tiresome to have to insert these kinds of `if` statements into every semantic action of a multi-pass parser, so **rdp** allows specification of the pass on which an action is to be executed. This semantic action is equivalent to the previous one:

```
[* printf("Executing actions on second pass"); *]@2
```

By appending an expression of the form `@`$n$ to an action then it is restricted to execution on pass $n$. **`rdp`** implements selective execution by simply wrapping the appropriate **`if`** statement around the action. An action without a trailing `@` expression will be executed on all passes.

# Chapter 6

# Directives

rdp directives are used to parameterise the parser: for instance most of the standard command line switches have default values which can be set up using directives. Other features controlled by directives include the instantiation of symbol tables in the generated parser with the SYMBOL_TABLE directive and the definition of new command line switches which can be added using the ARG_... family of directives. In addition, some global values such as the case sensitivity of the target language can be initialised using directives.

## 6.1 Flow control directives

INCLUDE("*filename*")

IBNF descriptions can span several files. The INCLUDE directive pulls in another .bnf file in exactly the same way as the #include preprocessor command in C. Included files can be nested to arbitrary depth: the only limit is the amount of available memory available to hold the list of nested file descriptor blocks.

## 6.2 Parser setup directives

USES("*filename*")

All rdp-generated parsers automatically include the header files for the scanner, text handler, memory manager, argument handler, graph handler, symbol table and set handling modules. Any user header files (such as the myth_aux.h file from the example in Figure 1.1) can be specified using this directive. Multiple USES directives may be issued, one for each included file, to generate a sequence of #include preprocessor directives in the C parser source file. The #include directives will appear in the same order as they are declared in the IBNF source file and will be followed with #include"*filename*.h" where *filename* is the name of the C parser header file. Token names for the grammar and symbol table data structures are defined in this header file.

`TITLE("`*string*`")`

The title of the generated parser, as reported in verbose mode and at the top of the help message is set using this directive. If no **TITLE** directive appears in an IBNF description then the default title of **rdparser** will be used.

`SUFFIX("`*string*`")`

The default filetype for the generated parser is set using the **SUFFIX** directive. **rdp** automatically appends a period (.) and the suffix to any source file name that is specified without a filetype. The *string* argument specified to the **SUFFIX** directive should not contain the leading period.

If no **SUFFIX** directive appears in an IBNF description then filetype processing is disabled and the user filename will be used literally.

`PARSER(`*start*`)`

The parser start rule is declared using this directive. If no **PARSER** directive appears in the grammar then the first rule encountered is taken to be the start rule.

`PRE_PARSE([*` *action* `*])`

The **rdp**-generated parser **main()** function checks command line arguments, initialises various subsystems and then makes a call to the parser function corresponding to the first IBNF rule found. If a **PRE_PARSE** directive is found in the IBNF description then the C language *action* is copied into the **main()** function immediately before the call to the parser.

`POST_PARSE([*` *action* `*])`

The **rdp**-generated parser **main()** function checks command line arguments, initialises various subsystems and then makes a call to the parser function corresponding to the first IBNF rule found. If a **POST_PARSE** directive is found in the IBNF description then the C language *action* is copied into the **main()** function immediately after the call to the parser.

`OUTPUT_FILE("`*file*`")`

The default value of the output filename for the generated parser is set using **OUTPUT_FILE**. It can be overridden on the command line with a **-o** directive. If no **OUTPUT_FILE** directive appears in the grammar then a default output name of **rdparser** is used.

`PASSES(`*count*`)`

Normally the **rdp**-generated parser contains a **main()** function that simply calls the function corresponding to the first IBNF rule, and then returns control to the user. For some applications, such as assemblers, it is convenient to have

the parser called multiple times. If a `PASSES` directive is encountered then the parser call is wrapped in a `for` loop causing the parser to be called *count* times. The internal variable `rdp_pass` may be referenced in semantic actions to check which pass is currently executing.

Note that the `POST_PARSE` routine is not called until all passes are complete, and that any listing requested by a `-l` command line option is not generated until the last pass.

`SYMBOL_TABLE(`*name size prime compare hash print* `[*` *data* `*])`

`rdp`-generated parsers make use of the hash coded symbol table package `symbol` which is described in [JS97b]. Each symbol table takes

- a *name* which must be a valid C identifier,
- an integer *size* which is the number of hash buckets to allocate,
- an integer *prime* which would ideally be a large prime number less than *size*,
- the name of a *compare* function,
- the name of a *hash* function,
- the name of a *print* function,
- a list of data fields.

*name* can be any identifier not used elsewhere in the grammar. The *size* of the table should be set to about 30–50% of the expected number of symbols to be placed in the table. The table will work correctly even if *size* is very small compared to the number of symbols but performance will suffer. *prime* must be coprime (*i.e.* not sharing any common factors greater than 1) with *size* for the standard hashing functions to work well.

*compare*, *hash* and *print* are pointers to functions that understand the layout of the user data. If your user data consists of a pointer to a string and a set of other fields, and if that string is the symbol table key (a common situation) then the standard routines supplied as part of the `symbol` package may be used.

The *data* fields are a list of semicolon delimited data declarations which are copied into the body of a `struct` by `rdp`.

For simple tables that are keyed on a string the following directive works well:

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

For each `SYMBOL_TABLE` directive, `rdp` creates global variable *name* which points to the table and then initialises it by calling `symbol_new_table()` before beginning the parse. In the header file, `rdp` also creates a data structure from the *data* fields and uses a `typedef` to create a name of the form *name*`_data` by which it may be called. Finally, a cast macro called *name*`_cast` is defined.

## 6.3 Command line argument definition directives

`rdp` builds ready-to-run parsers that include an automatically generated help facility: if a user mistypes a command line when trying to run an `rdp`-generated parser then a fatal error message will be issued which includes a condensed guide to the command line switches supported by the parser. By default, `rdp`-generated parsers support the command line arguments shown in Table 7.1. Extra command line arguments may be added using a family of four directives. They map onto the library functions declared in `rdp_supp\arg.c` which are described in the support library manual [JS97b]. The `rdp` source file `rdp.bnf` provides a large example of the use of command line argument definitions.

`ARG_BOOLEAN(`*key identifier key_string*`)`

Add a boolean command line argument. *key* should be a single alphabetic character. *identifier* is the name of a variable that will be automatically declared in the generated parser and initialised to zero. *key_string* is a descriptive string that will be reproduced if the help message is displayed.

For example, this declaration

```
ARG_BOOLEAN(X x_flag "Set X flag")
```

will add a `-X` command line argument and insert a variable called `x_flag` into the parser. The `x_flag` variable will be initialised to FALSE (zero), and then each instance of `-X` will invert the flag: hence a parse invocation of the form `rdparser -X myfile` will cause `x_flag` to be set to TRUE (1).

`ARG_NUMERIC(`*key identifier key_string*`)`

Add a numeric command line argument. *key* should be a single alphabetic character. *identifier* is the name of a variable that will be automatically declared in the generated parser and initialised to zero. *key_string* is a descriptive string that will be reproduced if the help message is displayed.

For example, this declaration

```
ARG_NUMERIC(N n_value "Set value of n")
```

will add a `-N` command line argument and insert a variable called `n_value` into the parser. The `n_value` variable will be initialised to zero and then a parser invocation of the form `rdparser -N25 myfile` will cause `n_value` to be set to the value of the numeric parameter (25 in this case).

`ARG_STRING(`*key identifier key_string*`)`

Add a string command line argument. *key* should be a single alphabetic character. *identifier* is the name of a variable that will be automatically declared in the generated parser and initialised to the empty string (`""`). *key_string* is a descriptive string that will be reproduced if the help message is displayed.

For example, this declaration

```
ARG_STRING(S s_value "Set value of s")
```

will add a `-S` command line argument and insert a variable called `s_value` into the parser. The `s_value` variable will be initialised to zero and then a parser invocation of the form as `rdparser -Sstring myfile` will cause `s_value` to be set to the value of the string parameter (`string` in this case).

`ARG_BLANK(`*key_string*`)`

Add a *blank* argument. No actual command line switch is instantiated: this declaration is used to add a line to the help message. *key_string* is a descriptive string that will be reproduced if the help message is displayed.

## 6.4 Command line default directives

`TAB_WIDTH(`*count*`)`

This directive sets the default number of spaces to expand tabs to when producing listings. If no `TAB_WIDTH` directive appears in the IBNF source then 8 is assumed. It may be over-ridden with the `-t` command line directive.

`TEXT_SIZE(`*count*`)`

This directive sets the default size of the scanner's text buffer in bytes. If no `TEXT_SIZE` directive appears in the IBNF source then 20,000 is assumed. It may be overridden with the `-T` command line directive.

Note that your operating system and compiler may impose their own limits on the size of the buffer: for instance MS-DOS 16-bit compilers often limit the size of a single heap object to 64K bytes, which acts as an effective limit to the size of the text buffer. `rdp` will exit with a fatal memory allocation error if you exceed the operating system limit. You can usually get a good idea of the limitations of your system by looking at the definition of the ANSI C datatype `size_t` which is used to represent the size of memory objects. If, as in Borland C++ version 3.1, `size_t` is a 16 bit number then the 64K limit applies.

`MAX_ERRORS(`*count*`)`

This directive sets the maximum number of errors that will be reported before parsing is aborted. If no `MAX_ERRORS` directive appears in the IBNF source then 25 is assumed.

`MAX_WARNINGS(`*count*`)`

This directive sets the maximum number of warnings that will be reported before parsing is aborted. If no `MAX_WARNINGS` directive appears in the IBNF source then 100 is assumed.

## 6.5 Scanner directives

`CASE_INSENSITIVE`

By default, **rdp**-generated parsers are case sensitive. For languages such as Pascal which are case insensitive, the scanner may be set by a `CASE_INSENSITIVE` directive so as to force all characters outside of strings and comments, including tokens and identifiers, to be lower case. Since conversion is from upper to lower case, tokens in the IBNF description of a case insensitive language should be written in lower case. The file **pascal.bnf** supplied in the standard **rdp** distribution provides an example of the use of this directive.

`SHOW_SKIPS`

After detecting an error, **rdp**-generated parsers consume input until a token that might reasonably be used to restart the parse is found. This process is known as *skipping*, and if a `SHOW_SKIPS` directive appears in the IBNF description then an extra warning message is enabled that marks the end of the skipped passage. This is useful when debugging error handling.

## 6.6 Tree generation directives

For completeness, this section summarises the directives that enable automatic tree generation. For fuller documentation, please refer to Chapter 9.

`TREE([*` *data* `*])`

Switch on tree generation and (optionally) define extra data fields to be added to each tree node. The trees will have epsilon nodes deleted: leaf nodes containing epsilon are simply removed and internal epsilon nodes are removed with their children being promoted to be at the same level as the internal epsilon node was at before pruning. Fuller documentation on tree generation will be found in Chapter 9.

`EPSILON_TREE([*` *data* `*])`

Switch on tree generation and (optionally) define extra data fields to be added to each tree node. Epsilon nodes will be left in the tree. Fuller documentation on tree generation will be found in Chapter 9.

`ANNOTATED_EPSILON_TREE([* data *])`

Switch on tree generation and (optionally) define extra data fields to be added to each tree node. Epsilon nodes will be left in the tree as with the **EPSILON_TREE** directive but each such node will be annotated with the string `#:`*name* where *name* is the name of the subrule that generated the epsilon. Fuller documentation on tree generation will be found in Chapter 9.

# Chapter 7

# Running `rdp`

This chapter describes the **`rdp`** command line options. **`rdp`** reads a single IBNF source file (of default filetype **`.bnf`**) and writes out a header file and a parser file. If no output filename is supplied then the files are written to **`rdparser.h`** and **`rdparser.c`** respectively.

The **`rdp`** command accepts parameters which can either be *option switches* which are denoted by a leading minus sign (`-`) followed by a letter, or *filename arguments* which are anything else. Options and filenames can be intermixed: it is not necessary to place the filename after the options. **`rdp`** expects a single filename — if you issue multiple filenames then only the leftmost one will be used.

All **`rdp`** options are processed in strict left to right order. This is significant because some options can override the actions of other options: in such cases the rightmost instance of an option will override any earlier ones.

## 7.1 `rdp` filename parameters

Any **`rdp`** parameter that does not consist of minus sign (`-`) followed by a non-whitespace character will be taken as a filename.

The **`rdp`** scanner attempts to add a default filetype to the filename you specify. It starts at the rightmost character, and looks backwards for a period (`.`). If it encounters one *before* it finds the start of the filename or an instance of either the Unix or MS-DOS directory separators (`/` and `\`) then it assumes that you have supplied your own filetype and leaves the filename untouched. If it does not find a period then it appends **`.bnf`** to your filename.

## 7.2 `rdp` option parameters

### `-e` write out expanded IBNF

This option enables an extended listing mode that causes **`rdp`** to render all of its internal and external rules as human readable BNF, as well as enumerating the first and follow sets for each rule, and giving a count of the number of times the rule is called in the grammar, that is the number of times a particular rule names appears on the right hand side of rules.

An internal rule (or *subrule*) is the expansion of one of the **rdp** extended rule types that are described in Chapter 3: the `(...)`, `[...]`, `{...}` or `<...>` bracket pairs or the expansion of an iterator operator `@`. Each internal rule inherits its parent rule's name with the string `rdp_` prepended and the string `_`$n$ appended, where $n$ is a unique integer.

This option usually generates a lot of output, but can be very educational when analysing first and follow sets for simple languages.

## -E add rule name to error messages

When debugging a grammar it is sometimes helpful to know which rule was being processed when an error occurred. If you regenerate the parser using **rdp** and add the **-E** flag to the **rdp** command line, then the message **In rule** '*name*' is prepended to all syntax errors displayed by the generated parser where *name* is the name of the active rule at the time the error was found.

## -f filter mode

In filter mode, input is read from the standard input and written to the standard output. The **-f** option sets the input to **stdin** (either the keyboard, or the output of a previous operation within a pipe) and the output to **stdout**, but subsequent **-o** options or filenames can be used to override this. Similarly, this option overrides any previous **-o** option.

## -F force creation of output files

Any ambiguities or left recursion in the supplied grammar will cause **rdp** to report LL(1) error messages, and inhibit production of the output files. Most real languages have at least one ambiguity (the IF ... THEN ... ELSE problem) and several others (such as C) have ambiguities based on the use of identifiers in different contexts. Careful design of the source grammar can result in correctly working parsers even in the face of these ambiguities because **rdp** will accept the first matching production alternate in a rule, in which case the **-F** option can be used to force **rdp** to produce its output files.

## -l make a listing

Usually, only lines containing syntax errors are echoed to the screen whilst an **rdp**-generated parser is running. When a **-l** option is issued each line of the source file is echoed to the message stream as it is read. Usually the message stream is the standard error stream, but you can change this by altering the macro **TEXT_MESSAGES** which is defined in **textio.h** and recompiling the whole system. It is also possible to change the message stream at run time using the **text_redirect()** routine: see the support manual [JS97b] for more information.

### -ofilename write output to filename

By default rdp creates output files rdparser.h and rdparser.c. When a -o option is issued, the characters immediately following the o up to the next whitespace character are taken as the filename. Any filetype is stripped off, and the remaining characters are used as a filename body.

### -p make parser only

It is often useful to be able to disable semantic actions and produce a pure parser so as to debug the grammar without interference from embedded C semantic actions. The -p option causes rdp to suppress the writing of semantic actions into the parser source code which may then be compiled into a pure syntax checker.

### -R add rule entry and exit messages

When debugging a grammar it is useful to be able to get a trace of the parser's execution path. One way to do this is to add semantic actions to each rule which print out a message on entry to the rule and on exit. It would be tedious to do this by hand: the -R option instructs rdp to automatically add these messages for all rules. This option can cause generated parsers to produce voluminous output.

### -s echo each scanner symbol as it is read

When debugging scanners it can be very helpful to get a diagnostic dump of each lexeme as it is passed to the parser. This option produces one line of output *per* lexeme and is only recommended for detailed debugging since it generates a lot of output.

### -S print summary symbol table statistics

rdp-generated parsers use hash coded symbol tables that are declared in the IBNF source file. The number of hash buckets in each table is specified by the user in the grammar and should be kept large enough to keep the number of entries per bucket below about four or five for efficient parsing. The -S option prints out a histogram of bucket utilisation frequencies and the mean bucket utilisation figure for each declared table. Note that the scanner uses one table internally to hold the keywords from the IBNF specification, and statistics for that table are printed too. If the tables are becoming congested then increase the size in the corresponding SYMBOL_TABLE() directive (see section 6.2) and regenerate.

### -tn tab expansion width

When echoing lines read in by the scanner it is important that tabs are correctly expanded or the user's formatting may be lost. rdp supports fixed tab stops

| | |
|---|---|
| `-f` | Filter mode |
| `-l` | Make a listing |
| `-ofilename` | Write output to **filename** |
| `-s` | Echo scanner symbols |
| `-S` | Print symbol statistics |
| `-tn` | Tab expansion width |
| `-Tn` | Text buffer size |
| `-v` | Set verbose mode |
| `-Vfilename` | Write derivation tree to **filename** in VCG format |

**Table 7.1** Standard command line options

every $n$ characters where $n$ defaults to 8, but may be set to some other value with a `-t` option.

### `-Tn` text buffer size in bytes for scanner

The **rdp** scanner allocates a fixed text area at initialisation time. The text buffer is used efficiently, but will eventually fill up when parsing long files. The `-T` option may be used to override the default text buffer size for the running parser.

Note that your operating system and compiler may impose their own limits on the size of the buffer: for instance MS-DOS 16-bit compilers often limit the size of a single heap object to 64K bytes, which acts as an effective limit to the size of the text buffer. **rdp** will exit with a fatal memory allocation error if you exceed the operating system limit. You can usually get a good idea of the limitations of your system by looking at the definition of the ANSI C datatype **size_t** which is used to represent the size of memory objects. If, as in Borland C++ version 3.1, **size_t** is a 16 bit number then the 64K limit applies.

### `-v` set verbose mode

In verbose mode **rdp** issues a running commentary on its progress, reporting all stages of the grammar checking process. All **rdp**-generated parsers also report CPU time usage in verbose mode. The value of the verbose flag (o if it is absent on the command line and 1 if it is present) is held in global variable **rdp_verbose** which may be examined from within semantic actions. This feature can be used to provide extra output from generated parsers in verbose mode.

### `-V` dump derivation tree in VCG format

**rdp**-generated parsers built from specifications that include one of the three **TREE** directives automatically build derivation trees using the graph manipulation support library. These trees can be output in a textual form that is suitable for reading into the VCG compiler graph visualisation tool [San95]. Activation of this option causes the tree to be dumped out at the end of the final parser

pass. Further details on tree construction and manipulation will be found in Chapter 9.

## 7.3 Options understood by `rdp`-generated parsers

All **rdp**-generated parsers automatically include all of the option flags listed in tablestandard. In addition, extra options can be specified in the grammar using the **ARG** directives described in the next chapter.

# Chapter 8

# rdp global variables

## 8.1 Monitoring parser status at runtime

Each rdp-generated parser has a set of global variable definitions written into it that are initialised before the parser start rule is called. Semantic actions in the parser file can access these variables.

### rdp_error_return

The contents of this variable supply an error return status to the operating system on normal completion. By default it is set to zero but semantic actions may set it to any value. A fatal error always returns a fatal status to the operating system regardless of the contents of this variable.

### rdp_outputfilename

The value of the -o output file command line switch, or "-" if a -f argument was last seen.

### rdp_pass

The current pass number. Passes are numbered 1 to *n* where *n* is the number defined in the PASSES() directive. Pass expressions of the form @*n* may be appended to semantic actions to control which pass they are executed on.

### rdp_sourcefilename

The value of the source filename supplied on the command line.

### rdp_tree

A pointer to the root of the derivation tree for parsers generated from specifications that include one of the TREE directives.

`rdp_verbose`

The value of the verbose mode flag. By default it is set to 0, but if a `-v` command line switch in encountered it is set to 1.

## 8.2 Defining the message stream

The initial destination for messages created by the `text_message()` and `text_printf()` functions is controlled by the value of the `TEXT_MESSAGES` macro. By default this is defined to be `stderr`, the standard error stream. On MS-DOS in particular it is sometimes useful to redefine this to be `stdout` because of the difficulty of capturing the standard error stream to a file. The message stream can also be redirected in mid-parse using the `text_redirect()` routine. See the support library manual [JS97b] for further details.

## 8.3 Adding reserved words to the dangerous identifier list

`rdp` checks that all your identifiers are valid C identifiers that will not clash with C or C++ reserved words. The macro `RDP_RESERVED_WORDS` which is defined in `rdp_aux.h` specifies the list of reserved words. The standard distribution contains the ANSI reserved words and a few standard library functions. You can add strings to this list in any order: you might have your own standard library functions, for instance. Checking for reserved words is efficient — at the end of parsing a complete IBNF specification all of the identifiers will be in the `rdp` symbol table. During the grammar checking phase `rdp` looks to see if any of the words specified in `RDP_RESERVED_WORDS` is in the symbol table, and issues error messages accordingly. Hence checking time is linear in the number of reserved words and is independent of the length of the source text.

# Chapter 9

# Derivation tree construction and visualisation

Some translation tasks are difficult to perform during a parse, even if a multi-pass parser is employed. In such cases, it is normal to construct an internal representation of the source text during parsing which may be traversed efficiently, and to use an *intermediate form* for tasks such as optimisation.

High quality compilers can perform many different code improvement transformations as part of an optimisation phase. Typically, optimisations work by relating together separate parts of the source text and so are very difficult to implement in a single pass compiler which only 'sees' a very small part of the input at any one time.

Take for example, *common sub-expression elimination* which is one of the most commonly applied optimisations. Consider two 2-dimensional arrays declared as

```
int a[10][20], b[10][20];
```

We can copy one element of `b` to the corresponding position in `a` as follows:

```
a[i][j] = b[i][j];
```

This simple assignment actually hides two indexing calculations which we can render explicitly in C using address arithmetic. In detail the computer has to perform this calculation:

```
*(a + (i*10) + j) = *(b + (i*10) + j);
```

Here, the index `i` is multiplied by the width of the array (10 in this case) and then added to the value of `j` and the base address of `a` to get the machine location of element `a[i][j]`, and then essentially the same calculation is performed to find the location of `b[i][j]`.

A single pass compiler is pretty much limited to producing this kind of repetitive code, but a compiler which is capable of gathering together information from potentially widely separated parts of the source program can implement the common sub-expression separately, producing this more efficient code:

```
int temp = (i*10) + j;
*(a + temp) = *(b + temp);
```

If a multiple pass translator is to be used then it is usual to construct a data structure in memory that represents the input program in a manner which may be efficiently processed. Simply storing the original program text is inefficient because discovering a derivation for an input text is time consuming — that is after all the primary function of the parsers that **rdp** constructs and it would clearly be wasteful to run the process several times[1].

Leaving aside issues of efficiency, making multiple independent passes over the source text does not of itself allow us to make connections between widely separated parts of the text because the parsers generated by **rdp** only look at a single symbol at a time: they do not of themselves keep track of complete sentences or program statements. However, **rdp** can be set to build a *derivation tree* whilst it performs a parse. This tree reflects explicitly relationships between symbols in the source program, and since it is held as a pointer-based data structure rather than as a single long text string, it can be traversed and rearranged efficiently.

## 9.1 Derivation trees

Informally, a derivation tree is a *trace* of the parser's behaviour during a particular parse. The derivation tree is constructed top down, left to right by creating a new tree node every time a nonterminal or terminal is matched. Every nonterminal when matched creates a new internal tree node and every terminal when matched causes a new tree leaf node to be added. In addition, when an iterator with a lower bound of zero (or the shorthands `[ ... ]` and `{ ... }`) match the *empty* string $\epsilon$ (or `#` in **rdp** syntax) an empty tree leaf node is added. The idea is that the nodes created whilst matching the body of a nonterminal will be attached as children of the node corresponding to that nonterminal.

This rather complicated recipe is best illustrated with an example. Let us revisit the example used in section 5.1 which describes a minimilist grammar which can generate arithmetic expressions made up of an addition followed by a multiplication:

```
start ::= INTEGER '+' expr.
expr  ::= INTEGER '*' INTEGER.
```

When presented with the source string `3 + 6 * 7` the parser generated by **rdp** from the above grammar will construct the tree shown in Figure 9.1.

In this example, tree construction starts by making a root node labeled `start`. The scanner then matches an `INTEGER` with lexeme `3` and so a suitable `INTEGER` leaf node is added to the tree. The token `+` is then matched and another leaf node is added. At this point in the parse, the parser function for

[1] Of course, just because making multiple passes over the source code is a wasteful process it need not stop us using it where applicable and **rdp** provides the `PASSES` directive for precisely this purpose. Simple multi-pass applications, such as the implementation of a translator from a machine's assembly language to its machine code, may usefully exploit this strategy. You can read about the design and implementation of such as assembler in the case study manual [JS97a].

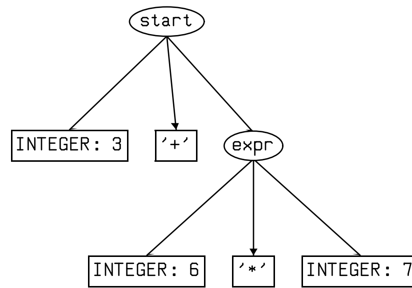


**Figure 9.1** A simple derivation tree

rule **start** calls the function corresponding to rule **expr** so a matching child node is added that then becomes the parent node for subsequent leaf nodes.

This picture was made with the VCG (Visualisation of Compiler Graphs) tool [San95] which you can obtain from the **rdp** archive as described in appendix A. Any **rdp**-generated parser that uses the automatic tree generation capability described in this chapter may be displayed on screen and printed using VCG if you are running under Windows or X-windows on Unix. You will find further information on using VCG in section 9.3.

Derivation trees grow rather rapidly. In the standard **rdp** distribution there is grammar for the Pascal language (**pascal.bnf**) and a corresponding test file (**test.pas**) containing 283 lines of Pascal. The tree produced contains 7167 nodes, one of which has 67 children. It is quite difficult to visualise these large structures although VCG provides useful navigation facilities.

### 9.1.1 A larger example

Let us examine a grammar which describes a language of assignment expressions. We allow the usual four arithmetic operators along with exponentiation (denoted by the operator `**`), monadic `+` and `-` operators and parenthesised expressions. The exponentiation operator is right associative and the other operators are left associative.

```
program ::= { statement ';' }.
statement ::= ID '=' e1.
e1 ::= e2 { ('+' | '-') e2 }. (* Add or subtract              *)
e2 ::= e3 { ('*' | '/') e3 }. (* Multiply or divide           *)
e3 ::= e4 | ('+' | '-') e3.   (* Monadic positive or negative *)
e4 ::= e5 [ '**' e4 ].        (* Exponentiate                 *)
e5::= ID                      (* Variable or ...              *)
      [ '(' (e1)@',' ')' ] |  (* ... function call            *)
      INTEGER              |  (* Numeric literal              *)
      '(' e1 ')'.             (* Bracketed subexpression      *)
```

Figure 9.2 shows the result of using this grammar to parse the string

```
a = 2;
```

```
b = a - 1 - 2 * (4 - 3) ** 4 ** 5 ** 6 / --+- 7;
```

This large tree displays several interesting features. The first thing to note is that the trees are designed to be read in a depth-first, left-to-right fashion. One useful side effect of this is that if the rectangular terminal nodes are written out in left-to-right order we recover the original string. This list of leaf nodes is sometimes called the *yield* of a tree.

It can also be useful to examine a horizontal section of the tree. When printed, the trees display all nodes having the same depth in the tree on the same horizontal line. Since the depth in the tree is dictated by the number of nested grammar rules active at any given point in the parse, a horizontal strip can tell you what was matched within a single body. The very top of the tree shown in Figure 9.2 for instance shows that the first rule expanded by the parser was `program` and the expansion was `statement ; statement ;`

Rules `e1, e2` and `e4` in our grammar each have the form of a call to a nonterminal followed by an optional phrase as in:

```
e2 ::= e3 { ('*' | '/') e3 }.
```

When an optional phrase matches the empty string $\epsilon$ (or `#` in `rdp` terminology) an empty node is added to the tree, and the tree shown in Figure 9.2 shows many examples of this.

Finally, note that different forms of operator specification generate different tree forms. In this case, the operators specified using `{ }` iterator brackets such as (`+, -, *` and `/`) generate horizontal runs of operators as in the sub-expression `b=a-1-2`. On the other hand, operators specified using right recursion such as `**` generate a descending sequence of nodes.

Since the full derivation tree is so large, it is conventional to discard some parts of the tree, retaining only those nodes that convey information needed by later stages of processing. `rdp` provides a set of *promotion operators* that allow nodes to be moved back up the tree, potentially overlaying earlier nodes. The operators are described later in this chapter, and you will find large scale examples of their use in the `rdp` case study document [JS97a]. To give you a flavour of what is possible, Figure 9.3 shows the result of applying these promotion operators to the tree shown in Figure 9.2.

The *modified derivation tree* has been obtained by

1. moving certain arithmetic operator terminals up so that they overlay their parents,

2. deleting some tokens which are redundant in the tree representation such as ;, ( and ),

3. deleting all empty ($\epsilon$) nodes left after the previous steps have been performed.

In addition, we ensure that chains of left associative arithmetic operators are converted to left descending sub-trees in a way that is symmetric with the right descending sub-trees used in the original tree to represent the right associative exponentiation operator,

**Figure 9.2** Derivation tree for expression grammar

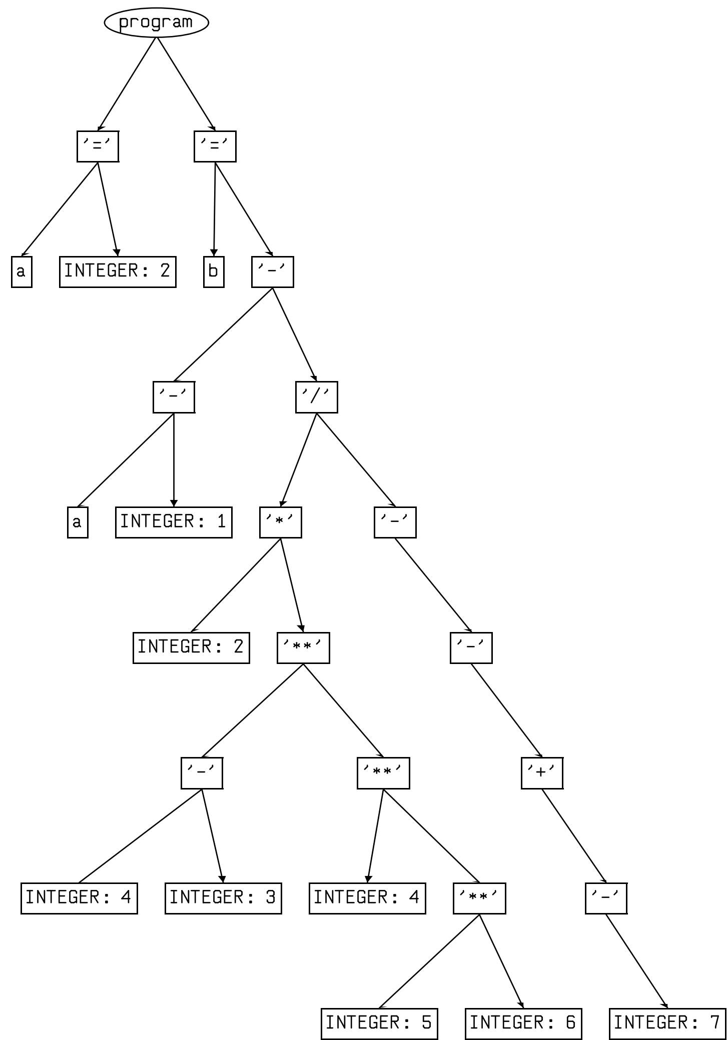


**Figure 9.3** A modified derivation tree

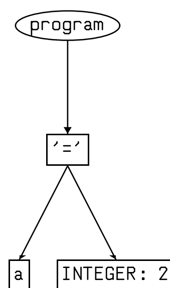


**Figure 9.4** Effect of the `TREE` directive

## 9.2 Tree generation directives

By default, `rdp`-generated parsers do not generate trees because the tree construction process does impose some overhead on the parsing process, and for simple single pass parsers this would be extravagant. Tree generation is switched on by adding one of the three tree directives to a grammar and then regenerating the parser by running the grammar through `rdp`.

`TREE([*` *data* `*])`

Switch on tree generation and (optionally) define extra data fields to be added to each tree node. The trees will have epsilon nodes deleted: leaf nodes containing epsilon are simply removed and internal epsilon nodes are removed with their children being promoted to be at the same level as the internal epsilon node was at before pruning.

`EPSILON_TREE([*` *data* `*])`

Switch on tree generation and (optionally) define extra data fields to be added to each tree node. Epsilon nodes will be left in the tree.

`ANNOTATED_EPSILON_TREE([*` *data* `*])`

Switch on tree generation and (optionally) define extra data fields to be added to each tree node. Epsilon nodes will be left in the tree as with the `EPSILON_TREE` directive but each such node will be annotated with the string `#:`*name* where *name* is the name of the subrule that generated the epsilon.

Figures 9.4–9.6 show the different effects of these directives when added to the expression grammar given above and used to parse the string `a=2;`

## 9.3 Using VCG to visualise derivation trees

All `rdp`-generated parsers accept the `-V`*filename* command line switch. For parsers that have been generated from grammars that do not include one of

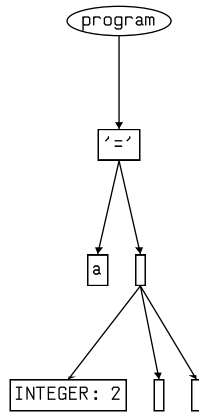


**Figure 9.5** Effect of the `EPSILON_TREE` directive

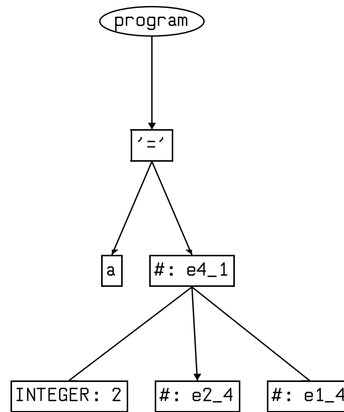


**Figure 9.6** Effect of the `ANNOTATED_EPSILON_TREE` directive

the three **TREE** directives this switch simply generates a warning message, but for parsers that do have tree generation enabled a text file will be created containing a specification of the derivation tree in the language understood by VCG. If the optional *filename* is specified then this will be the name of the VCG file, otherwise the default name of **rdparser.vcg** will be used.

On MS-Windows and Unix systems running X-windows the VCG tool may be started by typing

```
vcg rdparser.vcg
```

or the equivalent for your own file name. VCG will read the tree in (which may take a little while for a large tree) and then draw it on the screen. You can use VCG's navigational facilities to move around within the tree, zoom in and out, and print out the whole tree or a portion of it. VCG is a powerful tool and you should read the VCG manual supplied with the VCG distribution to get a full understanding of the tool. We are grateful to the author of VCG for permission to distribute it with **rdp**.

# Chapter 10

# Tree manipulation

Full derivation trees consume a lot of space, and often contain nodes that are of little use in subsequent language processing. Most books on compiler theory describe *concrete* and *abstract syntax trees* (often called AST's). There is rather little agreement on the formal definition of an AST, and in practice most language tool designers design an *ad hoc* representation which is built on the fly during the parsing phase. By embedding semantic actions in the specification of an `rdp`-generated parser it is, of course, possible to adopt this approach using `rdp`, but `rdp` provides a set of *promotion operators* which allow common AST forms to be automatically generated from the derivation tree. The advantage of this approach is that the grammar itself directly dictates the shape of the modified derivation tree whereas traditional AST's are only loosely related to the actual derivation tree. As a result, maintaining a language processor based on the traditional twin-track grammar and AST structure requires two independent tree-like forms to be described whereas in `rdp` the grammar itself fulfills both functions. The disadvantage is that the `rdp` promotion operators are not very easy to use, and we view them as somewhat experimental at this stage. The authors would be interested to hear of user experiences, both good and bad.

## 10.1 Normal tree construction

The promotion operators are applied on the fly during tree construction, and it is possible for a sequence of nodes to be promoted above each other. A full understanding of the effects of the promotion operators therefore requires an understanding of the order in which the tree is constructed.

At any given time during a parse there will be a *current parent*, that is a particular tree node which is the one to which children are being added. Immediately before a parse begins a root node representing the first call to the parser function for the start production is created and this is made the current parent. In the absence of promotion operators, subsequent tree growth occurs as a result of one of three processes:

1. Whenever a terminal is encountered within a rule a new node is added as a child of the current parent labeled with the terminal's lexeme. The

current parent does not change, therefore matches against terminals cause tree nodes to be added from from left to the right without changing the level.

2. Whenever an optional subphrase arising from `[...]` or `{...}` brackets or from an iterator with a low count of zero matches against the null string $\epsilon$, an epsilon node is added as a child of the current parent. The current parent does not change.

3. Whenever a nonterminal is encountered within a rule, a new child node is added to the current parent and labeled with the name of that nonterminal *and that node is made the current parent for any nodes that are created as a result of matches against that nonterminal's productions.* When the parser finishes matching against that nonterminal's productions, the original current parent is restored.

The effect of these rules is to go down one level in the tree each time a nonterminal is encountered and to go back up a level as the parser completes the matching of each production.

## 10.2 Modifying tree construction with promotion operators

The `rdp` promotion operators act so as to modify the rules above. There are four possible operators:

### 10.2.1 Promote underneath parent

The `^` (promote underneath) operator forces the node to be promoted to the parent node but the parent node's fields overwrite those of the node being promoted. The resulting node becomes the current parent for subsequent operations.

### 10.2.2 Promote on top of parent

The `^^` (promote on top of) operator forces the node to be promoted to the parent node and the parent node's fields are overwritten by those of the node being promoted. The resulting node becomes the current parent for subsequent operations.

### 10.2.3 Promote above parent

The `^^^` (promote above) operator forces the node to be promoted so as to become the parent of the current parent, that is it is inserted above the current parent rather than as a child of the current parent. The resulting inserted node becomes the current parent for subsequent operations.

### 10.2.4 Insert here (no promotion)

The ^_ (no promotion) operator forces the node to be inserted under the current parent in the usual way, that is the ^_ operator forces the rules described in the previous section to be observed for the grammar element to which the operator is applied. The current parent is unchanged. This operator is usually only used to apply the normal behaviour to a nonterminal whose default behaviour has been overridden, as described in the next section.

## 10.3 Valid contexts for promotion operators

Promotion operators may be applied in four contexts:

1. immediately after a terminal:

   ```
   a::=b '+'^^ c.
   ```

   in which case it indicates that the corresponding terminal node should be promoted,

2. immediately after a nonterminal on the right hand side of a rule:

   ```
   a::=b^^ '+' c.
   ```

   in which case it indicates that the corresponding nonterminal node should be promoted,

3. immediately after a nonterminal on the left hand side of a rule

   ```
   a^^::=b '+' c.
   ```

   in which case it specifies that the default promotion for instances of that nonterminal on the right hand side of rules should be changed from ^_ (the normal default) ,

4. in the default action clause of an optional phrase arising from [ ] or { } brackets or from an iterator with a low count of zero:

   ```
   a::=[b '+' c]:^^.
   ```

   in which case it indicates that any epsilon node created as a result of matching the optional phrase against the null string $\epsilon$ should be promoted.

Each grammar element (terminal or nonterminal) in an **rdp** grammar has an attached promotion operator which specifies the way that the corresponding tree nodes will be built into the tree during a parse. The default operation is ^_, so in effect any grammar element without an explicit promotion operator attached has an implicit ^_ operator following it, and such a node will be processed according to the rules given in the previous section[1].

[1] For nonterminals only, new defaults may be established by applying a promotion operator to the left hand side of the rule definition.

## 10.4 A complete example

In this section we show how to apply the promotion operators to the grammar for a simple expression-based language. The original grammar (without promotions) was given above and generates trees of the form shown in Figure 9.2. This modified grammar was used to produce the much more space-efficient tree shown in Figure 9.3.

```
TREE
program ::= { statement ';'^ }.
statement ::= ID '='^^ e1.
e1 ::= e2^^ { ('+'^^^ | '-'^^^) e2 }. (* Add or subtract (LA)    *)
e2 ::= e3^^ { ('*'^^^ | '/'^^^) e3 }. (* Multiply or divide (LA) *)
e3 ::= e4^^ | ('+'^^ | '-'^^) e3.     (* Monadic positive or negative (RA) *)
e4 ::= e5 [ '**'^^ e4 ]:^^.           (* Exponentiate (RA)       *)
e5::= ID^^                            (* Variable or ...         *)
      ['('^ (e1)@','^ ')'^ ] |        (* ... function call       *)
      INTEGER^^              |        (* Numeric literal         *)
      '('^ e1^^ ')'^.                 (* Bracketed subexpression *)
```

### 10.4.1 Removing syntactic sugar

The simplest application of the promotion operators is to simply remove unnecessary tokens from the tree. The ^ operator has this effect because it promotes the node up *under* the current parent. Since the resulting node contains the label from the original parent, the contents of the new node is effectively discarded. Using this operator, then, allows syntactic sugar to be deleted from the language. In this grammar, the parenthesis tokens in rule `e5` have been treated this way. In essence, parentheses in a programming language are usually used to represent *nesting* of some sort, and of course a tree allows nesting to be structurally shown simply by making the contents of a parenthesised expression into a child node. There is no reason, therefore, for the parentheses to be retained in the tree. Another example of redundant syntactic sugar is the semicolon used to terminate statements and this is similarly deleted.

Here is a small example that shows the effect of promoting the parentheses up under their parents:

```
s ::= e ['+' s].
e ::= t ['*' e].
t ::= INTEGER | '('^ s ')'^.
```

Figure 10.1 shows the full tree that will be produced without the promotion operators when parsing the string `(2+3)*4`, and Figure 10.2 shows the result of adding the promotions.

### 10.4.2 Making operators parent nodes to their operands

Classically, parse trees represent arithmetic expressions for diadic operators as binary trees with each operator node having two children corresponding to its

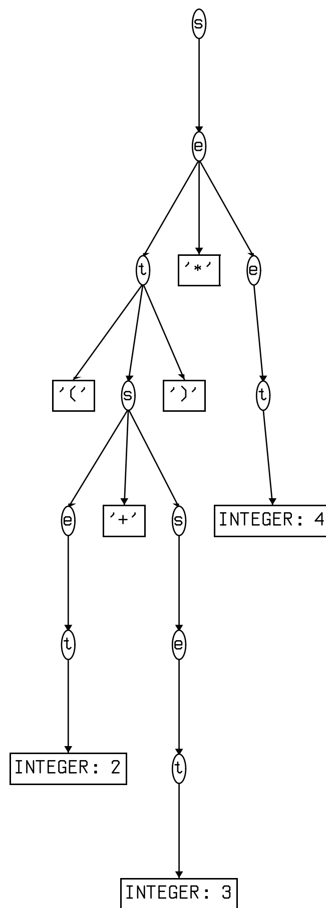


**Figure 10.1** Simple expression: full tree

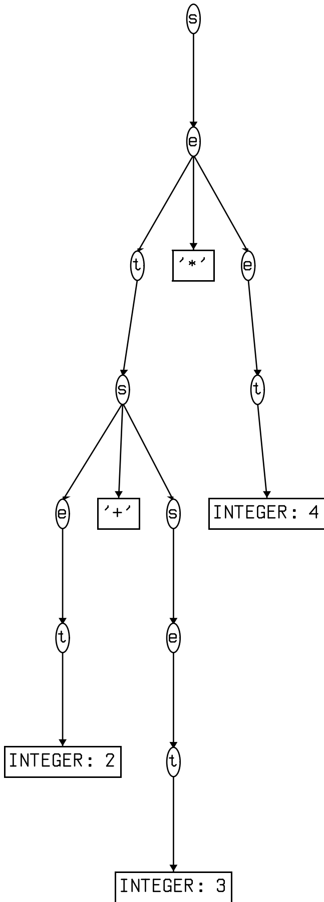


**Figure 10.2** Simple expression: result of adding promotion operators

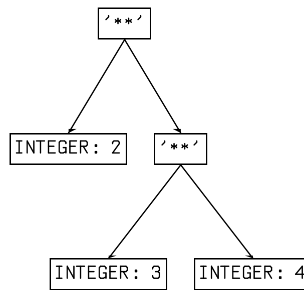


**Figure 10.3** Right associative operator tree

operands. We get this effect in `rdp` by promoting the operator node onto the parent nonterminal node using the `^^` operator. This is illustrated in rule `e4` which shows the implementation of the right associative (RA) exponentiation operator `**`. In detail, it turns out that the $\epsilon$ node created at the end of a run of RA operators must be promoted too: hence the default promotion in rule `e4`.

Here is a simplified grammar illustrating the construction of a right associative operator tree. The result of parsing the string `2**3**4` is shown if Figure 10.3.

```
s ::= e ['**'^^ s]:^^.
e ::= INTEGER^^ | '('^ s^^ ')'^.
```

## 10.4.3 Handling left associative operators

Arithmetic expressions containing left associative (LA) operators present a little more difficulty which is perhaps unfortunate since LA operators are the norm. From the parsing point of view the recursive rules used to recognise a LA operator create their tree nodes in the 'wrong' order and simply promoting the operator token node onto its parent yields a tree whose semantics do not match that of normal algebraic usage. The correct solution is to promote the operator token to the node *above* its parent, thus building that part of the tree in what amounts to a bottom-up fashion. This approach is illustrated in rules `e1` and `e2` above.

Here is a simplified grammar illustrating the construction of a left associative operator tree. The result of parsing the string `2-3-4` is shown if Figure 10.4.

```
s ::= e^^ {'-'^^^ e}.
e ::= INTEGER^^ | '('^ s^^ ')'^.
```

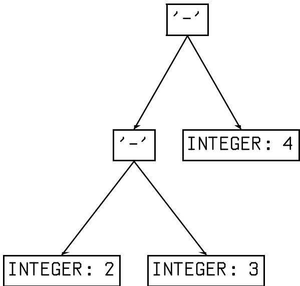


**Figure 10.4** Left associative operator tree

# Chapter 11

# Error and informational messages

All `rdp` error messages are issued *via* the `text_message()` routine which is part of the `textio` package described in [JS97b]. The routine supports four classes of message:

- *fatal errors* which cause immediate termination of the run,
- *errors* which cause termination of the run after grammar analysis unless the `-F` flag is set,
- *warnings* which may indicate a problem that should be checked and
- *informational messages* which provide feedback only and do not indicate a problem.

In each case, a message may or may not cause an *echo* of the current scanner input line, followed by an arrow indicating the current position of the scanner.

A typical message looks like this:

```
33: Error 1 (zzz.bnf): unexpected character 0x2C ',' in source file
33: bad , syntax ,
33: ----1
```

In this case an illegal character has been found in the source file. Up to nine error messages *per* line of source code may be reported, with the error messages themselves followed by an echo of the line in error and a marker line showing the location of the errors.

In detail, whenever `rdp` detects a syntax error in the source IBNF it prints out a line of the file with a digit marking the last character of the first token *after* the token that caused the error. This can be confusing if the error token is the last on a line, because the *next* line will be printed with an arrow pointing to the start of the line.

Fatal, error and warning messages are preceded by the relevant message severity. Informational messages are only preceded by a space character. This makes it easier to spot the errors by scanning the leftmost column of the output listing.

The rest of this chapter lists all `rdp` error messages in alphabetical order by class. Some of these messages can also be reported by `rdp`-generated parsers.

## 11.1 Fatal errors

`errors detected in source file`

This message is issued at the end of syntax analysis if syntax errors have been reported. It causes termination of the run before the `POST_PARSE` routine is called.

`internal error - expecting alternate`

The internal data structures representing the grammar have become corrupted. This error can only occur as a result of a programming error within `rdp`: please submit a bug report to `A.Johnstone@rhbnc.ac.uk` which includes an example IBNF file that generates the error along with a note of your computer model, operating system name and version and compiler vendor and version.

`internal error - unexpected alternate in sequence`

The internal data structures representing the grammar have become corrupted. This error can only occur as a result of a programming error within `rdp`: please submit a bug report to `A.Johnstone@rhbnc.ac.uk` which includes an example IBNF file that generates the error along with a note of your computer model, operating system name and version and compiler vendor and version.

`internal error - unexpected kind found`

The internal data structures representing the grammar have become corrupted. This error can only occur as a result of a programming error within `rdp`: please submit a bug report to `A.Johnstone@rhbnc.ac.uk` which includes an example IBNF file that generates the error along with a note of your computer model, operating system name and version and compiler vendor and version.

`no rule definitions found`

The source files processed by `rdp` did not contain any rule definitions, so there is nothing to do.

`no source file specified`

No filename was found on the command line, and a `-f` (filter mode) flag had not been issued. `rdp` prints a summary help message after issuing this error.

`run aborted without creating output files - rerun with -F to override`

Errors were detected during grammar analysis and so no output files were created. Many languages (including C and Pascal) contain at least one ambiguous rule (the `if...then...else` problem) and so *when* you are sure that all other problems in your grammar have been eradicated, rerun `rdp` with a `-F` flag which will override this message and generate the output files.

`source file not found`

The source file does not exist, or is read locked against the user. `rdp` and `rdp`-generated parsers print a summary help message after issuing this error.

`unable to open header output file '`*filename*`' for writing`

`rdp` was unable to open the named header file for writing. This may be because there is no disk space left, or there may already exist a file of that name that is write protected.

`unable to open parser file '`*filename*`' for writing`

`rdp` was unable to open the named parser file for writing. This may be because there is no disk space left, or there may already exist a file of that name that is write protected.

`unable to open VCG file '`*filename*`' for writing`

`rdp` was unable to open the named VCG output file specified with a `-V` switch for writing. This may be because there is no disk space left, or there may already exist a file of that name that is write protected.

`unrecognised option -`*c*

`rdp` or a generated parser found a command line switch it did not understand. After issuing this message `rdp` prints out a summary help page.

## 11.2 Errors

`comment delimiter tokens must be less than three characters long`

Due to the rather crude state machine used during comment parsing, comment close delimiter tokens must be one or two characters long, so the Algol-68 `comment...tnemmoc` brackets can not be handled (and quite right too in many people's opinion).

`doubly declared symbol '`*name*`'`

*name* appears more than once on the left hand side of a rule definition. Merge the rules using alternates.

`empty tokens are not allowed: use [ ... ] instead`

`rdp` does not allow use of the explicit null token `''`. Only the iteration operator `@`, the zero-or-more bracket `{ }` and optional bracket `[ ]` can introduce null rules into the grammar.

**identifier '*name*' begins with a reserved name**

All **rdp** internal identifiers begin with one of a set of ten reserved prefixes listed in Table 2.1. To avoid clashes between user identifiers and these internal names, **rdp** rejects any user defined identifiers that begin with one of those prefixes.

Error 1 (*filename*): expecting one of '*token1*', ...
*line containing an error*
-------1

This is the generic **rdp** syntax error report. After printing the error messages the line containing the errors is echoed to the error stream along with a pointer line. The pointer line contains up to nine digits that each mark the token *after* the token that has generated an error.

**identifier '*name*' is a C++ reserved word or library identifier**

Surprising compile time errors would result from declaring a rule called, for instance **register** because **register** is, of course, a C reserved word and may therefore not be used as a function name. All identifiers in the IBNF file are checked against a list of dangerous names which includes all C keywords and a few of the more common library functions. You can add extra names to the list by adding extending the definition of parameter RDP_RESERVED_WORDS in file **rdp_aux.h**

**illegal grammar element: a colon may not appear here**

A syntax error in the **rdp** source file has been detected.

**illegal grammar element: a real may not appear here**

A syntax error in the **rdp** source file has been detected.

**illegal grammar element: an integer may not appear here**

A syntax error in the **rdp** source file has been detected.

**illegal grammar element: expressions may not return values**

A syntax error in the **rdp** source file has been detected

**illegal grammar element: perhaps you intended to write '*string*'**

A syntax error in the **rdp** source file has been detected. A double quote delimited string has been found where only a single quote delimited string is allowed.

**include file '*filename*' not found**

The named include file does not exist, or is read locked against the user.

`iteration count too low`

The `rdp` rule `example ::= ('a' 'b')4@6# 'z'` matches the following strings:

```
abababababz
ababababababz
abababababababz
```

`rdp` performs this match by iterating in the `('a' 'b')` sub-rule at least four and at most six times. In detail, `rdp` iterates round the body and then checks the number of times it matched `('a' 'b')` when it eventually finds a non-match in the input. If `rdp` finds that it went round less than four times, it issues this message. So, in general, the message indicates that there were too few instances of the sub-rule in the input to meet the iterator specification.

`iterator high count must be greater than low count`

An iterator of the form `(body)5@3` is illegal because it requires `body` to be matched at least five, but no more than three, times which is meaningless.

**`LL(1) violation - rule '`*name*`' alternates`**
**`''`*alternate*`''`**
**`and ''`*alternate*`''`**
**`share these start tokens:`**
*tokens*

This is the most common LL(1) problem: a pair of alternates share at least one start terminal and so cannot be disambiguated by the parser. A simple example is

```
bad_first ::= 'a' 'b' 'c' | 'a' 'd'.
```

The error can often be eliminated by factorising the grammar, for example

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

**`LL(1) violation - rule '`*name1*`' and '`*name2*`' are both nullable`**

A construction like `['a'] ['b']` is ambiguous because `rdp` could match either bracket against a null input string.

**`LL(1) violation - rule '`*name*`'`**
**`contains null but first and follow sets both include:`**
*tokens*

When deciding whether to enter an iteration or optional bracket the parser must be able to distinguish between tokens that belong to the rules inside the bracket and those that belong to the rules following the brackets. If there are any tokens in both the *first* and *follow* sets for the subrule then the parser cannot disambiguate the brackets.

A simple example is

```
bad_null ::= 'x' {'a' 'b' 'c'} 'a'.
```

The error can sometimes be eliminated by refactorising the grammar, for example

```
good_null ::= 'x' 'a' {'b' 'c' 'a'}.
```

**LL(1) violation - rule '*name*' is left recursive**

In top-down parsers, immediate or indirect left recursion creates an infinite loop and must be eliminated.

**LL(1) violation - rule '*rule*' is nullable but contains the nullable subrule *rule***

It is illegal to nest nullable sub-rules (constructs such as `{ [ 'a' 'b' ] }` ) because `rdp` generated parsers could match either the inner square brackets or the outer braces to a null string. Rewrite as `{ 'a' 'b' }`.

**LL(1) violation - subrule '*name*' is empty**

The IBNF syntax analyser will accept a rule of the form

```
bad ::= 'a' 'b' | [* semantic action *] | 'z'.
```

but this is not meaningful IBNF, because the middle alternate will never be entered. In fact, this is effectively an empty alternate as far as the parser is concerned which is also illegal.

The only context in which alternates containing only semantic actions are allowed is the special case of a *semantic rule*. See section 5.9 for details.

**obsolete directive:**

```
obsolete directive:  HASH_PRIME replaced by SYMBOL_TABLE at version 1.4
obsolete directive:  HASH_SIZE replaced by SYMBOL_TABLE at version 1.4
obsolete directive:  INTERPRETER mode deleted at version 1.4
obsolete directive:  POST_PROCESS renamed POST_PARSE at version 1.3
obsolete directive:  PRE_PROCESS renamed PRE_PARSE at version 1.3
obsolete directive:  SET_SIZE deleted at version 1.4
obsolete scanner primitive:  ALT_ID deleted at version 1.4
obsolete scanner primitive:  NEW_ID deleted at version 1.4
obsolete scanner primitive:  NUMBER renamed INTEGER at version 1.3
```

A grammar for a previous version of `rdp` has been parsed. Replace obsolete directives and primitives.

**rule *name* is empty**

A rule with no body has been declared.

`string delimiter tokens must be exactly one character long`

Due to the rather crude state machine used when parsing strings, the close token must be exactly one character long. We would be interested to hear if you have an application that requires multi-character string delimiters.

`tokens must not contain spaces or control characters`

White space is stripped by the scanner, so a token definition in the IBNF file that contained white space or non-printing characters could never be matched by the scanner.

`undeclared symbol '`*name*`'`

A rule name has been referenced that is not defined elsewhere in the current set of input files.

## 11.3 Warnings

`grammar is not LL(1) but -F switch set: writing files`

This message appears instead of the fatal abort message when the `-F` switch is used.

`rule '`*name*`' will not appear in the output file`

It is sometimes useful to define rules that are not explicitly referenced in the grammar, typically to specify comment and string definitions. `rdp` searches the entire grammar recursively from the start rule looking for unused rules and marks them so that no equivalent code is produced in the parser output files. This avoids warning messages from the C compiler about the unused function definitions.

## 11.4 Informational messages

*count* `CPU seconds used`

In verbose mode all `rdp`-generated parsers report CPU time consumption with this message at the end of the run. Note that the figure is a measure of mill time, not elapsed time. The accuracy of the figure depends on your implementation of the ANSI C `clock()` routine. Some PC libraries are known to be a little unreliable on this score.

*count* `rules,` *count* `tokens,` *count* `actions and` *count* `subrules`

In verbose mode, `rdp` reports summary grammar statistics with this message. A *subrule* is the expansion of a grammar bracket.

`adding continuation token 'token'`

The **rdp** scanner matches *punctuation tokens* (i.e. tokens made up of non-alphanumeric characters) by repeatedly looking in the scanner symbol table and matching the longest token it can find. This strategy requires that all substrings of a token be in the symbol table, so that the token `::=` requires that `::` and `:` are also loaded. A *continuation token* is any token required for matching that has not already been declared by the user.

`checking for clashes with reserved words`

**rdp** checks all rule and attribute names to ensure that they are valid C identifiers that do not clash with C reserved words or library names. The list of names checked by **rdp** is maintained in macro **RDP_RESERVED_WORDS** which is defined in file **\rdp_supp\rdp_aux.h**. You can add names to this list by appending them to the macro definition. Note that the order of definition is not significant.

`checking for continuation tokens`

The **rdp** scanner matches *punctuation tokens* (i.e. tokens made up of non-alphanumeric characters) by repeatedly looking in the symbol table and matching the longest token it can find. This strategy requires that all substrings of a token be in the symbol table, so that the token `::=` requires that `::` and `:` are also loaded. A *continuation token* is any token required for matching that has not already been declared by the user. This message is issued in verbose mode at the start of continuation checking.

`checking for disjoint first sets`

**rdp** checks that rules are LL(1) by ensuring that the start sets of all alternates are disjoint for each rule in the grammar. This message is issued in verbose mode at the start of disjoint set checking.

`checking for empty alternates`

Syntactically it is possible for an **rdp** grammar to contain only semantic actions even when it is not part of a semantic rule. **rdp** checks all alternates and reports this error whenever it finds a rule that has no tokens in it. This message is issued in verbose mode at the start of empty checking.

`checking for nested nullable rules`

**rdp** checks that rules are not ambiguous as a result of nesting nullable subrules such as **{body}** or **[body]** within each other. This message is issued in verbose mode at the start of null rule checking.

**`checking nullable rules`**

**rdp** checks that rules are LL(1) by ensuring that the first and follow sets of each rule that can match the null token are disjoint. This message is issued in verbose mode at the start of null rule checking.

**`dumping derivation tree to VCG file to`** *'filename'*

This message is issued immediately before writing the VCG derivation tree file.

**`dumping header file to`** *'filename'*

This message is issued immediately before writing the header file.

**`dumping parser file to`** *'filename'*

This message is issued immediately before writing the main parser file.

**`entered`** *'rule-name'*

When an **rdp**-generated parser is generated using the `-R` option, the parser is enhanced to output trace information when run. Each time a grammar rule is activated, the parser prints out an entry and an exit message which can be used to track the nesting of parser function calls during a run.

**`exited`** *'rule-name'*

When an **rdp**-generated parser is generated using the `-R` option, the parser is enhanced to output trace information when run. Each time a grammar rule is activated, the parser prints out an entry and an exit message which can be used to track the nesting of parser function calls during a run.

**`follow sets stabilised after`** *count* **`passes`**

Follow set calculation usually requires a number of passes over the whole grammar. The number of passes depends on both the complexity of the grammar and the order in which the rules are listed. We would be interested in receiving a copy of any real grammar that requires more than twenty passes.

**`generating first sets`**

This message is issued immediately before the start of first set generation.

**`generating follow sets`**

This message is issued immediately before the start of follow set generation.

**`no continuation tokens needed`**

The **`rdp`** scanner matches *punctuation tokens* (i.e. tokens made up of non-alphanumeric characters) by repeatedly looking in the symbol table and matching the longest token it can find. This strategy requires that all substrings of a token be in the symbol table, so that the token `::=` requires that `::` and `:` are also loaded. A *continuation token* is any token required for matching that has not already been declared by the user. This message is issued in verbose mode at the end of continuation checking if no such tokens were needed.

**`updating follow sets`**

After the main grammar analysis takes place, **`rdp`** adds the first sets to the follow sets for iteration brackets `{...}` which aids error recovery. The follow sets then need to be recalculated.

# Chapter 12

# Understanding and debugging rdp-generated parsers

In this chapter we give a simplified overview of the structure of an `rdp`-generated parser along with some advice on using `rdp`'s facilities to monitor the execution of a running parser.

`rdp` writes out two files whenever it successfully generates a parser — a *header* file with a suffix of `.h` and a *main* file with a suffix of `.c`. These files are designed to be human-readable so that inserted semantic actions may be traced by stepping through the parser with a conventional debugger. The purpose of this section is to explain the basic techniques that are used within a parser by looking at real `rdp`-generated code.

We shall begin by looking at the parser generated by `rdp` for this small grammar.

```
start ::= INTEGER '+' expr.
expr  ::= INTEGER '*' INTEGER.
```

## 12.1 The header file

The header file contains declarations for datatypes that might be needed for use in semantic actions, such as any symbol table or tree data fields declared in the user's `.bnf` file. It also contains an `enum` declaration corresponding to the tokens declared in the grammar (described in more detail in the next section) and a macro which expands to the number of passes used in this parser. The header file for our example grammar is shown in Figure 12.1.

## 12.2 The rdp scanner

Traditional parser generators work only at the level of language tokens, and it is the user's responsibility to supply a suitable lexical analyser that digests the source text into a stream of tokens for consumption by the parser proper. One way of providing this lexical analysis function is to use a lexical analyser generator, which is rather like a scaled down parser generator with features targeted specifically at the construction of lexical analysis functions. `rdp` does

```
#include "scan.h"

/* Maximum number of passes */
#define RDP_PASSES 1

/* Token enumeration */
enum
{
RDP_T_17 /* * */ = SCAN_P_TOP, RDP_T_18 /* + */,
RDP_TT_TOP
};

/* Parser start production */
void start(void);
```

**Figure 12.1** Extracts from an `rdp`-generated header file

not need a separate lexical analyser generator: the parser generator and the lexical analyser are integrated together under the control of a single IBNF specification.

When analysing a grammar, `rdp` extracts information about tokens and uses it to parameterise the built-in scanner. This is convenient, but you should be aware that the `rdp` scanner is not completely general — in many versions of the BASIC language, for instance, string identifiers begin with a dollar sign and it is not possible to write an `rdp` grammar that enforces this rule exactly[1]. The scanner itself is a function called `scan_()` the source for which resides in `rdp_supp/scanner.c`. In principle new kinds of lexical structure can be defined by adding in new sections of code, but this turns out to require a good understanding of `rdp`'s internals so you might like to contact the authors for advice before embarking on this course.

### 12.2.1 The token enumeration

`rdp` makes a list of all the tokens used and then writes out a C enumeration which has the effect of allocating a unique integer value to represent each token. All grammars automatically include the scanner elements listed in Chapter 4 and so the first sixteen or so elements of the token enumeration are allocated to the scanner primitives. The scanner element enumeration `enum scan_primitive_type` is defined in `rdp_supp\scan.h`: each element name comprises the string `SCAN_P_` concatenated with the name of the scanner element as used within the `rdp`-IBNF language.

```
enum scan_primitive_type
{
```

[1] It is easy to define an identifier such as `id ::= '$' ID.` but this will accept 'identifiers' with a space between the `$` sign and the rest of the identifier.

```
  SCAN_P_IGNORE, SCAN_P_ID, SCAN_P_INTEGER, SCAN_P_REAL,
  SCAN_P_CHAR, SCAN_P_CHAR_ESC,
  SCAN_P_STRING, SCAN_P_STRING_ESC,
  SCAN_P_COMMENT, SCAN_P_COMMENT_VISIBLE,
  SCAN_P_COMMENT_NEST, SCAN_P_COMMENT_NEST_VISIBLE,
  SCAN_P_COMMENT_LINE, SCAN_P_COMMENT_LINE_VISIBLE,
  SCAN_P_EOF, SCAN_P_EOLN, SCAN_P_TOP
};
```

There are two elements in this enumeration which do not correspond with real scanner elements. The first `SCAN_P_IGNORE` is used by the scanner when it finds an illegal character or a comment in the source file. It is a signal to the scanner to 'go round again' and attempt to find a new, valid, token. The parser proper will never see an instance of this pseudo-token because the scanner will keep restarting until it finds something other than an `IGNORE` element.

The second pseudo-element is `SCAN_P_TOP` which simply takes a value one more than the highest real scanner element. Its value is then used to initialise the first element of the user-defined token enumeration written to the header file:

```
enum {RDP_T_17 /* * */ = SCAN_P_TOP, RDP_T_18 /* + */, ...
```

This ensures that the scanner elements and the tokens from the grammar are mapped to a contiguous sequence of integer values.

### 12.2.2 Interaction between the scanner and the parser

The parser calls the scanner function `scan_()` each time it needs to read a new token from the input file. The scanner begins by reading and discarding any whitespace characters (such as tabs, spaces and, for parsers which do not use the `EOLN` scanner element newline characters). The scanner then reads characters until a complete lexeme has been recognised. It loads a global variable called `text_scan_data` with a string containing the lexeme itself and an integer value from the token enumeration indicating which token the scanner has recognised. In what follows we shall refer to this global variable as the *scanner variable*.

The scanner variable is a structure containing several fields, not all of which are used by every token. The file `scan.h` contains the relevant definitions. In the case of scanner elements such as `INTEGER` which can return a synthesized attribute value, the scanner is also responsible for calculating the attribute and loading that into the scanner variable.

The scanner itself makes use of the routines in the text handling library `text.c` to read the source input file, deal seamlessly with nested input files and handle the generation of an output listing.

After the scanner variable has been loaded, the scanner returns control to the parser which must decide which rule expansion to use next on the basis of the contents of the scanner variable and the present state of the derivation. The scanner variable effectively provides a single token of lookahead, and as

such must be loaded *before* a parser function is started. In `rdp`, as in most top-down parsers, the parser functions assume that the scanner variable has been correctly initialised before entry to the parser function and the function takes responsibility for leaving the scanner variable correctly set up for the next parser function. This leads to a somewhat counter-intuitive organisation in which the scanner is called at the end of each parser function rather than at the beginning. In the next section we shall look at the details of a complete parser which will make this point clearer.

## 12.3 The main file

The `rdp`-generated main file contains the parser proper. Figure 12.2 shows extracts from the main file for the grammar given at the beginning of this chapter. The full file contains declarations and initialisation code for the FIRST and STOP sets for each parser function. You can read more about the calculation of these sets in the tutorial manual [JS97c] or in most standard compiler texts.

Apart from the initialisation code, an `rdp`-generated main file contains exactly one function for each nonterminal declared in the grammar (called the parser function for that nonterminal) plus a `main()` function that initialises the text and scanner subsystems before calling the function corresponding to the start nonterminal. Each parser function must

- ◇ assume that a (possibly empty) section at the beginning of the input has already been read by the scanner and matched against rules in the grammar by the parser functions,
- ◇ assume that the scanner variable has already been loaded with the first token to be matched against the current parser function's rule in the grammar,
- ◇ by looking at the current contents of the scanner variable decide which of the alternate productions within the rule is to be matched,
- ◇ match the rule against the input, calling the scanner each time a token is successfully matched so as to advance to the next token,
- ◇ ensure that at the end of a successful match the scanner variable is loaded with the first token to be matched by the succeeding parser function,
- ◇ in the case of an unsuccessful match generate an error message and attempt to read tokens from the input until the parser function sees a plausible place for parsing to continue.

### 12.3.1 Implementing parser functions

Parser functions make use of three functions from the scanner module:

- ◇ `void scan_(void)` the scanner function which has been described above,

```
static void expr(void)
{
  {
    scan_test(NULL, SCAN_P_INTEGER, &expr_stop);
    scan_();
    scan_test(NULL, RDP_T_17 /* * */, &expr_stop);
    scan_();
    scan_test(NULL, SCAN_P_INTEGER, &expr_stop);
    scan_();
    scan_test_set(NULL, &expr_stop, &expr_stop);
  }
}

void start(void)
{
  {
    scan_test(NULL, SCAN_P_INTEGER, &start_stop);
    scan_();
    scan_test(NULL, RDP_T_18 /* + */, &start_stop);
    scan_();
    expr();
    scan_test_set(NULL, &start_stop, &start_stop);
  }
}

int main()
{

.....  /* Some initialisation code omitted */

  for (rdp_pass = 1; rdp_pass <= RDP_PASSES; rdp_pass++)
  {
.....  /* Pass level initialisation (including source file opening) omitted */

    scan_();
    start();            /* call parser at top level */
    if (text_total_errors() != 0)
      text_message(TEXT_FATAL, "error%s detected in source file\n", text_total_errors() == 1 ? ""
  }
  text_print_total_errors();

.....  /* Clean up code omitted */

  return rdp_error_return;
}
```

**Figure 12.2** Extracts from an `rdp`-generated parser main file

◇ `int scan_test(const char *production, const int valid, set_ * stop)` and

◇ `int scan_test_set(const char *production, set_ * valid, set_ * stop)`.

The two `..._test` functions check the contents of the scanner variable against a supplied parameter called `valid`. The only difference between them is that `scan_test` checks against a single valid token value and `scan_test_set` checks against a set of of valid tokens.

The two parameters `production` and `stop` are used to control the generation of error messages in the case of a mismatch. If only a simple test is required then both parameters will be set to `NULL`, but if the `stop` parameter is non-null then *when* the test function finds a mismatch between the current token and the `valid` token or token set it issues an error message before returning `false`. Immediately after issuing the error message an attempt is made to resynchronise the input stream by using the scanner to read and discard tokens until a token is found that is in the `stop` set. All stop sets are initialised to contain at least the end-of-file token so that the synchroniser will not go into an infinite loop at the end of of the source file. In detail, the stop set for a parser function is the follow set for the corresponding grammar rule UNION the end of file token. You can use the `-e` flag to instruct `rdp` to display the stop sets for each nonterminal of a grammar. The `production` parameter is a simple character string that `rdp` uses to pass the name of the currently executing parser function to the error handling routine. It is always `NULL` unless the `-E` flag has been used to ask `rdp` to add the current rule name to error messages.

The body of the `start` production shows how these routines are used in practice. The corresponding grammar rule is

```
start ::= INTEGER '+' expr.
```

and `rdp` writes out the following lines

```
scan_test(NULL, SCAN_P_INTEGER, &start_stop);
scan_();
scan_test(NULL, RDP_T_18 /* + */, &start_stop);
scan_();
expr();
scan_test_set(NULL, &start_stop, &start_stop);
```

The function call `scan_test(NULL, SCAN_P_INTEGER, &start_stop)` asks for the current scanner variable to be tested against `SCAN_P_INTEGER` and if a valid integer such as `23` or `0xFF` is *not* found then orders an appropriate error message to be issued after which the input should be resynchronised on the set of tokens `start_stop`. Assuming the test did succeed then the scanner is called to get the next token which is tested against `RDP_T_18` (the token enumeration symbol for the `+` lexeme). If that succeeds then the `expr()` parser function is called. The last thing that each parser function does is to test that the scanner variable has been loaded with a token that can validly follow the corresponding nonterminal by testing against its `_stop` set.

## 12.4 Selecting alternate productions

If a grammar rule has more than one alternate production then the FIRST sets for the productions are used to control the selection of a production for matching. Here is a small grammar that illustrates the process:

```
multi ::= A 'b' 'c' | X 'y'.
A ::= 'a' | 'aa'.
X ::= 'x' | 'y'.
```

When **rdp** processes a grammar rule with multiple alternate productions it gives each of them a name comprising the prefix **rdp_** followed by the name of the rule followed by an integer which is incremented after each use. The two alternate productions in rule **multi** are therefore called **rdp_multi_0** and **rdp_multi_1**. **rdp** will calculate FIRST sets which it names **rdp_multi_0_first** and **rdp_multi_1_first**.

The generated parser function for rule **multi** is as follows.

```
void multi(void)
{
  {
    if (scan_test_set(NULL, &rdp_multi_0_first, NULL))
    {
      A();
      scan_test(NULL, RDP_T_b, &multi_stop);
      scan_();
      scan_test(NULL, RDP_T_c, &multi_stop);
      scan_();
    }
    else
    if (scan_test_set(NULL, &rdp_multi_1_first, NULL))
    {
      X();
      scan_test(NULL, RDP_T_y, &multi_stop);
      scan_();
    }
    else
      scan_test_set(NULL, &multi_first, &multi_stop)     ;
    scan_test_set(NULL, &multi_stop, &multi_stop);
   }
}
```

The **if** statements here act as gateways to the different branches of the the rule. If the current value of the scanner variable is a token that is in **rdp_multi_0_first** then the first branch will be taken and parser function **A()** will be called. If not, then the scanner variable is tested against **rdp_multi_1_first** and if successful then the second branch is taken. If neither branch is taken then an error has occurred and the production of an error message is forced by testing the scanner variable against the FIRST set for the whole rule **multi_first**.

## 12.5 Parsing iterators

An iterator is created either explicitly by using the iterator operator `@` or by using one of the four bracket forms `(  )`, `[  ]`, `{  }` `and \verb¡` ¿+. As for alternate productions, **rdp** gives each a unique name and calculates a FIRST set for it. The high and low iteration counts, the delimiter token and the FIRST set for the iterator are then used to control the parsing with the help of a `while` loop. Here is an iterator rule that uses most of the features of the iterator operator:

```
iter ::= ('a')3@4 'b'.
```

The generated parser function for rule `iter` is as follows.

```
void iter(void)
{
  {
    { /* Start of rdp_iter_1 */
      unsigned long rdp_count = 0;
      while (1)
      {
        {
          scan_test(NULL, RDP_T_a, &iter_stop);
          scan_();
        }
        rdp_count++;
        if (rdp_count == 4) break;
        if (SCAN_CAST->token != RDP_T_b) break;
        scan_();
      }
      if (rdp_count < 3)  text_message(TEXT_ERROR_ECHO,
                            "iteration count too low\n");
    } /* end of rdp_iter_1 */
    scan_test_set(NULL, &iter_stop, &iter_stop);
  }
}
```

A local variable `rdp_count` is declared to keep a count of the number of times the body of the iterator has been successfully matched. The iterator loop is implemented as an 'infinite' `while (1)` loop which contains `break` statements that can cause control to be transferred out of the loop This rather inelegant arrangement is used because there are several different conditions that can cause loop termination and it is more efficient to simply break out of the loop than to, say, set a flag to be tested at the bottom of the loop.

On entry to the iterator loop, the code performs a match against the body of the iterator rule: in this case a simple match against the token `a`. Upon successful matching, the counter `rdp_count` is incremented and if its value exceeds the high limit for the iterator (4 in this case) then the loop terminates. Otherwise a test is performed to see if the current scanner symbol is the delimiter

token for the iterator (`b` in this case). If not, the loop terminates otherwise the iterator body is matched again. Once the iterator loop does finally exit, a test is made to ensure that the loop counter exceeds the low value for the iterator, otherwise an error message is issued.

**rdp** applies a series of optimisations to the generation of iterator parser functions which are not documented in detail here. As a simple example: if the iterator high and low values are integers in the range of $0 \ldots 1$ then it is never necessary to keep a track of the number of iterator loops so the code associated with the variable **rdp_count** is omitted. Although these optimised forms of the iterator template all differ from the version documented, each iterator function follows the same general form and is flagged up in the code with a comment of the form **/* Start of rdp_... */**. You might find it interesting to try different kinds of iterator and examine the generated code.

## 12.6 Debugging rdp-generated parsers

Debugging a machine generated parser is always more challenging than debugging a normal human-written program because of the multiple levels at which errors can be introduced and the difficulty of distinguishing between an error in the grammar proper and an error in the semantic actions inserted into the grammar. In this section we shall distinguish between the different kinds of error that can arise in terms of the point within the process at which the error will be detected and give advice on the use of **rdp**'s facilities to make the task of diagnosing the cause the error easier. Errors can be detected at the following times.

- During the initial parse of the IBNF specification as a result of syntax errors or because of illegal use of **rdp**'s features, such as requesting an iterator of the form `4@3` in which the low bound is higher than the high bound.
- During the grammar analysis phase, in which LL(1) violations of various forms may be reported.
- During string testing of the parser. The generated parser may turn out to accept inputs that are not legal in the intended language or reject inputs that are. In either case, this indicates a mismatch between the language the designer had in mind and the language specified by the grammar processed by **rdp**.
- During testing of the semantic actions of a parser.

## 12.7 Errors reported by rdp when parsing a specification

Syntax errors in the IBNF specification are detected by **rdp** and reported using the standard error reporting mechanism. New users are most often caught out by one of the following common errors.

1. Using the Pascal-style assignment operator `:=` instead of the **rdp**-IBNF 'expands-to' symbol `::=`.
2. Omitting the full stop (period) at the end of each production.
3. Quoting nonterminals instead of terminals.
4. Quoting scanner elements. A grammar containing a rule of the form `item ::= ID | 'INTEGER'.` is perfectly acceptable to **rdp** but will *not* match to integers: it will attempt to match against the keyword **INTEGER** instead.
5. Putting in superfluous commas within parameter lists and directives. **rdp**-IBNF does not use the comma in any context.

## 12.8 LL(1) errors reported by `rdp` during the analysis phase

**rdp** will, of course, reject grammars that have empty alternates or that make use of production rules that have not been declared since it assumes that these grammars are incomplete. **rdp** will also reject a grammar that contains two or more definitions for the same rule. You must combine such multiple rules together with the alternate operator (|).

There is a set of errors that **rdp** may detect whilst analysing a grammar. Even a well formed grammar (in the sense of the previous paragraph) may be unacceptable because **rdp**-generated parsers can only handle grammars that may be parsed top-down using a single symbol of lookahead. In practice this means that at every point in the grammar where a running parser may have to choose between two or more courses of action it must be able to make a decision simply by looking at the single lookahead token held in the scanner variable. The three broad classes of problem and the best approach to their correction are as follows.

◇ Left recursive rules, that is ones that may call themselves before consuming any input tokens are not allowed. Many left recursive constructions can be recast as iterators and thus converted to acceptable grammars. Left recursion removal is a common requirement if you are adapting grammars that were developed for bottom up parser generators such as YACC, because these tools do allow left recursive rules.

  There do exist standard algorithms for left recursion removal. Unfortunately these algorithms in general remove the left recursion but introduce other forms of LL(1) error into the resulting grammar so they are not a panacea, in spite of the claims occasionally made in their favour. In fact some left recursive grammars have no simple counterpart which is LL(1) although in practice most grammars can be massaged into the necessary form.

  The most common source of such problems is in the description of operator expressions with left and right associativity. We suggest that you study

the examples in the tree generation chapter of this manual and in the tutorial manuals, and copy the techniques used there.

- Alternate productions within a rule must start with different tokens, that is their FIRST sets must be disjoint. A rule such as

  ```
  bad ::= 'a' 'b' 'c' | 'a' 'y' 'z'.
  ```

  has two alternate productions both of which starts with the token `a`. It can be recast using left factorisation to be acceptable to `rdp` as follows:

  ```
  good ::= 'a' ('b' 'c' | 'y' 'z').
  ```

- Iterators with a low bound of zero (and that includes the `[ ]` and `{ }` bracket shorthands) can match the empty string. If the body of the iterator can start with tokens that can also follow the iterator then the parser cannot know, in general, whether the existence of such a token on the input indicates that it should step into or step over the iterator. As a result, for rules and iterators that can match epsilon, the FIRST and FOLLOW sets must be disjoint.

## 12.9 Refining a grammar

Once you have a compiled parser, you may find that it does not behave as you intended. If your parser rejects strings that it 'ought' to accept, or accepts strings that it should reject then before tracing the code try checking that all alternates are correctly separated by alternate operators (|). In a long list of alternates it is easy to leave one of the bars off:

```
alpha ::= 'a' |
          'b'
          'c' |
          'd' .
```

This rule does not accept the language { `a, b, c, d` } which is perhaps what was intended but rather the language { `a, bc, d` }.

Assuming that no typographic errors are found at this stage then it will be necessary to trace the behaviour of the parser. `rdp` provides a range of levels at which tracing may be performed.

### Examining scanner lexemes with the -s flag

If you run the generated parser with a `-s` flag, the scanner will report the value of every lexeme seen by the parser. This is particularly useful if your commenting convention is causing the problem or if the handling of newlines is suspect. We recommend that you use the `-l` option in tandem with `-s` so as to generate a line-by-line listing as well, otherwise it can become hard to follow the output.

### Adding rule names to error messages with the -E flag

Just seeing the stream of lexemes processed by the parser gives little information on the state of the parser at the point of error. If you regenerate the parser with `rdp` and add a `-E` flag to the `rdp` command line then all syntax error messages produced by the generated parser will include the name of the rule that was being matched when the error occurred. This can be a great help in tracking down the part of the grammar that is in error. Note that the use of this flag is very different to that of the `-s` flag above: all `rdp`-generated parsers have the `-s` flag built in but the `-E` flag is a command to `rdp` that governs the way that it generates the parser, so you must regenerate the parser from scratch to see its effect.

### Enabling a full parser trace with the -R flag

If all else fails, you can instruct `rdp` to generate extremely verbose parsers that issue a message very time a parser function is entered or exited. This allows a complete parse to be traced, but can generate very long listings.

### Tracing with VCG

An alternative way of tracing a parse is to enable tree generation using one of the directives described in Chapter 9 and examine the parse tree with VCG, although you will of course need to be using a computer that supports VCG (such as a PC running Windows or an X-windows Unix system).

## 12.10 Debugging semantic actions

`rdp` blindly copies semantic actions in the IBNF specification into the generated parser and does not attempt to check them for syntactic or logical correctness. If you write a syntactically incorrect action, by for instance omitting the semicolon at the end of a C-language statement then it is only when you attempt to compile the generated parser that you will receive an error message, and the details of the message you receive depends on the particular compiler that you are using.

### Inhibiting semantic actions with the -p flag

You are strongly advised to check that your parser is functioning correctly as a standalone parser before attempting to check the semantics. To make this easier, `rdp` has an option to suppress the copying of semantic actions into the generated parser main file: if you generate the parser by running `rdp` with the `-p` flag then all semantic actions in the IBNF file will be ignored.

### Tracing parsers with a debugger

Once the semantic actions have been introduced into the parser then it is best to use the debugging facilities of your compiler to trace the behaviour of the parser.

We have successfully used the Microsoft and Borland integrated development environments and, on Unix, the `gdb` debugger with the GNU compilers. We suggest that you set a breakpoint on the parser function corresponding to the start rule and then single step through the parser whilst looking at a short input string.

## Examining the contents of symbol tables with the `-S` flag

If your parser makes use of the built-in symbol table handler then you can order the parser to print out the contents of its hash-coded symbol tables at the end of a run by adding a `-S` flag to the command line. This is useful for checking that identifiers are being added correctly and also to ensure that the tables are not becoming congested. We recommend that the length of each hash bucket is kept to around three–four on average. Longer chains indicate that the size of the table should be increased, although the tables will operate correctly even if they are overfull: there will simply be a performance penalty.

# 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/rdp**$x_y$**.tar** or if you are an MS-DOS user download **pub/rdp/rdp**$x_y$**.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


[Bac60] J. W. Backus. The syntax and semantics of the proposed International Algebraic Language of the Zurich ACM-GAMM conference. In R. Oldenburg, editor, *Proc. Internat'l Conf. Information Processing, UNESCO, Paris, 1959*, pages 125–132, London, 1960. Butterworths.

[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_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.

[JS97c] Adrian Johnstone and Elizabeth Scott. A tutorial guide to `rdp` for new users. Technical Report TR-97-24, Royal Holloway, University of London, Computer Science Department, December 1997.

[San95] Georg Sander. *VCG Visualisation of Compiler Graphs.* Universität des Saarlandes, 66041 Saarbrücken, Germany, February 1995.