rdp – an iterator-based recursive descent parser generator with tree promotion operators

Adrian Johnstone
A.Johnstone@rhbnc.ac.uk

Elizabeth Scott
E.Scott@rhbnc.ac.uk

Department of Computer Science,
Royal Holloway, University of London,
Egham, Surrey, TW20 0EX, UK

Abstract

rdp is a parser generator which accepts *Iterator Backus Naur Form* productions decorated with attributes and ANSI-C actions and produces recursive descent parsers. It has special support for the generation of tree-based intermediate forms, built-in symbol table handling for the implementation of context-sensitive components of the language syntax and a support library that includes a generalised graph handling module that can output graphs in a form suitable for use with well known visualisation tools.

Keywords: Parser generator, EBNF, iterator, derivation tree, tree promotion operator, LL(1) grammar

Introduction

rdp is a parser generator that accepts context-free grammar specifications written in *Iterator Backus Naur Form* (IBNF) and outputs recursive descent parsers written in ANSI-C. rdp is targeted at neophyte users but includes the following features that make it a powerful tool in the hands of the more experienced:

- ◇ a grammar specification language that comprises standard BNF extended by a single construct (called an *iterator*) that subsumes as special cases the optional phrase, Kleene closure and positive closure found in traditional extensions to BNF,
- ◇ automatic construction of derivation trees,
- ◇ a set of *promotion operators* which may be used to produce reduced derivation trees that conform to common Abstract Syntax Tree forms,
- ◇ a built-in graph-handling library with an interface to the VCG [San95] graph visualisation tool,
- ◇ a paramaterisable scanner which supports runtime extension of the keyword set of the language to be parsed,
- ◇ built-in handling of multiple symbol tables,
- ◇ straightforward handling of named synthesised and inherited attributes that may be accessed by semantic actions written in ANSI-C or C++, and
- ◇ special semantic rules that are instantiated inline using ANSI-C macros.

rdp itself, and the language processors it generates, use standard library modules to manage symbol tables, sets, graphs, memory allocation, text buffering, command line argument processing and scanning. The rdp scanner is programmed by loading tokens into a symbol table at the start of each run. In this way, the rdp scanner can be used to support runtime extensible language features, such as user defined operators in Algol-68.

rdp offers a high level of integration between its component parts. An unusual feature of rdp is that it produces complete runnable programs with built-in help information and command line switches that are specified as part of the IBNF file. In this sense rdp output is far more shrink-wrapped than the usual parser generators which can be a great help to new users. rdp also provides integrated I/O: the text buffering routines and built-in scanner work together to automatically handle nested files, error message reporting and text data buffering.

rdp generates itself from an IBNF file which describes the syntax of rdp's IBNF source language and specifies semantic actions. This demonstrates the bootstrapping technique used for porting compilers to new architectures. The rdp distribution comes with extensive documentation which includes a tutorial manual for new users and a large case

study in which a family of interpreters and compilers are developed.

Iterator BNF

`rdp` language specifications use the standard notions of context-free grammar rules first employed in the Algol-60 report [Bac60]. Rule names are unquoted alphanumeric identifiers and keywords are delimited by single quotes. In addition to the usual notions of sequencing and alternation from BNF `rdp` provides a generalised iterator operator. 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*

which 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 represents 'nothing' or the empty string (sometimes represented by ϵ). In this case no separating token is looked for.

Traditional EBNF forms `rdp` supports the traditional Wirth-style EBNF constructs [Wir77] for optional, do-first and Kleene closure brackets as shorthands for special cases of the iterator construct. We have extended Wirth's set with a positive closure operator. The correspondences are shown in Table 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*).

Using iterators to implement lists Delimited lists are common in high level languages. Consider, for instance, a function call in C made up of a parenthesised comma-delimited list of identifiers:

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

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 ].
```

which 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) @ ',' ] ')'.
```

Attributes and semantic actions

Table driven parsers usually maintain an *attribute stack* which may be accessed by semantic actions in the running parser to synthesize run-time attribute values. Careful use of the attribute stack may also allow implementation of inherited attributes. One of the key advantages of recursive descent parsers is that inherited and synthesized attribute passing between grammar rules maps naturally onto the function parameter and function return value mechanisms in conventional programming languages. The `rdp` tutorial and case study manuals [JS97f, JS97a] show how this mapping is exploited to provide individual named (rather than numbered) attributes along with strongly type checked inherited and synthesized attribute handling without requiring a separate attribute stack.

`rdp` rules may have a *return type* which is used to pass the value of a synthesized attribute up the

do-first	`( ... )`	→	`( ... ) 1@1 #`
positive closure (one-or-many)	`< ... >`	→	`( ... ) 1@0 #`
optional (zero-or-one)	`[ ... ]`	→	`( ... ) 0@1 #`
Kleene closure (zero-or-many)	`{ ... }`	→	`( ... ) 0@0 #`

Table 1 Iterator:bracket correspondences

derivation tree. Inherited attributes are defined using typed parameters. All types refer to type names defined in the underlying ANSI-C implementation which may include user defined type names (including the use of aggregates). Semantic actions are ANSI-C fragments enclosed in `[* *]` brackets that are simply copied into the generated parser verbatim. For instance, the rule

```
a_rule(x1:integer x2:integer):integer::=
  'a' [* result = x1; *]
  { 'b' [* result = x2; *]
  }.
```

matches a single `a` followed by zero or more `b`'s. If no `b`'s are seen then `a_rule` returns the value of the `x1` parameter, otherwise the value of `x2` is returned. The rule maps to the following ANSI-C skeleton:

```
integer a_rule(integer x1, integer x2)
{
  /* ..code to match a.. */
  result = x1;

  if (current is b)
  {
    /* ..code to match multiple b's.. */
    result = x2;
  }

  return result;
}
```

The value returned by `a_rule` may be accessed in a rule that calls `a_rule` by appending an attribute name to the call:

```
upper_rule ::= a_rule(0 1): saw_b.
```

Here, the synthesized attribute `saw_b` will be set to `1` if rule `a_rule` matched any `b`'s, otherwise `saw_b` will be set to `0`.

Default actions For iterators with a lower bound of zero (which include `[ ]` and `{ }`) it is often convenient to have a semantic action that is executed only if the iterator matches ϵ, that is a *default* action that is executed if the syntax matched by the iterator is not found. Such actions are defined by following the iterator with a colon and an action:

```
rule_with_default ::=
  'a' [
       'b' [* printf("Found a 'b'\n"); *]
      ] : [* printf("No 'b' found\n"); *]
```

This rule matches the strings `ab` and `a`, printing appropriate messages.

Integrated library features

`rdp`-generated parsers use a set of general purpose support modules known collectively 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; a set handler which supports dynamically resizable sets; 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; a memory manager which wraps fatal error handling around the standard ANSI C heap allocation routines; a text handler which provides line buffering and string management without imposing arbitrary limits on input line length; a command line argument parsing package that allows Unix style options to be implemented in a standardised way; and scanner support routines for handling tokens in recursive descent parsers.

Symbol tables Symbol tables are fundamental to the implementation of almost all useful translators

since in practice the language accepting power of context free grammars must be augmented with context sensitive type checks. In addition, interpreters require storage space for their run-time variables and associated attributes, and an efficient symbol table organisation is critical to the performance of such tools. The `rdp_supp` library includes a hash-coded symbol table handler interfaced to the `rdp` source language *via* the `SYMBOL_TABLE` declaration. An arbitrary number of symbol tables may be declared, each with user defined data fields and a user defined record comparison function. An arbitrary number of nested scope levels are supported.

Generalised graph handling The `rdp` graph handling package provides a general framework for building graph data structures that may then be output in a form suitable for display with the VCG (Visualisation of Compiler Graphs) tool. `rdp` generated parsers can be set to automatically build derivation trees in a form suitable for human viewing. VCG runs on Windows 3.1, Windows-95 and Unix/X-windows systems. We are grateful to the author of VCG for permission to supply VCG with `rdp`: you can fetch a copy of VCG from the home FTP site for `rdp` (`ftp://ftp.dcs.rhbnc.ac.uk/pub/rdp`)

Scanning The `rdp` scanner uses the same compiled constructs as the generated parser functions rather than employing traditional Finite Automata based techniques. Alphanumeric keywords and punctuation are recognised *via* longest-match comparisons with the contents of the scanner's symbol table which is loaded at startup with the tokens referenced in the associated IBNF grammar. This dynamically organised scanner allows new operators and keywords to be added to the scanner's set during translation, a feature designed to support the use of Algol-68 style operator definitions (as opposed to the more restricted overloading of operators allowed in C++). The scanner is tightly integrated with a text buffering package that manages the flow of source text through the translator and provides a range of messaging and text handling services.

Help and command line processing functions One of the aims of `rdp` is to allow neophyte users to get a complete translator up and running in the minimum of time by producing a complete, runnable program that includes built-in help processing at command line level and customisable handling of command line switches. The generated parsers automatically include processing for a set of standard command line arguments and the user may additionally specify command line switch using directives in the IBNF source file. The results of command line processing are available as attributes within the running translator.

Derivation tree construction and manipulation

`rdp`-generated parsers can automatically build complete derivation trees which can be used as the basis of an intermediate form suitable for input to tree-walking code generators and optimisers. The trees are constructed using the graph library, and therefore can also be output to a text file and displayed using the VCG graph visualisation tool.

Full derivation trees consume a lot of space, and often contain nodes that are of little use in subsequent language processing. In practice, translation tools use simpler Abstract Syntax Trees (ASTs) but 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 it is, of course, possible to adopt this approach with `rdp`-generated parsers, 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 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 structures requires two independent tree-like forms to be described whereas in `rdp` the grammar itself fulfills both functions.

```
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 *)
e4 ::= e5 [ '**'^^ e4 ]:^^.           (* Exponentiate (RA)       *)
e5::= ID^^                            (* Variable or ...         *)
      ['('^ (e1)@',' ')'^ ] |         (* ... function call       *)
      INTEGER^^             |         (* Numeric literal         *)
      '('^ e1^^ ')'^.                 (* Bracketed subexpression *)
```

Figure 1 An `rdp` expression grammar showing tree promotion operators

Modifying tree construction with promotion operators There are four promotion operators. 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. 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. 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. The `^_` (no promotion) operator is used to apply the normal behaviour to a nonterminal whose default behaviour has been overridden.

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 will be inserted into the derivation tree as a child of the current parent. The grammar shown in Figure 1 describes a small operator language with both left and right associative operators. Figure 2 shows the full derivation tree that results from using this grammar to parse the string

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

Figure 3 shows the reduced derivation tree that is produced when the promotion operators in the grammar are enabled.

Documentation

Four manuals describe the `rdp` system and its applications. The user guide [JS97d] 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 [JS97e]. 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 [JS97f]. 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.

Availability

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 and Windows-NT, 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. `rdp` has been in use at a variety of indus-

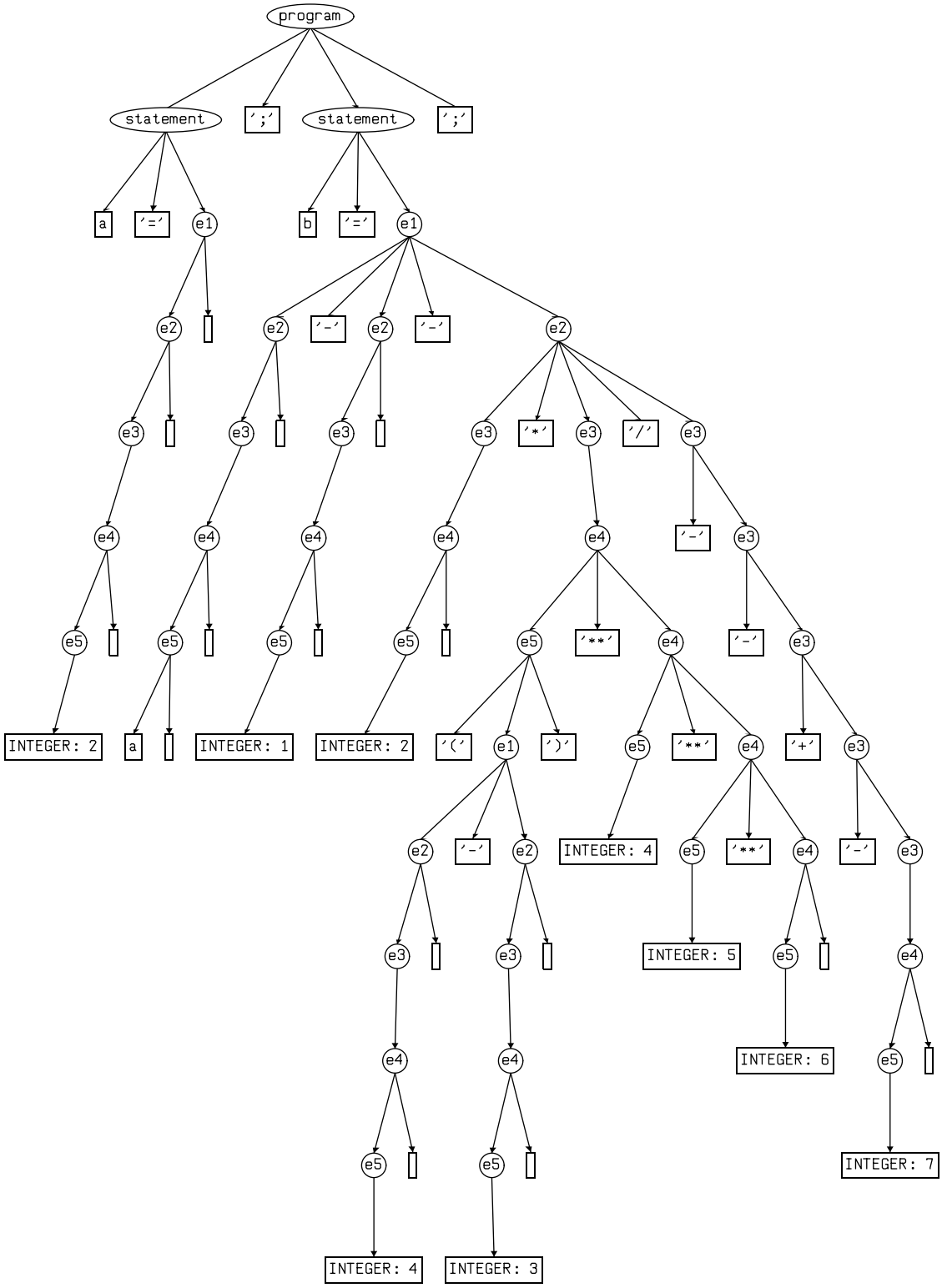

Figure 2 Full derivation tree for expression grammar

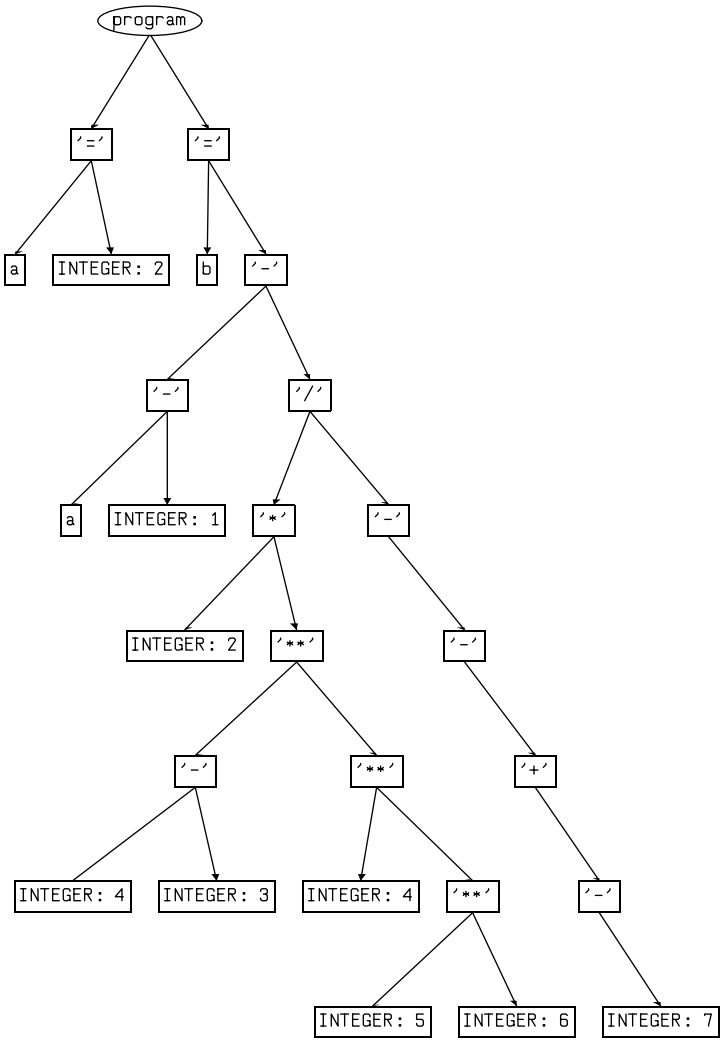

Figure 3 A reduced derivation tree

trial and academic sites since 1994 for both teaching and the generation of production translators. The current version (1.5) is the sixth functionality release and the authors would like to acknowledge the many suggestions for features and improvements that have been provided by our users.

`rdp` may be fetched using anonymous ftp to `ftp.dcs.rhbnc.ac.uk`. Unix users should download the file `pub/rdp/rdp1_5.tar`. MS-DOS, Windows 3.1 and Windows-95 users should download `pub/rdp/rdp1_5.zip`. The `rdp` distribution may also be accessed *via* the `rdp` Web page `http://www.dcs.rhbnc.ac.uk/research/languages`.

Future work

The `rdp` iterator and tree construction features have lead to further work on generalised backtracking parsers that will form the basis of a new tool currently under development provisionally called the Permutation Grammar Toolbox (PGT). We have reported elsewhere on the theoretical and practical aspects of our generalised recursive descent parsers [JS97b, JS97c, JS98] and the Web site hosts reports and prototype versions of the generalised parser generator.

References

[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. Generalised recursive descent. Part 1: language design and parsing. Technical Report TR-97-18, Royal Holloway, University of London, Computer Science Department, October 1997.

[JS97c] Adrian Johnstone and Elizabeth Scott. Generalised recursive descent. Part 2: some underlying theory. Technical Report TR-97-19, Royal Holloway, University of London, Computer Science Department, October 1997.

[JS97d] 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.

[JS97e] 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.

[JS97f] 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.

[JS98] Adrian Johnstone and Elizabeth Scott. Generalised recursive descent parsing and follow determinism. In Kai Koskimies, editor, *Proc. 7th Intnl. Conf. Compiler Construction (CC'98), Lecture notes in Computer Science 1383*, pages 16–30, Berlin, 1998. Springer.

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

[Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions. *Communications of the ACM*, 20(11), November 1977.