rdp_supp – support routines for the rdp compiler compiler

User manual for version 1.5

Adrian Johnstone Elizabeth Scott

Technical Report

CSD–TR–97–26

December 20, 1997

Royal Holloway University of London

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

Abstract

rdp is a system for implementing language processors. It accepts a grammar written in an extended Backus-Naur Form annotated with inherited and synthesized attributes and C-language semantic actions. rdp checks that the grammar is LL(1), providing detailed error messages pinpointing the source of any problems. A parser written in ANSI C may then be output. rdp is particularly suited to student use because it constructs ready to run parsers and interpreters and provides detailed diagnostics.

This report describes the user accessible parts of the rdp support library — rdp_supp for short. rdp_supp comprises seven packages to manage memory (memalloc.c), sets (set.c), graphs (graph.c), scanners (scan.c), text buffers (textio.c), command line arguments (arg.c) and symbol tables (symbol.c). These packages are designed to be useful for general purpose programming and may be used independently of the rdp system. The internal operation of the packages is not documented here, but the rdp distribution pack does include commented source code for all parts of the system.

The rdp source code is public domain and has been successfully built using Borland C++ 3.1 and Microsoft C++ version 7 on MS-DOS, GNU gcc and g++ running on OSF/1, Ultrix, MS-DOS, Linux and SunOS, and a variety of vendor's own compilers running on many flavours of Unix. rdp has also been built for the Macintosh using a console support library, and a version modified to take account of the unusual filenaming conventions on the Acorn Archimedes is available on request.

The version of rdp described here is version 1.50 dated 20 December 1997.

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

Contents

Chapter 1

Introduction

rdp is a system for implementing language processors. Compilers, assemblers and interpreters may all be written in the **rdp** source language (an extended Backus-Naur Form) and then processed by the **rdp** command to produce a program written in ANSI C which may then be compiled and run.

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 set of routines for creating and manipulating general graph data structures which can also output graphs in a form that may be displayed on-screen by the VCG tool (`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 set of routines for processing command line arguments and automatically building help routines (`arg.c`),
- ◇ scanner support routines for testing tokens in recursive descent parsers (`scan.c`).

Writing effective language processors in **rdp** requires a detailed understanding of these modules. This report documents the user accessible functions in the **rdp_supp** library. Implementation details are hidden except where a knowledge of the underlying data structures is required for efficient exploitation of the library. The full source code of the **rdp_supp** library is available in directory **rdp_supp** of the **rdp** distribution, and the various **rdp** tools provide examples of the use of **rdp_supp** routines.

This manual is part of a four manual series. In addition to this support library manual, the user manual [JS97b] describes the **rdp** source language, command switches and error messages.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.

Chapter 2

arg – command line argument services

The **arg** library provides automatic processing for Unix style command line arguments. The library is used to implement **rdp**'s **ARG_...** directives in which command line switches are associated with variables in the parser called *switch variables*. When the command line is processed, the switch variables are loaded with values from the command line switches supplied by the user.

The **arg** library is set up at run time by calling one of a family of routines to declare command line switches. When all of the switches have been set up, the command line can be processed by passing the normal **argc** (argument count) and **argv** (argument vector) parameters from the ANSI-C **main()** function to the library. Each command line switch has an associated *switch variable* which will be updated during command line processing and a *description string* which gives a short summary message describing the switch's function. The library creates a *help message* by concatenating these descriptions which may be issued along with a fatal error message if an invalid command line is detected.

2.1 Command line format

The model supported by the library is that of a command line made up of *file arguments* and *switches* separated by spaces made up of space or tab characters. Switches are distinguished by a leading minus sign (`-`). Any space delimited field beginning with a `-` character is a switch and anything else is a file argument.

Switches are distinguished one from another by their *key character* which immediately follows the `-` character. Switches are processed in strict left-to-right order as they appear on the command line and may be of three types.

1. *Boolean switches* declared using the function **arg_boolean** which take an integer switch variable that is initialised to FALSE (integer 0). Each instance of the boolean switch in the command line toggles the state of the switch variable by exclusive OR-ing its value with logical TRUE.

2. *Numeric switches* declared using the function **arg_numeric** which take an unsigned long switch variable that is initialised to zero. Each instance of the numeric switch in the command line must be immediately followed by a decimal integer without any intervening spaces. The ASCII coded

number on the command line is converted to binary and loaded into the switch variable, overwriting any previous value.

3. *String switches* declared using the function `arg_string` which take a string (`char *`) switch variable that is initialised to `NULL`. Each instance of the numeric switch in the command line must be followed by a string of characters which will be collected and loaded into the switch variable, overwriting any previous value. No intervening spaces are allowed between the switch key and the actual string: a string switch key followed by a space will be interpreted as an empty (zero length) string parameter.

2.2 The help message

It is usual to provide a summary help message that can be issued by a program if it receives invalid command parameters. The `arg` library automatically constructs such a message by concatenating the *description lines* from the declared command line switches. The routine `arg_help()` may be called to issue this message.

2.3 arg_boolean

```
void arg_boolean(char key, char* description, int *intvalue)
```

Declare a boolean switch with key character `key`, help message `description` and switch variable `intvalue`.

2.4 arg_help

```
void arg_help(char *msg)
```

Issue a fatal error message `msg` followed by the help message formed by concatenating the description lines from each declared command line switch. The program exits after calling this function with exit status `EXIT_FAILURE`.

2.5 arg_message

```
void arg_message(char* description)
```

Declare a line to be added to the help message without an associated command line switch. This function is useful for adding blank spacing lines or titles and other general information to the help message.

2.6 arg_numeric

```
void arg_numeric(char key, char* description, unsigned long *unsignedvalue)
```

Declare a numeric switch with key character `key`, help message `description` and switch variable `unsignedvalue`.

2.7 arg_process

```
char ** arg_process(int argc, char *argv[])
```

Process the command line parameters held in **argv** according to the switches declared using the switch definition functions. All the non-switch (filename) arguments are collected into an array of pointers to strings (a **char **** variable) which is returned by the function. If no filename arguments are seen, then NULL is returned.

2.8 arg_string

```
void arg_string(char key, char* description, char **str)
```

Declare a string switch with key character **key**, help message **description** and switch variable **str**.

2.9 An example program

The example shown in Figure 2.1 is an extract from the source of the **rdp** tool which illustrates the use of most of the **arg** routines. The output produced by the **arg_help()** function when **rdp** is called with no source file is shown in Figure 2.2.

2.10 Limitations

Unix commands use a wide variety of conventions for command line switches, not all of which are supported by the **arg** library. Here is a list of such limitations.

1. Command line switches can only be of the three kinds described above: there is no built-in facility for real number switches, for instance, although a string switch could be used to collect the characters for later processing.
2. There is no straightforward way to allow embedded spaces in string switches. This is a side-effect of the way in which the ANSI-C standard command line handler parses the fields in a command line.
3. Command line switch keys can only be made up of a single character.
4. No spaces are allowed between a key and its argument.
5. There is no way to associate command line switches with particular file parameters. Consider, for instance a switch **-l** which is intended to switch on the listing for a source file. It would be reasonable to interpret a command line of the form

   ```
   mytool first_file -l second_file third_file -l
   ```

```
arg_message("Recursive descent parser generator V1.50 (c) Adrian Johnstone 1997\n\n"
            "Usage: rdp [options] source[.bnf]");
arg_message("");  /* Add a blank line to the help message */
arg_boolean('f', "Filter mode (read from stdin and write to stdout)", &rdp_filter);
arg_boolean('l', "Make a listing", &rdp_line_echo);
arg_string ('o', "Write output to filename", &rdp_outputfilename);
arg_boolean('s', "Echo each scanner symbol as it is read", &rdp_symbol_echo);
arg_boolean('S', "Print summary symbol table statistics", &rdp_symbol_statistics);
arg_numeric('t', "Tab expansion width (default 8)", &rdp_tabwidth);
arg_numeric('T', "Text buffer size in bytes for scanner (default 20000)", &rdp_textsize);
arg_boolean('v', "Set verbose mode", &rdp_verbose);
arg_string ('V', "Write derivation tree to filename in VCG format", &rdp_vcg_filename);
arg_message("");
arg_boolean('e', "Write out expanded BNF along with first and follow sets", &rdp_expanded);
arg_boolean('E', "Add rule name to error messages in generated parser", &rdp_error_production_name);
arg_boolean('F', "Force creation of output files", &rdp_force);
arg_boolean('p', "Make parser only (omit semantic actions from generated code)", &rdp_parser_only);
arg_boolean('R', "Add rule entry and exit messages", &rdp_trace);
arg_message("");
arg_message("You can contact the author (Adrian Johnstone) at:");
arg_message("");
arg_message("Computer Science Department, Royal Holloway, University of London");
arg_message("Egham, Surrey, TW20 OEX UK. Email: A.Johnstone@rhbnc.ac.uk");

rdp_sourcefilename = *arg_process(argc, argv);

if (rdp_sourcefilename == NULL)
   arg_help("No source file specified");
```

Figure 2.1 Example usage of the arg library

```
Fatal - No source file specified

Recursive descent parser generator V1.50 (c) Adrian Johnstone 1997
Generated on Dec 20 1997 12:04:45 and compiled on Dec 20 1997 at 12:02:49

Usage: rdp [options] source[.bnf]

-f     Filter mode (read from stdin and write to stdout)
-l     Make a listing
-o <s> Write output to filename
-s     Echo each scanner symbol as it is read
-S     Print summary symbol table statistics
-t <n> Tab expansion width (default 8)
-T <n> Text buffer size in bytes for scanner (default 20000)
-v     Set verbose mode
-V <s> Write derivation tree to filename in VCG format

-e     Write out expanded BNF along with first and follow sets
-E     Add rule name to error messages in generated parser
-F     Force creation of output files
-p     Make parser only (omit semantic actions from generated code)
-R     Add rule entry and exit messages

You can contact the author (Adrian Johnstone) at:

Computer Science Department, Royal Holloway, University of London
Egham, Surrey, TW20 0EX UK. Email: A.Johnstone@rhbnc.ac.uk
```

Figure 2.2 Output from the `arg_help()` function as an instruction to process files the three files **first_file**, **second_file** and **third_file** with the source listing being switched on for the first and third files but switched off for the second file. However, the **arg** library processes all command line switches in left to right order and then returns the file parameters in a block, so the interleaving of command line arguments and file parameters is not preserved.

Chapter 3

`graph` – a graph data structure handler

A *graph* is a collection of nodes and edges, often drawn as a collection of round nodes and arrows representing the edges. There may or may not be data associated with individual nodes and edges. In a general graph, there is no limit to the number of edges leaving or entering a node, and there is no limit of the number of nodes in a graph.

Graphs are fundamental objects in computing, being used to represent relationships between objects. Special cases of graphs, such as *linked lists* or *trees* have restrictions on the number of edges that may enter or leave nodes and the kinds of paths that may be traced through the graph. A *singly linked list*, for instance is a collection of nodes each with either one or zero edges entering and one or zero edges leaving. Every node in the list has exactly one edge entering it and one edge leaving it except for one node (called the head) which has no edge entering it and one node (called the tail) which has no edge leaving it.

These special cases along with the properties of more general graphs are described in most standard books on data structures. The `graph` library described in this chapter provides a completely general mechanism for implementing unrestricted graphs in an efficient manner. It is possible to provide more space (and time) efficient implementations of some important special cases such as trees of fixed order, queues and circular buffers but the implementation used here is the cheapest simple method we know of for handling completely unconstrained graphs.

Graphs can be very complex, and debugging a program which is based on graph structures can be hard because tracing through the edges using a conventional ANSI-C debugger is confusing and time consuming. A major advantage of the `rdp graph` library is that any graph can be output as a text file written in the language of the `VCG` graph visualisation tool. VCG can display a graph at various resolutions, trace graphically through the nodes and edges of a graph and format the graph for printing on a wide variety of devices. The various tree diagrams shown in the `rdp` manuals were produced in this way: `rdp` uses the graph library to build parse trees and `rdp` generated parsers provide a `-V` command line switch which is used to output the tree in VCG compatible format. The VCG tool is not a part of the `rdp` distribution but the author of VCG has kindly given his permission for VCG to be distributed alongside `rdp` — you will find versions for Windows or Unix in the `rdp` FTP server as described in Appendix A.

3.1 Internal structure of a graph

The `graph` library uses a hierarchy of linked lists of *graph atoms* to represent graphs that may be manipulated using a family of routines for inserting and deleting nodes and edges. A graph atom may be used to represent

1. a graph header,
2. a graph node, or
3. a graph edge.

Each atom has a predecessor and a successor pointer which are used to form doubly-linked lists of atoms representing the same kind of atom and an ancillary pointer which is used to point to atoms of another type. Each atom also has a unique number which may be displayed as part of a graph dump. The number is only present to aid debugging: it is not used by any graph routine and may not be changed during a run.

The library maintains a single doubly-linked list of graph headers, one header for each graph in use by the program. Graphs may be added or deleted during program execution and the graph list may be empty. The ancillary pointer of each graph header points to a doubly-linked list of graph nodes, which may be empty. Each graph node represents a single, unique, node in the graph represented by the parent header. The ancillary pointer of each graph node points to a doubly-linked list of graph edges, which may be empty. There will be one graph edge in the list for each edge leaving that node. The ancillary pointer for each edge points to the node that the edge is directed towards. In this representation, edges are inherently unidirectional: an undirected graph may be represented inserting both forward and reverse edges between each related pair of nodes.

Figure 3.1 shows an example of a small derivation tree generated by the `minitree` compiler and its internal representation using graph atoms is shown in Figure 3.2. In these examples, no data is associated with the edges. For graph atoms without data, the VCG representation shows the type of the atom followed by a colon and the unique number of the atom. Hence, in Figure 3.2 the graph header node is labeled `Graph:1` and the edges are labeled as `Edge:5` and so on. The *nodes* in the graph do have user supplied labels (in this case, the `minitree` compiler that produced the derivation tree has labeled the nodes with the scanner lexeme for terminals or the rule name for non-terminals) and these are used as VCG node labels where they exist.

3.2 Graph data and handles

All graph atoms can carry data, be they graph headers, nodes or edges. When a graph, node or edge atom is inserted into the current set, extra space can be reserved for the user data in that atom. It is not possible to change the size of

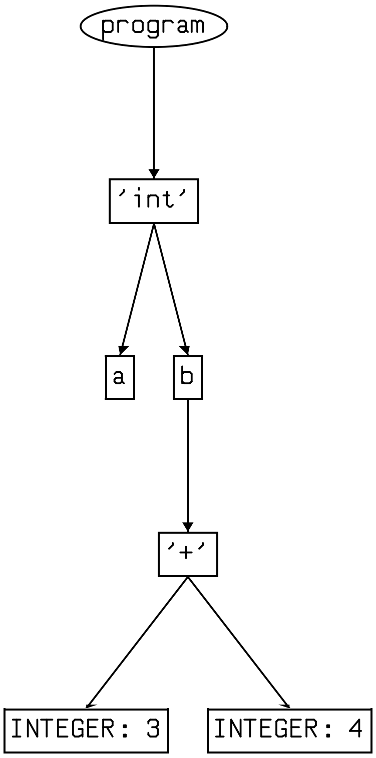

Figure 3.1 A small tree built using the **graph library**

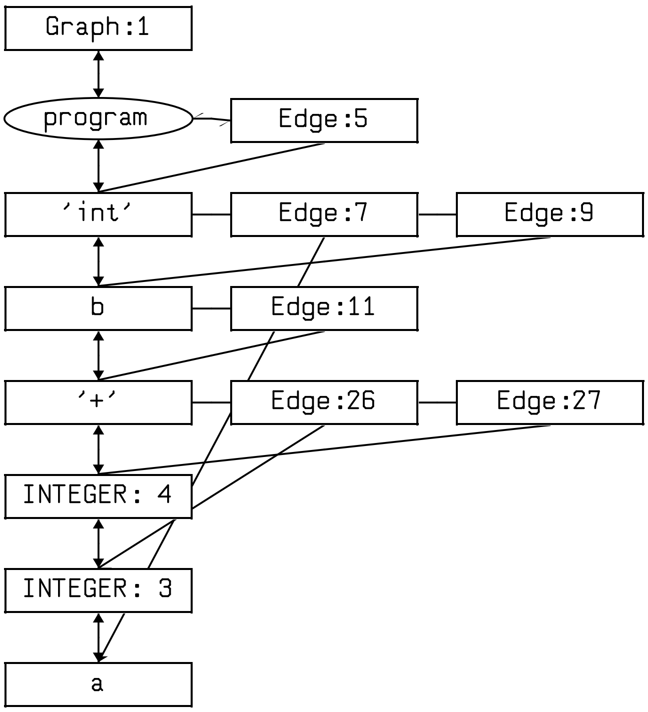

Figure 3.2 A small tree showing the internal graph structure

the data space in a graph atom once it is created, so although different graph nodes and edges can contain different amounts of data each node must stay the same size throughout their life.

The functions to insert atoms into graphs return a *handle* to the atom that has been created. In detail, it turns out that the handle is a `void` pointer to the start of the user data area in the atom, or (equivalently) a `void` pointer to the location one past the atom's internal pointer data block. These handles may not be manipulated but they are used to refer to individual atoms and can be passed into other functions to cause atoms to be printed out, set as the target of an edge, deleted and so on. User data is accessed by *casting* the handle of an atom to a pointer to the user datatype. The fields in the user data block can then be accessed using the usual ANSI-C `->` operator.

3.3 graph_compare_double

```
int graph_compare_double(void *left, void *right)
```

Compare double precision real fields for equality. The first element of the user data structure must be a `double`. Return 0 if they are equal, $+1$ if `left` $>$ `right` or -1 if `right` $<$ `left`, just like the ANSI routine `strcmp()`.

3.4 graph_compare_long

```
int graph_compare_long(void *left, void *right)
```

Compare long unsigned integer fields for equality. The first element of the user data structure must be a `long unsigned int`. Return 0 if they are equal, $+1$ if `left` $>$ `right` or -1 if `right` $<$ `left`, just like the ANSI routine `strcmp()`.

3.5 graph_compare_mem

```
int graph_compare_mem(void *left, void *right, size_t size)
```

Compare memory blocks for equality. The first element of the user data structure must be a pointer and the two memory blocks are compared for string equality over the first `size` bytes. Return 0 if they are equal, $+1$ if `left` $>$ `right` or -1 if `right` $<$ `left`, just like the ANSI routine `strncmp()`.

3.6 graph_compare_string

```
int graph_compare_string(void *left, void *right)
```

Compare string fields for equality. The first element of the user data structure must be a `char *`. Return 0 if they are equal, $+1$ if `left` $>$ `right` or -1 if `right` $<$ `left`, just like the ANSI routine `strcmp()`.

3.7 graph_delete_edge

```
void graph_delete_edge(void *edge)
```

Remove `edge` from its parent graph. The parent and target nodes for `edge` are unchanged.

3.8 graph_delete_graph

```
void graph_delete_graph(void *graph)
```

Remove `graph` from the list of graphs. All of the nodes and edges in `graph` are also deleted and the memory returned to the free list.

3.9 graph_delete_node

```
void graph_delete_node(void *node)
```

Remove `node` from its parent graph. All of the edges emanating from `node` are also deleted and the memory returned to the free list. The nodes pointed to by those edges are unchanged.

3.10 graph_get_atom_number

```
unsigned long graph_get_atom_number(const void *graph_or_node_or_edge)
```

Return the unique atom number for a graph atom. Atom numbers are allocated in an ascending sequence starting from 1 in the order in which atoms are inserted. Atom numbers are never reused, even after atoms have been deleted.

3.11 graph_get_next_edge

```
void *graph_get_next_edge(const void* node_or_edge)
```

Get the next member of the edge list emanating from a node or an edge. If this routine is passed an atom that corresponds to a node, then the returned edge will be the first edge in that node's list. If the routine is passed an atom that corresponds to an edge, then the successor to that edge will be returned. If there is no next edge, then `NULL` is returned.

3.12 graph_get_edge_target

```
void * graph_get_edge_target(const void * edge)
```

Return a handle to the node atom pointed to by the `edge`, that is the value of the ancillary pointer for atom `edge`.

3.13 graph_get_final_edge

```
void * graph_get_final_edge(const void * node_or_edge)
```

Get the final member of the edge list emanating from a node or an edge. If this routine is passed an atom that corresponds to a node, then the returned edge will be the last edge in that node's list. If the routine is passed an atom that corresponds to an edge, then the last element of that atoms's successor list will be returned. If the edge list is empty, then NULL is returned.

3.14 graph_get_final_node

```
void * graph_get_final_node(const void * node_or_edge)
```

Get the final member of the node list emanating from a graph atom or a node atom. If this routine is passed an atom that corresponds to a graph, then the returned node will be the last node in that graph's list. If the routine is passed an atom that corresponds to an node, then the last element of that atoms's successor list will be returned. If the node list is empty, then NULL is returned.

3.15 graph_hash_double

```
unsigned graph_hash_double(unsigned hash_prime, void *data)
```

Hash a double precision real number. See Chapter 7 on symbol tables for more information on hashing.

3.16 graph_hash_long

```
unsigned graph_hash_long(unsigned hash_prime, void *data)
```

Hash an unsigned long integer. See Chapter 7 on symbol tables for more information on hashing.

3.17 graph_hash_mem

```
unsigned graph_hash_mem(unsigned hash_prime, void *data)
```

Hash a length encoded block of memory. See Chapter 7 on symbol tables for more information on hashing.

3.18 graph_hash_string

```
unsigned graph_hash_string(unsigned hash_prime, void *data)
```

Hash a zero terminated string. See Chapter 7 on symbol tables for more information on hashing.

3.19 graph_insert_edge

```
void *graph_insert_edge(size_t size, void* target_node, void* node_or_edge)
```

Insert an edge into a graph, allocating `size` bytes for user data and setting `target_node` as the destination of the edge. If parameter `node_or_edge` is passed a node atom then the new edge is inserted at the first element of that node's edge list with the original list as the successor to the new edge. If parameter `node_or_edge` is passed an edge atom then the new edge is inserted at the successor to that edge.

3.20 graph_insert_graph

```
void *graph_insert_graph(char *id)
```

Insert a new graph into the library's graph list, allocating space for a character pointer which is set to `id`.

3.21 graph_insert_node

```
void *graph_insert_node(size_t size, void* node_or_graph)
```

Insert a node into a graph, allocating `size` bytes for user data. If parameter `node_or_graph` is passed a graph atom then the new node is inserted at the first element of that graph's node list with the original list as the successor to the new node. If parameter `node_or_graph` is passed a node atom then the new node is inserted at the successor to that node.

3.22 graph_insert_node_child

```
void *graph_insert_node_child(size_t node_size, size_t edge_size,
                              void* parent_node)
```

Insert a new node and an edge from `parent_node` to the new node, reserving `node_size` bytes of space in the new node atom and `edge_size` bytes of space in the new edge atom.

3.23 graph_insert_node_parent

```
void *graph_insert_node_parent(size_t node_size, size_t edge_size,
                               void* child_node)
```

Insert a new node and an edge from it to `child_node`, reserving `node_size` bytes of space in the new node atom and `edge_size` bytes of space in the new edge atom.

3.24 graph_vcg

```
void graph_vcg(void *graph,
               void (*graph_action)(const void *graph),
               void (*node_action) (const void *node),
               void (*edge_action) (const void *edge)
              )
```

Output **graph** in VCG format to the current **textio** output stream which by default is the screen. See Chapter 8 for information on how to redirect the textio output stream. The three function pointers **graph_action**, **node_action** and **edge_action** are *callback* functions that will be called once for each graph, node and edge atom respectively in the graph with the handle of the graph atom as a parameter. These callback functions can be used to output VCG specific tags so as to, for instance, change the colour and shape of a node or the size of the arrowhead on an edge. Figure 3.1 was produced using this function.

3.25 graph_vcg_atoms

```
void graph_vcg_atoms(void *graph,
                     void(* graph_action)(const void * graph),
                     void (*node_action) (const void *node),
                     void (*edge_action) (const void *edge)
                    )
```

Output the atoms in **graph** in VCG format to the current **textio** output stream which by default is the screen. See Chapter 8 for information on how to redirect the textio output stream. The three function pointers **graph_action**, **node_action** and **edge_action** are *callback* functions that will be called once for each graph, node and edge atom respectively in the graph with the handle of the graph atom as a parameter. These callback functions can be used to output VCG specific tags so as to, for instance, change the colour and shape of a node or the size of the arrowhead on an edge. Figure 3.2 was produced using this function.

Chapter 4

memalloc – memory management routines

The memalloc routines replicate the ANSI standard memory management routines but issue a fatal error message if an error occurs. They all return a void pointer which in general will need to be cast to the required pointer type. Several of these routines use parameters of type size_t: ANSI C translators provide an implementation dependent definition of size_t which is guaranteed to be able to represent the largest data object that may be created using that translator. Most often it will be either an unsigned int or an unsigned long.

4.1 mem_calloc

```
void *mem_calloc(size_t nitems, size_t size)
```

Allocate a block of memory large enough to hold nitems of size size and then clear the contents to zero. Return a void pointer to the first location in the block. Exit with a fatal error if insufficient memory is available to allocate the requested block size.

4.2 mem_free

```
void mem_free(void *block)
```

Free a block previously allocated by a call to one of the other mem_ routines. Exit with a fatal error if block is null. Attempting to a free a pointer that is not referring to a previously allocated block results in unpredictable behaviour.

4.3 mem_malloc

```
void *mem_malloc(size_t size)
```

Allocate a block of memory of size size. Return a void pointer to the first location in the block. The memory block is not initialised. Exit with a fatal error if insufficient memory is available to allocate the requested block size.

4.4 mem_print_statistics

```
void mem_print_statistics(void)
```

Print out the number of bytes of memory allocated using `mem_malloc`, `mem_calloc` and `mem_realloc` since the program started running.

4.5 mem_realloc

```
void *mem_realloc(void *block, size_t size)
```

Change the size of a previously allocated memory block to `size`. If necessary, a completely new memory block will be created and the necessary copying of data between old and new blocks performed automatically. Return a void pointer to the first location in the block. Any new memory area beyond the end of the old block memory block is not initialised, and will therefore contain unpredictable data immediately after a call to `mem_realloc`. Exit with a fatal error if insufficient memory is available to allocate the requested block size.

Chapter 5

scan – scanner support routines

The scanner is such an integral part of the rdp system that is unlikely to ever be used as a general purpose package: all rdp generated parsers automatically contain the necessary calls to these routines. They are documented here for completeness.

5.1 scan_column_number

```
unsigned long scan_column_number(void)
```

Return the start column number for the most recently scanned lexeme.

5.2 scan_init

```
void scan_init(const int case_insensitive,
               const int newline_visible,
               const int show_skips,
               const int symbol_echo,
               char *token_names)
```

Initialise the scanner subsystem. This routine *must* be called before calling any other scanner routines. It is an error to call this routine twice. As well as creating a symbol table to hold the scanner keywords,

`scan_init`

accepts parameters that control the overall behaviour of the scanner as follows:

- `case_insensitive` If true, then convert uppercase alphabetic characters to lowercase before lexical analysis except in extended tokens such as strings and comments.
- `newline_visible` If true, then pass newline characters to the parser as token `EOLN`, otherwise discard newlines in the scanner.
- `show_skips` If true, issue a `skipping to ...` message during error recovery.

◇ **symbol_echo** If true, print the token value of each symbol as it is scanned.

◇ **token_names** A string containing the token names in ASCII, each name terminated by a null character (`\0`). If null then error messages will only report the token number in decimal. If there are fewer strings than tokens defined, unexpected error messages will cause unpredictable behaviour.

5.3 scan_line_number

```
unsigned long scan_line_number(void)
```

Return the current line number of the file being scanned. The line number reported is the line number at the start of the most recently parsed token. Visible comment tokens can be many lines long, so the line numbers reported during a scan may not be contiguous.

5.4 scan_load_keyword

```
void scan_load_keyword(char *id1, const char *id2, const int token,
                       const int extended)
```

Load the keyword `id1` into the scanner's symbol table and mark it as token number `token`. Extended tokens such as `STRING_ESC` and `COMMENT` use the `id2` parameter to specify a supplementary token. The class of an extended token is specified using the `extended` parameter.

The scanner recognises keywords and punctuation symbols by comparing the input with the contents of its own symbol table. It is sometimes useful to add elements to the scanner table during program parsing. In particular, the C language `typedef` statement creates new names for types which may be indistinguishable from variable names without lookahead in certain contexts. Indeed, rather remarkably it is legal in C to have a type name and a variable name which are identical, that is the names must inhabit separate name space. In situations like these it is sometimes useful to be able to *create* new keywords during execution.

5.5 scan_prune_tree

```
void scan_prune_tree(scan_data * rdp_tree)
```

Prune empty (epsilon) nodes from the derivation tree.

5.6 scan_test

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

Test to see if the current token number equals `valid` and return a 1 if the test succeeds or else a 0 if the test fails. If the current token is *not* equal to `valid` and `stop` is not `NULL` then generate an error message and skip until a token in `stop` is detected. If `production` is not `NULL`, then preface the error message with the production name. RDP uses this parameter when the `-E` switch is used during parser generation.

5.7 scan_test_set

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

Test to see if the current token number is a member of set `valid` and return a 1 if the test succeeds or else a 0 if the test fails. If the current token is *not* in `valid` and `stop` is not `NULL` then generate an error message and skip until a token in `stop` is detected. If `production` is not `NULL`, then preface the error message with the production name. RDP uses this parameter when the `-E` switch is used during parser generation.

5.8 scan_vcg_print_node

```
void scan_vcg_print_node(const void* node)
```

RDP generated parsers can automatically build derivation trees during a parse which show how the productions in a grammar are activated: essentially the derivation tree is a trace of the path taken by the parser. These derivation trees can be written to a text file in a form suitable for input to the VCG (Visualisation of Compiler Graphs) tool [San95], which can then display the tree under MS-Windows, Windows-95 or X-windows. The derivation trees are built using the `graph` library. This routine is called once for each node in the tree during the last pass of the parser, and produces scanner specific labels in the displayed tree.

Chapter 6

set – a dynamic set handler

Set manipulation is central to many parsing algorithms, and a space efficient set representation is an important part of the `rdp_supp` library. Sets are represented as variable length bit strings, and the common operations such as set union are implemented as bitwise logical operations.

The set handler can handle sets of integers or `enum` elements (which in C are really integers in disguise) in the range 0 to (`MAX_UNSIGNED` − 1) where `MAX_UNSIGNED` is the ANSI standard C macro that expands to the largest encodable `unsigned` number. Each set is represented by a structure which contains an unsigned integer called `size`, and a pointer to a block of memory on the heap as shown in Figure 6.1. When initially created a set contains a null pointer, that is its total memory consumption is `(sizeof(unsigned) + sizeof(void *))`.

As elements are added to a set, the set package automatically grows the set by allocating extra storage. Sets can only grow in multiples of a byte, so set size is always rounded up to the nearest eight bits.

Simply clearing bits in a set is not enough to release the memory. The only routines that can cause a set to shrink are `set_free()` which clears the set and returns allocated memory to the heap, and `set_normalise()` which removes empty bytes from the end of a set.

To reduce the number of reallocation calls made to the memory manager it is possible to define a minimum size for a set below which it will not shrink. By default the minimum set size is zero bytes.

Set names are always passed by address since C passes structures by value not name.

The basic set operations such as union and intersection are implemented in three different forms — so-called *element*, *list* and *set* forms. Each form takes the address of a set as the destination parameter, but the source might be either a single integer (the element form), a list of integers (the list form) or another set (the set form). A list is terminated by the constant `SET_END` which is defined in `set.h` to be `MAX_UNSIGNED`. This extract illustrates the use of the three different forms:

```
set_unite_element(&first, 5)
set_unite_list(&first, 8, 1, 3, SET_END)
set_unite_set(&first, &second)
```

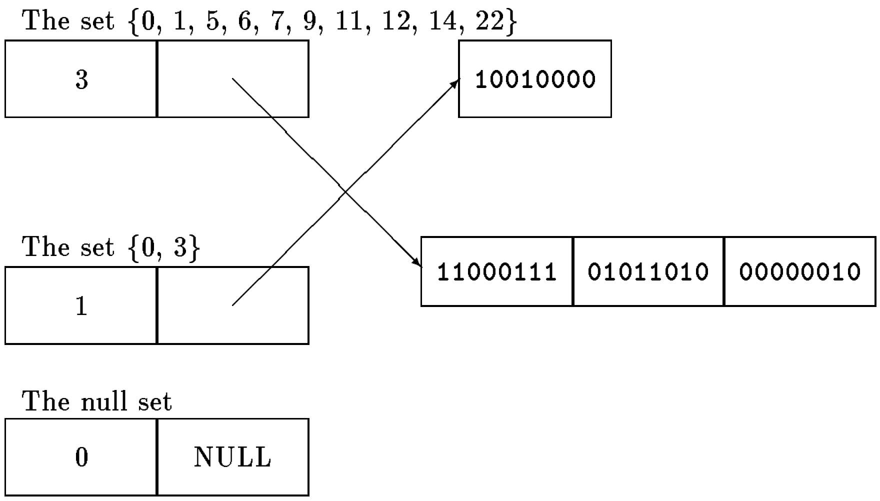

Figure 6.1 Set data structure

6.1 set_array

`unsigned *set_array(const set_ * src)`

When iterating over the contents of a set it is inefficient to test each bit individually. This routine takes a set and returns an array of `unsigned` integers, one for each element of the set. A final element is set to `SET_END`. Once this array has been created, set iterations may be implemented by iterating over the elements of the array:

```
set_ src
unsigned *elements = set_array(src)

while (*elements != SET_END)
{
  ...

  elements++
}
```

When you have finished using the array created by calling `set_array`, you can free the memory by simply calling `mem_free()`:

```
mem_free(elements)
```

6.2 set_cardinality

`unsigned set_cardinality(const set_ * src)`

Return the number of elements in set `src`.

6.3 set_assign_element

```
void set_assign_element(set_ * dst, const unsigned element)
```

Clear `dst` and then assign the single `element` to it.

6.4 set_assign_list

```
void set_assign_list(set_ * dst,...)
```

Clear `dst` and then assign the list of elements to it.

6.5 set_assign_set

```
void set_assign_set(set_ * dst, const set_ * src)
```

Copy `src` to `dst`.

6.6 set_compare

```
int set_compare(set_ * dst, set_ * src)
```

Return 0 if `src` and `dst` are each subsets of each other (i.e. they contain exactly the same elements). It is not necessary for the sets to be the same length for this test to succeed. The routine returns -1 if `src` is 'less than' `dst` and $+1$ if `src` is 'greater than' `dst`. The exact definition of greater than and less than is not significant: the existence of a collation sequence allows sets to be used as symbol table keys.

6.7 set_difference_element

```
void set_difference_element(set_ * dst, const unsigned element)
```

Remove `element` from `dst`. It is not an error to remove an element that is not in a set.

6.8 set_difference_list

```
void set_difference_list(set_ * dst,...)
```

Remove each member of the list of elements from `dst`. It is not an error to remove an element that is not in a set.

6.9 set_difference_set

```
void set_difference_set(const set_ * dst, const set_ * src)
```

Remove every element in `src` from `dst`. It is not an error to remove an element that is not in a set.

6.10 set_free

```
void set_free(set_ * dst)
```

Clear `dst` and return the bit vector storage to the heap.

6.11 set_grow

```
void set_grow(set_ * dst, const unsigned length)
```

Expand `dst` so that it is `length` bytes long and therefore capable of holding elements in the range $0 \ldots (\texttt{length} \times 8) - 1$.

6.12 set_includes_element

```
int set_includes_element(set_ * dst, const unsigned element)
```

Return 1 if set `dst` contains `element` otherwise 0.

6.13 set_includes_list

```
int set_includes_list(set_ * dst,...)
```

Return 1 if set `dst` contains every element in the list otherwise 0.

6.14 set_includes_set

```
int set_includes_set(const set_ * dst, const set_ * src)
```

Return 1 if set `dst` contains every element in `src` otherwise 0.

6.15 set_intersect_element

```
void set_intersect_element(set_ * dst, const unsigned element)
```

Remove every element in `dst` apart from `element`.

6.16 set_intersect_list

```
void set_intersect_list(set_ * dst,...)
```

Remove every element in `dst` that is not in the list.

6.17 set_intersect_set

```
void set_intersect_set(set_ * dst, const set_ * src)
```

Remove every element in `dst` that is not in `src`.

6.18 set_complement

```
void set_invert(set_ * dst, const unsigned universe)
```

Form the complement of `dst` in universe 0, ..., `universe` by complementing all bits in the vector and then clearing bits corresponding to `universe` and above.

6.19 set_minimum_size

```
unsigned set_minimum_size(const unsigned minimum_size)
```

Set a minimum length below which `set_normalise()` will not shrink any set.

6.20 set_normalise

```
void set_normalise(set_ * dst)
```

Delete zero bytes from the end of a bit vector and update the size field, i.e. reduce a set to its minimum storage requirement. Do not shrink to less than the value set by the last call to `set_minimum_size`.

6.21 set_print_element

```
void set_print_element(const unsigned element, const char *element_names)
```

Print a single set element. If `element_names` is `NULL` then simply print the decimal representation of the element number. If `element_names` is non-null, it is assumed to be an ASCII string made up of null delimited substrings, one per element. The routine counts substrings from the left until it finds the name of the set element and prints that instead of the decimal number.

6.22 set_print_set

```
void set_print_set(const set_ * src, const char *element_names,
                   unsigned line_length)
```

Print all elements in `src`. If `element_names` is `NULL` then simply print the decimal representation of the element numbers. If `element_names` is non-null, it is assumed to be an ASCII string made up of null delimited substrings, one per element. The routine finds counts substrings from the left until it finds the name of the set element and prints that instead of the decimal number.

Whenever the routine starts to print out a new set element, it checks to see whether the length of the current output line exceeds `line_length`. If so, it prints a newline before proceeding. This parameter can be used to avoid printing very long lines by setting an upper bound on the start column of a set element. Note that this does not have the effect of limiting line length to the set value because the actual line lengths will depend on the length of the set element names.

This routine is used by `rdp` to build the error messages when a parser syntax error occurs. See the routine `scan_test_set()`.

6.23 set_unite_element

```
void set_unite_element(set_ * dst, const unsigned element)
```

Add `element` to `dst`.

6.24 set_unite_list

```
void set_unite_list(set_ * dst,...)
```

Add a list of elements to `dst`.

6.25 set_unite_set

```
void set_unite_set(set_ * dst, const set_ * src)
```

Add each element in `src` to `dst`.

Chapter 7

symbol – a hash coded symbol table manager

7.1 Data structures

An efficient symbol table manager is crucial to the performance of any translator. Many languages require symbol table access during scanning simply to resolve grammatical ambiguities, and semantic analysis usually requires symbol table manipulation if the underlying grammar is to remain context free. The **rdp** symbol table manager is particularly flexible, allowing multiple symbol tables to be managed. The user data associated with each symbol can be defined with complete freedom, and the internal links used to maintain the hash table are hidden. There is no inherent reason why symbols in a particular table should not carry different user data as long as the key fields are in the same place in each record. The **rdp** EBNF provides a `SYMBOL_TABLE` directive which automatically creates and initialises symbol tables. See the file **rdp.bnf** for a particularly complicated example of its use.

rdp maintains a linked list of symbol tables. Each symbol table is described by a header record that contains pointers to a hash table, a scope list, various maintenance functions and some book keeping data. The basic layout is illustrated in Figure 7.1.

Whenever a symbol is to be inserted into the table, its key fields are *hashed* generating a random number in the range 0 ... **size**. This hash number is then used to index into the hash table, selecting one of the linked lists. The symbol is then added to the head of the list. A lookup is performed by hashing the test symbol and then searching down the list for a match. Since the most recent additions are always examined first, the structure directly implements nested scope rules in that a new symbol will hide any symbols with the same key deeper in the table.

The hash lists are in fact doubly linked so that symbols can be quickly unlinked from the chain.

Whenever a symbol is added to a hash list, it is also added to the head of the current scope chain. New scope regions may be declared, in which case a new scope record is created and added to the head of the scope list. The scope pointers are represented in Figure 7.1 by curved arrows. Although not shown

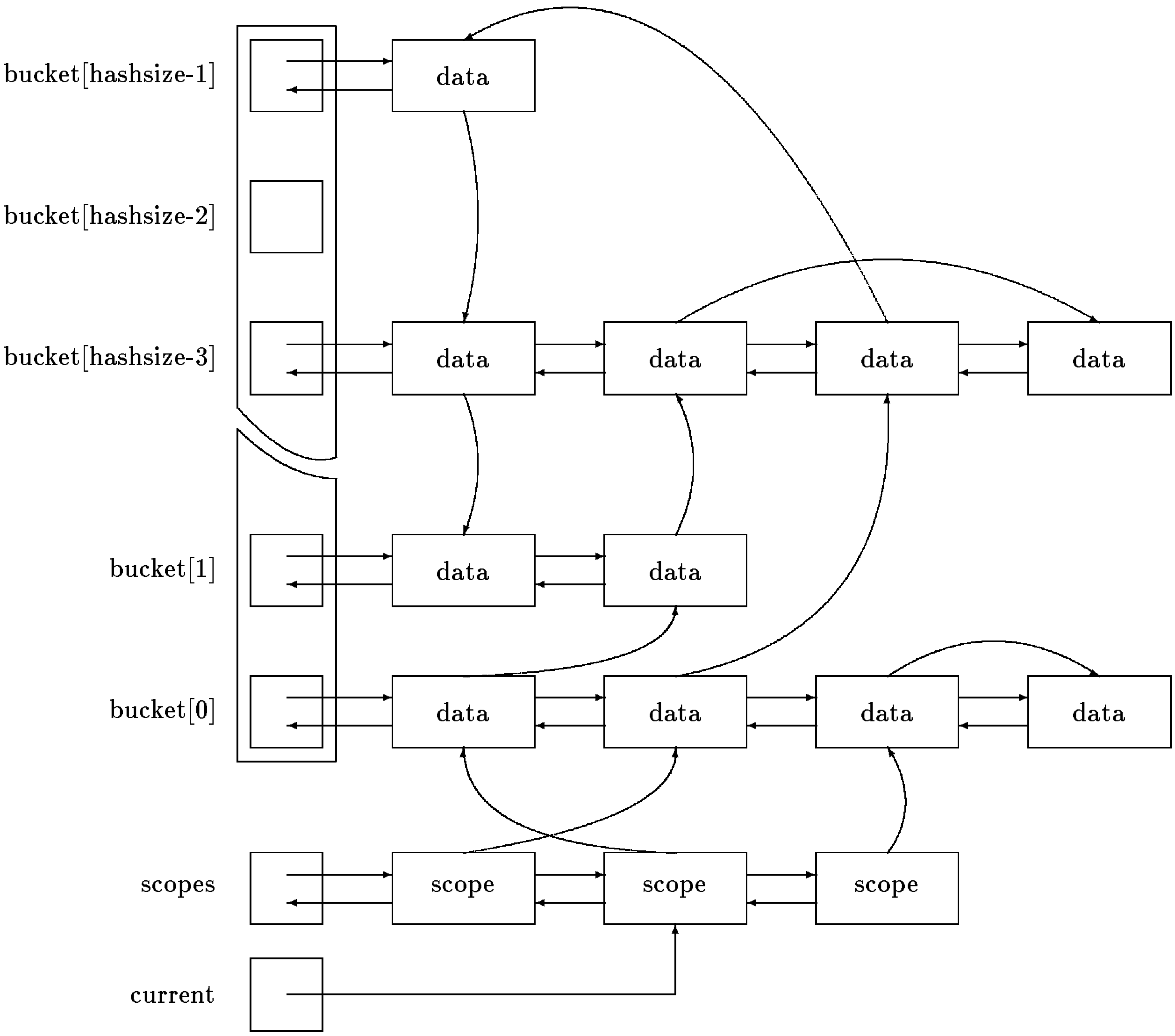

Figure 7.1 Symbol table data structure

on the diagram, each symbol maintains a back link to its scope record allowing efficient checking of the scope level for a particular symbol. The current scope may be reset to a previously declared scope.

Different kinds of user data record are allowed for by parameterising the functions that hash, compare and print symbols. These are supplied as function pointers when the symbol table is declared. Most symbol tables (certainly all those in the distributed grammars) simply use a single string as the key field. The library provides standard hash and compare functions for the special (although common) case of a symbol in which the first field is a character string (i.e. a pointer to `char`) which acts as the key field. Functions are also provided for an initial `set_` key field. If you need to do something more baroque, such as hashing on both a string and a numeric name space, then you will have to write your own functions. Try looking at the source code for `symbol_compare_string()`, `symbol_hash_string()` and `symbol_print_string()` for ideas.

All of the pointers embedded in the symbols are hidden from the user, and symbols are manipulated *via* `void` pointers to the first location in the user data block. `rdp` defines casting macros for each symbol table to make user data field access less verbose. See the file `miniplus.bnf` for examples.

The `symbol` table package was originally developed along the lines described in the 'Dragon Book' [ASU86]. The idea of hiding the pointers and using function pointers in a sort of poor man's object oriented programming was taken from Holub's book on compilers [Hol90] although I have implemented things rather differently and provided a more complete set of routines. I also took the idea of storing a symbol's hash number within it to allow fast lookup from the symbol table module that Terence Parr supplies with PCCTS compiler-compiler suite.

7.2 symbol_compare_double

```
int symbol_compare_double(void *left, void *right)
```

Compare double precision real keys for equality. The first element of the user data structure must be a `double`. Return 0 if they are equal, +1 if `left` > `right` or −1 if `right` < `left`, just like the ANSI routine `strcmp()`. For symbols that are keyed on a single `double`, this routine may be used as the *compare* parameter to `symbol_init()` and the `rdp` directive `SYMBOL_TABLE`.

7.3 symbol_compare_double_reverse

```
int symbol_compare_double_reverse(void *left, void *right)
```

Compare double precision real keys for equality with reverse polarity. The first element of the user data structure must be a `double`. Return 0 if they are equal, +1 if `left` < `right` or −1 if `right` > `left`. For symbols that are keyed on a single `double`, this routine may be used as the *compare* parameter to `symbol_init()` and the `rdp` directive `SYMBOL_TABLE`.

7.4 symbol_compare_long

```
int symbol_compare_long(void *left, void *right)
```

Compare long integer keys for equality. The first element of the user data structure must be a `long int`. Return 0 if they are equal, +1 if `left` > `right` or −1 if `right` < `left`, just like the ANSI routine `strcmp()`. For symbols that are keyed on a single long integer, this routine may be used as the *compare* parameter to `symbol_init()` and the `rdp` directive `SYMBOL_TABLE`.

7.5 symbol_compare_long_reverse

```
int symbol_compare_long_reverse(void *left, void *right)
```

Compare long integer keys for equality with reverse polarity. The first element of the user data structure must be a `long int`. Return 0 if they are equal, +1 if `left` < `right` or −1 if `right` > `left`, just like the ANSI routine `strcmp()`. For symbols that are keyed on a single long integer, this routine may be used as the *compare* parameter to `symbol_init()` and the `rdp` directive `SYMBOL_TABLE`.

7.6 symbol_compare_string

```
int symbol_compare_string(void *left, void *right)
```

Compare string keys for equality. The first element of the user data structure must be a `char*`. Return 0 if they are equal, +1 if `left` > `right` or −1 if `right` < `left`, just like the ANSI routine `strcmp()`. For symbols that are keyed on a single string, this routine may be used as the *compare* parameter to `symbol_init()` and the `rdp` directive `SYMBOL_TABLE`.

7.7 symbol_compare_string_reverse

```
int symbol_compare_string_reverse(void *left, void *right)
```

Compare string keys for equality with reverse polarity. The first element of the user data structure must be a `char*`. Return 0 if they are equal, +1 if `left` < `right` or −1 if `right` > `left`, just like the ANSI routine `strcmp()`. For symbols that are keyed on a single string, this routine may be used as the *compare* parameter to `symbol_init()` and the `rdp` directive `SYMBOL_TABLE`.

7.8 symbol_find

```
void symbol_find(const void *table, void *key, size_t key_size,
                 size_t symbol_size, void* scope,
                 enum SYMBOL_FIND_OP op)
```

```
enum SYMBOL_FIND_OP {SYMBOL_NEW, SYMBOL_OLD, SYMBOL_ANY}
```

7.9 symbol_free_scope

```
void symbol_free_scope(const void *scope)
```

Unlink all symbols in a scope chain and then free all memory associated with them. Unlink the scope record from the scope chain and free the memory associated with it.

7.10 symbol_free_symbol

```
void symbol_free_symbol(void *symbol)
```

Free the memory associated with a symbol. Unpredictable behaviour will occur if a symbol is freed before unlinking it from the symbol table.

7.11 symbol_free_table

```
void symbol_free_table(void *table)
```

Free all memory associated with a table and all symbols and scope records within it.

7.12 symbol_get_scope

```
void *symbol_get_scope(const void *table)
```

Return a pointer to the scope record for the current scope level.

7.13 symbol_hash_double

```
unsigned symbol_hash_double(unsigned hash_prime, void *data)
```

7.14 symbol_hash_long

```
unsigned symbol_hash_long(unsigned hash_prime, void *data)
```

7.15 symbol_hash_mem

```
unsigned symbol_hash_mem(unsigned hash_prime, void *data)
```

Hash a length encoded string. For symbols that are keyed on a single length encoded string, this routine may be used as the *hash* parameter to `symbol_init()` and the `rdp` directive `SYMBOL_TABLE`.

7.16 symbol_hash_string

```
unsigned symbol_hash_string(unsigned hash_prime, void *data)
```

Hash a zero terminated string. For symbols that are keyed on a single string, this routine may be used as the *hash* parameter to `symbol_init()` and the `rdp` directive `SYMBOL_TABLE`.

7.17 symbol_insert_key

```
void *symbol_insert_key(const void *table, char *str, size_t size)
```

Make a new symbol with a user data area `size` bytes long. Put a pointer to `str` in the user first user data field. Hash the symbol and insert in the table.

7.18 symbol_insert_symbol

```
void *symbol_insert_symbol(const void *table, void *symbol)
```

Hash an existing symbol and insert it in the table.

7.19 symbol_lookup_key

```
void * symbol_lookup_key(const void * table, void * key, void * scope)
```

Hash the key and lookup up the symbol. Return `NULL` if not found, otherwise a pointer to the base of the user data in the found symbol. Parameter `scope` restricts the search to scope level `scope`. If `scope` is `NULL`, then all scopes are searched.

7.20 symbol_new_scope

```
void *symbol_new_scope(void *table, char *str)
```

Create a new named scope and add it to the head of the scope list. Make the new scope current.

7.21 symbol_new_symbol

```
void *symbol_new_symbol(size_t size)
```

Allocate enough memory for the symbol table pointers plus `size` bytes of user data. Return a pointer to the base of the user data.

7.22 symbol_new_table

```
void *symbol_new_table(char *name,
                       const unsigned symbol_hashsize,
                       const unsigned symbol_hashprime,
                       int (*compare) (void *left_symbol, void *right_symbol),
                       unsigned (*hash) (unsigned hash_prime, void *data),
                       void (*print) (const void *symbol))
```

Create a new symbol table and add it to the head of the linked list of tables. Return a pointer to the table which may be used to name the table in subsequent calls. The table will have `size` hash buckets. See `rdp.c` for examples of use.

7.23 symbol_next_symbol

```
void *symbol_next_symbol(void *table, void *symbol)
```

Sometimes it is necessary to look down a hash chain beyond a found symbol, for instance to locate instances of symbols with the same key that were inserted previously. This routine takes a pointer to a symbol and then continues to search down the same chain until it finds another match or reaches the end of the list. Return `NULL` if no other matching symbol is found, otherwise a pointer to the base of the user data.

7.24 symbol_next_symbol_in_scope

```
void *symbol_next_symbol_in_scope(void *symbol)
```

This routine returns the next symbol in a scope chain. Prior to any sorting, symbols will be returned in the reverse order to that in which they were inserted.

7.25 symbol_print_all_table

```
void symbol_print_all_table(void)
```

Print a diagnostic dump of all symbol tables currently active.

7.26 symbol_print_all_table_statistics

```
void symbol_print_all_table_statistics(const int histogram_size)
```

Print summary statistics for all symbol tables currently active. `rdp` generated parsers call this routine when the `-S` command line option is active.

7.27 symbol_print_double

```
void symbol_print_double(const void *symbol)
```

Print the first element in the user data as a double precision real. For symbols that are keyed on a single real, this routine may be used as the *print* parameter to `symbol_init()` and the `rdp` directive `SYMBOL_TABLE`.

7.28 symbol_print_long

```
void symbol_print_long(const void *symbol)
```

Print the first element in the user data as a long integer. For symbols that are keyed on a single long integer, this routine may be used as the *print* parameter to `symbol_init()` and the `rdp` directive `SYMBOL_TABLE`.

7.29 symbol_print_string

```
void symbol_print_string(const void *symbol)
```

Print the first element in the user data as a pointer to string. For symbols that are keyed on a single string, this routine may be used as the *print* parameter to `symbol_init()` and the `rdp` directive `SYMBOL_TABLE`.

7.30 symbol_print_scope

```
void symbol_print_scope(const void *table, void *scope)
```

Print all symbols in the scope chain pointed to by `scope`.

7.31 symbol_print_symbol

```
void symbol_print_symbol(const void *table, const void *symbol)
```

Print a single symbol.

7.32 symbol_print_table

```
void symbol_print_table(const void *table)
```

Print the entire contents of the symbol table pointed to by `table`.

7.33 symbol_print_table_statistics

```
void symbol_print_table_statistics(const void *table,
                                   const int histogram_size)
```

Print summary statistics for the symbol table pointed to by `table`.

7.34 symbol_set_scope

```
void symbol_set_scope(void *table, void *scope)
```

Set the current scope to `scope`, which must be a pointer returned by a previous call to `symbol_new_scope()` or `symbol_get_scope()`.

7.35 symbol_sort_table

```
void symbol_sort_table(void *table)
```

Sort all scope chains in a table using the ordering defined by `compare` function.

7.36 symbol_sort_scope

```
void symbol_sort_scope(void *table, void *scope)
```

Sort a scope chain using the ordering defined by `compare` function. `rdp` uses this function to alphabetically sort token and production names.

7.37 symbol_unlink_scope

```
void symbol_unlink_scope(void *data)
```

Unlink all symbols in a scope chain from their hash chains. The symbols themselves (and the scope chain data) are preserved. This function is usually called at the exit from a scope block.

7.38 symbol_unlink_symbol

```
void symbol_unlink_symbol(void *data)
```

Unlink a single symbol from its hash chain. The symbol itself (and the scope chain data) are preserved.

7.39 symbol_unlink_table

```
void symbol_unlink_table(void *table)
```

Unlink all symbols in a table from their hash chains. The symbols themselves (and the scope chain data) are preserved.

Chapter 8

textio – text buffering and messaging services

Text buffering is a surprisingly troubling part of lexical analyser design. Supporting nested include files, source echoing and synchronised error messages requires careful design. The `rdp` text buffer manager maintains a single large area of memory. New strings can be inserted at low addresses and grow upwards.

The top of the region is used as a pushdown stack of line buffers for the set of included files. As each nested include file is opened, a record containing the previous state of the text manager is pushed onto a linked list and a new line buffer opened up. At the end of the included file, the buffer is released, the record list popped and scanning continue where it left off. End of file is not returned to the caller until the outermost file is completely consumed.

This arrangement allows arbitrary strings of arbitrary lengths to be stored, and files with arbitrarily long lines to be read. As each new line is read in, it is stored backwards at the top of the buffer. `rdp` does not run out of memory until the strings meet the line buffers, so memory can always be fully used. This data arrangement is illustrated in Figure 8.1.

As well as these text input routines, messaging routines are provided to centralise the production of error messages.

8.1 Global variables

8.1.1 `*text_bot`

A pointer to the first location in the text buffer.

8.1.2 `*text_top`

A pointer to the first free location above the string base.

8.1.3 `int text_char`

The last character read by *textio*.

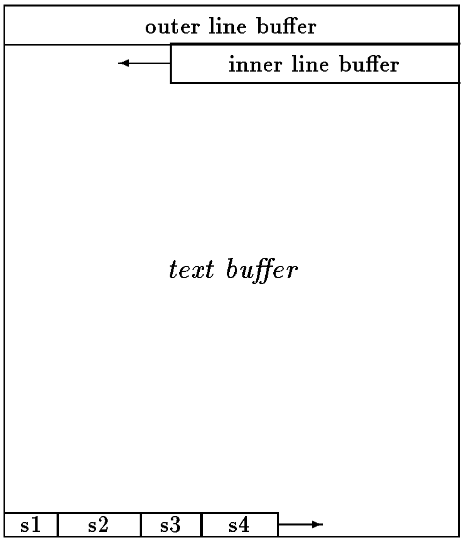

Figure 8.1 Text buffer structure

8.1.4 `void *text_scan_data`

A pointer to the last scanner symbol read by the scanner.

8.2 `text_capitalise_string`

```
char *text_capitalise_string(char *str)
```

Capitalise the first character of each space delimited word in string `str`.

8.3 `text_default_filetype`

```
char *text_default_filetype(char *fname, const char *ftype)
```

If `fname` has no filetype then add a period and the string `ftype` to it.

8.4 `text_dump`

```
void text_dump(void)
```

Print out (in order of creation time) all the inserted strings in the text buffer.

8.5 `text_echo`

```
void text_echo(const int i)
```

Enable listing for all lines.

8.6 text_extract_filename

```
char * text_extract_filename(char * fname)
```

Return the file name part of a path, after stripping off leading directories and the trailing file type.

8.7 text_find_ASCII_element

```
char * text_find_ASCII_element(int c)
```

Return a string representing the ASCII code for `c`. Non-printing character codes return a three digit mnemonic code.

8.8 text_force_filetype

```
char *text_force_filetype(char *fname, const char *ftype)
```

Force `fname` to have filetype `ftype` even if it already has one.

8.9 text_free

```
void text_free(void)
```

Release all memory held by the `textio` package. It is an error to access any `textio` functions after calling `text_free`.

8.10 text_get_char

```
void text_get_char(void)
```

Get a single character from the line buffer into `text_char`.

8.11 text_init

```
void text_init(const long max_text,
               const unsigned max_errors,
               const unsigned max_warnings,
               const unsigned tab_width)
```

Initialise the text subsystem with a buffer of `max_text` bytes.

8.12 text_insert_char

```
char *text_insert_char(const char c)
```

Insert a single character into the string buffer. Return a pointer to the inserted character.

8.13 text_insert_characters

```
char *text_insert_characters(const char *str)
```

Insert the string **str** into the string buffer, but omit the terminating null character. Return a pointer to the first character.

8.14 text_insert_integer

```
char *text_insert_integer(const unsigned n)
```

Insert the ASCII decimal representation of an unsigned integer into the text buffer. Return a pointer to the start of the string.

8.15 text_insert_string

```
char *text_insert_string(const char *str)
```

Insert the string **str** into the string buffer and include the terminating null character. Return a pointer to the start of the string.

8.16 text_insert_substring

```
char *text_insert_substring(const char * prefix, const char *str,
                            const unsigned n)
```

Insert the string **prefix** into th string buffer followed by the string **str** followed by an underscore and then insert the ASCII decimal representation of unsigned integer **n** with a terminating null character. Return a pointer to the start of the string. This routine is used to construct sub-production names in the **rdp** grammar checking routines.

8.17 long text_is_valid_C_id

```
int text_is_valid_C_id(char * s)
```

Return TRUE (integer 1) if **s** conforms to the rules for valid ANSI-C identifiers, otherwise return FALSE (integer 0).

8.18 long text_line_number

```
unsigned long text_line_number(void)
```

Return the current line number in the current file.

8.19 text_lowercase_string

`char *text_lowercase_string(char *str)`

Go through string `str`, converting all upper case letters to lower case and return that string.

8.20 text_make_C_identifier

`char *text_make_C_identifier(char * str)`

Use `text_find_ASCII_element` to construct a valid C identifier from the names of the characters in `str`.

8.21 text_message

`int text_message(const enum text_message_type type, const char *fmt, ...)`

Generate an error message. `type` is one of

- `TEXT_INFO` print the current filename and the message.
- `TEXT_WARNING` print `Warning`, the current filename and the message.
- `TEXT_ERROR` print `Error`, the current filename and the message.
- `TEXT_FATAL` print `Fatal`, the current filename and the message. Exit to the operating system after issuing the message.
- `TEXT_INFO_ECHO` echo the current source line, print the current filename and the message.
- `TEXT_WARNING_ECHO` echo the current source line, print `Warning`, the current filename and the message.
- `TEXT_ERROR_ECHO`echo the current source line, print `Warning`, the current filename and the message.
- `TEXT_FATAL_ECHO` echo the current source line, print `Warning`, the current filename and the message. Exit to the operating system after issuing the message.

Any valid `printf()` parameters may be supplied after `type`.

8.22 text_open

`FILE *text_open(char *s)`

Open a file. `s` is an ASCII string containing the file name. An error message will be issued if the file cannot be opened. There is no corresponding close function because files are automatically closed by the handler when an `EOF` is encountered.

8.23 text_print_C_char

`int text_print_C_char(char * string)`

Print the contents of `string` as an ANSI-C character literal, using escape sequences where necessary.

8.24 text_print_C_char_file

`int text_print_C_char_file(FILE * file, char * string)`

Print the contents of `string` as an ANSI-C character literal to file stream `file`, using escape sequences where necessary.

8.25 text_print_C_string

`int text_print_C_string(char * string)`

Print the contents of `string` as an ANSI-C string literal, using escape sequences where necessary.

8.26 text_print_C_string_file

`int text_print_C_string_file(FILE * file, char * string)`

Print the contents of `string` as an ANSI-C string literal to file stream `file`, using escape sequences where necessary.

8.27 text_print_statistics

`void text_print_statistics(void)`

Print summary text buffer statistics. Use this routine to find out how much free space is left in the text buffer.

8.28 text_print_time

`void text_print_time(void)`

Print the currently consumed CPU time for this run.

8.29 text_printf

`int text_printf(const char *fmt, ...)`

Send a formatted message to the message stream. Any valid `printf()` parameters are valid here.

8.30 text_print_total_errors

```
int text_print_total_errors(void)
```

Print the total number of errors across all input files.

8.31 text_redirect

```
void text_redirect(FILE* file)
```

At startup, messages are sent to the stream named in the `TEXT_MESSAGES` macro defined in `textio.h`, which is usually `stderr`. Output can be redirected to any other text file with this routine. `file` must be an initialised file variable pointer.

8.32 text_total_errors

```
unsigned text_total_errors(void)
```

Return the total number of errors across all input files.

8.33 text_total_warnings

```
unsigned text_total_warnings(void)
```

Return the total number of warnings across all input files.

8.34 text_uppercase_string

```
void text_uppercase_string(char *str)
```

Go through string `str`, converting all lower case letters to upper case and return that string.

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

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

[Hol90] Allen I. Holub. *Compiler design in C*. Prentice Hall, 1990.

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

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

[JS97c] Adrian Johnstone and Elizabeth Scott. 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.