# Designing and implementing language translators with `rdp` – a case study

Adrian Johnstone Elizabeth Scott

Technical Report

CSD – TR – 97 – 27

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 specification, written in an extended Backus-Naur Form, of a source language and produces as output a parser for the language, written in C. It is possible for the user to specify, in C, actions which should be taken when fragments of the source language are recognised by the generated parser. `rdp` produces as output a program written in C, which parses fragments of the specified language and carries out the specified corresponding actions. `rdp` can produce, for example, compilers (the actions specify the corresponding target code), interpreters (the actions evaluate the input fragments) and pretty printers (the actions reformat the input fragments).

This report describes the design and implementation of a family of language translators based around a simple procedural programming language called `mini`. The tools covered include two different interpreters, an assembler, a simulator for an idealised processor, a naïve single-pass compiler for that processor, and a multiple-pass compiler. We also include a pretty printer for ANSI-C and a list of sugestions for further project work. For each tool, we look at how the tool is used before covering the design and implementation of the translator. All of the tools are included in the `rdp` standard distribution pack and have been tested on MS-DOS, Windows-95 and Unix systems.

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



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

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

# Contents

# Chapter 1

# Translation tools

rdp is a system for implementing language processors. It accepts a specification, written in an extended Backus-Naur Form, of a source language and produces as output a parser for the language, written in C. It is possible for the user to specify, in C, actions which should be taken when fragments of the source language are recognised by the generated parser. rdp produces as output a program written in C, which parses fragments of the specified language and carries out the specified corresponding actions. rdp can produce, for example, compilers (the actions specify the corresponding target code), interpreters (the actions evaluate the input fragments) and pretty printers (the actions reformat the input fragments).

This report describes the design and implementation of a family of language translators based around a simple procedural programming language called mini. The tools covered include two different interpreters, an assembler, a simulator for an idealised processor, a naïve single-pass compiler for that processor, and a multiple-pass compiler. We also include a pretty printer for ANSI-C and a list of sugestions for further project work. For each tool, we look at how the tool is used before covering the design and implementation of the translator. All of the tools are included in the rdp standard distribution pack and have been tested on MS-DOS, Windows-95 and Unix systems.

If you have not used rdp before, we recommend that you read through the accompanying report entitled *'A tutorial guide to* rdp *for new users'* [JS97c] which is a step by step guide to running rdp and which also describes some of the theoretical underpinnings to parsing and translation of computer languages. There are also two reference manuals for rdp: the user guide [JS97a] and the support library manual [JS97b]. These reference guides provide detailed information on rdp's options, error messages and support library functions.

We begin by discussing the roles of interpreters and compilers, with some historical background.

## 1.1 The spectrum of language translators and the limitations of single pass translators

rdp can be used to construct many kinds of translator. In the tutorial guide [JS97c] we looked at a single pass *interpreter* for a very simple language called `mini`. These kinds of translators are limited to reading the source file once and executing embedded semantic actions on the fly. This makes it hard to implement loop constructs, which of course require parts of the source file to be executed over and over again. This is the reason why the mini language interpreter described in the tutorial manual does not support looping constructs.

One approach to handling loops within an interpreter might be to 'trick' the parser by resetting the input pointer to the start of the mini source code loop at the beginning of each loop iteration: a *rewindable* interpreter. This technique is feasible, but requires a detailed understanding of the internals of **`rdp`**. It also results in rather slow translation. Experiments with the `mini` interpreter show that when interpreting arithmetic expressions, about 90% of the time is spent performing the parse and only 10% of the time performing useful computation.

In fact, even this discouraging ratio represents the best-case. The use of comments and long variable names can significantly increase the proportion of time spent on parsing. This is unfortunate as it militates against use of meaningful names and embedded documentation, leading to cryptic and hard to understand programs. Treating loops using the rewinding trick would mean that the loop would be re-parsed over and over again, and such an interpreter would be slow. Nevertheless, this kind of trick is used in some real systems: in particular BASIC interpreters (such as the Visual Basic engine built in some Windows-95 tools) work this way. To improve the performance a little, it is normal for such tools to store the program in a format that strips out comments and white space, and replaces keywords with single characters. This eases the job of the scanner and helps to improve performance.

A *compiler* does not attempt to execute a program in the way that an interpreter does. Instead, it outputs a program in the machine language of some *target processor* which can be directly executed by that processor. The compiler's main task is to identify operations in the source program and map them to code templates in the target processor's language that have the same meaning, or *semantics*.

Full compilation undoubtedly provides the most efficient way of executing most real programs, but a different target program will be required for each kind of target processor, that is, the generated code is not portable between architectures (or in extreme cases, not even between different models of computer within the same architectural family). One approach to providing a measure of portability is to strictly separate the parsing stage (which is specified by the design of the language to be translated) and the generation phase, which is keyed to the architecture of the target processor.

This is usually achieved by allowing the parser to make one or more passes over the source program and by providing embedded semantic actions that translate the program into some simple *intermediate form* which captures the meaning of the program without requiring the large syntactic overhead of keywords and English-like syntax that are used in most human-readable programming languages. The compilers `miniloop` and `minitree` (described in Chapters 7 and 8 respectively) are examples of this approach.

## 1.2 Intermediate forms, and translation to virtual machine code

Intermediate forms used in real compilers fall into two basic types: a tree like-structure closely related to the derivation trees described in [JS97a] or alternatively a list of instructions for a paper architecture or *virtual machine*[1]. The virtual machine approach is illustrated by the `miniloop` compiler in Chapter 7 and the tree based approach by the `minitree` compiler described in Chapter 8. Virtual machines are superficially similar to real processors, but they offer a level of abstraction above that of a real processor. For instance, it is common in intermediate forms to retain the variable names from the original user's program rather than translating them into machine addresses as would be required for a real machine level program.

Both kinds of intermediate form allow a variety of *optimisations* to be applied, such as the evaluation of constant expressions or the replacement of multiplications by powers of two with shift operations. In general, an optimiser is supposed to take a program in the intermediate form and output another program written in intermediate form that has the same semantics, but is faster or more compact, or both. Sometimes optimisers fail to make improvements, and in some cases they may actually make things worse. In addition, some features of programming languages (such as the unrestricted use of pointers) can introduce subtle effects that make it hard for the optimiser to guarantee that the semantics are preserved.

After the optimiser has finished, code must be generated for the target processor. In general, there must be a different code generator for each processor, but at least all of the parsing and many of the optimisation components of the compiler can be common between target processors.

One way of providing the benefits of full portability whilst retaining much of the efficiency of a fully compiled solution is use an intermediate form that can itself be efficiently interpreted. In this case no final code generation phase is required. Instead, a software simulator for the virtual machine which can read and execute the intermediate form is supplied. This kind of approach was popularised by the UCSD P-system in the 1970's which was a combined operating system and Pascal compiler that was distributed as *P-code*. P-code [PD82b, PD82a] was in fact the machine language for a mythical stack based computer that could be efficiently simulated on real architectures. The system was so successful that a microprocessor manufacturer subsequently designed and marketed a hardware implementation of the P-code processor. On this processor, the P-code *was* native machine code so no software based interpretation was required.

P-code was successful because its only real competitors on the very small microprocessor based systems of the time were interpreters for BASIC. These

[1] This use of the term virtual machine to denote an architecture that is independent of any physically implemented machine should not be confused with the use of the term in operating systems and computer architecture contexts, where it denotes the ability of an architecture to support multiple simultaneously executing processes each of which appears to own the full resources of the host machine.

fully interpreted languages were slow compared to the P-code simulator. As microprocessor systems matured, true compilers for languages such as C and Turbo-Pascal that compiled to the host machine's machine code became widely available and the UCSD P-system fell out of favour because it was much slower than these so-called native-mode compilers.

Recently, virtual machine based approaches have become popular again because of the need to distribute executable programs around the Internet. Portability between different computer architectures must be absolutely guaranteed even though there are a very wide variety of systems connected to the net, and the programs must run in an identical fashion wherever they are executed. In addition, the programs must be run in such a way that any suspicious behaviour that might undermine the host system's security can be caught. In practice, actually allowing arbitrary machine language programs to execute is too dangerous. Instead, languages like Java compile to an intermediate virtual machine, and Web browsers provide an interpreter for that virtual machine that can in prinipal catch illegal memory accesses and attempts to access operating system services that could threaten system integrity. The Java virtual machine simultaneously acts as a reasonably efficient portable platform for executing programs and as a filter on the actions of those programs that protects the underlying operating system.

## 1.3 The `mini` family

In the following chapters we will illustrate the interpreted virtual machine approach to compilation by describing the development of single and multiple-pass compilers for `mini` which work in this fashion, outputting instructions for a 'paper' processor called the Mini Virtual Machine (MVM). Along the way we will look at fully interpreted versions of `mini` and the design of an assembler and simulator for MVM. The level of presentation is aimed at readers who are familiar with the principles of parser generators and the ANSI-C programming language. If you are completely new to translator design you may find it helpful to read the `rdp` tutorial manual [JS97c] and the accompanying user [JS97a] and support [JS97b] manuals.

In detail we will develop the following tools.

1. A syntax checker and interpreter for `minicalc`, a language that provides declarations, assignment, expression evaluation and output.

2. An interpreter for `minicond` which has block statements, relational operators and an `if-then-else` statement in addition to the basic `minicalc` language.

3. A paper architecture called the Mini Virtual Machine (MVM) and its specification as a simulator for MVM (called `mvmsim`) written in ANSI-C.

4. An assembler called `mvmasm` that translates MVM assembly language into MVM binary code. The implementation of `mvmasm` illustrates the design

issues in assemblers which are culturally rather different from high level programming languages.

5. A single-pass compiler for the language `miniloop` which adds a while loop construct to `minicond` and outputs MVM assembler source, suitable for translation with `mvmasm` into MVM binary which may be executed by `mvmsim`.

6. A multiple-pass compiler in which the parser builds a tree-based intermediate form and a separate code-generation pass traverses the tree and outputs MVM assembler source code which may then be assembled and simulated.

7. A pretty-printer for ANSI-C which illustrates the use of a highly under-specified grammar to process a language which will be checked for syntactic correctness by another tool using a fully specified language.

We conclude this report by suggesting some design projects based on extensions to the compilers.

All of these tools are included in the `rdp` distribution and are automatically built and tested as part of the standard installation `makefile`. If you have successfully installed `rdp`, therefore, you should already have working versions of the tools, and all the source files described here will be found in the main `rdp` directory.

# Chapter 2

# The `minicalc` language: a simple calculator with declared variables

In this chapter we give a grammar and associated syntax checker for a tiny language, `minicalc`, which includes only the features at the core of any procedural programming language — expression evaluation and assignment to named variables. In `minicalc`, as in most modern languages, variables must be declared before they are used, so as to catch the elementary programming error of assigning to a variable whose name has been misspelled. In early high level programming languages, a system would quietly make a new variable with the misspelt name and assign the value there. Subsequent expressions using the value of the correctly spelt variable would then use the old value, causing hard to find errors.

Variable declarations are also used to establish the *type* of a variable which restricts the kinds of values that may be assigned, and the kinds of operations that may be applied to the declared variable. Type checking can catch programming errors such as trying to add a number to a string, or attempting to use an integer instead of a pointer value.

`minicalc` provides constructs for variable declaration, for assignment of the results of arithmetic operations to those variables and for the values of those variables to be printed out. It is, effectively, only as powerful as a desktop calculator with named variables. An example `minicalc` program listing is shown in Figure 2.1: it corresponds to the file `testcalc.m` in the standard `rdp` distribution. In later chapters we shall extend `mini` to include control structures such as loops and `if` statements.

## 2.1 `minicalc` features

`minicalc` programs comprise a sequence of declarations and statements. Each statement and declaration must be terminated by a semicolon, in much the same way as in an ANSI-C program. `minicalc` supports only integer variables. Variable declarations look like ANSI-C `int` declarations, taking an optional initialisation expression. Line 11 in the listing shows an example of two variables being declared, both with initialisation expressions.

```
 1: (******************************************************************************
 2: *
 3: * RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
 4: *
 5: * testcalc.m - a piece of Mini source to test the Minicalc interpreter
 6: *
 7: * This file may be freely distributed. Please mail improvements to the author.
 8: *
 9: ******************************************************************************)
10:
11: int a=3+4, b=1;
12:
13: print("a is ", a, "\n");
14:
15: b=a*2;
16:
17: print("b is ", b, ", -b is ", -b, "\n");
18:
19: int z = a ** 3;
20: print(a, " cubed is ", z, "\n");
21:
22: (* End of testcalc.m *)
```

**Figure 2.1** An example `minicalc` program (`testcalc.m`)

`minicalc` expressions are built up using the four basic diadic left associative arithmetic operators (`+, -, * and /`) along with unary `+` and `-` and the diadic right associative exponentiation operator (`**`). Operands may be either numeric literals or variable names. The result of an expression may be assigned to a variable, as shown in line 15, or used within a `print` statement, as shown in line 17. `print` statements take a parenthesised, comma delimited list of expressions or strings, which are evaluated and printed in left to right order, much like the Pascal `write` statement. The usual ANSI-C escape sequences may be used to output non-printing characters such an line end (represented by `\n`).

## 2.2 `minicalc` limitations

`minicalc` can only perform integer computations, and only allows strict sequencing of statements, there being no flow of control statements. In addition, there is no `read` input statement to accompany the `print` output statement. Enhanced versions of `minicalc` will be presented in later chapters.

## 2.3 A grammar for `minicalc`

It is easy to construct an LL(1) grammar for `minicalc` suitable for input to `rdp`. A parser generated from such a grammar with no semantic actions embedded within it acts as a pure parser or *syntax checker* for the `minicalc` language. Figure 2.2 shows a suitable grammar from which to generate a `mini` syntax checker: it is supplied with the `rdp` distribution as file `mini_syn.bnf` and will be discussed in the remainder of this section.

```
(******************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* mini_syn.bnf - a mini grammar for syntax checking
*
* This file may be freely distributed. Please mail improvements to the author.
*
*******************************************************************************)
TITLE("Mini_syn V1.50 (c) Adrian Johnstone 1997")
SUFFIX("m")

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

var_dec   ::= 'int' ( ID [ '=' e1 ] )@','.

statement ::= ID '=' e1 | 'print' '(' ( e1 | String )@',' ')'.

e1 ::= e2 { '+' e2 (* Add *)      | '-' e2 (* Subtract *) }.

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

e3 ::= e4 | '+' e3 (* Posite *)   | '-' e3 (* Negate   *).

e4 ::= e5 [ '**' e4 ] (* Exponentiate *).

e5 ::= ID (* Variable *) | INTEGER (* Numeric literal *) | '(' e1 ')'.

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

String  ::= STRING_ESC('"' '\\').     (* Strings for print *)

(* End of mini_syn.bnf *)
```

**Figure 2.2** An rdp grammar specification for minicalc (mini_syn.bnf)

A minicalc source program comprises an arbitrary number of variable declarations and statements, each terminated with a semicolon, as specified by rule program in line 13. Variable declarations, specified in line 15, comprise the keyword int followed by a comma delimited list of variable names which may be optionally followed by an = sign and an initialisation expression.

Statements in minicalc may be either assignments or print statements, as specified in line 17. The print statement takes parameters by way of a comma delimited list of strings and expressions.

Comments present particular problems for parser generators because the modern convention is to allow a comment to occur anywhere that whitespace is allowed: typically between any two tokens of the language. A full specification of this convention using only the grammar rules would require a call to a comment rule after every terminal in the grammar, and this would make the grammar very unwieldy. The usual solution is to instruct the scanner to detect and filter out comments in exactly the same way as whitespace (such as line ends, tabs and space characters) is discarded.

rdp offers a range of scanner primitives to support three different commenting conventions. You can read more about the use of these primitives, and the general problems of comment specification in Chapter 4 of the user manual [JS97a]. In mini_syn.bnf comments comprising matching (* and *) brackets are specified on line 29. This rule is only used to parameterise the scanner, and is never actually called by the main parser so it will be deleted when rdp processes the grammar.

## 2.3.1 Specifying expressions

The forms of mini expressions are specified on lines 19–27. Most programming languages provide support for expressions made up of operators for common arithmetic and logical operations. Some even allow control flow to be specified using operators: the ANSI-C if-then-else operator (? :) is perhaps the most well known example of this feature.

Syntactically, expressions are simply streams of operator and operand tokens. For an expression made up of diadic (two-operand) operators we expect to see expressions of the form

operand operator operand ... operand operator operand

Monadic (single operand) operators are distinguished by appearing between diadic operators and operands or next to other monadic operatirs. Here is an expression made up of monadic - (negate) and the diadic addition and subtraction operators.

```
3 + -4 - -6 + 2
```

The following is a simple grammar that generates expressions of this form:

```
expression ::= {monadic_op} operand { diadic_op {monadic_op} operand }.
operand ::= INTEGER.
monadic_op ::= '-'.
diadic_op ::= '+' | '-'.
```

If all we want to do is to check that an expression is syntactically valid then we can simply extend the definitions of `monadic_op` and `diadic_op` to include all the monadic and diadic operators present in the language being specified. Here, for instance is a suitable grammar for `mini` expressions which include monadic `+` and `-` and diadic `+`, `-`, `*`, `/` and `**` (exponentiation).

```
expression ::= {monadic_op} operand { diadic_op {monadic_op} operand }.
operand ::= INTEGER.
monadic_op ::= '+' | '-'.
diadic_op ::= '+' | '-' | '*' | '/' | '**'.
```

If we extend the definition of rule `operand` we can similarly allow named variables and non-integer literal operands such as `REAL`s.

Deeper problems arise when we consider the evaluation of an expression. The string `fred = 1 - 2 - 3` is a legal `minicalc` statement, but what value should actually be assigned to `fred`? We could begin evaluating from the left, effectively carrying out the steps

```
fred = 1
fred = fred - 2
fred = fred - 3
```

or from the right, effectively carrying out the steps

```
fred = 3
fred = 2 - fred
fred = 1 - fred
```

In the first case `fred` ends up with value -4 and in the second case with value 2. The usual convention is to evaluate subtractions from left to right, *i.e.* using the first of the two choices above, and this is called *left associativity.* The programmer may wish to override this and other order-of-evaluation conventions, and traditionally this is done using parentheses. The following grammar allows expressions of the form `1 - (2 - 3)` and `3 + (x * y) - 2`

```
expression ::= {monadic_op} operand { diadic_op {monadic_op} operand }.
operand ::= INTEGER | ID | '(' expression ')'.
monadic_op ::= '+' | '-'.
diadic_op ::= '+' | '-' | '*' | '/' | '**'.
```

Note that we have introduced recursion into the grammar: the `operand` rule accepts a left parenthesis after which it calls the `expression` rule before accepting a matching closing parenthesis. This nested structure incorporates the notion of 'do-first' into the grammar which is exactly what parentheses 'mean' in expressions — the parentheses in an expression are used to override the default order of expression evaluation so that the sub-expression in parentheses is evaluated first.

### Specifying operator priority

We have seen that parentheses can be used to override default evaluation orders in expressions. However, expressions with lots of parentheses can be hard to read (although LISP programmers seem to manage) and so the conventions of operator priority have grown up to allow expressions to be written with *implicit* parentheses. Most people use priority rules instinctively because they are taught to us when we first learn arithmetic at school. Multiplication and division, for instance, have higher precedence than addition and subtraction. This means that the expression `3 + 4 * 5` evaluates to `23` *not* `35` as would be the case if strict left-to-right evaluation were used. Effectively, the priority allows us to write `3 + 4 * 5` as a shorthand for `3 + (4 * 5)`. If we need to force left-to-right evaluation then we can write `(3 + 4) * 5`. In conventional arithmetic, exponentiation has the highest priority followed by negation, multiplication (and division) and finally addition (and subtraction). We can express these priorities by using the nest of production rules shown in lines 19–27 of Figure 2.2.

### Specifying operator associativity

In programming languages, operators take values of a particular type and return values with a type. Integer addition, for instance, takes two integers and returns an integer, and the 'greater than or equal' relational operator `>=` takes two integers and returns a boolean result. If the return-type of an operator is the same as the type of its acceptable operands then we can write expressions that contain a run of similar operators such as `3 + 4 + 5` which we can read as either `(3 + 4) + 5` or `3 + (4 + 5)`. With addition, both interpretations evaluate to `12` but if we use subtraction instead then an ambiguity arises: `(3 - 4) - 5` is `-6` but `3 - (4 - 5)` is `3 - (-1)` which is `4`. The *associativity* of an operator specifies which of the two interpretations should be selected: subtraction is in fact *left associative* so `3 - 4 - 5` is interpreted as `(3 - 4) - 5`.

In general, if we have several operators at the same level of priority, we need to decide in which order to evaluate the operands. In most cases we evaluate from left to right, so that `2 - 3 + 4` evaluates as `(2 - 3) + 4` not as `2 - (3 + 4)` and `2 / 3 * 4` evaluates as `(2 / 3) * 4`. Evaluating from left to right automatically gives each of the operators left associativity.

Left to right evaluation is most common, but strings of exponents, such as `2 ** 3 ** 4`, are traditionally evaluated in right to left order. The `mini` grammars demonstrate how to ensure left to right (and right to left) operand evaluation and we shall now discuss this in detail.

### Left to right evaluation and left associative operators

The left associative arithmetic operators `+` and `-` are specified with rules of the form

```
e1 ::= e2 { '+' e2 | '-' e2 }.
```

The rule specifying the left-associative operators (`e1` in this case) calls its immediate successor rule in the operator tree (`e2`) on both sides of the operators being recognised in e1. This has the effect of ensuring that higher priority operators on both sides of a + or - are evaluated first. In addition, the use of the *zero-or-many* iterator bracket { } ensures that a run of + and/or - operators is processed in strictly left to right order, i.e. the operators are evaluated in a left associative manner.

## Right to left evaluation and right associative operators

Right associative operators, such as exponentiation ** are specified using rules of the form

```
e4 ::= e5 { '**' e4 }.
```

The crucial difference between this kind of rule for right associative operators and the previous rule for defining left associative operators is that in this case the rule calls *itself* on the right hand side of the operator. Because of the way the iterator works, this ensures that, in a run of exponentiation operators all of the operators to the right of the first one will be processed before the first one is processed. By extension, it is easy to see that the effect of this kind of right-recursive rule is to ensure that a run of exponentiation operators is evaluated in the *reverse* order to that in which they are read, that is right to left, which is what we require for a right associative operator.

In detail, it is clear that the right recursion will absorb all instances in a run of exponentiation operators, so even though we have used the *zero-or-many* iterator bracket { } each invocation of rule e4 can only ever absorb zero or one instances of the ** operator, so in practice we write such rules in this way:

```
e4 ::= e5 [ '**' e4 ].
```

## Operators that do not combine

Some operators yield result values that are incompatible with their operands and therefore can not be used next to each other in expressions. It is not mathematically meaningful, for instance, to write an expression like 3 < 4 <= 6 because the result of evaluating 3 < 4 is a boolean truth value and this cannot reasonably be compared to the integer 6. We might loosely call such operators 'non-associative' but strictly speaking it is meaningless to speak of the associativity of such an operator. The arithmetic relationals are the standard examples of such operators, and we implement them using rules such as

```
rel_expr ::= exp [ '>' exp ].
```

The important point here is that the curly braces { } used for the expression rules e1 and e2 in Figure 2.2 to specify zero or more consecutive instances of an operator are replaced by square brackets [ ] which only allow zero or one occurrence of the relational operator, so a sequence of such operators in an expression will be rejected by the parser.

In the next chapter we shall add relational operators to mini using a rule of the form described here. It is, perhaps, worth noting that in some languages this issue is rather obscured by the confusion of boolean values with integer values. Languages such as Pascal are strict whereas others (such as ANSI-C) simply use integer values instead of true booleans and may even provide fully left associative relational operators.

## 2.4 Hints on selecting operator priority and associativity

Most common operators are left associative since this corresponds to a left-to-right evaluation rule which is natural for cultures that read left-to-right. Occasionally an operator is given right associativity for special reasons. The exponentiation operator `**` is an example of such an operator. The reason that exponentiation is traditionally right associative is that, when written in the traditional mathematical notation using position rather than a symbol for the operator, the expression $(x^y)^z$ can be trivially rewritten as $x^{yz}$. Since there is already such a straightforward way of writing left associating exponentiation it makes sense to define the exponent operator as right associative, so $x^{y^z}$ corresponds to $x^{(y^z)}$. Thus, using the programming language notation, `x ** y ** z` is interpreted as `x ** (y ** z)`.

Similar considerations may be used to decide the relative priorities of operators. Remember that the higher priority operators need to be evaluated first in an expression. In common usage, for instance, it is clear that the expression `-2 ** 2` (i.e. $-2^2$) is expected to yield `-4`, that is, it is a shorthand for `-(2 ** 2)` *not* `(-2) ** 2` which would yield `+4`. Hence we must give exponentiation higher priority than monadic `-`. On the other hand it is also clear that `-2 * -2` evaluates to `+4`, not `-4`, that is we should interpret the expression as `(-2) * (-2)` from which we deduce that multiplication has *lower* priority than monadic `-`.

The number of priority levels provided by a language is a fundamental design decision. Pascal provides rather few levels, and in particular expressions containing adjacent logical operators *must* be parenthesised. ANSI-C goes to the other extreme and provides so many priority levels that many C programmers are unsure of the relative priorities of unusual operators. From the perspective of the language user (as opposed to designer) the golden rule is: if in doubt, insert explicit parentheses.

## 2.5 A `minicalc` syntax checker

As it stands, `mini_syn.bnf` can be processed by `rdp` in the normal way to make a syntax checker for `mini`. Such a checker can detect badly spelled keywords, and syntactically ill formed expressions but is not able to check that variables have been declared before use.

The `rdp` make file contains the commands for constructing a syntax checker from `mini_syn.bnf` and running it on a test file called `testcalc.m`. These commands will be executed if you type

```
make ms_test
```

```
******:
     1: (* Erroneous minicalc input *)
     2:
     3: int a;
     4:
     5: Error 1 (error.m) Scanned ID whilst expecting '='
     5: innt b = 3;     (* should be int b; *)
     5: -----1
     6:
     7: b=a*2;          (* a used before being initialised *)
     8:
     9: Error 1 (error.m) Scanned '*' whilst expecting one of ID, INTEGER, '(', '+', '-'
     9: a = 3 - * 4;    (* should be a = 3 * - 4; *)
     9: --------1
    10:
    11: bb = b * 3;     (* undeclared variable *)
    12:
******: 2 errors and 0 warnings
******: Fatal - errors detected in source file
```

**Figure 2.3** Error reporting in the syntax checker

This make file target is automatically built as part of the standard installation, so if you have already built `rdp` using the make file then the already-compiled syntax checker will simply be run on the test file.

Figure 2.3 shows the output of the syntax checker for an erroneous program and illustrates the syntax checker's limitations. The misspelling of `int` in line 5 and the incorrect orderings of the arithmetic operators in line 9 have been detected, but the use of an uninitialised variable in line 7 and the assignment to an undeclared variable in line 11 have been ignored.

These kinds of errors can only be detected by checking *long range* relationships between program symbols. A variable declaration may occur a long way before that variable is used, but the context free grammars used by `rdp` are essentially only powerful enough to check local features of the language. A *context sensitive* grammar may be written in such a way as to support type checking, but efficient parsing techniques for practical context sensitive grammars are not available. Instead, we use an external symbol table and embedded semantic actions to keep track of the declaration and use of identifiers. Our next tool, which is a full interpreter for `minicalc` can check for undeclared variables without any extra overhead: the interpreter needs a symbol table anyway to keep track of computed results and it turns out that adding checks for undeclared variables is straightforward.

# Chapter 3

# An interpreter for `minicalc`

The primary purpose of **`rdp`** is to construct a parser for the language generated by an **`rdp`**-IBNF specification. Such a parser may be used as a syntax checker for the language, as we have seen in the previous chapter. Syntax checkers are useful, but we really want to be able to write *translators* that perform some useful action as a side effect of performing a parse. An *interpreter* is a translator that executes actions specified in the parser whilst a parse is occurring. To be suitable for interpretation, the language grammar must be designed to be both parsable and executable in a single linear pass, and **`minicalc`** is an example of a language which has such a grammar. An interpreter which is very similar to **`minicalc`** is described in the tutorial manual [JS97c].

**`rdp`** allows us to embed *semantic actions* into a grammar. **`rdp`**'s semantic actions are written in C, and are copied into the generated parser so that as soon as a fragment of the language to be parsed is recognised the action can be performed. For instance, on recognition of the **`minicalc`** fragment

```
int temp
```

we can make a new symbol table entry for a variable called **`temp`**. If the fragment is followed by an `=` token then we can go on to parse and evaluate an arithmetic expression, placing the result into the symbol table record for **`temp`**. Figures 3.1 and 3.2 show the specification for a full interpreter for `minicalc` that operates in this way. This interpreter IBNF specification uses the same grammar as that for the **`mini`** syntax checker described in the last chapter. The only differences are the addition of semantic actions and synthesised attributes to allow declaration of variables, evaluation of arithmetic operations and the printing of results.

You can read more about semantic actions in Chapter 5 of the user manual [JS97a] and Chapter 6 of the tutorial manual [JS97c]. To understand the **`minicalc`** interpreter you also need to be familiar with the use of **`rdp`**'s built-in symbol table package which you can read about in Chapter 7 of the support library manual [JS97b] and Chapter 7 of the tutorial manual [JS97c].

## 3.1 Declaring symbol tables

Lines 14–19 of Figure 3.1 specify the creation of a symbol table to hold the variables declared in a **`minicalc`** program. A symbol table is a repository for

```
(*******************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* minicalc.bnf - a decorated mini calculator grammar with interpreter semantics
*
* This file may be freely distributed. Please mail improvements to the author.
*
*******************************************************************************)
TITLE("Minicalc interpreter V1.50 (c) Adrian Johnstone 1997")
SUFFIX("m")
USES("math.h")

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

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

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

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

              'print' '(' ( e1:val [* printf("%li", val); *] |
                            String:str [* printf("%s", str); *]
                          )@','
                      ')'.

```

**Figure 3.1** An rdp specification for the minicalc interpreter: part 1

```
e1:integer ::= e2:result {'+' e2:right [* result += right; *] |          (* Add *)
                          '-' e2:right [* result -= right; *] }.         (* Subtract *)

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

e3:integer ::= '+' e3:result |                                           (* Posite *)
               '-' e3:result [* result = -result; *] |                   (* Negate *)
               e4:result.

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

e5:integer ::= ID:name
               [* if (symbol_lookup_key(mini, &name, NULL) == NULL)
                 {
                   text_message(TEXT_ERROR, "Undeclared variable '%s'\n", name);
                   symbol_insert_key(mini, &name, sizeof(char*), sizeof(mini_data));
                 }
              *]
               [* result = mini_cast(symbol_lookup_key(mini, &name, NULL))->i; *] | (* Variable *)
               INTEGER:result |                                          (* Numeric literal *)
               '(' e1:result ')'.                                        (* Do-first *)

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

String: char * ::= STRING_ESC('"' '\\'):result.                          (* Strings for print *)

(* End of minicalc.bnf *)
```

**Figure 3.2** An `rdp` specification for the `minicalc` interpreter: part 2 records which may be stored and retrieved using a *key*. Typically the key is a string corresponding to the name of an identifier, but **rdp** allows keys to be made up of combinations of different fields. The symbol table package itself is quite general — the user must supply a set of routines for comparing, hashing and printing keys which effectively tune the package to use tables with a particular kind of key. The library itself comes with suitable functions for the very common case in which the single string key field is the first field in the symbol table record, and it is those routines that are used here. For these routines to work we must be sure to set up the data part of the symbol table record correctly: in this case a single `char*` field to hold the variable's identifier is the first field and then an `integer` field is declared to hold the working value of a variable.

## 3.2 Using synthesized attributes

A synthesized attribute is a value (which may be of any C type, including primitive types such as characters and integers as well as complex types such as structures and arrays) which is passed back up a parse tree, or equivalently in **rdp** terms 'returned' by a grammar rule or scanner primitive. A simple example may be found in the definition for rule `e5`, part of which is reproduced here:

```
e5:integer ::= ... | INTEGER:result | ... .
```

When rule `e5` matches against an `INTEGER` the scanner can be asked to return the binary number corresponding to the `INTEGER` lexeme just recognised. The specification here indicates that the return value should be loaded into an attribute called `result`. At the end of a rule, the current value of `result` is returned to the caller of the rule, so the effect of this **rdp** IBNF fragment is to parse an integer and return the corresponding binary number to the caller.

## 3.3 Expression evaluation

The expression tree (rules `e1`–`e5` specified in lines 45–70) evaluates expressions by collecting the values of operands in rule `e5` and passing them back up through the tree, performing any calculations specified by operators *en route*.

Each operator has an attached semantic action which evaluates its operands into the `result` return value. The semantic actions just use the equivalent operator in the underlying C language except in the case of the exponentiation operator which does not exist in ANSI-C. Exponentiation is therefore handled by calling the `pow()` standard library function (ensuring that the `integer` operands supplied by the `mini` code are re-cast as `double` precision real numbers). The header file for the maths library must be added to the list of files which are `#include`ed into the parser, and this is specified with the `USES` directive in line 12.

We must be particularly cautious with the divide operator `/` because an attempt to divide by zero would generate an arithmetic trap on some computer architectures (or, even worse, quietly generate undefined results on some others!) The semantic action for the divide operator checks for this condition before attempting to evaluate any divisions and issues a fatal error message if necessary which will abort interpretation. You can read about the routine `text_message()` which is used to issue error messages in Chapter 8 of the support library manual [JS97b].

## 3.4 Accessing the symbol table

When a new variable is declared in a `mini` program using an `int` declaration we must create a new symbol table entry which will hold the value of the variable. When the corresponding variable identifier appears within an expression we must access the symbol table to retrieve the value, and when an identifier appears on the left hand side of an assignment we must access the symbol table to update the variable's value field. The symbol table library provides two routines `symbol_lookup_key()` and `symbol_insert_key()` to search for and insert keys. You should look at Chapter 7 of the support library manual [JS97b] for a complete description of these routines. Lines 30–36 illustrate the use of these functions to look up an identifier in the symbol table:

```
30: statement ::= ID:name
31:               [* if (symbol_lookup_key(mini, &name, NULL) == NULL)
32:                  {
33:                    text_message(TEXT_ERROR, "Undeclared variable '%s'\n", name);
34:                    symbol_insert_key(mini, &name, sizeof(char*), sizeof(mini_data));
35:                  }
36:               *]
```

The `ID` scanner primitive will accept an alphanumeric identifier whose lexeme is returned in attribute `name`. The semantic action calls `symbol_lookup_key()` to search the symbol table called `mini` for the identifier `name` in any scope region. If the symbol table does not contain `name` then `symbol_lookup_key()` will return a `NULL` value, in which case the action issues an error message and then inserts the identifier into the table. This is done so as to suppress subsequent error messages that might be triggered by later references to the variable.

# Chapter 4

# The `minicond` language: interpretation with conditionals

The `minicalc` language discussed in the last chapter is only really as powerful as an integer-only pocket calculator with a large number of memories. Historically, calculators were distinguished from full blown computers on the basis of their control capability: to be worthy of the name a computer must be capable of making decisions. A 'decision' in this context usually means conditionally executing some parts of a program on the basis of calculations performed whilst the program is running (that is, at *run-time*). By this definition, `minicalc` has the abilities of a calculator, not a computer. We shall progressively add capabilities to our `mini` language. We start in this chapter by making these additions:

1. relational operators (`>`, `>=`, `<`, `<=`, `==` and `!=`) with lower priority than any of the arithmetic operators, and

2. an `if-then-else` statement which allows conditional interpretation of programs.

The result is a language `minicond` whose programs look like `minicalc` programs with some additional features — `minicalc` is a strict subset of the `minicond` language so any `minicalc` program will be correctly evaluated by a `minicond` interpreter. Figure 4.1 shows an example `minicond` program. The output produced when this is run through the `minicond` interpreter is shown in Figure 4.2.

In a later chapter we shall see how to add looping constructs and a facility for grouping statements together in blocks.

## 4.1 A grammar for `minicond`

The grammar for `minicond` shown in Figures 4.3 and 4.4 follows the general form of the interpreter presented in the previous chapter, except that large semantic actions have been placed in their own *semantic rules*, and the necessary syntax and semantic actions have been added to support relational expressions and the `if` statement.

```
(******************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* testcond.m - a piece of Minicond source to test the Minicond interpreter
*
* This file may be freely distributed. Please mail improvements to the author.
*
******************************************************************************)
int a=3+4, b=1;
print("a is ", a, "\n");
b=a*2;
print("b is ", b, ", -b is ", -b, "\n");
print(a, " cubed is ", a**3, "\n");
int z = a;
if z==a then print ("z equals a\n") else print("z does not equal a\n");
z=a - 3;
if z==a then print ("z equals a\n") else print("z does not equal a\n");
(* End of testcond.m *)
```

**Figure 4.1** An example `minicond` program (`testcond.m`)

```
a is 7
b is 14, -b is -14
7 cubed is 343
z equals a
z does not equal a
```

**Figure 4.2** `minicond` output for example program

```
(*******************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* minicond.bnf - a decorated mini-conditional grammar with interpreter semantics
*
* This file may be freely distributed. Please mail improvements to the author.
*
*******************************************************************************)
TITLE("Minicond interpreter V1.50 (c) Adrian Johnstone 1997")
SUFFIX("m")
USES("math.h")

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

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

(* semantic rules - implemented as macros in the C code *)
_insert(id)       ::= [* if (interpret)
                           symbol_insert_key(minicond, &id, sizeof(char*),
                                             sizeof(minicond_data));
                      *].
_lookup(id ret)   ::= [* {
                           void * sym = symbol_lookup_key(minicond, &id, NULL);
                           if (sym == NULL) /* not found! */
                           {
                             text_message(TEXT_ERROR_ECHO, "Undeclared variable, '%s'\n", id);
                             sym = symbol_insert_key(minicond, &id,
                                                     sizeof(char*), sizeof(minicond_data));
                           }
                           ret = minicond_cast(sym)->i;
                         }
                       *].
_update(id val)   ::= [* if (interpret)
                         {
                           void * sym = symbol_lookup_key(minicond, &id, NULL);
                           if (sym == NULL) /* not found! */
                           {
                             text_message(TEXT_ERROR_ECHO, "Undeclared variable, '%s'\n", id);
                             sym = symbol_insert_key(minicond, &id,
                                                     sizeof(char*), sizeof(minicond_data));
                           }
                           minicond_cast(sym)->i = val;
                         }
                      *].
_and(dst a b)     ::= [* dst = a && b;  *].
_and_not(dst a b) ::= [* dst = !a && b; *].
_local_int(a)     ::= [* integer a; *].

```

**Figure 4.3** An rdp specification for the minicond interpreter: part 1

```
var_dec(interpret:integer) ::=
  'int' ( ID:name _insert(name)                                  (* Declaration *)
          ['=' e0:val _update(name val) ]                        (* Initialisation *)
        )@','.
statement(interpret:integer) ::=
  ID:name '=' e0:value _update(name value) |                     (* Assignment *)
  _local_int(flag)
  'if' e0:cnd 'then' _and(flag cnd interpret) statement(flag)    (* if statement *)
     [ 'else' _and_not(flag cnd interpret) statement(flag) ] |
  'print' '(' ( e0:value [* if (interpret) printf("%li", value); *] | (* output *)
                String:str [* if (interpret) printf("%s", str); *]
              )@','
          ')'.
e0:integer ::=
  e1:result ['>' e1:right  [* result = result > right; *] |      (* Greater than *)
             '<' e1:right  [* result = result < right; *] |      (* Less than *)
             '>=' e1:right [* result = result >= right; *] |     (* Greater than or equal *)
             '<=' e1:right [* result = result <= right; *] |     (* Less than or equal *)
             '==' e1:right [* result = result == right; *] |     (* Equal *)
             '!=' e1:right [* result = result != right; *] ].    (* Not equal *)
e1:integer ::= e2:result {'+' e2:right [* result += right; *] |  (* Add *)
                          '-' e2:right [* result -= right; *] }. (* Subtract *)
e2:integer ::= e3:result {'*' e3:right [* result *= right; *] |  (* Multiply *)
                          '/' e3:right                           (* Divide *)
                          [* if (result == 0)
                               text_message(TEXT_FATAL_ECHO, "Divide by zero attempted\n");
                             else result /= right;
                          *]
                         }.
e3:integer ::= '+' e3:result |                                   (* Posite *)
               '-' e3:result [* result = -result; *] |           (* Negate *)
               e4:result.
e4:integer ::= e5:result [ '**' e4:right                         (* Exponentiate *)
                           [* result = (integer) pow((double) result, (double) right); *]
                         ].
e5:integer ::= ID:name _lookup(name result) |                    (* Variable access *)
               INTEGER:result |                                  (* Numeric literal *)
               '(' e1:result ')'.                                (* Do-first *)
comment    ::= COMMENT_NEST('(*' '*)').                          (* Comments *)
String: char* ::= STRING_ESC('"' '\\'):result.                   (* Strings for print *)
(* End of minicond.bnf *)
```

**Figure 4.4** An rdp specification for the minicond interpreter: part 2

## 4.2 Adding conditional operators

The rule `e0` defined in lines 72–78 adds a specification for the six new relational operators at a priority level below that of the operators defined in the `minicalc` grammar expression tree. It is not meaningful to write a sequence of relational operators; an expression such as `(3 < 2) < 4` attempts to compare a boolean value, in this case FALSE (the result of $3 < 2$) with the integer 4. This issue is confused in some languages such as C where booleans are not directly supported, integers being used in their place. This expression actually yields TRUE in ANSI-C because the subexpression `(3<2)` yields `0` which is indeed less than `4`. So as to avoid this kind of confusion in `minicond` the rule will only accept individual instances of relational operators: that is an expression such as

`a + b > 3`

is legal but

`a > b > 3` and `(a > b) > 3`

are not. This is achieved by using the *zero-or-one* `[ ]` construct rather than the *zero-or-many* `{ }` bracket in rule `e0` and only allowing bracketed expressions to contain arithmetic operators as specified on line 101.

In other respects the rule is entirely conventional: on recognition of an operator subexpression the parser will execute the associated semantic action and in each case we have simply used the equivalent operator in the underlying ANSI-C language.

## 4.3 Using inherited attributes

In the last chapter we saw how information about a token could be passed from the scanner to the parser's semantic actions using attributes, and we also saw how information calculated within a parser rule could be passed back to a calling rule using a similar mechanism. These kinds of attributes are called *synthesized attributes* because the information is synthesized at the lower level (either within the scanner or a rule) and passed back *up* the chain of production rules. Synthesized attributes correspond roughly to the return values of functions in a programming language, and as we have seen this is precisely how they are implemented in `rdp`.

Sometimes we need to reverse this process and pass information down into production rules. We can think of this as a rule inheriting information from the rule which called it, and so these kinds of attributes are called *inherited attributes*. They correspond roughly to function parameters in conventional programming languages, and that is how they are implemented in `rdp`. You can read more about the use of inherited attributes in Chapter 5 of the user manual [JS97a] and Chapter 6 of the tutorial manual [JS97c].

In the `minicond` interpreter we use inherited attributes in some rules to pass in a flag called `interpret` which controls the execution of semantic actions. The use of this flag will be explained more fully in section 4.5 below. Here we simply note that an inherited attribute is specified on the left hand side of a rule definition by adding parenthesised parameters to the rule name. In line 50 of `minicond.bnf`, for instance, the original `minicalc` variable declaration rule is redefined as

```
var_dec(interpret:integer) ::= ...
```

This specifies that rule `var_dec` has a single inherited attribute called `interpret` of type `integer`. `rdp` grammar rules can have multiple inherited attributes each of which must be specified with an accompanying type. When such a rule is called, the parameter values must be filled in using either literal numbers, literal strings or the names of other attributes. You can see examples in line 21 where the top level rule calls subrules with a literal integer:

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

and in line 64 where the `statement` rule is called and passed the value of an attribute.

## 4.4 Using semantic rules

In complex translators the semantic actions can become very large, and reading a decorated `rdp` grammar can become difficult as the C-language semantic actions obscure the underlying form of the grammar. One solution to this problem is to parcel all but the most trivial actions into separate C functions that reside in the auxiliary file, in which case the semantic actions in the grammar may be reduced to function calls. This certainly allows the grammar to 'show through' but then a full understanding of the translator requires two files (the `rdp` IBNF file and the C language auxiliary file) to be coordinated. Semantic rules are a sort of half way house in which the C language actions may be separated out from the main part of the grammar whilst still residing in the same source file.

A semantic rule is one which contains *only* a single sequence of semantic actions. As such, these rules do not affect the language generated by the grammar (or, equivalently, matched by the parser generated from the grammar). By convention, semantic rule names begin with a leading underscore so that when reading the grammar we can mentally delete them from consideration of the language generated.

Semantic rules are implemented using ANSI-C macros rather than as functions. This is to allow the semantic rule to automatically have access to all of the attributes in the rule that calls the semantic rule but you should be aware that each instance of a semantic rule will result in the complete body of the rule being instantiated into the parent rule, so casual use could lead to very large generated parsers.

Semantic rules can take inherited attributes but may not return synthesized attributes — since the semantic rule automatically has access to the complete state of the calling rule it can access the parents attributes directly. The inherited attributes are treated slightly differently for semantic rules than for normal rules in that the attributes are made into macro parameters and are thus available for textual substitution within the semantic actions. As such they do not take a type and they follow the rules for macro parameters in the ANSI-C macro preprocessor.

The `minicond` interpreter contains six semantic rules.

◇ `_local_int(a) ::= [* integer a; *].` generates a macro that will declare a new local variable of type integer. The name of the variable will be whatever identifier is supplied as the actual parameter in a call to this semantic production.

◇ `_and(dst a b) ::= [* dst = a && b;  *].` the destination attribute is set to the logical AND of attributes `a` and `b`.

◇ `_and_not(dst a b) ::= [* dst = !a && b; *].` the destination attribute is set to the logical AND of the inverse of attribute `a` and attribute `b`.

◇ `_insert(id)` the identifier `id` is added to the symbol table `minicond`.

◇ `_lookup(id ret)` the attribute `ret` is set to the integer value field of the symbol table record for identifier `id`. If `id` is not found in the symbol table then an error message is issued and the symbol is added in, so as to suppress subsequent messages.

◇ `_update(id val)` the integer value field of the symbol table record for identifier `id` is set to `val`. If `id` is not found in the symbol table then an error message is issued and the symbol is added in, so as to suppress subsequent messages.

## 4.5 Adding conditional execution

Our first task when adding an `if` statement to `mini` is to define the necessary syntax to the grammar. This is done in lines 64 and 65 of `minicond.bnf`:

```
63:   ...
64:   'if' e0:cnd 'then' _and(flag cnd interpret) statement(flag)
65:      [ 'else' _and_not(flag cnd interpret) statement(flag) ] |
66:   ...
```

If we strip out the semantic actions, the semantic rules and the attributes from this rule we see that its effect on the language is to define an `if` statement with this syntax:

```
statement ::= 'if' e0 'then' statement [ 'else' statement ].
```

This recursive rule allows nesting of `minicond` statements, which leads to an ambiguity in the grammar. Consider this fragment of `minicond` code:

```
int x = 0, a = 10, b = 20, c = 30;
if a>b then
  if b>c then
    x = 1
  else
    x = 2
```

Here we see two nested `if` statements, with one optional `else` clause present. The ambiguity in the grammar means that we cannot tell, just by looking at lines 1-5 whether the `else` clause at line 5 belongs to the `if` statement at line 3 or the `if` statement at line 2. The ambiguity is reflected in the error message that `rdp` generates when presented with the full grammar:

```
******: Error - LL(1) violation - rule
 rdp_statement_2 ::= [ 'else' _and_not statement ] .
 contains null but first and follow sets both include: 'else'
```

Since an `else` clause may be immediately followed by another `else` clause *and* the `else` clause is optional we have an LL(1) violation because the IBNF phrase `[ 'else' statement]` is (1) optional, and (2) starts with the keyword `else` and may be followed by the keyword `else`. In practice, it is almost universally agreed by programming language designers that an `else` clause should bind to the nearest `if` statement, in this case to the `if` statement at line 3.

Whilst `rdp` is an LL(1) parser generator it is not strictly true that it can only generate parsers for LL(1) grammars. When `rdp` is presented with a non-LL(1) grammar it is effectively being asked to parse strings that may provide matches for more than one alternative at some point in the derivation. By default, `rdp` simply rejects such grammars with an appropriate error message, but if we add a `-F` flag to the `rdp` command line then `rdp` will be forced to output a parser that disambiguates such cases by choosing the alternative that is lexically first in the grammar. As long as the grammar writer is able to achieve the non-LL(1) behaviour required by putting the most important alternative first then the generated parser will operate correctly. In the case of iterators (including the optional bracket `[ ]` here) `rdp` will choose to go into an iterator rather than skipping over it if the currently parsed token is in both the FIRST and FOLLOW sets of the iterator. This rule has the effect of parsing the `else` clause in such a way that the derivation tree shows the `else` clause as bound to the nearest `if`.

There are other techniques for handling the so-called dangling-else problem. Perhaps the simplest is to change the language syntax so that `if` statements are explicitly terminated. The Algol-68 language for instance uses a rule of this form:

```
statement ::= 'if' e0 'then' statement [ 'else' statement ] 'fi'.
```

The closing `'fi'` (which is `if` backwards) marks the end of each `if` statement and removes both the grammatical ambiguity for dangling-else's and the LL(1) breach caused by the presence of the keyword `else` in both the FIRST and FOLLOW sets of the optional `else` clause. The modern trend in programming languages is to insist on this kind of explicit termination because it has been observed that programmers are more likely to accidentally leave out tokens than to add in spurious tokens. Forcing the programmer to mark the end of compound statements is a useful discipline. However, languages designed during the 60's and 70's such as Pascal and C typically allow unterminated control statements.

### 4.5.1 Semantic actions for conditional execution

When we write an `if` statement in a program we think of the computer 'jumping' over one of the two branches. In a compiler we can generate instructions that do indeed cause a section of code to be jumped over as we shall see in Chapter 7, but in an interpreter we cannot simply jump over part of the input stream because the whole source text must be checked for syntactic correctness. Hence the parser will read and process both the **`then`** and the **`else`** branches of an `if` statement, executing the semantic actions in both branches as it goes. In practice, of course, we only want to execute the semantic actions for one branch or the other so we need to be able to dynamically disable semantic action execution at run-time. We do this by supplying an inherited attribute to the productions `statement` and `var_dec` which is a boolean value. If the attribute is true then the embedded semantic actions are executed and if not they are skipped over.

At the top level, rule `program` in line 21 calls the `statement` and `var_dec` rules with the attribute set to `1` (i.e. TRUE) so at the start of a program all semantic actions will be executed. When an `if` statement is encountered, the conditional expression is evaluated and the result is logically AND-ed with the value of the `interpret` attribute. This new value is then supplied as a parameter to a new (nested) instance of the `statement` rule.

The semantic actions in the expression tree are *not* switched on and off in this way because they do not need to be: during the evaluation of an expression attributes are calculated and passed back up the tree of expression rules but no changes are made to the variables declared in a `minicond` program until an assignment is executed. The `_update` semantic rule associated with the assignment statement *is* guarded by a check against the `interpret` attribute, so the result of an expression is simply discarded if interpretation has been switched off.

## 4.6 Next steps

The techniques used in this chapter to handle conditional interpretation can not easily be extended to handle looping because our parsers are designed to make complete passes over the source program, and a loop construct would require us to skip *backwards* in the parse. This is certainly not impossible to implement but would require detailed knowledge of `rdp`'s internals and is not the recommended approach. Instead we shall move to a full compiler for `mini` which outputs instructions for a very simple computer called the Mini Virtual Machine (MVM). By providing an assembler and simulator for MVM we can build a complete system that models the tasks of a compiler for a real processor.

# Chapter 5

# The Mini Virtual Machine (MVM)

MVM is a paper architecture designed to support efficient interpretation on a host architecture. We will use MVM to illustrate the techniques of virtual machine simulation, assembly language translation and compilation. In this chapter we shall describe the MVM architecture and a simulator for that architecture written in ANSI-C. As well as providing a means to execute MVM programs, the simulator source code can be treated as an exact specification of the architecture. We will begin by describing the architecture informally.

The Mini Virtual Machine is a very simple architecture based around a conventional memory to memory processor. This means that all MVM operations execute directly on the contents of memory locations: there are no registers or stacks available for storing data, although there are two internal registers used to hold the address and contents of the currently executing instruction.

MVM is a *16-bit* processor in which arithmetic operations take place on 16-bit quantities and in which memory addresses also fit into 16-bit words. This limitation to 16-bit memory addresses does constrain the size of the programs that we can write, but is sufficient for demonstrating the ideas behind the development of a compiler. It also means that the tools can be compiled and run on older 16-bit computers such as ordinary MS-DOS machines. If you have a 32-bit system and a suitable C compiler, it is quite easy to extend the MVM specification and its simulator to support 32-bit operations and addresses.

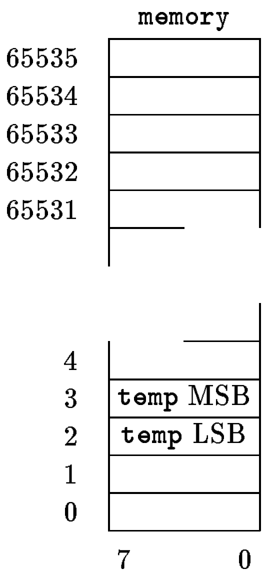
Understanding a new processor is made easier if we list the capabilities of the architecture under three main headings:

1. the memory resources provided by the architecture,

2. the various ways in which operands may be fetched during instruction execution (the *addressing modes*) and

3. the collection of operations that may be programmed on that architecture (the *instruction set*).

In each of these three areas MVM provides very limited facilities. This makes MVM easy to understand, easy to program and easy to write software simulators for, but it does not make MVM a good target for efficient hardware implementation. That need not concern us: MVM is only really intended to




Total of 65536 cells addressed as `memory[0]` to `memory[65535]`.

Each cell contains eight bits numbered 0 – 7

The 16-bit variable `temp` is stored as two bytes at successive addresses.

The address of `temp` is 2: the address of the Least Significant Byte (LSB)

**Figure 5.1** MVM memory structure

be used *via* a software simulator, so the many clever devices that hardware designers have introduced into real architectures to aid hardware realisation are irrelevant to our purpose. You should bear in mind, however, that writing a compiler for a real processor is more complex than writing a compiler for MVM: the principles remain the same, but the large scale design of a real compiler requires much more detail to be handled.

## 5.1 MVM memory

All MVM data and instructions are stored in a single main memory. MVM has no special or general purpose registers for data. The MVM memory can be regarded as an array of eight-bit (byte) locations, individually addressed. The size of the MVM memory is fixed at 64K ($= 65536_{10}$) bytes. This allows all MVM addresses to be specified using 16-bit numbers. A diagrammatic representation of the MVM memory is shown in Figure 5.1.

Since the memory cells can only hold eight bit numbers, and since MVM usually operates on 16-bit quantities, in general two adjacent cells are used to hold each data item. When accessing 16-bit numbers, the address of the least significant byte is specified, (say $n$) and the operand is understood to be made up of the contents of the addressed cell concatenated with the contents of the next highest address ($n$+1). In the example shown in the figure, the 16-bit variable `temp` resides at locations 2 and 3. If `memory[2]` contains $9_{10}$ ($1001_2$) and `memory[3]` contains $3_{10}$ ($0011_2$) then the value of `temp` is 0003 concatenated with 0009, which is $3_{10} \times 256_{10} + 9_{10} = 777_{10}$ ($1100001001_2$).

MVM instructions range in size from two to eight bytes. By analogy with the addressing for data items, the address of the instruction is taken to be the address of the least significant byte which will be the lowest address of the range of locations occupied by the instruction.

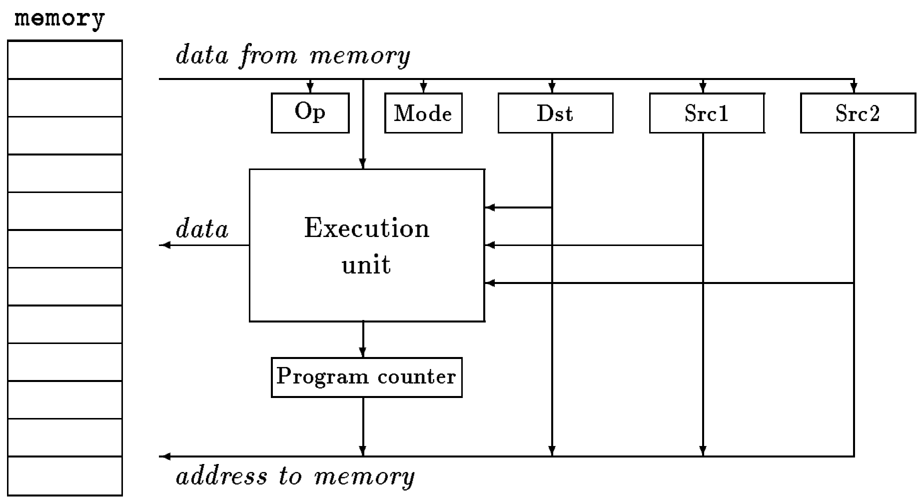



**Figure 5.2** MVM internal structure

## 5.2 MVM instruction execution

A block diagram of the internal structure of an MVM processor is shown in Figure 5.2. MVM is an example of a *Von Neumann* processor, as indeed are most real computers in use today. In a Von Neumann machine, instructions and data can co-exist in the same memory as described in the previous section. It is not possible, just by looking at the contents of memory to distinguish between instructions and data.

The processor maintains a pointer to memory (that is, a register which holds the address of a location in memory) called the *program counter*. Before a program can be executed, it must be loaded into memory and then the program counter initialised with the address of the first instruction to be executed.

Once the machine starts running the program, it reads the instruction pointed to by the program counter into some internal registers collectively called the *Instruction Register*. In this case, the instruction register has space for an operation code, an address mode and up to three operands called the destination, source 1 and source 2 operands. These registers between them can hold all of the information needed to execute a single MVM instruction: the significance of the individual registers will be described in the following sections. For now, note that the data from memory may be loaded into the instruction register or sent to the execution unit; that memory addresses may be supplied by the program counter or by the operand registers; that only the execution unit can generate data to be written back into memory and that the contents of the operand registers may connected directly to the execution unit.

After the instruction register has been loaded with a new instruction ready for execution, the program counter is incremented so that it points to the location just past the end of the instruction that has just been read. The processor then performs whatever action is specified by the instruction in the instruction register, which might for instance be the addition of two numbers or the copying of a data item from one memory location to another. These kinds of instructions are called *data manipulation* instructions because they allow data to be modified in various ways.

In addition to the data manipulation instructions, Von Neumann processors like MVM have *control manipulation* instructions which affect the order in which the program's instructions are executed. By default, the program counter is simply incremented to point to the next instruction after the current one. In the absence of any control manipulation instructions, therefore, all programs would simply be executed once only in strict order by address. This is the kind of program that we can write using the simple `minicalc` language which has no control flow constructs.

The control manipulation instructions allow sequences of instructions to be jumped over. They work by loading a new value directly into the program counter which overrides the simple sequential execution. Often the new value is only loaded if some condition is true. An **`if`** *condition* **`then`** *action* statement can be implemented as a test of *condition*. If *condition* is false, then the program counter is loaded with the address of the instruction after the code corresponding to *action*, and this has the effect of skipping over *action* without executing it.

## 5.3 MVM addressing modes

Most high level programming languages provide both variables and numeric literals. In `minicalc`, for instance, the assignments

```
temp = x + y
```

and

```
temp = x + 12
```

are both valid. In practice, they will be compiled into an `ADD` instruction with three operands: a destination and two sources corresponding to the left and right sides of the `+` operator. The variables `x` and `y` will be stored at specific memory addresses. What if variable `y` were to be stored at location 12? How would an MVM processor distinguish between an instruction to add the number 12 and an instruction to add the contents of a variable stored at location 12? The answer is to provide some extra information called the *addressing mode*. It is the responsibility of the compiler to specify the correct addressing mode when it generates an MVM instruction. We shall look at how the modes are specified in the next section.

MVM provides only two addressing modes: literal and variable. Real processor architectures often provide many complex addressing modes which, for instance, might allow an access within a two dimensional array to be specified as a single machine instruction. The trend in recent years has been to discourage the use of any but the most straightforward addressing operations because they complicate the use of *pipelining* in hardware implementations. Pipelined processors are very efficient, but their execution units are disrupted by the overhead of having to decode complicated addressing modes. Broadly speaking, Complex Instruction Set Computer (CISC) architectures such as the DEC VAX, the Motorola 68000 and the Intel 80x86 family have many exotic addressing modes, and Reduced Instruction Set Computer (RISC) architectures such as the MIPS, Sun SPARC and Dec Alpha have essentially only three modes. MVM is simple, but is not comparable to a real RISC architecture because it does not have any data registers, and the efficient use of such registers is perhaps the defining characteristic of a true RISC architecture.

## 5.4 MVM instructions

MVM instructions are made up of a string of bytes. Depending on the instruction, the string may be between two and eight bytes long: every MVM instruction has an operation code (opcode) byte followed by an address mode byte, and most instructions also contain some operands. Each operand is represented by a 16-bit number, so each operand adds two bytes to the length of an instruction one for the most significant byte (MSB) and one for the least significant byte (LSB). The format of a three address instruction, therefore is

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *opcode* | *mode* | *dst MSB* | *dst LSB* | *src1 MSB* | *src1 LSB* | *src2 MSB* | *src2 LSB* |

Instructions with less than three operands follow this general format but simply omit the unused operand fields. Even zero address instructions such as HALT have a mode byte.

### 5.4.1 Instruction set capabilities

The opcode byte could encode up to 256 unique instructions, but in fact MVM only uses the first 17 codes, numbered 0–16. As we shall see in the next chapter, it in more convenient to use hexadecimal (base 16) than decimal (base 10) to represent machine level quantities, so Table 5.1 which shows the complete MVM instruction set gives the hexadecimal encodings for the instructions.

The functional description of each instruction in Table 5.1 uses a C-like syntax to explain the actions of the MVM processor on receipt of each instruction. Main memory is modeled as an array of locations called `mem[]` and the program counter as a variable called `PC`. The function `resolve()` looks at the addressing mode of its corresponding operand and fetches the actual data. Later in this chapter we give extracts from the source code of a simulator for MVM instructions which shows exactly how these functional descriptions may be turned into executable code.

### 5.4.2 Address mode encoding

The mode byte is split into two four-bit nibbles called mode fields [1]. Each of the two mode fields encode the address mode for one of the two source operands:

[1] Since we only have two addressing modes to encode, we could make do with only a single bit for each field, but we wish to leave some capacity so that, for instance, a register based variant of MVM could be easily defined.

| Opcode | Mnemonic | Operands | | | Function |
|---|---|---|---|---|---|
| 00 | HALT | – | – | – | Stop the processor |
| 01 | ADD | *dst* | *src1* | *src2* | `mem[dst] = resolve(src1) + resolve(src2)` |
| 02 | SUB | *dst* | *src1* | *src2* | `mem[dst] = resolve(src1) - resolve(src2)` |
| 03 | MUL | *dst* | *src1* | *src2* | `mem[dst] = resolve(src1) * resolve(src2)` |
| 04 | DIV | *dst* | *src1* | *src2* | `mem[dst] = resolve(src1) / resolve(src2)` |
| 05 | EXP | *dst* | *src1* | *src2* | `mem[dst] = resolve(src1) ** resolve(src2)` |
| 06 | EQ | *dst* | *src1* | *src2* | `mem[dst] = resolve(src1) == resolve(src2)` |
| 07 | NE | *dst* | *src1* | *src2* | `mem[dst] = resolve(src1) != resolve(src2)` |
| 08 | GT | *dst* | *src1* | *src2* | `mem[dst] = resolve(src1) > resolve(src2)` |
| 09 | GE | *dst* | *src1* | *src2* | `mem[dst] = resolve(src1) >= resolve(src2)` |
| 0A | LT | *dst* | *src1* | *src2* | `mem[dst] = resolve(src1) < resolve(src2)` |
| 0B | LE | *dst* | *src1* | *src2* | `mem[dst] = resolve(src1) <= resolve(src2)` |
| 0C | CPY | *dst* | *src1* | – | `mem[dst] = resolve(src1)` |
| 0D | BNE | *target* | *src1* | – | `if resolve(src1) != 0 then PC = target` |
| 0E | BEQ | *target* | *src1* | – | `if resolve(src1) == 0 then PC = target` |
| 0F | PRTS | 0 | *src1* | – | Print `resolve(src1)` as string |
| 10 | PRTI | 0 | *src1* | – | Print `resolve(src1)` as decimal integer |

**Table 5.1** The MVM instruction set

the most significant nibble encodes for *src1* and the least significant nibble for *src2*. For instructions that do not use one or both of the source operands, the corresponding mode fields are set to zero. No mode field is required for the destination operand because the destination must clearly always be an address: it is never meaningful to assign a result to literal! On some real architectures, multiple destination addressing modes are provided but MVM has no need of them.

## 5.5 Example MVM instructions

MVM programs are made up of sequences of MVM instructions which will usually include both data manipulation and control manipulation instructions. Each valid program must finish with a `HALT` instruction which the simulator interprets as an instruction to finish interpreting instructions and return control to the user. It is not an accident that the `HALT` instruction uses opcode number 0. Within the simulator, the memory is initialised throughout to zero. If a user programming error causes the simulator to try executing from memory that has not been loaded with instructions, the simulator will immediately terminate because those zeros will be interpreted as `HALT` instructions.

### 5.5.1 Data manipulation instructions and address modes

In section 5.3 we distinguished between literal and variable addressing. Here we look at the MVM instructions that correspond to the `minicalc` code fragments `temp = x + y` and `temp = x + 12`.

If the variable `x` is resident at location $10_{10}$ (location 000A in hexadecimal) and `y` resides at location $12_{10}$ (location 000C in hexadecimal) then the instruction to add them together and store them in a variable called `temp` at location 4 is

| *Op* | *Mode* | *Dst* | *Src1* | *Src2* |
|---|---|---|---|---|
| 01 | 11 | 0004 | 000A | 000C |

Here we have the operation code for `ADD` ($01_{16}$) followed by a mode byte that specifies variable mode addressing for both source operands ($11_{16}$). Then we have three operands specified as the addresses of the destination ($0004_{16}$) and the two sources ($000A_{16}$ and $000C_{16}$).

By contrast, if we wish to add the number 12 to the contents of `x` and put the result in `temp` then the correct instruction is

| *Op* | *Mode* | *Dst* | *Src1* | *Src2* |
|---|---|---|---|---|
| 01 | 10 | 0004 | 000A | 000C |

The only difference between these instructions is that the mode field for operand `src2` is 1 in the first example and 0 in the second corresponding to variable mode addressing and literal mode addressing respectively.

### 5.5.2 Control manipulation instructions

Control manipulation instructions are used in the implementation of `if` statements, loop statements and `goto` statements. Consider the `minicond` fragment

```
if temp then a = a + 1;
z = z - 2;
```

If `temp` resides at location $000A_{16}$, `a` at location $000E_{16}$ and `z` at location $001C_{16}$, then the following sequence of instructions based at location $2134_{16}$ corresponds to the `minicond` fragment.

| *Location* | *Op* | *Mode* | *Dst* | *Src1* | *Src2* |
|---|---|---|---|---|---|
| 2134: | 0E | 10 | 2142 | 000A | |
| 213A: | 01 | 10 | 001C | 001C | 0001 |
| 2142: | 02 | 10 | 000E | 000E | 0001 |

The instruction at location $2134_{16}$ is a `BEQ` which will restart execution at address $2142_{16}$ if the variable at location $000A_{16}$ is zero. The next line adds one to the variable at location $001C_{16}$, and the final line subtracts one from the variable at location $000E_{16}$. The overall effect of the fragment is to skip over the middle instruction if the value of `temp` at location $001C_{16}$ is zero.

## 5.6 Using an assembler to program MVM

Writing MVM programs in this numerical code is time consuming and highly error prone. An *assembler* is a translator for a very simple language that offers English-language like mnemonic names for the machine instructions and can also perform branch calculations automatically, so that instructions can be referred to by a symbolic label rather than by their numeric address. Assemblers do not make machine level programming easy, but they do free the programmer from a great deal of bookkeeping work. Assemblers are in themselves examples of an interesting class of translator. We shall describe the implementation of an assembler for MVM (called `mvmasm`) in the next chapter. We complete this chapter with the description of a simulator, `mvsim`, for MVM code.

## 5.7 `mvmsim` – a simulator for MVM byte codes

The MVM instruction set is designed to be efficiently implemented as a simulator. In this section we look at the design and use of such a simulator.

### 5.7.1 Using `mvmsim`

The function of the simulator is two-fold: firstly it provides a concrete model of the behaviour of an MVM processor, and secondly it allows instruction execution to be *traced* by printing out each instruction as it is executed. The `mvmsim` executable is built as part of the standard **`rdp`** installation, so if you have already run **`make`** on the supplied **`makefile`** you should have a working simulator. To check, type `mvmsim` at the command line prompt. You should see the following output:

```
Fatal: No source file specified

mvmsim v1.5 - simulator for MVM

Usage: mvmsim [options] source

-l      Show load sequence
-t      Print execution trace
-v      Set verbose mode

You can contact the author (Adrian Johnstone) at:

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

This is the standard `mvmsim` help message: in this case it has been triggered because no source file was specified. It tells you about the three optional flags that can be supplied to `mvmsim`:

- `-l` tells `mvmsim` to echo the data it is writing into the simulator's memory during the load phase
- `-t` switches on *trace* mode in which instructions are echoed as they are executed
- `-v` sets verbose mode which causes `mvmsim` to print out a title line, and then at the end of a run, the total CPU time along with the number of MVM instructions executed

### 5.7.2 The mvmsim input file format

The MVM program to be simulated is read by mvmsim from an input file which contains binary information rendered as a hexadecimal dump of the required memory contents. Each line of the dump file starts with a 16-bit hexadecimal number which specifies the base address to which the rest of the data on that line will be loaded. The data is specified as zero or more pairs of hexadecimal digits. Each pair specifies the contents of one eight-bit memory cell, and the corresponding memory locations are loaded in ascending order starting with the base address.

Spaces are allowed (but not required) between each pair of digits. Blank lines are also allowed.

This kind of load format is commonly used by assemblers for real processors, although executable file formats used by commercially available processors are usually in pure binary to save space. In a pure binary file each location could be represented by a single binary byte but in our case each location requires two bytes, each representing a hexadecimal character. Therefore, MVM executable files are likely to be at least twice as large as their pure binary equivalents. On the other hand, pure binary files can not easily be read into an editor or printed out.

As well as specifying memory contents, the input file must tell the simulator which memory location contains the first instruction to be executed. The mvmsim input file uses a special format to specify this *transfer address* comprising an asterisk followed by the transfer address itself.

The short example below shows the contents of an mvmsim input file for a program comprising three instructions.

```
1000 0C 01 000A 007B
1006 10 11 0000 000A
100C 00 11
100E *1000
```

The first instruction, based at location $1000_{16}$ copies the number $123_{10}$ ($7B_{16}$) to the memory location $000A_{16}$. The next instruction prints out the contents of that location as a decimal number and the third instruction is a HALT which will cause the simulator to terminate. The final line specifies a transfer address of $1000_{16}$: the transfer address is denoted with a leading asterisk (*) which warns the assembler not to attempt to load the data on that line into memory. The file may be found in the standard **rdp** distribution as **examples/rdp_case/mvmsim.sim**.

### 5.7.3 Running a simulation

We can run the simulator on the above test file with all options enabled by issuing the command

```
mvmsim -l -t -v examples/rdp_case/mvmsim.sim
```

The output of this command is shown below:

```
mvmsim v1.5 - simulator for mvm

Load address 1000 0C01000A007B
Load address 1006 10110000000A
Load address 100C 0011
Load address 100E *1000

1000 CPY  000A, 007B, 0000 -> 007B
1006 PRTI 0000, 007B, 0000 -> 0000  123
100C HALT 0000, 0000, 0000 -> 0000   -- Halted --


0.006 CPU seconds used, 3 MVM instructions executed
```

After the title line, `mvmsim` echoes to the output the contents of the input file as it is loaded into the internal memory. Execution then begins, starting at the transfer address. As each instruction is executed, `mvmsim` outputs the address of the instruction, its mnemonic and then the three operands in the order *dst*, *src1* and *src2*. If an operand is not used by an opcode, then `0000` is output. The operands are printed *after* the addressing mode has been resolved, that is the actual data to be operated on is displayed rather than its address. Hence, when the instruction at location $1006_{16}$ is being executed, its second operand is shown as $7B_{16}$ not as $000A_{16}$ which is the address specified in the load file.

The value written back to memory by the instruction is shown after a 'yields' sign (`->`). If no value is written back (as for instance in the case of the PRTI instruction) then a zero is displayed.

Any output produced by PRTI or PRTS instructions is displayed after the instruction. If the `-t` option is not used on the command line, then the instruction display is completely suppressed, so only program output appears.

Finally, when the simulator encounters a HALT instruction, it prints the message `-- Halted --` and terminates.

## 5.7.4 Implementing `mvmsim`

The full source code of the simulator runs to a little over 300 lines of ANSI-C which may be found in the file `mvmsim.c`. In this section we shall look at the overall structure of the simulator and look in detail at the code corresponding to the MVM instruction execution unit.

About half of the code in `mvmsim.c` is concerned with processing the command line options, parsing the input file and then loading of the internal memory. These functions are easy to understand, and we shall not discuss them further here. The parts of the code we are interested in are those that model the MVM architecture's memory and program counter, and the function that controls the simulated execution of the MVM instructions.

```
18: #define MEM_SIZE 65536lu

22: unsigned char memory[MEM_SIZE];
23: unsigned long pc = 0;

39: static int get_memory_byte(unsigned long address)

52: static int get_memory_word(unsigned long address)

65: static void put_memory_byte(unsigned long address, int data)

71: static void put_memory_word(unsigned long address, int data)
```

**Figure 5.3** Extracts from the `mvmsim` simulator: memory declarations

## Memory and program counter declarations

Figure 5.3 shows the declarations that model the MVM memory. Line 18 specifies the size of the simulated memory, which is restricted to `65536lu` (65536 as a long unsigned number or 64K) bytes in this 16-bit MVM simulator. It is possible to reduce the size of the MVM memory by adjusting this figure, but of course MVM programs must then ensure that they only work within the available memory. The MVM internal memory itself is modeled by an array of `unsigned char` (line 22) and the program counter by an `unsigned long` integer (line 23).

We could access the MVM memory by simply reading and writing to the memory array, but one of the characteristics of machine level programming is that programs often contain errors. A bad error might cause the simulator to run amok and start executing from illegal host addresses. So as to control this kind of problem, all memory access is channeled through the routines `get_memory_byte()`, `get_memory_word()`, `put_memory_byte()` and `put_memory_word()` which are declared in lines 39–76. These routines validate the memory address, issuing a fatal error message if the program being simulated tried to access a non-existent location.

The `get_memory_byte()` and `get_memory_word()` routines take an address and return either a single byte or a single word which is formed by concatenating the addressed byte with the contents of the location `address + 1`. In this case, the addressed byte forms the least significant byte of the returned word. The `put_memory_word()` function takes an address and a 16-bit data word. The least significant byte of the data word is written into memory at the specified address, and the most significant byte is loaded to location `address + 1`

## The main execution loop

After the simulator has loaded the `memory` array and set the program counter to the value of the transfer address, the function `mvmsim_execute()` is called. This function loops until a HALT instruction is encountered, executing one instruction *per* iteration. The full source of the `mvm_execute()` function is shown in Figures 5.4–5.6.

```
160: static void mvmsim_execute(void)
161: {
162:   int stop = 0;
163:
164:   while (!stop)
165:   {
166:     unsigned op = get_memory_byte(pc),
167:              mode = get_memory_byte(pc + 1);
168:     int dst = get_memory_word(pc + 2),
169:         src1 = get_memory_word(pc + 4),
170:         src2 = get_memory_word(pc + 6);
171:
172:     exec_count++;
173:
174:     /* do indirections on modes */
175:     if ((mode >> 4) == 1)
176:       src1 = get_memory_word(src1);
177:
178:     if ((mode & 7) == 1)
179:       src2 = get_memory_word(src2);
180:
```

**Figure 5.4** Extracts from the `mvmsim` simulator: the execute function part 1

Variable `stop` declared at line 162 is used as a flag to signal termination of the simulation. It is initialised to false, and only set to true when a HALT instruction is encountered. The main simulation loop comprises lines 164 to 279. The body of the loop comprises the code to fetch the new instruction (lines 166–170), the address mode resolution code in lines 174–179 and a large switch statement which decodes the operation code (lines 181–278 shown in Figures 5.5 and 5.6.). Within the switch statement, each of the 17 cases includes a call to the `display()` function which provides the trace output if a `-t` option has been specified on the simulator command line. Each case finishes with an increment of the program counter by the length in bytes of the decoded instruction.

The address mode resolution code examines the mode byte loaded from the instruction. If the corresponding mode field (see section 5.3) is a one, then the source operand is reloaded with the contents of the memory location addressed by the data. There is no need to resolve the destination operand, because destinations are always assumed to be addresses not literal data.

The code within the simulation loop provides a detailed specification of the meaning of each instruction in terms of the semantics of ANSI-C. The BNE instruction for instance (lines 243–249) tests the value of the first source operand against zero, and if the test succeeds the program counter is loaded with the address specified in the destination operand. If the test fails, the program counter is simply incremented in the normal way, thus passing control to the next instruction.

```
switch (op)
{
case OP_ADD:
  put_memory_word(dst, src1 + src2);
  display("ADD ", dst, src1, src2);
  pc += 8;
  break;
case OP_SUB:
  put_memory_word(dst, src1 - src2);
  display("SUB ", dst, src1, src2);
  pc += 8;
  break;
case OP_MUL:
  put_memory_word(dst, src1 * src2);
  display("MUL ", dst, src1, src2);
  pc += 8;
  break;
case OP_DIV:
  put_memory_word(dst, src1 / src2);
  display("DIV ", dst, src1, src2);
  pc += 8;
  break;
case OP_EXP:
  put_memory_word(dst, (int) pow((double) src1, (double) src2));
  display("EXP ", dst, src1, src2);
  pc += 8;
  break;
case OP_EQ:
  put_memory_word(dst, src1 == src2);
  display("EQ  ", dst, src1, src2);
  pc += 8;
  break;
case OP_NE:
  put_memory_word(dst, src1 != src2);
  display("NE  ", dst, src1, src2);
  pc += 8;
  break;
case OP_GT:
  put_memory_word(dst, src1 > src2);
  display("GT  ", dst, src1, src2);
  pc += 8;
  break;
```

**Figure 5.5** Extracts from the mvmsim simulator: the execute function part 2

```
    case OP_GE:
      put_memory_word(dst, src1 >= src2);
      display("GE  ", dst, src1, src2);
      pc += 8;
      break;
    case OP_LT:
      put_memory_word(dst, src1 < src2);
      display("LT  ", dst, src1, src2);
      pc += 8;
      break;
    case OP_LE:
      put_memory_word(dst, src1 <= src2);
      display("LE  ", dst, src1, src2);
      pc += 8;
      break;
    case OP_CPY:
      put_memory_word(dst, src1);
      display("CPY ", dst, src1, src2);
      pc += 6;
      break;
    case OP_BNE:
      display("BNE ", dst, src1, src2);
      if (src1 != 0)
        pc = dst;
      else
        pc += 6;
      break;
    case OP_BEQ:
      display("BEQ ", dst, src1, src2);
      if (src1 == 0)
        pc = dst;
      else
        pc += 6;
      break;
    case OP_PRTS:
      display("PRTS", dst, src1, src2);
      printf("%s", memory + src1);
      pc += 6;
      break;
    case OP_PRTI:
      display("PRTI", dst, src1, src2);
      printf("%i", (int) src1);
      pc += 6;
      break;
    case OP_HALT:
      display("HALT", dst, src1, src2);
      printf(" -- Halted --\n");
      stop = 1;
      pc += 2;
      break;
    default:
      display("----", dst, src1, src2);
      text_printf("\n");
      text_message(TEXT_FATAL, "illegal instruction encountered\n");
      break;
    }
  }
}
```

**Figure 5.6** Extracts from the `mvmsim` simulator: the execute function part 3

# Chapter 6

# mvmasm – an assembler for MVM

Writing MVM programs directly in the binary machine code is very error prone. In the early days of computing it was not unusual for programmers to take great pride in their ability to remember all the binary codes for instructions, but even if the machine code is easy to remember (as indeed it is for the very simple MVM processor) it is still hard to keep track of lots of variables if they can only be referred to by their numeric machine addresses.

Assemblers evolved as the earliest available programming aids. Most assemblers provide two basic facilities:

◇ a set of mnemonic names for the machine instructions

◇ the ability to *label* instructions and data locations allowing jump targets and variable addresses to be referred to using symbolic names rather than numeric values.

In addition, assemblers usually allow *arithmetic* to be performed on symbolic addresses. This allows the address calculations associated with array indexing, record field selection and jump branch selection to be done by the assembler, rather than by the programmer.

## 6.1 A first example

The `mvmasm` source code corresponding to the short example used in section 5.7.3 is shown below: a variable is loaded with decimal 123 and then printed.

```
;Simulator example file
 DATA 0x000A     ;start assembling data at address 000A hex
temp: WORD 1     ;declare an integer variable called temp

 CODE 0x1000     ;switch to assembling code at address 1000 hex
start:
 CPY temp, #123  ;load temp with decimal 123
 PRTI temp       ;print the value of temp as an integer
 HALT            ;terminate the simulator

 END start       ;transfer address is code start
```

Each line of mvmasm source code may contain a label such as **temp:** or **start:**, an instruction such as **PRTI temp** and a comment which comprises anything between a semicolon ; and the end of a line. All three of these fields are individually optional, so lines containing only a label, only an instruction or only a comment are valid as indeed are blank lines.

Most instructions in an assembler program correspond to machine opcodes, but some are *directives* which are instructions to the assembler. In the example above, the instructions DATA, CODE, WORD and END are directives.

### 6.1.1 Assembler output

The effect of assembling opcodes and executing directives is best seen by examining the assembler's output. The example source code is available within the **rdp** distribution as file **examples/rdp_case/mvmsim.mvm**. Executing the command

```
mvmasm -l examples/rdp_case/mvmsim
```

produces the following output listing:

```
******:
0000                      1: ;Simulator example file
0000                      2:  DATA 0x000A     ;start assembling data at address 000A hex
000A 0001                 3: temp: WORD 1     ;declare an integer variable called temp
000C                      4:
000C                      5:  CODE 0x1000     ;switch to assembling code at address 1000 hex
1000                      6: start:
1000 0C01000A007B         7:  CPY temp, #123  ;load temp with decimal 123
1006 10110000000A         8:  PRTI temp       ;print the value of temp as an integer
100C 0011                 9:  HALT            ;terminate the simulator
100E                     10:
100E *1000               11:  END start       ;transfer address is code start
******: Transfer address 00001000
******: 0 errors and 0 warnings
```

#### Listing format

This listing shows the familiar line numbered source file listing on the right, with the assembler generated output on the left. The first field of the output is the current *assembly address*, that is the MVM memory address to which any data or instructions following on the line will be loaded. A single space is followed by a string of pairs of hexadecimal digits representing the assembled output. You will see that the output is in the same format as the input for the mvmsim.

#### Assembly using the DATA and CODE pointers

Two internal counters are maintained by mvmasm called the *current data address* and the *current code address*. Their values are set by the DATA and CODE directives respectively and they keep track of the next available data and code memory locations. Since MVM is a Von Neumann processor, data and code may be loaded at any memory locations, but it is conventional to separate them into blocks. Line 2 (`DATA 0x000A`) sets the current data address to hexadecimal 000A and makes the data pointer the current assembly address. All subsequent instructions will be assembled into succeeding locations until such time as another `DATA` or `CODE` directive is encountered. If a `DATA` or `CODE` directive appears on its own without an operand then it simply switches the current assembly address to the current data address or current code address respectively.

### Labels

Labels have the same syntactic form as C language identifiers, that is an alphabetic character or an underscore followed by zero or more alpha-numeric characters or underscores. A label definition must be followed by a colon (`:`). When a label is encountered, it is given the value of the current assembly address. Whether a label gets the current data address or the current code address depends upon which of the `DATA` and `CODE` directives was most recently encountered. Hence, in the above example `temp` is given the value $000A_{16}$ (the current data address) and `start` the value $1000_{16}$ (the current code address).

### Machine instructions and addressing modes

Lines 7, 8 and 9 show actual machine instructions being assembled. Each line comprises one of the operation codes from Table 5.1 followed by between zero and three operands. An operand may be either an address or literal data, which is distinguished by a preceding hash `#` sign. Hence the instruction `CPY temp, #123` assembles to `0B 01 000A 007B` where 000A is the value of the label `temp` and 007B is the hexadecimal form of the literal decimal constant `#123`. As in ANSI-C, hexadecimal numbers are marked by the prefix `0x`. Numeric values lacking this prefix are assumed to be decimal.

### Data declaration directives

Data may be declared using the `WORD` directive, which specifies that enough space be reserved for a machine word (two bytes) and in this case also provides an initialisation expression so that temp is initialised to 1. There are other data declaration directives which may be used to reserve larger blocks of storage. These other directives are described below in section 6.2.5.

### The `END` directive

Line 11 shows an `END` directive which both marks the end of the assembler input file and specifies the transfer address, that is the address of the first instruction to be executed by the simulator. In this case, the value of the `start` label, which is $1000_{16}$

## 6.1.2 Using the assembler and the simulator together

The assembler is usually used to prepare input for the `mvmsim` simulator, and if the assembler is invoked with a `-x` option then the simulator will be automatically run in trace mode on the assembler output. Hence issuing this command

```
mvmasm -l -x examples/rdp_case/mvmsim.mvm
```

produces this output:

```
******:
0000                       1: ;Simulator example file
0000                       2:  DATA 0x000A     ;start assembling data at address 000A hex
000A 0001                  3: temp: WORD 1     ;declare an integer variable called temp
000C                       4:
000C                       5:  CODE 0x1000     ;switch to assembling code at address 1000 hex
1000                       6: start:
1000 0C01000A007B          7:  CPY temp, #123  ;load temp with decimal 123
1006 10110000000A          8:  PRTI temp       ;print the value of temp as an integer
100C 0011                  9:  HALT            ;terminate the simulator
100E                      10:
100E *1000                11:  END start       ;transfer address is code start
******: Transfer address 00001000
******: Calling simulator: mvmsim -t -v mvmasm.out

mvmsim v1.5 - simulator for mvm


1000 CPY  000A, 007B, 0000 -> 007B
1006 PRTI 0000, 007B, 0000 -> 0000  123
100C HALT 0000, 0000, 0000 -> 0000   -- Halted --


0.030 CPU seconds used, 3 MVM instructions executed
******: 0 errors and 0 warnings
```

## 6.2 Assembler syntax reference

In this section we describe the features of the mvmasm assembler in terms of the lexical structure, the available arithmetic operators, the directives and the machine instructions. The implementation of mvmasm as an rdp translator specification is described in the next section. Large examples of mvmasm code which exercise most of the features may be found in the following chapters, which describe compilers that translate to mvmasm.

### 6.2.1 Line oriented and free format languages

The mvmasm syntax follows the tradition of assemblers in being *line oriented* with only one statement allowed *per* line. Early high level programming languages were line oriented in this way, but most programming languages designed since the early 1960's have been free format, allowing whitespace and line breaks to appear between any two language tokens. Low level languages such as assemblers have tended to retain the older style, not least because it can be simpler to hand write a parser for a line oriented language. In particular, error recovery is eased: if a syntax error is detected on a line then after reporting the error the parser can simply restart at the start of the next line. The re-synchronisation of the parser after an error in a free format language can be much harder, and as a result errors in free format languages can generate an avalanche of spurious error messages.

### 6.2.2 Lexical elements

Identifiers in `mvmasm` follow the rules for identifiers in ANSI-C, that is an identifier may begin with an alphabetic character or an underscore and continue with zero or more alphabetic characters, digits or underscores. The length of an identifier is limited only by the available memory within the running assembler and is for practical purposes unbounded.

Numbers start with a digit. If the first digit is a zero (`0`) and this is immediately followed by a lower or upper case `x` character, then the rest of the number is assumed to be in hexadecimal format, otherwise the number is assumed to be decimal. Decimal numbers are made up of the digits `0`–`9`. Hexadecimal numbers can additionally use the letters `A`–`F` in either upper or lower case to represent hexadecimal 10–15 respectively.

Within a line, space and tab characters may be used to format the source. Comments are marked by a leading semicolon (`;`). Any characters between a semicolon and the end of a line are ignored by the assembler. A comment may start in any column.

### 6.2.3 Expressions

In any `mvmasm` context requiring a numeric value, an expression may be used. Expression operands may be identifiers or numbers as defined above, or the predefined identifiers `TRUE` and `FALSE` which are synonyms for the values `1` and `0` respectively. The full set of ANSI-C numeric operators is provided, augmented by the operator `**` which stands for exponentiation. The supported operators, with their priorities on a scale of 1 (the lowest) to 11 (the highest) are listed in Table 6.1. Internally, all assembler arithmetic is done with the precision of a `long integer`. Most C compilers treat this as a 32-bit integer.

### 6.2.4 Instructions and addressing modes

The 17 MVM instructions are assembled using the mnemonics listed in Table 5.1. Operands are separated by commas, and take the form of an expression as defined above. For source, but not destination, operands the expression may be preceded by a hash sign (`#`) denoting literal addressing mode. The hash has no effect on the value returned by the expression, but sets the addressing mode for the operand containing the `#` to literal mode.

The following are all valid instructions:

```
ADD temp, x, y       ;sum x and y

SUB temp, temp, #1   ;decrement temp by 1
SUB temp, temp, #x   ;decrement temp by the value of x
SUB temp, temp, x    ;decrement temp by the value of the contents
                     ;of memory location with address x
```

| Operator | Priority | Function |
|---|---|---|
| > | 1 | Greater than |
| < | 1 | Less than |
| >= | 1 | Greater than or equal to |
| <= | 1 | Less than or equal to |
| == | 1 | Equal to |
| != | 1 | Not equal to |
| \|\| | 2 | Logical inclusive OR |
| && | 3 | Logical AND |
| ^ | 4 | Bitwise exclusive OR |
| \| | 5 | Bitwise inclusive OR |
| & | 6 | Bitwise AND |
| << | 7 | Shift left |
| >> | 7 | Shift right |
| + | 8 | Add |
| - | 8 | Subtract |
| * | 9 | Divide |
| / | 9 | Multiply |
| - | 10 | Monadic - (negate) |
| + | 10 | Monadic + (posite) |
| ~ | 10 | Bitwise complement |
| ! | 10 | Logical not |
| ** | 11 | Exponentiate |

**Table 6.1** Operator priorities in `mvmasm`

```
CPY temp, #(300 + 6) * 2  ;assign 612 to temp

HALT                      ;terminate execution

BEQ temp, start           ;If temp is zero, branch to start otherwise
                          ;continue execution with the next instruction
```

### 6.2.5 Directives

#### File inclusion

`INCLUDE ("`*filename*`")` – Open *filename* for reading at this point, continuing to read the parent file after the end of *filename* has been reached. This directive works just like the `#include` preprocessor directive in ANSI-C and the `INCLUDE` directive in `rdp`'s IBNF source language.

#### Assembly pointer manipulation

`CODE` *optional-expression* – Use the `CODE` pointer for subsequent assembly. If the *optional-expression* is present, set the `CODE` pointer to its value, otherwise carry on assembling starting at the most recent value of the `CODE` pointer. The `CODE` pointer is initialised to zero when the assembler starts.

`DATA` *optional-expression* – Use the `DATA` pointer for subsequent assembly. If the *optional-expression* is present, set the `DATA` pointer to its value, otherwise carry on assembling starting at the most recent value of the `DATA` pointer. The `DATA` pointer is initialised to zero when the assembler starts.

#### Data declaration directives

`WORD` *expression* – Reserve a word (two bytes) of memory and initialise the contents to the value of *expression*.

`BLOCKW` *expression* – Reserve *expression* words ($2 \times$ *expression* bytes) of memory. No initialisation of the memory is performed.

`STRING "`*character-string*`"` – Reserve sufficient bytes to hold the number of characters in *character-string* plus one, and initialise them to hold the *character-string* and a terminating zero byte.

#### Symbol assignment

*label*: `EQU` *expression* – Force the value of *label* to be the value of *expression*. This allows symbols to be set to arbitrary values, rather than the default behaviour which is for symbols to acquire the value of the assembly pointer at the time they are translated. The following are legal uses of `EQU`:

```
large_prime: EQU 131     ;large_prime <- 131

top_bit: EQU 128         ;top_bit <- 128
```

```
length: EQU top_bit + 1  ;length <- 129
```

### Transfer address specification

`END` *expression* – Mark the last valid line of the assembler source file and set the transfer address to the value of *expression*. The transfer address is written into the output file, and is used by the simulator to specify the address of the first instruction to be executed by the simulator.

## 6.3 Implementing `mvmasm`

Assembler syntax is designed to be easy to parse, and it is quite straightforward to design an assembler completely by hand. However, this tutorial manual is about `rdp`, so we shall use `rdp` to implement `mvmasm`. It turns out that two special `rdp` features are needed to efficiently implement assemblers: support for multiple passes and support for line oriented languages.

### 6.3.1 Multiple pass parsers

An interesting aspect of nearly all high level languages is that they may be translated in a single pass. This requires variables and functions to be declared before they are used, a rule rigidly enforced by Pascal and loosely enforced with the help of default behaviour in C. In an assembler, however, such a rule would make writing programs very tedious because of the large number of forward jumps present in real code. A forward jump has this form:

```
  BEQ temp, done
  ...
  ...
done:
  ...
```

Recall that labels take the value of the current assembly address at the point of their declaration, so declaring a label before it is used is not helpful here. On the other hand, the actual value of `done` will be unknown when the `BEQ` instruction is encountered for the first time.

There are three solutions to this predicament: we can either ban forward references; we can use a *fixup*; or we can use a multiple pass assembler. The first option is draconian since it means that only backward jumps are allowable. Remarkably, the standard assembler for at least one real computer (the Digico M16 minicomputer, a 16-bit machine with an architecture similar to that of the 12-bit DEC PDP-8) did indeed enforce this restriction. This machine assembled from paper tape, and internal memory was very limited. In addition, the design of the instruction set meant that forward jumps were less common than on modern machines so the designers thought that only having to feed the source papertape through once was a sufficient advantage to justify banning forward references.

A fixup assembler assembles the source into a buffer in memory. When it encounters the **BEQ** instruction above, it assumes a value of zero for the value of the label `done`, but also adds the instruction to a list of references for `done`. When it subsequently assembles the definition of `done`, it then knows the correct value and so can go back through the list for `done` filling in the correct value wherever it has been used. This approach, called fixing up the forward references, allows assembly to be completed in a single pass of the source file, but it requires the assembler to maintain an internal buffer which is as large as the largest possible program that could be assembled, and in addition a potentially large number of reference lists. In practice, a fixup based assembler can be rather complicated and might require a large amount of runtime storage.

By far the most common solution to the problem is to make two or more passes over the source file. On the first pass, a symbol table is loaded with the labels and their values as they are encountered. If they are first seen as an operand, then the corresponding table entry is loaded with an arbitrary value. By the end of the first pass, however, definitions will have been seen for all the symbols if the source program is syntactically well formed. The assembler then repeats the entire process, but making use of the label information from the first pass. The success of this approach relies on the fact that all instructions use a fixed size field to hold symbol values. As a result, the position of each instruction and data item in memory is fixed during the first pass, so symbol values will not change as a result of other symbols changing their value.

Are two passes sufficient? Well, if our only concern is forward references the two passes are enough, but consider this use of the **EQU** directive:

```
first: EQU second + 2
second: EQU third + 3
third: EQU 100
```

Here we have a chain of forward references. On pass one, labels `first` and `second` are to receive the value of expressions which include unknown data but label `third` will be correctly set to 100. On pass two label `second` can be correctly set to 103, but label `first` is still indeterminate, so in this case two passes is not sufficient. In general, we need as many passes as there are levels of forward referencing plus one. Since we can always add another level of forward referencing to a source file, any fixed number of passes is insufficient.

In practice, this situation is rather artificial, and real assemblers typically put an upper limit on the number of passes although it is not hard to simply keep re-parsing until all the symbols in the symbol table are determined. The `mvmasm` parser makes three passes, so it can in fact handle the situation shown above, but no more than two levels of forward referencing are allowed. As we shall see in a later section, **rdp** generated parsers can be set to make multiple passes by adding a directive of the form `PASSES(3)` to the **rdp** BNF specification file.

### 6.3.2 The EOLN scanner primitive

As noted in section 6.2.1 assemblers (and some early languages like the original FORTRAN) are line oriented, in that a maximum of one statement *per* line of source file is allowed. The **rdp** generated parsers that we have looked at previously have been *free format* in that line ends and white space may be introduced arbitrarily between language tokens so as to format the source file for human convenience. **rdp** provides a special scanner primitive denoted by **EOLN** which matches against the line end marker, and a special comment primitive **COMMENT_LINE** which can be used to specify comments which are introduced by a grammar token and terminated by a line end. If an **rdp** grammar does not include any instances of **EOLN** then line end markers are suppressed and treated as whitespace.

## 6.4 The mvmasm grammar

A listing of **mvmasm.bnf**, the **rdp** specification for **mvmasm** is shown in Figures 6.1–6.3. The main body of the **mvmasm** grammar is shown in Figure 6.2. The third part of the listing (Figure 6.3) shows a self contained interpreter for arithmetic expressions based on the C-language operators. This part of the grammar is a useful starting point for any small language based around expression evaluation. We shall look at the three parts in turn.

### 6.4.1 Directives for setting up the parser

**mvmasm** source files have default file type .mvm as specified in line 11. Three header files are used by the grammar: **mvm_aux.h** which contains the function prototypes for the auxiliary functions described in the next section, **mvm_def.h** which contains an enumeration listing the MVM operation codes (see Figure 6.4) and the ANSI-C library file **math.h** which is used to implement the exponentiation operator.

### 6.4.2 The MVM definition header

The parser uses three passes to resolve forward references in the way described above. Output file handling is performed by the **PRE** and **POST_PARSE** functions (**init()** and **quit()**, respectively) declared on lines 18 and 19. The -x command line switch is set up using an **ARG_BOOLEAN** directive in line 21 along with some other additional information for the help message.

**mvmasm** uses a symbol table to keep track of labels and their contents. When a label is first declared it is added to the symbol table and given the value of the assembler's current location counter. However, the **EQU** directive can be used to assign arbitrary numeric values to labels by evaluating expressions, and label values may of course be used in those expressions. Thus labels in **mvmasm** perform the same rôle as variables in the **minicalc** interpreter and it is perhaps to be expected that the symbol table declaration in lines 28–35 is essentially identical to the symbol table declaration in the **minicalc** and

```
(******************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* mvmasm.bnf - an assembler for Mini Virtual Machine assembler language
*
* This file may be freely distributed. Please mail improvements to the author.
*
******************************************************************************)
TITLE("mvmasm v1.5 - absolute assembler for mvm")
SUFFIX("mvm")
USES("mvm_aux.h")
USES("math.h")
USES("mvm_def.h")
PASSES(3)
PARSER(unit) (* name of start production *)

PRE_PARSE([* init(rdp_outputfilename); *])
POST_PARSE([* quit(rdp_outputfilename); *])

ARG_BOOLEAN(x execute_sim "execute assembled code using mvmsim simulator")
ARG_BLANK("")
ARG_BLANK("You can contact the author (Adrian Johnstone) at:")
ARG_BLANK("")
ARG_BLANK("Computer Science Department, Royal Holloway, University of London")
ARG_BLANK("Egham, Surrey, TW20 OEX UK. Email: A.Johnstone@rhbnc.ac.uk")

SYMBOL_TABLE(mvmasm 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char *id;
                integer val;
             *]
            )

unit ::= [* emit_code = (rdp_pass == 3);
            data_location = code_location = 0;  /* clear location counters */
            location = &code_location;          /* make code counter current */
            dummy_label = symbol_new_symbol(sizeof(mvmasm_data));  /* initialise error symbol */
         *]
         { code }.

code  ::= [* emit_eoln(); emit_loc(); last_label = NULL; *]
          [label ':'] [instr] [* emit_fill(); *] EOLN.

```

**Figure 6.1** An rdp BNF specification for mvmasm part 1: rdp directives and the start production

```
label ::= ID:lab
          [* if ((last_label = symbol_lookup_key(mvmasm, &lab, NULL)) == NULL)
               last_label = symbol_insert_key(mvmasm, &lab, sizeof(char*), sizeof(mvmasm_data));
             mvmasm_cast(last_label)->val = *location;
          *].

instr   ::= diadic | copy | branch | print | halt | directive.

diadic ::= [* int op, m1 = 1, m2 = 1; *]
           (
           'ADD' [* op = OP_ADD; *] e1: dst ',' ['#' [* m1=0; *] ] e1: src1 ',' ['#' [* m2=0; *] ] e1:src2 |
           'SUB' [* op = OP_SUB; *] e1: dst ',' ['#' [* m1=0; *] ] e1: src1 ',' ['#' [* m2=0; *] ] e1:src2 |
           'MUL' [* op = OP_MUL; *] e1: dst ',' ['#' [* m1=0; *] ] e1: src1 ',' ['#' [* m2=0; *] ] e1:src2 |
           'DIV' [* op = OP_DIV; *] e1: dst ',' ['#' [* m1=0; *] ] e1: src1 ',' ['#' [* m2=0; *] ] e1:src2 |
           'EXP' [* op = OP_EXP; *] e1: dst ',' ['#' [* m1=0; *] ] e1: src1 ',' ['#' [* m2=0; *] ] e1:src2 |

           'EQ'  [* op = OP_EQ;  *] e1: dst ',' ['#' [* m1=0; *] ] e1: src1 ',' ['#' [* m2=0; *] ] e1:src2 |
           'NE'  [* op = OP_NE;  *] e1: dst ',' ['#' [* m1=0; *] ] e1: src1 ',' ['#' [* m2=0; *] ] e1:src2 |
           'GT'  [* op = OP_GT;  *] e1: dst ',' ['#' [* m1=0; *] ] e1: src1 ',' ['#' [* m2=0; *] ] e1:src2 |
           'GE'  [* op = OP_GE;  *] e1: dst ',' ['#' [* m1=0; *] ] e1: src1 ',' ['#' [* m2=0; *] ] e1:src2 |
           'LT'  [* op = OP_LT;  *] e1: dst ',' ['#' [* m1=0; *] ] e1: src1 ',' ['#' [* m2=0; *] ] e1:src2 |
           'LE'  [* op = OP_LE;  *] e1: dst ',' ['#' [* m1=0; *] ] e1: src1 ',' ['#' [* m2=0; *] ] e1:src2
           ) [* emit_op(op, dst, src1, src2, m1, m2, 3); *] .

copy   ::= [* int m1 = 1; *]
           'CPY' e1: dst ',' ['#' [* m1=0; *] ] e1: src [* emit_op(OP_CPY, dst, src, 0, m1, 1, 2); *] .

branch ::= 'BEQ' e1: src ',' e1: label [* emit_op(OP_BEQ, label, src, 0, 1, 1, 2); *] |
           'BNE' e1: src ',' e1: label [* emit_op(OP_BNE, label, src, 0, 1, 1, 2); *] |
           'BRA' e1: label [* emit_op(OP_BEQ, label, 0, 0, 0, 1, 2); /* force immediate mode */ *] .

print  ::= [* int m1 = 1; *]
           (
           'PRTS' e1: src [* emit_op(OP_PRTS, 0, src, 0, 0, 1, 2); /* force immediate mode */ *] |
           'PRTI' ['#' [* m1=0; *] ] e1: src [* emit_op(OP_PRTI, 0, src, 0, m1, 1, 2); *]
           ).

halt   ::= 'HALT' [* emit_op(OP_HALT, 0, 0, 0, 1, 1, 0); *] .

directive  ::= 'INCLUDE' '(' string: filename ')'
               [* if (text_open(filename) == NULL)
                     text_message(TEXT_ERROR_ECHO, "include file '%s' not found\n", filename);
                  else
                  {
                    text_get_char();
                    scan_();
                  }
               *] |

               'CODE' [* location = &code_location; *] [ e1:n [* *location = n; *] ] |
               'DATA' [* location = &data_location; *] [ e1:n [* *location = n; *] ] |
               'WORD' e1:val [* emit2(val); *] |
               'BLOCKW' e1:val [* *location += 2 * val; *] |
               'STRING' string:str [* while (*str!=0) emit1(*str++); emit1(0); *] |
               'EQU' e1:val [* mvmasm_cast(current_label())->val = val; *] |
               'END' e1: val [* transfer = val; emit_transfer(); *] .
```

**Figure 6.2** An `rdp` BNF specification for `mvmasm` part 2: instructions

```
(* Expression interpreter using C operators and long int data ***************)

e1:integer ::= e2:result ['>'  e2:right [* result = result > right;  *] |  (* Greater than *)
                         '<'  e2:right [* result = result < right;  *] |  (* Less than *)
                         '>=' e2:right [* result = result >= right; *] |  (* Greater than or equal to *)
                         '<=' e2:right [* result = result <= right; *] |  (* Less than or equal to *)
                         '==' e2:right [* result = result == right; *] |  (* Equal to *)
                         '!=' e2:right [* result = result != right; *] ]. (* Not equal to *)

e2:integer ::= e3:result {'||' e3:right  [* result = result || right; *]}. (* Logical inclusive OR *)

e3:integer ::= e4:result {'&&' e4:right  [* result = result && right; *]}. (* Logical AND *)

e4:integer ::= e5:result {'^' e5:right  [* result ^= right; *]}.           (* Bitwise exclusive OR *)

e5:integer ::= e6:result {'|' e6:right  [* result |= right; *]}.           (* Bitwise inclusive OR *)

e6:integer ::= e7:result {'&' e7:right  [* result &= right; *]}.           (* Bitwise AND *)

e7:integer ::= e8:result {'<<' e8:right [* result <<= right; *] |          (* Shift left  *)
                         '>>' e8:right [* result >>= right; *] }.          (* Shift right *)

e8:integer ::= e9:result {'+' e9:right  [* result += right; *] |           (* Add *)
                         '-' e9:right  [* result -= right; *] }.           (* Subtract *)

e9:integer ::= e10:result {'*' e10:right  [* result *= right; *] |         (* Divide *)

e10:integer ::= '+' e10:result |                                           (* Posite *)
                '-' e10:result [* result = -result; *] |                   (* Negate *)
                '~' e10:result [* result = ~result; *] |                   (* Bitwise complement *)
                '!' e10:result [* result = !result; *] |                   (* Logical not *)
                e11:result.

e11:integer ::= e0:result ['**' e10:right [* result = (integer) pow((double) result, (double) right); *]].

e0:integer ::= [* mvmasm_data* temp; *]
               ID:name
               [* temp = mvmasm_cast(symbol_lookup_key(mvmasm, &name, NULL));
                  if (temp == NULL)
                  {
                    if (rdp_pass == 3)
                      text_message(TEXT_ERROR_ECHO,"Undefined symbol '%s'\n", name);
                    result = 0;
                  }
                  else
                    result = temp->val;
               *] |                                                        (* Variable *)
               INTEGER:result |                                            (* Numeric literal *)
               'TRUE' [* result = 1; *] |                                  (* Logical TRUE *)
               'FALSE' [* result = 0; *] |                                 (* Logical FALSE *)

               '(' e1:result ')'.                                          (* Parenthesised expression *)

string: char* ::= STRING_ESC('"' '\\'):result.

Comment ::= COMMENT_LINE(';').
```

**Figure 6.3** An `rdp` BNF specification for `mvmasm` part 3: expressions

```
/*******************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* mvm_def.h - Mini Virtual Machine opcode definitions
*
* This file may be freely distributed. Please mail improvements to the author.
*
*******************************************************************************/
enum opcodes{OP_HALT, OP_ADD, OP_SUB, OP_MUL, OP_DIV, OP_EXP,
  OP_EQ, OP_NE, OP_GT, OP_GE, OP_LT, OP_LE,
  OP_CPY,
  OP_BNE, OP_BEQ,
  OP_PRTS, OP_PRTI};
```

**Figure 6.4** The MVM opcode definitions

`minicond` grammars. The alphanumeric label indentifier is stored in the `char*` field `id` and the value of the label is held in the `val` field.

### 6.4.3 The main `mvmasm` grammar

The top level rule `unit` accepts zero or more lines of assembler source. `unit` itself is activated three times, once for each pass over the source. `rdp` automatically resets the input at the start of each pass, but some `mvmasm` variables need to be re-initialised each time as shown in lines 37–40. The boolean flag `emit_code` is set to TRUE on pass three and is used to control the output to the binary file: code emission is inhibited on passes one and two when this flag is FALSE. Both the data and code location counters are zeroed, and the code location is set as the default assembly address. Finally, `dummy_label` is initialised to point to a new symbol table record. This record is used to handle errors involving the `EQU` assembler directive as will be described below.

Each line of assembler source is processed by the `code` rule which can match an optional label, an optional instruction and a line end (`EOLN`) primitive. At the start of each line, an end of line character is sent to the output by calling `emit_eoln()` followed by the value of current location counter. The global variable `last_label` is set to `NULL` to indicate that no label has been seen yet on this line. After processing the contents of the line the `code` rule calls `emit_fill()` to pad the binary output to column 16. This ensures that the mixed binary/source output listing is properly aligned.

Labels are processed by rule `label`. The string representing the label's name is returned in attribute `lab` and the semantic action in lines 48–51 first looks up the label in the symbol table (inserting it if not already present) before loading the current value of the location counter into the symbol's `val` field. It should be clear that labels may be redefined in `mvmasm`, that is they may appear more than once in a label field. This is primarily intended to allow labels to have new values assigned with the `EQU` directive, but it does mean that a label might accidentally be used several times within a code segment and it is hard to imagine a situation where this would not represent a programming error. The reader might like to consider whether it would be appropriate to issue a warning message when such a doubly-declared label is encountered.

Rule `instr` on line 53 splits the handling of instructions into six cases: the first five handle the five syntactically different classes of MVM instructions and the sixth handles the assembler directives.

The diadic instructions (matched by rule `diadic`) illustrate the general principles used in all MVM instruction processing. Three local variables `op`, `m1` and `m2` are used to collect the operation code and the addressing modes for the two destination operands. Upon recognition of the opcode `op` is set to the corresponding member of the `opcodes` enumeration shown in Figure 6.4. The addressing modes are set to 1 (variable mode addressing) by default but are set to 0 (constant mode addressing) if the corresponding operand starts with a `#` character. The expression evaluator (which is described in the next section) is called for each operand. After the line has been processed the auxiliary function `emit_op()` is called to output the binary pattern corresponding to the instruction.

The other MVM instructions are processed similarly. The eight assembler directives are handled by rule `directive`. The `INCLUDE` directive collects a filename and calls `text_open()` to lookup and open the file. If the file is not found, `text_open()` returns `NULL` and an error message is issued. If the file is successfully opened, the text handler and scanner are initialised (lines 91–92). There is no need to restore the scanner and text handler context when an included file is closed because the text handler performs this task automatically.

The `CODE` and `DATA` directives switch the current assembly location to the code or data pointer respectively. They also optionally take an expression and update the location accordingly.

The `WORD`, `BLOCKW` and `STRING` directives allocate storage space for data. `WORD` takes an expression which is evaluated and emitted directly. The `BLOCKW` also takes an expression which is then used to update the location counter which has the effect of reserving a block of storage without initialising it. The `STRING` directive accepts a double quote delimited string and then emits along with a terminating zero (the ASCII NUL character).

The `EQU` directive takes an expression and updates the current label's `val` field accordingly. The `END` directive marks the end of the assembly unit and specifies the start address of the unit.

## 6.4.4 The expression evaluator

The expression evaluator follows the general principles used in the `minicalc` interpreter. In `mvmasm` a more complete set of operators is available than in `minicalc`, corresponding to the complete set of ANSI-C integer operators augmented with the exponentiation operator `**`. Two literal values have also been added, TRUE and FALSE, which yield 1 and 0 respectively. Identifiers are checked for validity on the final pass (lines 143–150): undefined labels on earlier passes are simply ignored.

```
 1: /******************************************************************************
 2: *
 3: * RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
 4: *
 5: * mvm_aux.c - Mini Virtual Machine assembler semantic routines
 6: *
 7: * This file may be freely distributed. Please mail improvements to the author.
 8: *
 9: ******************************************************************************/
10: #include <stdarg.h>
11: #include <stdio.h>
12: #include <stdlib.h>
13: #include "scan.h"
14: #include "memalloc.h"
15: #include "textio.h"
16: #include "mvmasm.h"
17: #include "mvm_aux.h"
18:
19: int emit_code = 0;
20: int execute_sim = 0;
21:
22: static FILE * objfile = NULL;
23:
24: unsigned long * location;
25: unsigned long data_location;
26: unsigned long code_location;
27: unsigned long transfer = 0;
28:
29: void * last_label = NULL;     /* pointer to most recently seen label */
30: void * dummy_label = NULL;    /* dummy symbol returned by current label on error */
31:
32: static int emitted = 0;       /* Count of bytes emitted this line */
33:
```

**Figure 6.5** mvmasm auxiliary functions part 1: declarations

## 6.5 mvmasm auxiliary functions

The mvmasm auxiliary functions shown in Figures 6.5 –6.7 perform file handling and output to the binary object file. Function `emitf()` (lines 34–51) forms the heart of the output routines: it simulates the behaviour of the ANSI-C `printf()` output function by accepting a formatted output string and an arbitrary number of output fields and then using ANSI-C `vprintf()` and `vfprintf()` functions to format the output. The ANSI-C standard library macros `va_list`, `va_start` and `va_end` are used to handle the variable number of arguments which `emitf()` may be passed — see any good book on ANSI-C for an explanation of their use. If the `emit_code` flag is false, the output is simply discarded, but if it is true (as it will be on pass 3) then the output is sent to the object file. In addition, if text echoing is enabled with a `-l` command line option (to construct an assembler listing) then up to the first 16 characters are also echoed to the screen.

The functions `emit_transfer()`, `emit_loc()` and `emit_fill()` call `emitf()` to print the transfer address and the current value of the location counter and to pad the line with spaces to column 16.

```
static int emitf(char * fmt, ...) /* conditional print to object file */
{
  int i;
  va_list ap;                 /* argument list walker */

  va_start(ap, fmt);

  if (emit_code)              /* no-op if not emitting... */
  {
    if (emitted < 16 && text_get_echo())
      i = vprintf(fmt, ap);
    vfprintf(objfile, fmt, ap);  /* ... otherwise pass to fprintf() */
  }

  va_end(ap);

  return(i);                  /* for completeness, although not used here */
}

void emit_eoln(void)
{
  if (emit_code)
    fprintf(objfile, "\n");
}

void emit_transfer(void)
{
  if (emit_code)
    emitted += emitf("*%.4lX", transfer);
}

void emit_loc(void)
{
  emitted = 0;
  emitf("%.4lX ", * location);
}

void emit_fill(void)
{
  if (text_get_echo())
  {
    while (emitted++ < 16) printf(" ");
      printf(" ");
  }
}

void emit_op(int op, unsigned long oper1, unsigned long oper2, unsigned long oper3,
             int mode1, int mode2, int opers)
{
  emit1((unsigned long) op);  /* output opcode */
  emit1((unsigned long)((mode1 << 4)| mode2));  /* output addressing modes */
  if (opers > 0)
    emit2(oper1);
  if (opers > 1)
    emit2(oper2);
  if (opers > 2)
    emit2(oper3);
}

```

**Figure 6.6** mvmasm auxiliary functions part 2: main output routines

```
void emit1(unsigned long val)
{
  emitted += emitf("%.2lX", val);
  (* location)++;
}

void emit2(unsigned long val)
{
  emitted += emitf("%.4lX", val);
  (* location)+= 2;
}

void * current_label(void)    /* check that there is a valid label on this line */
{
  if (last_label == NULL)
  {
    text_message(TEXT_ERROR_ECHO, "Missing label on directive\n");
    return & dummy_label;
  }
  else
    return last_label;
}

void init(char * outputfilename)
{
  if (* outputfilename == '-')
    objfile = stdout;
  else if ((objfile = fopen(outputfilename, "w"))== NULL)
    text_message(TEXT_FATAL, "Unable to open object file");
}

int quit(char * outputfilename)
{
  fclose(objfile);

  text_message(TEXT_INFO, "Transfer address %.8lX\n", transfer);

  if (execute_sim && * outputfilename != '-')
  {
    #define COMMAND "mvmsim -t -v "
    char * command =(char *) mem_calloc(1, strlen(outputfilename)+ strlen(COMMAND)+ 1);

    command = strcat(command, COMMAND);
    command = strcat(command, outputfilename);

    text_message(TEXT_INFO, "Calling simulator: %s \n", command);

    if (system(command)!= 0)
      text_message(TEXT_FATAL, "Not enough memory or simulator not found\n");
  }

  return 0;
}
```

**Figure 6.7** `mvmasm` auxiliary functions part 3: housekeeping functions

The emit1() and emit2() functions in lines 92–102 output one and two byte (two and four hexadecimal digit) values and then update the current assembly location accordingly. They are used by the emit_op() function to output complete MVM instructions. Each instruction comprises an opcode byte and a mode byte constructed from the two mode fields passed into the function. Between zero and three 16-bit operands are then output.

Function current_label() (lines 104–113) returns the value of the last label seen. At the start of each line of assembler source, the parser resets the variable last_label to NULL. This flags the error condition for the EQU directive — if no label has been seen an error message is issued and a pointer to dummy_label is returned instead. This is so as to ensure that subsequent assignments to the label fields of the symbol table record returned by current_label() do not need to check for a NULL pointer.

The init() function in lines 115–121 is straightforward: it simply attempts to open the output file and issues an error message if the file open fails. The quit() function closes the object file and echoes the transfer address to the screen. If the -x command line option has been used then the flag execute_sim will be true. Assuming that the object file was not sent to stdout, i.e. that a file containing the object code exists, a command is constructed that will run the simulator on the object file and then the ANSI-C library function system() is called to pass control to the simulator.

# Chapter 7

# A single pass compiler for `miniloop`

This chapter describes the first of two full compilers for an extended version of the `minicond` syntax that provides a `while` loop and a compound statement delimited by `begin` and `end` keywords. The compiler works by recognising compilable fragments of the source code, such as an individual assignment or an arithmetic operation, and then emitting the corresponding MVM assembler instruction. The output of the compiler is a complete assembler program with the same semantics as the `miniloop` source program, and this can then be assembled using `mvmasm` and executed using `mvmsim`.

In this chapter we shall describe the language features added to `miniloop`, give an example of the compiler's output and then describe the assembler code patterns that are used to implement the `miniloop` high level language constructs. We shall then describe in detail the `rdp` grammar and auxiliary routines that are used to implement `miniloop`. In the next chapter we shall describe another compiler called `minitree` which compiles from the same source language to the same MVM assembler code as `miniloop`. The difference between the two compilers is that `miniloop` emits assembler code during the parse whereas `minitree` builds an internal representation of the source program (a modified derivation tree) and then, in a separate phase, traverses the tree to output the assembler code. The two compilers are functionally almost identical as they stand, but `minitree` allows code optimisations such as rearranging the order of instructions to be performed. Since `miniloop` is a single pass compiler it cannot perform code re-ordering.

## 7.1 `miniloop` features

`miniloop` programs look like `minicond` programs with some additional features — the `minicond` and `minicalc` languages are almost strict subsets of the `miniloop` language so any `minicalc` or `minicond` program will be correctly handled by the `miniloop` compiler. The only exception to this rule is that `miniloop` variable names *must not* start with two underscore characters. This is because `miniloop` generates internal identifier names with that form, and we do not want user identifiers and internal identifiers to clash. Figure 7.1 shows an example `miniloop` program.

```
(******************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* testloop.m - a piece of Miniloop source to test the Miniloop compiler
*
* This file may be freely distributed. Please mail improvements to the author.
*
******************************************************************************)

int a=3+4, b=1;

print("a is ", a, "\n");

b=a*2;

print("b is ", b, ", -b is ", -b, "\n");

print(a, " cubed is ", a**3, "\n");

int z = a;

if z==a then print ("z equals a\n") else print("z does not equal a\n");

z=a - 3;

if z==a then print ("z equals a\n") else print("z does not equal a\n");

a = 3;

while a > 0 do
begin
  print("a is ", a, "\n");
  a = a - 1
end;

(* End of testloop.m *)
```

**Figure 7.1** An example `miniloop` program (`testloop.m`)

The output produced when this is run through the `miniloop` compiler and then assembled and simulated by `mvmasm` and `mvmsim`, is shown in Figure 7.2. The assembler code produced by `miniloop` is shown in Figures 7.6–7.8 and discussed in section 7.8.

## 7.1.1 The `begin end` block (compound statement)

It is useful to be able to group statements together into blocks so that a single `if` statement can control the execution of a list of statements. In `minicond` only a single statement could be placed within the `then` or `else` clause of an `if` statement. The `begin end` brackets allow statements to be grouped and treated as a single, compound, statement. It is worth noting that `miniloop` is strict about the placement of semicolons which are statement *separators* not statement *terminators* as they are in ANSI-C. The last statement in a `begin end`

```
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 --
```

**Figure 7.2** `mvmsim` output for assembled output from `miniloop` for the example program

block cannot by definition therefore have a semicolon following it, so it is always an error to have a semicolon before an **end** statement. This usage follows that of Pascal and Algol-68 although Pascal does allow an *empty* statement which in most cases allows spurious semicolons to be accepted.

### 7.1.2 The `while` loop

Lines 31–35 of Figure 7.1 illustrate the use of the **while** loop which is essentially identical to the **while** loop in Pascal. A relational expression is repeatedly evaluated and the statement after the **do** keyword is evaluated as long as the expression is TRUE. The statement may be either a simple statement (such as **print** or indeed another **while**) or it may be a compound statement, as in the example.

## 7.2 Arranging data and code in memory

MVM is limited to 64K bytes of memory because the address fields in the instructions are only 16 bits long and $2^{16} = 65536 = 64$K. MVM instructions can be executed from any location and operands can also reside anywhere in memory, but `miniloop` places code in a single sequence starting at location $1000_{16}$ and data in a single block starting at location $8000_{16}$. Within the data block, internal temporary variables created during the compilation of expressions are placed at the end. This memory map is shown in Figure 7.3.

We establish this memory map by setting the `CODE` and `DATA` assembly pointers appropriately. At the start of each program, `miniloop` issues the following assembler directives:

```
                 DATA 0x8000
__MPP_DATA:
                 CODE 0x1000
__MPP_CODE:
```

This has the effect of initialising the start address for data assembly to $8000_{16}$ and setting the label `__MPP_DATA` to the address of the first data item,

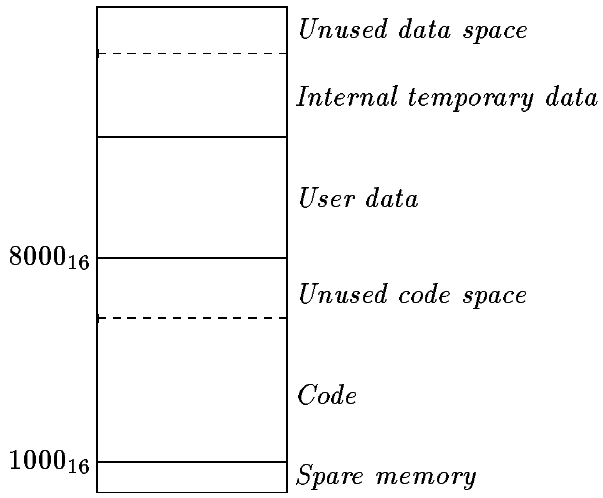



**Figure 7.3** MVM memory map for programs compiled by `miniloop`

and then initialising the start address for code assembly to $1000_{16}$ and setting the label `__MPP_CODE` to the address of the first instruction, which will subsequently be used as the transfer address.

At the end of each program `miniloop` writes out directives of this form:

```
        DATA
__temp: BLOCKW 9  ;declare array of temporaries

        END __MPP_CODE
```

Here assembly is switched to the data region and a block of temporaries (in this case nine words long) is specified. These form the array of temporary variables used during expression evaluation (as described in section 7.4) and represented by the region *Internal temporary data* in Figure 7.3. Finally, the value of `__MPP_CODE` is established as the transfer address by naming it in an `END` directive.

## 7.3 Compiling declarations

A declaration in `miniloop` such as

```
int a;
```

reserves space for one integer in memory and makes the identifier `a` a synonym for the address of that variable. Whenever the compiler encounters a declaration it switches to the `DATA` location and assembles a `WORD` directive:

```
   DATA
a: WORD 0
```

The WORD directive reserves one word (two bytes) of memory and initialises them, in this case to zero. Following such a declaration we can use the identifier a to refer symbolically to the location holding the contents of the variable a just as we would in the high level source code.

## 7.4 Compiling arithmetic expressions

The parser breaks expressions down into individual operations taking account of operator priority and associativity as discussed in Chapter 2. Each operation is then compiled into the corresponding MVM instruction with the destination operand being a temporary variable. A sub-expression of the form 3 + 4 will be compiled to

```
ADD  __temp + 0,#3,#4    ;__temp + 0 := #3 + #4
```

The temporary variables do not need to be separately declared in the way that user variables were handled in the previous section. Instead, the compiler keeps count of the number of temporaries used and declares them in a block at the end of the program. The temporaries are always referred to as __temp + *n* where *n* us the number of the temporary. This uses the address calculation capability of the assembler to avoid the need for a large number of separate labels.

## 7.5 Compiling print statements

The print statement can take an arbitrary number of parameters of either string or integer type. The MVM instruction set provides two opcodes specifically for printing strings and integers.

For an integer parameter, code to evaluate the arithmetic expression is issued which leaves a value in a temporary variable *t*. The compiler then simply issues an instruction of the form

```
PRTI __temp + t  ;print integer
```

For the case of an expression made up of a single variable, the expression evaluator returns the name of that variable instead of the name of a temporary, so code of the form

```
PRTI v                ;print integer
```

will be issued, where *v* is the name of the variable.

For a string parameter the compiler outputs code of the form

```
       DATA
__STR_2: STRING "b is "

       CODE
       PRTS __STR_2
```

The string is stored in data space and given a unique label (in this case __STR_2). The compiler then switches back to code space and emits a PRTS instruction.

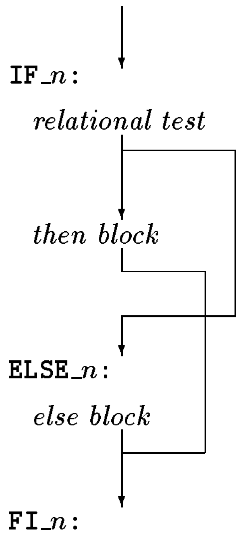



**Figure 7.4** Flow of control through a compiled `if-then-else` statement

## 7.6 Compiling `if` statements

An `if-then-else` statement defines three blocks of code: a relational expression, a `then` block and an `else` block. Figure 7.4 illustrates the code template used by `miniloop` and the allowed forms of control flow through the construct. `miniloop` maintains an internal label counter which is advanced each time a new unique label is required. Such labels are needed for labelling the strings used when assembling `print` statements containing string parameters and whenever a structured statement (such as `if-then` or `while-do`) is encountered. In the case of an `if-then-else` statement the start of the statement (corresponding to the first assembler instruction in the compiled version of the relational expression) is labeled `__IF_`$n$ where $n$ is the current value of the label counter. $n$ is called the number of the control statement. Similarly, the end of the statement is labeled `__FI_`$n$ and the start of the `else` block is labeled with `__ELSE_`$n$.

The relational expression is compiled first, yielding a temporary variable which will contain a zero if the expression evaluates to FALSE and a one otherwise. The compiler then issues the assembler instruction

```
BEQ  __temp + t, __ELSE_n   ;ifn __temp + t go to __ELSE_n
```

where $t$ is the number of the temporary containing the result of evaluating the relational expression and $n$ is the number of the control statement as defined above. This has the effect of jumping to the `else` block if the condition was false.

The compiler then emits the code for the `then` block followed by the assembler instruction

```
BRA  __FI_n      ;go to __FI_n
```

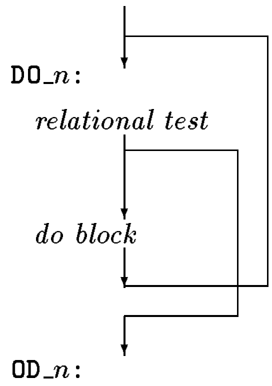



**Figure 7.5** Flow of control through a compiled `while-do` statement

which causes control to flow unconditionally to the end of the `if` statement. Finally, the compiler emits the `__ELSE_`*n* label and the code for the `else` block (which may be empty) finishing off with the `__FI_`*n* label.

## 7.7 Compiling `while` loops

A `while-do` statement is similar to an `if-then` statement (with no `else`) clause which is followed by a jump back to the relational test. There are two blocks of code: a relational expression, and the `do` block. Figure 7.5 illustrates the code template used by `miniloop` and the allowable control flow through the construct.

The compiler emits two labels for each `while-do` loop: one of the form `__DO_`*n* to mark the start of the statement and one of the form `__OD_`*n* to mark the end, where $n$ is the number of the control statement.

The relational expression is compiled first, yielding a temporary variable which will contain a zero if the expression evaluates to FALSE and a one otherwise. The compiler then issues the assembler instruction

```
BEQ  __temp + t, __OD_n     ;ifn __temp + t go to __OD_n
```

where $t$ is the number of the temporary containing the result of evaluating the relational expression and $n$ is the number of the control statement. This has the effect of jumping to the end of the `while-do` if the condition was false. The code for the `do` block is then emitted followed by the `__OD_`*n* label.

## 7.8 Typical compiler output

Figures 7.6–7.8 show the compiled output for the test program in Figure 7.1 which contains instances of all the constructs described above. In particular, note the setup and wrapup code in lines 3–6 and 137–142, the declaration in lines 8–9, the arithmetic expression evaluation in line 12 and the assignment of its result to user variable **a** in line 13, the **if-then-else** statement at lines 97–114 and the **while-do** statement at lines 116–135.

## 7.9 Implementing miniloop

The **miniloop** compiler makes a single pass over the source file emitting MVM instructions as it goes. We shall examine the grammar first and then look at the auxiliary functions which perform the actual code output.

### 7.9.1 A grammar for miniloop

The overall form of the **miniloop** grammar shown in Figures 7.9 and 7.10 is similar to the **minicond** grammar with the addition of syntax for a **while-do** loop in lines 56–61 and syntax for the **begin-end** compound statement on line 67. The symbol table declared in lines 17–22 is used only for keeping track of whether a variable has been correctly identified and so the symbol table data specified in line 21 includes only the **id** field — there is no need for an integer data field as there was for the **minicalc** and **minicond** interpreters. When a variable is declared it is checked for validity (line 39–40): **miniloop** variable names must not begin with a double underscore (**__**) because these might clash with the internal label names. Lines 44–67 show the statement compiler. This emits code according to the templates described in the previous sections.

A significant difference between the **miniloop** grammar and the earlier **minicond** and **minicalc** grammars is that here the expression rules return **char*** attributes rather then **integer** ones. In the previous grammars, the expression rules formed an *interpreter* that returned **values**. In **miniloop** the rules return the labels of locations that will contain the values at run time. Each level of the expression tree has this basic form:

```
79: e1:char* ::= [* char* dst; *] e2:left { [* dst = new_temporary(); *]
80:                ( '+' e2:right [* emit("ADD", "+", dst, left, right); *] |
81:                  '-' e2:right [* emit("SUB", "-", dst, left, right); *]
82:                )
83:                [* left = dst; *]
84:              } [* result = left; *].
```

The auxiliary function **emit()** outputs one assembler instruction constructed from the supplied opcode and operand parameters. The **new_temporary()** function constructs a string of the form **__temp +** $n$ where $n$ is the name of the next available temporary variable. This temporary then becomes the destination operand for the assembler instruction corresponding to the operator being processed.

### 7.9.2 miniloop auxiliary functions

The **miniloop** auxiliary functions are shown in Figures 7.11 and 7.12. They perform file handling, output to the assembler object file and some housekeeping concerned with the generation of unique labels. Function **emitf()** (lines 24–31) forms the heart of the output routines: it simulates the behaviour of the

```
0000                        1: ; testloop.mvm - generated from 'testloop.m'
0000                        2:
0000                        3:  DATA 0x8000
8000                        4: __MPP_DATA:
8000                        5:  CODE 0x1000
1000                        6: __MPP_CODE:
1000                        7:
1000                        8:  DATA
8000 0001                   9: a: WORD 0
8002                       10:
8002                       11:  CODE
1000 0100807400030004      12:  ADD  __temp + 0, #3, #4        ;__temp + 0 := #3 + #4
1008 0C1180008074          13:  CPY  a, __temp + 0     ;a := __temp + 0
100E                       14:
100E                       15:  DATA
8002 0001                  16: b: WORD 0
8004                       17:
8004                       18:  CODE
100E 0C0180020001          19:  CPY  b, #1     ;b := #1
1014                       20:
1014                       21:  DATA
8004 612069732000          22: __STR_0: STRING "a is "
800A                       23:
800A                       24:  CODE
1014 0F0100008004          25:  PRTS __STR_0
101A 101100008000          26:  PRTI a ;print integer
1020                       27:
1020                       28:  DATA
800A 0A00                  29: __STR_1: STRING "\n"
800C                       30:
800C                       31:  CODE
1020 0F010000800A          32:  PRTS __STR_1
1026 0310807580000002      33:  MUL  __temp + 1, a, #2         ;__temp + 1 := a * #2
102E 0C1180028075          34:  CPY  b, __temp + 1     ;b := __temp + 1
1034                       35:
1034                       36:  DATA
800C 622069732000          37: __STR_2: STRING "b is "
8012                       38:
8012                       39:  CODE
1034 0F010000800C          40:  PRTS __STR_2
103A 101100008002          41:  PRTI b ;print integer
1040                       42:
1040                       43:  DATA
8012 2C202D6220697320      44: __STR_3: STRING ", -b is "
801B                       45:
801B                       46:  CODE
1040 0F0100008012          47:  PRTS __STR_3
1046 0211807600008002      48:  SUB  __temp + 2, 0, b  ;__temp + 2 := 0 - b
104E 101100008076          49:  PRTI __temp + 2        ;print integer
1054                       50:
```

**Figure 7.6** miniloop compiled output for the example program: part 1

```
1054                       51:  DATA
801B 0A00                  52: __STR_4: STRING "\n"
801D                       53:
801D                       54:  CODE
1054 0F010000801B          55:  PRTS __STR_4
105A 101100008000          56:  PRTI a ;print integer
1060                       57:
1060                       58:  DATA
801D 2063756265642069      59: __STR_5: STRING " cubed is "
8028                       60:
8028                       61:  CODE
1060 0F010000801D          62:  PRTS __STR_5
1066 0510807780000003      63:  EXP  __temp + 3, a, #3        ;__temp + 3 := a ** #3
106E 101100008077          64:  PRTI __temp + 3        ;print integer
1074                       65:
1074                       66:  DATA
8028 0A00                  67: __STR_6: STRING "\n"
802A                       68:
802A                       69:  CODE
1074 0F0100008028          70:  PRTS __STR_6
107A                       71:
107A                       72:  DATA
802A 0001                  73: z: WORD 0
802C                       74:
802C                       75:  CODE
107A 0C11802A8000          76:  CPY  z, a     ;z := a
1080                       77: __IF_7:
1080 06118078802A8000      78:  EQ   __temp + 4, z, a  ;__temp + 4 := z == a
1088 0E11109A8078          79:  BEQ  __temp + 4,__ELSE_7      ;ifn __temp + 4 go to __ELSE_7
108E                       80:
108E                       81:  DATA
802C 7A20657175616C73      82: __STR_8: STRING "z equals a\n"
8038                       83:
8038                       84:  CODE
108E 0F010000802C          85:  PRTS __STR_8
1094 0E0110A00000          86:  BRA  __FI_7    ;go to __FI_7
109A                       87: __ELSE_7:
109A                       88:
109A                       89:  DATA
8038 7A20646F6573206E      90: __STR_9: STRING "z does not equal a\n"
804C                       91:
804C                       92:  CODE
109A 0F0100008038          93:  PRTS __STR_9
10A0                       94: __FI_7:
10A0 0210807980000003      95:  SUB  __temp + 5, a, #3        ;__temp + 5 := a - #3
10A8 0C11802A8079          96:  CPY  z, __temp + 5     ;z := __temp + 5
10AE                       97: __IF_10:
10AE 0611807A802A8000      98:  EQ   __temp + 6, z, a  ;__temp + 6 := z == a
10B6 0E1110C8807A          99:  BEQ  __temp + 6,__ELSE_10     ;ifn __temp + 6 go to __ELSE_10
10BC                      100:
```

**Figure 7.7** miniloop compiled output for the example program: part 2

```
10BC                        101:  DATA
804C 7A20657175616C73       102: __STR_11: STRING "z equals a\n"
8058                        103:
8058                        104:  CODE
10BC 0F010000804C           105:  PRTS __STR_11
10C2 0E0110CE0000           106:  BRA  __FI_10   ;go to __FI_10
10C8                        107: __ELSE_10:
10C8                        108:
10C8                        109:  DATA
8058 7A20646F6573206E       110: __STR_12: STRING "z does not equal a\n"
806C                        111:
806C                        112:  CODE
10C8 0F0100008058           113:  PRTS __STR_12
10CE                        114: __FI_10:
10CE 0C0180000003           115:  CPY  a, #3     ;a := #3
10D4                        116: __DO_13:
10D4 0810807B80000000       117:  GT   __temp + 7, a, #0         ;__temp + 7 := a > #0
10DC 0E111108807B           118:  BEQ  __temp + 7,__OD_13       ;ifn __temp + 7 go to __OD_13
10E2                        119:
10E2                        120:  DATA
806C 612069732000           121: __STR_14: STRING "a is "
8072                        122:
8072                        123:  CODE
10E2 0F010000806C           124:  PRTS __STR_14
10E8 101100008000           125:  PRTI a ;print integer
10EE                        126:
10EE                        127:  DATA
8072 0A00                   128: __STR_15: STRING "\n"
8074                        129:
8074                        130:  CODE
10EE 0F0100008072           131:  PRTS __STR_15
10F4 0210807C80000001       132:  SUB  __temp + 8, a, #1         ;__temp + 8 := a - #1
10FC 0C118000807C           133:  CPY  a, __temp + 8     ;a := __temp + 8
1102 0E0110D40000           134:  BRA  __DO_13   ;go to __DO_13
1108                        135: __OD_13:
1108                        136:
1108 0011                   137:  HALT
110A                        138:
110A                        139:  DATA
8074                        140: __temp: BLOCKW 9  ;declare array of temporaries
8086                        141:
8086 *1000                  142:  END __MPP_CODE
```

**Figure 7.8** miniloop compiled output for the example program: part 3

```
(*******************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* miniloop.bnf - a decorated mini loop grammar with single pass compiler semantics
*
* This file may be freely distributed. Please mail improvements to the author.
*
*******************************************************************************)
TITLE("Miniloop compiler V1.50 (c) Adrian Johnstone 1997")
SUFFIX("m")
PARSER(program)
USES("ml_aux.h")
TREE
OUTPUT_FILE("miniloop.mvm")

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

check_declared ::= [* if (symbol_lookup_key(mini, &dst, NULL) == NULL)
                      {
                        text_message(TEXT_ERROR, "Undeclared variable '%s'\n", dst);
                        symbol_insert_key(mini, &dst, sizeof(char*), sizeof(mini_data));
                      }
                   *].

program   ::=  [* emit_open(rdp_sourcefilename, rdp_outputfilename); *]
               { [var_dec | statement] ';'}
               [* emit_close(); *].

var_dec ::= 'int' ( ID:dst
                    [* emitf(" \n DATA\n%s: WORD 0\n\n CODE\n",dst); *]
                    ['=' e0:left [* emit("CPY", "", dst, left, NULL); *] ]
                    [* symbol_insert_key(mini, &dst, sizeof(char*), sizeof(mini_data));
                       if (*dst == '_' && *(dst+1) == '_')
       text_message(TEXT_ERROR_ECHO, "variable names must not begin with two underscores\n");
                    *]
                  )@','.                       (* Declaration *)

statement ::= ID:dst check_declared
              '=' e0:left [* emit("CPY", "", dst, left, NULL); *] | (* assignment *)

              [* integer label = new_label(); *]                  (* if statement *)
              [* emitf("__IF_%lu:\n", label); *]
              'if' e0:left
              [* emitf(" BEQ  %s,__ELSE_%lu\t;ifn %s go to __ELSE_%lu \n",left,label,left, label); *]
              'then' statement
              [* emitf(" BRA  __FI_%lu\t;go to __FI_%lu\n__ELSE_%lu:\n", label, label, label); *]
              [ 'else' statement ]
              [* emitf("__FI_%lu:\n", label); *] |

```

**Figure 7.9** An rdp BNF specification for miniloop part 1: statements

```
              [* integer label = new_label(); *]        (* while do statement *)
              [* emitf("__DO_%lu:\n", label); *]
              'while' e0:left
              [* emitf(" BEQ  %s,__OD_%lu\t;ifn %s go to __OD_%lu \n",left,label,left, label); *]
              'do' statement
              [* emitf(" BRA  __DO_%lu\t;go to __DO_%lu\n__OD_%lu:\n", label, label, label); *] |

              'print' '(' ( e0:left [* emit_print('I', left); *] |
                            String:left [* emit_print('S', left); *]
                          )@',' ')' |           (* print statement *)

              'begin' (statement)@';' 'end'.  (* compound statement *)

e0:char* ::= [* char* dst; *] e1:left [ [* dst = new_temporary(); *]
               ('>' e1:right [* emit("GT ", ">", dst, left, right); *]  |  (* Greater than *)
                '<' e1:right [* emit("LT ", "<", dst, left, right); *]  |  (* Less than *)
                '>=' e1:right [* emit("GE ", ">=", dst, left, right); *]|  (* Greater than or equal *)
                '<=' e1:right [* emit("LE ", ">=", dst, left, right); *]|  (* Less than or equal *)
                '==' e1:right [* emit("EQ ", "==", dst, left, right); *]|  (* Equal *)
                '!=' e1:right [* emit("NE ", "!=", dst, left, right); *]   (* Not equal *)
               ) [* left = dst; *]
             ] [* result = left; *].

e1:char* ::= [* char* dst; *] e2:left { [* dst = new_temporary(); *]
               ( '+' e2:right [* emit("ADD", "+", dst, left, right); *] |   (* Add *)
                 '-' e2:right [* emit("SUB", "-", dst, left, right); *]     (* Subtract *)
               )
               [* left = dst; *]
             } [* result = left; *].

e2:char* ::= [* char* dst; *] e3:left { [* dst = new_temporary(); *]
               ( '*' e3:right [* emit("MUL", "*", dst, left, right); *] |   (* Multiply *)
                 '/' e3:right [* emit("DIV", "/", dst, left, right); *]     (* Divide *)
               )
               [* left = dst; *]
             } [* result = left; *].

e3:char* ::= [* int negate = 0; char* dst;*]

             {('+'|'-' [* negate ^= 1; *])} e4:result (* Posite or negate *)
[* if (negate) {dst = new_temporary(); emit("SUB", "-", dst, "0", result); result = dst; } *].

e4:char* ::= [* char *dst; *]
             e5:left
             [ [* dst = new_temporary(); *]
               '**' e4:right [* emit("EXP", "**", dst, left, right);  *] (* Exponentiate *)
               [* left = dst; *]
             ] [* result = left; *].

e5:char* ::= ID:dst check_declared [* result = dst; *] |                 (* Variable access *)
             INTEGER:val [* result = (char*) mem_malloc(12); sprintf(result, "#%lu", val); *] |
            '(' e1:result ')'.    (* Parenthesised expression *)

comment ::= COMMENT_NEST('(*' '*)').  (* Comments: stripped by lexer *)
String:char*  ::= STRING_ESC('"' '\\'):result.     (* Strings for print *)

(* End of miniloop.bnf *)
```

**Figure 7.10** An `rdp` BNF specification for `miniloop` part 2: expressions

ANSI-C `printf()` output function by accepting a formatted output string and an arbitrary number of output fields and then using ANSI-C `vprintf()` and `vfprintf()` functions to format the output. The ANSI-C standard library macros `va_list`, `va_start` and `va_end` are used to handle the variable number of arguments which `emitf()` may be passed — see any good book on ANSI-C for an explanation of their use.

The `emit_open()` and `emit_close()` functions open and close the output file as well as writing the wrapper code that appears at the start and end of every compiled program (see section 7.2). The function `emit()` is used to output a single assembler instruction along with a comment that renders the operation in an algebraic form to make reading the output easier for those not used to assembler format. The `emit_print()` function is a specialised output routine for handling the `print` statement in `miniloop`. It generates the code templates discussed in section 7.5. The `new_temporary()` function allocates a block of memory to hold the name of the temporary and then uses the `sprintf()` function to construct the name.

```
/*******************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* ml_aux.c - miniloop one pass compiler semantic routines
*
* This file may be freely distributed. Please mail improvements to the author.
*
*******************************************************************************/
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include "textio.h"
#include "memalloc.h"
#include "ml_aux.h"

FILE * outfile;

static long unsigned temp_count = 0;

int emitf(const char * fmt, ...)
{
  int i;
  va_list ap;                 /* argument list walker */

  va_start(ap, fmt);          /* pass parameters to vprintf */
  i = vfprintf(outfile, fmt, ap);  /* remember count of characaters printed */
  va_end(ap);                 /* end of var args block */

  return i;                   /* return number of characters printed */
}

void emit_open(char * sourcefilename, char * outfilename)
{
  if ((outfile = fopen(outfilename, "w"))== NULL)
    text_message(TEXT_FATAL, "unable to open output file \'%s\' for writing\n", outfilename);
  emitf("; %s - generated from \'%s\'\n\n", outfilename, sourcefilename);
  emitf(" DATA 0x8000\n__MPP_DATA:\n CODE 0x1000\n__MPP_CODE:\n");
}

void emit_close(void)
{
  emitf("\n HALT\n\n DATA\n__temp: BLOCKW %lu  ;declare array of temporaries\n\n"
        " END __MPP_CODE\n", temp_count);
  fclose(outfile);
}

```

**Figure 7.11** miniloop auxiliary functions part 1: low level output

```
void emit(char * asm_op, char * alg_op, char * dst, char * src1, char * src2)
{
  emitf(" %s  %s, %s", asm_op, dst, src1);
  if (src2 != NULL)
    emitf(", %s", src2);

  /* Now output algebraic style */
  emitf(" \t;%s := %s %s", dst, src1, alg_op);
  if (src2 != NULL)
    emitf(" %s", src2);
  emitf("\n");
}

void emit_print(char kind, char * src)
{
  if (kind == 'S')
  {
    unsigned long label = new_label();

    emitf("\n DATA\n__STR_%lu: STRING \"", label);
    text_print_C_string_file(outfile, src);
    emitf("\"\n\n CODE\n PRTS __STR_%lu\n", label);
  }
  else
  {
    emitf(" PRTI ");
    text_print_C_string_file(outfile, src);
    emitf("\t;print integer\n");
  }
}

char * new_temporary(void)
{
  char * ret =(char *) mem_malloc(30);

  sprintf(ret, "__temp + %lu", temp_count++);

  return ret;
}

unsigned long new_label(void)
{
  static long unsigned label = 0;

  return label++;
}

/* End of ml_aux.c */
```

**Figure 7.12** miniloop auxiliary functions part 2: high level output and housekeeping

# Chapter 8

## minitree – a multiple pass compiler

Some translation tasks are difficult to perform during a parse, even if a multi-pass parser is employed. High quality compilers, for instance, can perform many different code improvement transformations as part of an optimisation phase. Typically, optimisations work by relating together widely separated parts of the source text. Take for example, *common sub-expression elimination* which is one of the most commonly applied optimisations: an assignment between array elements in ANSI-C such as

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

actually contains two identical calculations if the sizes of the `a` and `b` arrays are the same. (In detail, `i` must be multiplied by the width of the array and added to `j`.) A single pass translator has to process each of these identical calculations in isolation and so is unlikely to be able to rearrange the calculations into the equivalent but more efficient form

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

If a multiple pass translator is to be used then it is usual to construct a data structure in memory that represents the user program in a manner which may be efficiently processed. Simply storing the original program text is inefficient because discovering a derivation for an input text is so time consuming: that is after all the primary function of the parsers that `rdp` constructs and it would clearly be wasteful to run the process several times. (Of course, just because this is a wasteful process it need not stop us using it where applicable and `rdp` provides the `PASSES` directive for precisely this purpose. Simple multi-pass applications, such as the implementation of a translator from a machine's assembly language to its machine code, may usefully exploit this strategy. You can read about the design and implementation of such as assembler in Chapter 6.)

Leaving aside issues of efficiency, making multiple independent passes over the source text does not allow us to make connections between widely separated parts of the text because the parsers generated by `rdp` only look at a single symbol at a time: they do not of themselves keep track of complete sentences or program statements. However, `rdp` can be set to build a *derivation tree* whilst it performs a parse. This tree shows explicitly the relationships between symbols in the source program that are only implicitly present in the original text, and can be traversed and rearranged efficiently.

This chapter is about a compiler called `minitree` that accepts the same source language as the `miniloop` compiler described in the previous chapter and which outputs almost identical MVM assembler code, but which uses a tree as an internal data structure. During the parse, the `rdp` generated `minitree` parser automatically constructs the intermediate form, and then a `POST_PARSE` function called `code_generate()` is called which traverses the tree, emitting MVM instructions as it goes. In principle, optimising phases could be inserted between the parse phase and the code generation phase that would rearrange the tree to create more efficient code, although we do not describe such optimisations here.

We strongly recommend that before proceeding with this chapter you read Chapters 9 and 10 of the `rdp` user manual [JS97a] which describe `rdp`'s tree generation facilities in detail.

## 8.1 `minitree` intermediate form

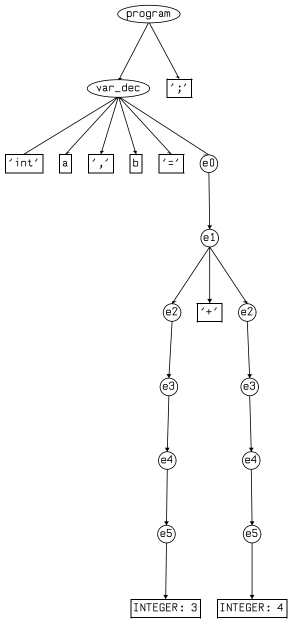
When designing a tree-based compiler, the central decision concerns the information to be retained in the tree after parsing. One extreme option is to simply use the entire derivation tree which contains all the terminals matched as well as a node for every rule instance activated during the parse.

The small programs in Figures 8.1–8.6 exercise all of the major syntactic features of `minitree` including declarations (with and without initialisation); assignment of expressions to variables; print statements; both `if-then` and `if-then-else` statements; a `while-do` statement; and a compound `begin-end` statement. Each program fragment is accompanied by a full derivation tree and the corresponding reduced derivation tree used as an intermediate form by `minitree`.

Full derivation trees for a parse grow rapidly with program length: putting all the program fragments together into a ten-line program yields a tree containing 184 nodes. The tree is mostly broad and flat with long 'catkins' hanging off of some nodes. The catkins are generated by the expression rules: every time an integer or a variable is referenced the parser must recurse right down to the bottom of the expression tree giving rise to these long vertical chains. More than a quarter of the nodes in the derivation tree are of this form, and the proportion would be even higher if the expression tree had more levels (that is, if we had more priority levels in the expressions as we do in the `mvmasm` grammar, for instance). Our reduced derivation trees typically contain only one quarter of the nodes of a full derivation tree and yet the original program may be reconstructed from a reduced derivation tree. In particular, the expression rules no longer generate 'catkins' but are only as deep as they need to be to show the operators actually used in the source expression.

An efficient intermediate form for a compiler should retain all the information needed to reconstruct the original program *but no more*. Text books on compilers often distinguish between *Concrete Syntax Trees* and *Abstract Syn-*

```
1: int a,
2:      b = 3 + 4;
```



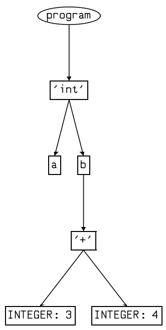
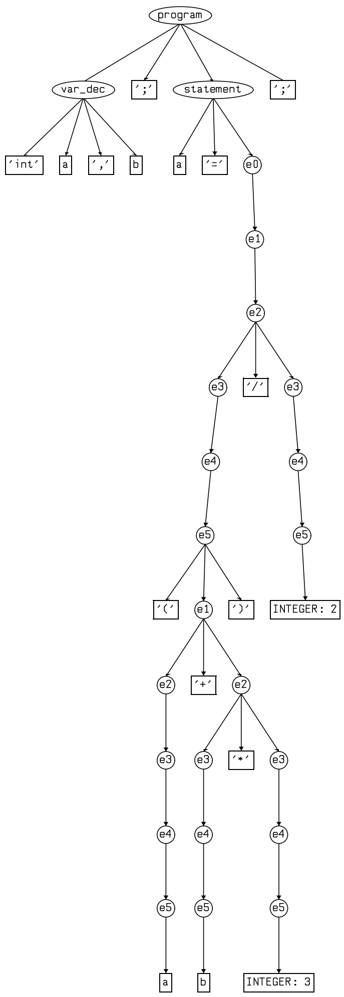




**Figure 8.1** A `minitree` declaration, its full derivation tree and a reduced derivation tree

```
1: int a, b;
2:
3:  a = (a + b * 3) / 2;
```



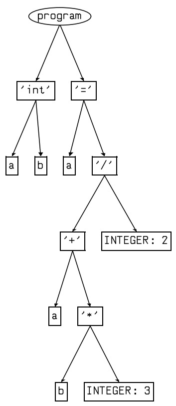




**Figure 8.2** A `minitree` expression, its full derivation tree and a reduced derivation tree

```
int a;

 print("a is ", a, "\n");
```

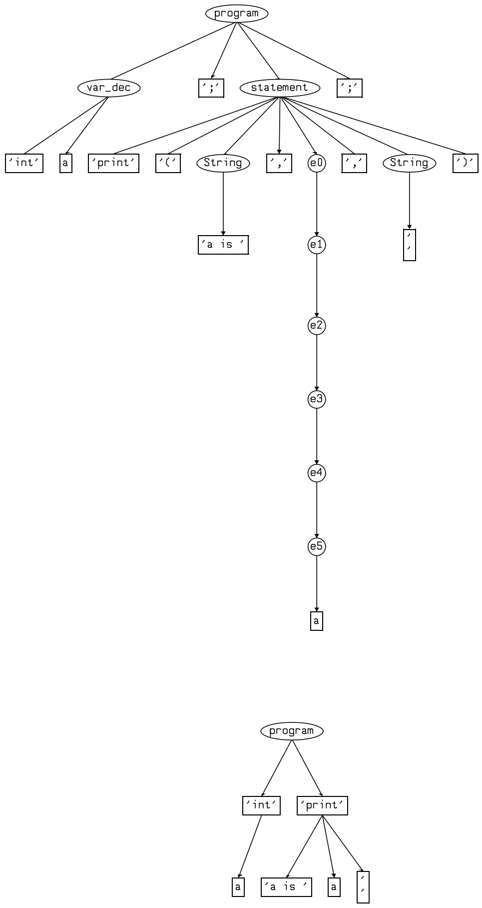



**Figure 8.3** A **minitree print** statement, its full derivation tree and a reduced derivation tree

```
int a,b;

 begin
  a = a + b;
  b = b - 1
 end;
```

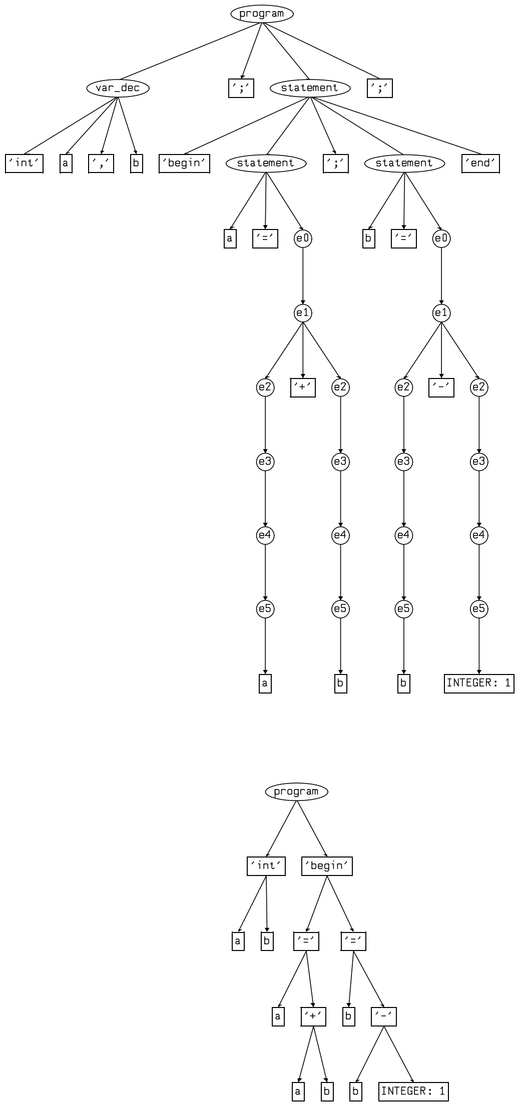



**Figure 8.4** A `minitree` compound statement, its full derivation tree and a reduced derivation tree

```
int a = 1, b = 1;

 if a == 1 then a = 0;

 if a > b then a = 0 else a = 1;
```

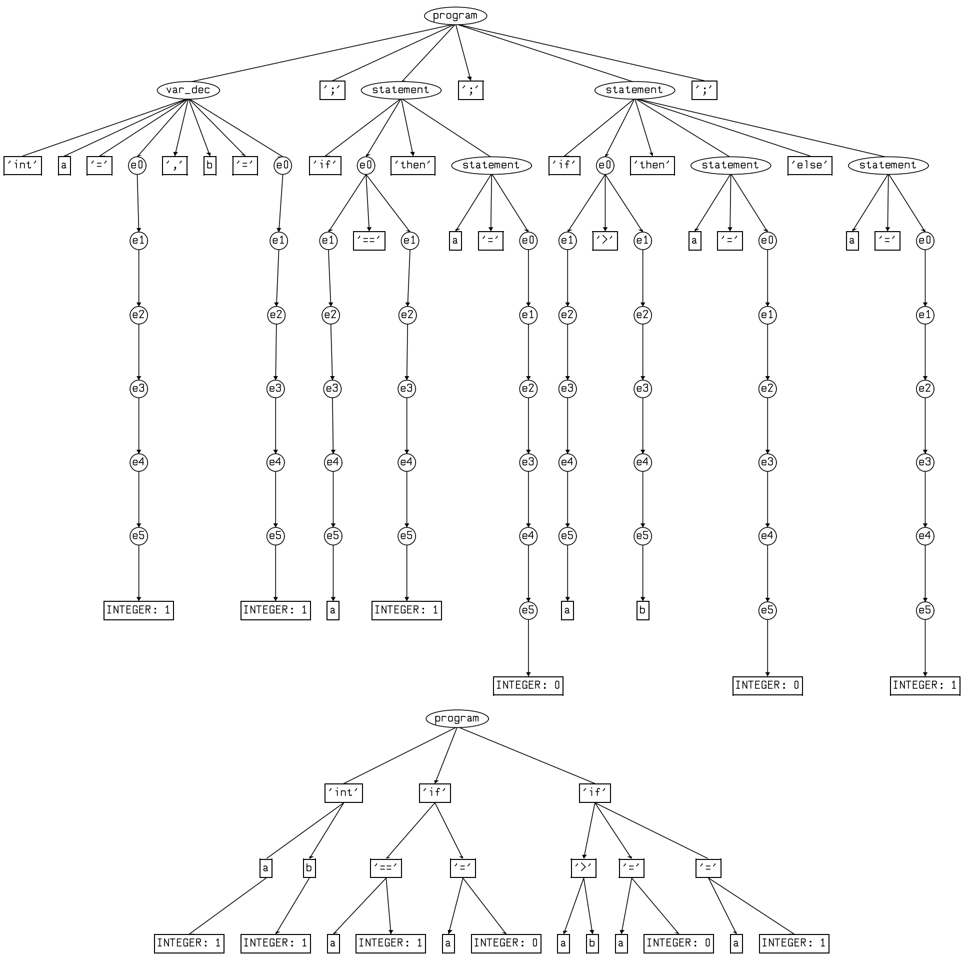



**Figure 8.5** A **minitree if** statement, its full derivation tree and a reduced derivation tree

```
int a = 1, b =10;

 while a < b do
  begin
   print(a);
   a = a + 1
  end;
```

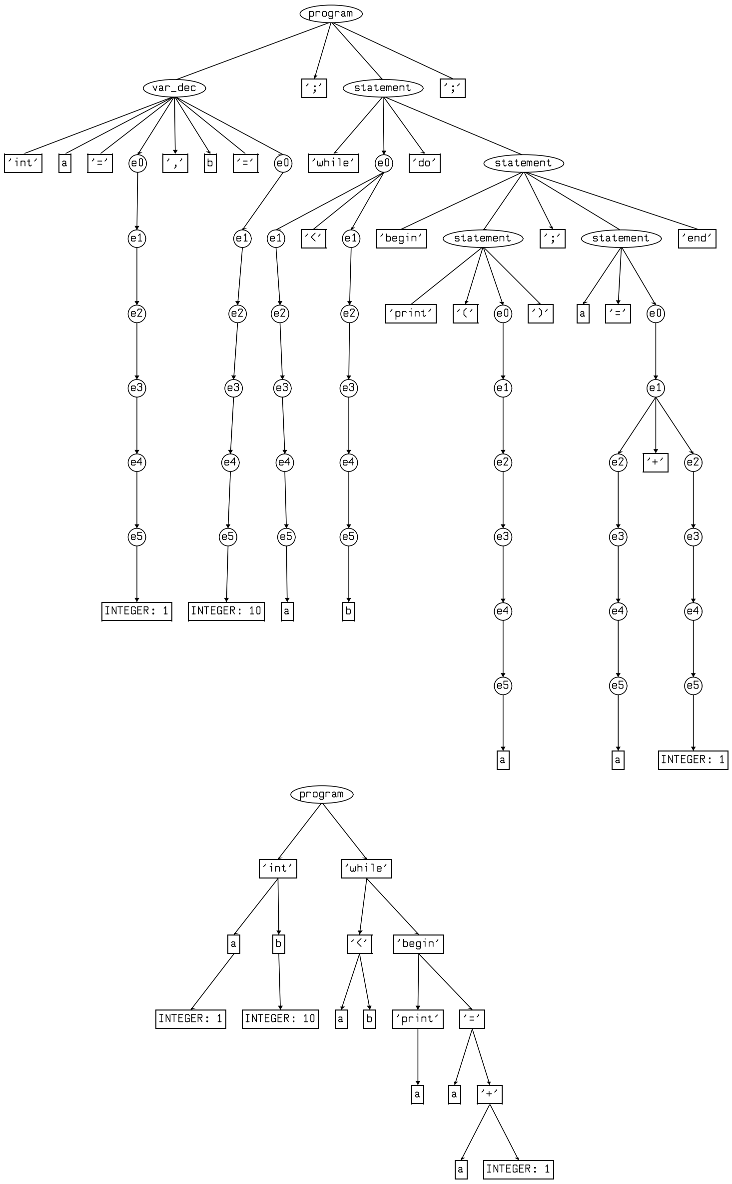



**Figure 8.6** A `minitree` `while` statement, its full derivation tree and a reduced derivation tree

*tax Trees* (AST's). There is little agreement on the formal definition of these objects, but broadly speaking a Concrete Syntax Tree is either a full derivation tree or a *parse tree* made up of just the terminal nodes whilst an AST is usually a tree made up of *some* of the terminals.

The reason for the distinction between concrete and abstract forms is that some terminals in real programming languages are just there to make the program more readable (so-called 'syntactic sugar') and some are only there to represent the two-dimensional nature of programs. The concrete form includes all such terminals but they may be dropped in the abstract forms.

An example of the first case is the parentheses that appear around the conditional expression in the ANSI-C `if () else` and `while () do` statements: it is perfectly straightforward to write an unambiguous grammar that does not include these parentheses and in fact the equivalent Pascal statements do not require them. They are just there to 'please the eye' and may be omitted from the intermediate form.

The second case is represented by the many kinds of brackets used in programming languages including parentheses in arithmetic expressions, the brackets around array index expressions and the `begin end` constructs (or { } in ANSI-C). These brackets are used to show the nesting in a program, but any tree form can show nesting naturally in terms of the parent-child relationships between nodes, so the bracketing terminals are redundant.

The intermediate tree forms used by real compilers tend to be rather *ad hoc*. `rdp` provides a standardised way to build trees by applying promotion operators to nodes within the full derivation tree. The user manual [JS97a] contains examples of standard approaches to common language features and we have applied these to the implementation of `minitree`.

## 8.2 Implementing `minitree`

One of the advantages of a multi-pass implementation scheme is that it allows a clean separation between the grammar and the semantics of code generation. The only semantic actions left in the grammar file `minitree.bnf` are those that use the symbol table to check that all variables encountered have been correctly declared. All of the code generation calls to the various `emit...()` auxiliary functions have been shifted to the tree walker code. `minitree` uses the same auxiliary semantic functions as `miniloop` and so needs to be linked with the functions in `ml_aux.c`. In addition, three extra functions to handle the tree walking are contained in the auxiliary file `mt_aux.c`. We shall look at the grammar first, and then the new auxiliary functions.

### 8.2.1 A grammar for `minitree`

The starting point for the `minitree` grammar is the `miniloop` grammar stripped of its semantic actions apart from those associated with the symbol table. We then add promotion operators to terminals and nonterminals so as to prune the derivation tree into the forms shown in Figures 8.1–8.6. The first task is to remove nodes that are pure syntactic sugar such as the semicolon and comma nodes. These are used to separate items in lists when represented as a linear text, but within a tree we can simply represent the list items as siblings under a parent node. Hence, in line 33 the semicolon node is promoted under its parent, and thus effectively deleted from the tree. Similarly, in line 46, the parentheses in the `print` statement and the comma separating the parameters to be printed are deleted.

The `if` and `while` statements also contain sugar nodes that are deleted: the `if` statement is represented in the tree as a single `if` node with two or three children, the first being the expression tree for the relational condition and the second and third corresponding to the `then` block and the optional `else` block.

The expression tree in lines 49–74 uses the techniques described in the user manual to build operator trees with the usual priority and associativity relationships built into their structure. The promote-above operator (`^^^`) is used to handle the left associative operators and the natural tree ordering ensures that the operator priorities are correctly implemented. We assume that the tree is to be traversed in a depth-first, left-to-right manner so that higher priority operators will appear deeper in the tree.

## 8.3 `minitree` auxiliary functions

`minitree` makes use of the `miniloop` auxiliary functions described previously for handling output to the assembler file and opening and closing the file. You should refer to the previous chapter for a discussion of these code emission functions. The `minitree` auxiliary file `mt_aux.c` shown in Figures 8.9–8.13 contains three extra functions:

1. a top level function (`code_generate()` at lines 188–193) that is called as the `POST_PARSE` function from the grammar,

2. a depth-first, left-to-right tree traversal function that processes expression trees (`expression_walk()` at lines 20–76), and

3. a depth-first, left-to-right tree traversal function that processes statements and calls the expression walker where appropriate (`tree_walk()` at lines 78–186).

The `code_generate()` function is straightforward: it calls the `emit_open()` and `emit_close()` functions used by `miniloop` to initialise and close the assembler output file, and between them the tree walker is called.

### 8.3.1 Use of the graph library

`rdp`'s trees are built using the `graph` library which you can read about in the support library manual [JS97b]. The base of the tree is held in global variable `rdp_tree` which is a pointer to the graph header node in the tree. The first node in the underlying graph is the root of the derivation tree, and the

```
(******************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* minitree.bnf - a mini parser which builds an intermediate form
*
* This file may be freely distributed. Please mail improvements to the author.
*
*******************************************************************************)
TITLE("Minitree compiler V1.50 (c) Adrian Johnstone 1997")
SUFFIX("m")
PARSER(program)
USES("ml_aux.h")
USES("mt_aux.h")
OUTPUT_FILE("minitree.mvm")
TREE
POST_PARSE([* code_generate(rdp_sourcefilename, rdp_outputfilename, rdp_tree); *])

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

check_declared ::= [* if (symbol_lookup_key(mini, &name, NULL) == NULL)
                      {
                        text_message(TEXT_ERROR, "Undeclared variable '%s'\n", name);
                        symbol_insert_key(mini, &name, sizeof(char*), sizeof(mini_data));
                      }
                   *].

program   ::=  { [var_dec | statement] ';'^}.

var_dec ::= 'int'^^ (dec_body)@','^.

dec_body ::= ID:name^^ ['='^ e0 ]:^
             [* symbol_insert_key(mini, &name, sizeof(char*), sizeof(mini_data));
                if (*name == '_' && *(name+1) == '_')
text_message(TEXT_ERROR_ECHO, "variable names must not begin with two underscores\n");
            *].

statement ::= ID:name check_declared '='^^ e0  |                   (* assignment *)
              'if'^^ e0 'then'^ statement [ 'else'^ statement ] |  (* if statement *)
              'while'^^ e0 'do'^ statement |                       (* while do statement *)
              'print'^^ '('^ ( e0  | String )@','^ ')'^ |          (* print statement *)
              'begin'^^ (statement)@';'^ 'end'^.                   (* compound statement *)

```

**Figure 8.7** An rdp BNF specification for minitree part 1: statements

```
e0 ::= e1^^ [ '>'^^^ e1  |  (* Greater than *)
              '<'^^^ e1  |  (* Less than *)
              '>='^^^ e1 |  (* Greater than or equal *)
              '<='^^^ e1 |  (* Less than or equal *)
              '=='^^^ e1 |  (* Equal *)
              '!='^^^ e1    (* Not equal *)
            ] .

e1 ::= e2^^ { '+'^^^ e2  |  (* Add *)
              '-'^^^ e2     (* Subtract *)
          } .

e2 ::= e3^^ { '*'^^^ e3  |  (* Multiply *)
              '/'^^^ e3     (* Divide *)
            } .

e3 ::= e4^^ |
       '+'^ e3  |  (* Posite: note suppression from intermediate form! *)
       '-'^^ e3 .  (* Negate *)


e4 ::= e5 [ '**'^^ e4 ]:^^.

e5 ::= ID:name^^ check_declared | (* Variable access *)
       INTEGER^^  |    (* Numeric literal *)
       '('^ e1^^ ')'^.    (* Parenthesised expression *)

comment ::= COMMENT_NEST('(*' '*)').  (* Comments: stripped by lexer *)
String^  ::= STRING_ESC('"' '\\').     (* Strings for print *)

(* End of minitree.bnf *)
```

**Figure 8.8** An `rdp` BNF specification for `minitree` part 2: expressions

```
/*******************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* mt_aux.c - Minitree multiple pass compiler semantic routines
*
* This file may be freely distributed. Please mail improvements to the author.
*
*******************************************************************************/
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include "graph.h"
#include "memalloc.h"
#include "textio.h"
#include "minitree.h"
#include "ml_aux.h"
#include "mt_aux.h"

```

**Figure 8.9** `minitree` auxiliary functions part 1: declarations

```
char * expression_walk(rdp_tree_data * root)
{
  /* Postorder expression walk */
  if (root->token == SCAN_P_ID)
    return root->id;
  else if (root->token == SCAN_P_INTEGER)
  {
    char * result =(char *) mem_malloc(12);

    sprintf(result, "#%lu", root->data.u);
    return result;
  }
  else
  {
    void * left_edge = graph_get_next_edge(root);
    void * right_edge = graph_get_next_edge(left_edge);

    char * left = expression_walk((rdp_tree_data *) graph_get_edge_target(left_edge));

    if (right_edge == NULL)   /* monadic operator */
    {
      char * dst = new_temporary();

      switch (root->token)
      {
        case RDP_T_26         /* - */ : emit("SUB", "-", dst, "0", left); break;
        default:
        text_message(TEXT_FATAL, "unexpected monadic operator found in expression walk: "
        "token number %i, identifier \'%s\'\n", root->token, root->id);
      }
      return dst;
    }
    else
    {
      char * right = expression_walk((rdp_tree_data *) graph_get_edge_target(right_edge));
      char * dst = new_temporary();

      switch (root->token)
      {
        case RDP_T_17         /* != */ : emit("NE ", "!=", dst, left, right); break;
        case RDP_T_22         /* * */ : emit("MUL", "*", dst, left, right); break;
        case RDP_T_23         /* ** */ : emit("EXP", "**", dst, left, right); break;
        case RDP_T_24         /* + */ : emit("ADD", "+", dst, left, right); break;
        case RDP_T_26         /* - */ : emit("SUB", "-", dst, left, right); break;
        case RDP_T_27         /* / */ : emit("DIV", "/", dst, left, right); break;
        case RDP_T_29         /* < */ : emit("LT ", "<", dst, left, right); break;
        case RDP_T_30         /* <= */ : emit("LE ", "<=", dst, left, right); break;
        case RDP_T_32         /* == */ : emit("EQ ", "==", dst, left, right); break;
        case RDP_T_33         /* > */ : emit("GT ", ">", dst, left, right); break;
        case RDP_T_34         /* >= */ : emit("GE ", ">=", dst, left, right); break;
        default: text_message(TEXT_FATAL, "unexpected diadic operator found in expression walk: "
                              "token number %i, identifier \'%s\'\n", root->token, root->id);
      }
      return dst;
    }
  }
}

```

**Figure 8.10** minitree auxiliary functions part 2: expression walker

```
void tree_walk(rdp_tree_data * root)
{
  /* Preorder tree walk */
  if (root == NULL)
    return;
  else
  {
    void * this_edge = graph_get_next_edge(root);

    switch (root->token)
    {
      case 0:                /* scan root or begin node's children */
      case RDP_T_begin:
      {
        void * this_edge = graph_get_next_edge(root);

        while (this_edge != NULL) /* walk children, printing results */
        {
          tree_walk((rdp_tree_data *) graph_get_edge_target(this_edge));
          this_edge = graph_get_next_edge(this_edge);
        }
        break;
      }

      case RDP_T_31          /* = */ :
      emit("CPY",
      "",
      ((rdp_tree_data *) graph_get_edge_target(this_edge))->id, expression_walk(
      (rdp_tree_data *) graph_get_edge_target(graph_get_next_edge(this_edge))), NULL);
      break;

      case RDP_T_int:
      {
        void * this_edge = graph_get_next_edge(root);

        while (this_edge != NULL) /* walk children, declaring each variable */
        {
          void * child_edge;
          rdp_tree_data * this_node =(rdp_tree_data *) graph_get_edge_target(this_edge);

          emitf(" \n DATA\n%s: WORD 1\n\n CODE\n", this_node->id);
          if ((child_edge = graph_get_next_edge(this_node))!= NULL)
            emit("CPY", "", this_node->id,
          expression_walk((rdp_tree_data *) graph_get_edge_target(child_edge)), NULL);
          this_edge = graph_get_next_edge(this_edge);
        }
        break;
      }

```

**Figure 8.11** `minitree` auxiliary functions part 3: program, assignment and declaration

```
    case RDP_T_print:
    {
      void * this_edge = graph_get_next_edge(root);

      while (this_edge != NULL) /* walk children, printing results */
      {
        rdp_tree_data * this_node =(rdp_tree_data *) graph_get_edge_target(this_edge);

        if (this_node->token == RDP_T_18 /* " */)
          emit_print('S', this_node->id);
        else
          emit_print('I', expression_walk(this_node));

        this_edge = graph_get_next_edge(this_edge);
      }
    }
    break;

    case RDP_T_if:
    {
      char * relation;
      rdp_tree_data
      * rel_stat =(rdp_tree_data *) graph_get_edge_target(this_edge),
      * then_stat =(rdp_tree_data *) graph_get_edge_target(graph_get_next_edge(this_edge)),
      * else_stat =(rdp_tree_data *) graph_get_edge_target(graph_get_next_edge(
      graph_get_next_edge(this_edge)));

      integer label = new_label();
      emitf("__IF_%lu:\n", label);
      relation = expression_walk(rel_stat);
      emitf(" BEQ  %s,__ELSE_%lu\t;ifn %s go to __ELSE_%lu \n", relation, label, relation, label);
      tree_walk(then_stat);
      emitf(" BRA  __FI_%lu\t;go to __FI_%lu\n__ELSE_%lu:\n", label, label, label);
      tree_walk(else_stat);
      emitf("__FI_%lu:\n", label);
      break;
    }

```

**Figure 8.12** minitree auxiliary functions part 4: print and if

```
      case RDP_T_while:
      {
        char * relation;
        rdp_tree_data
        * rel_stat =(rdp_tree_data *) graph_get_edge_target(this_edge),
        * do_stat =(rdp_tree_data *) graph_get_edge_target(graph_get_next_edge(this_edge));

        integer label = new_label();
        emitf("__DO_%lu:\n", label);
        relation = expression_walk(rel_stat);
        emitf(" BEQ  %s,__OD_%lu\t;ifn %s go to __OD_%lu \n", relation, label, relation, label);
        tree_walk(do_stat);
        emitf(" BRA  __DO_%lu\t;go to __DO_%lu\n__OD_%lu:\n", label, label, label);
        break;
      }

      default:
      text_message(TEXT_FATAL, "unexpected tree node found: "
      "token number %i, identifier \'%s\'\n", root->token, root->id);
    }
  }
}

void code_generate(char * source, char * output, void * tree_root)
{
  emit_open(source, output);
  tree_walk((rdp_tree_data *) graph_get_next_node(tree_root));
  emit_close();
}

/* End of mt_aux.c */
```

**Figure 8.13** minitree auxiliary functions part 5: while and POST_PARSE function edges emanating from that node point to the first level children. The function `graph_get_next_edge()` is used to traverse the edge list, and the function `graph_get_edge`target()+ is used to retrieve the node pointed to by a particular edge.

### 8.3.2 The tree walker

The expression walker is recursively called, once for each node in the expression tree. At each call a *subtree* is passed as a parameter, and the function examines the token number of the root node in that subtree. The token numbers comprise either one of the scanner primitives such as `SCAN_P_ID` (the `ID` primitive) or a keyword from the `minitree` grammar such as `RDP_T_17` (the `!=` token). The definitions of the primitives may be found in `rdp_supp/scan.h` and the definitions of the `minitree` tokens in `minitree.h`. The expression walker returns at each level the name of the variable containing the result of the calculation performed at that level in exactly the same way as the expression rules in `miniloop` transmit the names of locations back up the tree.

The leaf nodes in an expression must be either `INTEGER` or `ID` tokens. In these two cases `expression_walk()` simply returns a string corresponding to the lexeme of the token. Any other nodes will be operator nodes, and the expression walker will call their children before emitting an assembler instruction corresponding to the operator. The left child is called at line 37. The right child is then examined in line 39 and if it is `NULL` (empty) then the node must be a monadic operator so assembler code for the monadic operators (only monadic `-` in this case) is emitted *via* the switch statement at lines 43–49. For non-monadic operators, the right child is processed and then the switch statement at lines 57–72 is used to select the assembler instruction corresponding to the operator.

The tree walker has this following outline form:

```
78: void tree_walk(rdp_tree_data * root)
79: {
80:   /* Preorder tree walk */
81:   if (root == NULL)
82:     return;
83:   else
84:   {
85:     void * this_edge = graph_get_next_edge(root);
86:
87:     switch (root->token)
88:     {
89:       case 0:                /* scan root or begin node's children */
90:       case RDP_T_begin:
91:       {
92:         void * this_edge = graph_get_next_edge(root);
93:
94:         while (this_edge != NULL) /* walk children, printing results */
95:         {
96:           tree_walk((rdp_tree_data *) graph_get_edge_target(this_edge));
97:           this_edge = graph_get_next_edge(this_edge);
98:         }
```

```
 99:            break;
100:        }

                case ...:
                .
                .
                .
            }
        }
```

The function is designed to be called recursively and at each level to look at the type of node in the root of the subtree being processed and act accordingly. The first case, shown here, corresponds to a `begin` node or the root node of the reduced derivation tree which has token number 0. `begin` nodes do not of themselves generate any output code but their children are `minitree` statements that must be recursively processed, hence in line 96 we see `tree_walk()` being called on the current node's children.

Five other statements are handled within the tree walker. Assignment (lines 102–107) emits a `CPY` assembler instruction with the first child of the root as the destination operand. The source is obtained by calling the expression walker on the right child of the root.

Declarations are denoted by a sub-tree with an `INT` root node which have one or more child nodes containing the names of the variables to be declared. The `while` loop at lines 113–123 walks the children outputting a `WORD` assembler directive for each variable labeled with the name of that variable. The optional initialisation expression is represented in the tree as an expression sub-tree hanging under the node containing the name of the variable being declared, so if this tree is non-null then the expression walker is called to generate code to evaluate the initialisation expression.

`print` statements are handled in lines 127–143. The `while` loop in lines 131–141 walks the children of the `print` node. If the child is a string (marked with a node type of `RDP_T_18` corresponding to the " token) then `emit_print()` is called to emit a print string instruction. If not, then the expression walker is called on the child and `emit_print()` is called to emit a print integer instruction.

The code to handle `if` and `while` statements is at lines 145–163 and 165–179 respectively. The general format of the code is exactly the same as for the semantic actions in the `miniloop` grammar except that the tree walker is called in lines 158, 160 and 176 to generate the code for the `then`, `else` and `do` blocks.

It would be quite straightforward to integrate the expression and statement walker functions together into a single function. We have separated them for clarity, but the reader may like to consider how to combine them together. Further ideas for projects are given in the final chapter.

# Chapter 9

# A pretty-printer for ANSI-C

A pretty-printer is a tool that rearranges the formatting of a program so as to meet some standard for indentation and comment placement. It turns out that ANSI-C and its embedded preprocessor present some difficult challenges in the design of a pretty-printer which we shall explore in this chapter. The tool described here is called `pretty_c` and you can see some examples of its output in Figures 9.4–9.7.

`rdp` is usually used to specify parsers that describe a language *tightly*, that is the parser should accept inputs that are in the language and reject inputs that are not. It can be very hard to ensure that a parser does have this property, and we know that some aspects of language (such as type checking) are not amenable to specification using just context-free grammars. In these cases we must use semantic checks to increase the checking power of the parser.

For our pretty-printer, we look at a radically different approach to language parsing in which a minimilist `rdp` grammar is constructed that will parse all valid ANSI-C programs as well as a large number of syntactically invalid ones. The rationale here is that an ANSI-C programmer who wishes to use the pretty-printer will also have access to an ANSI-C compiler which *will* be able to detect syntactically invalid programs, so we can reasonably assume that the ANSI-C program presented to the pretty-printer will *already* have been checked for validity. Therefore we can safely use a parser for a superset of the ANSI-C language and not bother to check every detail. This allows us to use a very significantly simplified grammar, but the limitation is that our pretty-printer has to make formatting decisions on the basis of the current input lexeme and its immediate predecessor. The pretty-printer never 'knows' whether it is inside a function definition or processing global definitions, for instance, and as a result it cannot vary formatting according to the kind of construct it is processing.

## 9.1 Using the pretty-printer

The pretty-printer is built during installation of `rdp` as a side effect of running the command `make`. To check whether all is well type

```
pretty_c
```

and you should receive the following help message:

```
Fatal - No source file specified

C pretty-printer V1.50 (c) Adrian Johnstone 1997
Generated on Dec 20 1997 21:55:41 and compiled on Dec 20 1997 at 21:51:16

Usage: pretty_c [options] source[.c]
-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 - not available in this parser)

-i <n> Number of spaces per indent level: 0 means use tabs (default 2)
-c <n> Preferred start column for trailing comments (default 30)
```

These command line options are described below. Now type

```
pretty_c test.c
```

The pretty printer will reformat the file `test.c` (which is part of the standard distribution) and print out

```
test.c,2133,12267,5.75
```

The first field is the name of the file that was formatted, the second is the number of lines in the file (`2133`) and the third is the number of language tokens processed (`12267` in this case). The final field is the average number of tokens per line.

### 9.1.1 Command line options

The pretty-printer provides the normal `rdp`-generated parser command line options along with the following two pretty-printer specific flags.

#### `-i` indent spacing

The default indentation spacing is two spaces. A larger value makes the indentation clearer (whilst making the lines longer) and some standards require the use of tab characters to show indentation. A flag of `-i0` will force `pretty_c` to use one tab character *per* indent. A non-zero value (such as `-i4`) will set the pretty-printer to use that number of spaces *per* indent.

#### `-c` comment start column

The pretty-printer handles comments specially, as will be described in the next section. `pretty_c` attempts to line up comments by moving them across to the comment start column, which is column 30 by default. This flag may be used to change the comment start column.

### 9.1.2 File usage

`pretty_c` is a single pass parser which reads the lexemes in the input file from left to right in the usual way and writes a reformatted version to a temporary file. By default, this file is called `pretty.c` but a different temporary file name can be specified with the `-o` option. At the end of a successful run, `pretty_c` renames the source file to a file with the same name but a filetype of `.bak` and then renames the temporary file to the original source file name. It is a fatal error to try to make the temporary output file the same name as the input file because under some operating systems (such as MS-DOS) the temporary file will overwrite the input file during processing which results in a corrupted file.

### 9.1.3 Making a listing

`pretty_c` can be used to make a line-numbered listing of a program by using the `-l` option. However, bear in mind that it is the input file that will be listed, not the pretty-printed file. If you run the pretty-printer twice on the same file, then a listing generated on the second run will show the formatted file.

### 9.1.4 Error messages

Although `pretty_c` accepts a very loose C grammar it will reject files that contain invalid C lexemes. In such cases `pretty_c` issues the usual syntax error messages. In addition, one of the following three fatal error messages may appear if `pretty_c` has difficulty accessing files.

`temporary output filename is the same as the source filename`

An output file name that is the same as the source file name has been specified. It is a fatal error to try and make the temporary output file the same name as the input file because under some operating systems (such as MS-DOS) the temporary file will overwrite the input file during processing which results in a corrupted file. Use a different output file name.

`unable to open output file` *output filename*

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

`unable to rename` *filename 1* `to` *filename 2*

`pretty_c` was unable to rename the first file to the second file. This may be because there is no disk space left, or there may already exist a file called *filename 2* that is write protected.

## 9.2 Pretty-printer features

The first requirement of a pretty-printer is that it should only modify the spacing of a program and not change its meaning: a pretty-printer is an interesting example of a translator whose input and output language are the same! The particular details of the formatting changes are essentially a matter of taste. A variety of standards exist for C formatting, but there is no universal agreement on how a C program should be indented. We choose to follow the format that `rdp` uses for its machine generated parsers. In detail, `pretty_c` uses the following conventions.

1. Each line of a program has an *indentation level.* The indentation level of the first line of a program is 0.

2. All of the original spacing in the file to be pretty printed is discarded, except for the contents of comments, preprocessor directives and string literals which are preserved.

3. Each output line is preceded by a (possibly zero-length) space, the length of which is proportional to the indentation level. By default, each indentation level is represented by two spaces, but the user can specify *via* the `-i` command line argument, the use of a single tab character or an arbitrary non-zero number of space characters *per* indentation level.

4. Some lexemes are output with a preceding *inter-token space.* Diadic operators such as `>>` or `%`, for instance, are always surrounded by single space characters. In detail, `pretty_c` classifies each language token into one of 16 *kinds* and maintains a $16 \times 16$ array of boolean values that specify whether an ordered pair of language tokens should be separated by a space or not.

5. All line endings are preserved. (Some pretty-printers attempt to ensure that a blank line is inserted after each block of declarations and in some other contexts. `pretty_c` preserves whatever convention for vertical spacing already exists in the file to be formatted: the only changes made are within a line.)

6. An opening brace `{` increases the indentation level by one, and a closing brace `}` decreases the indentation level by one.

7. The keywords `do`, `while`, `for`, `if`, `else` and `switch` are *indenting keywords.* The line after an indenting keyword will have its indentation level increased by one unless it starts with an open brace `{`. Subsequent lines will not be affected and will be indented as they would have been if the indenting keyword had not been encountered.

8. A comment that starts in the first column is never indented.

9. A comment that does not start in the first column but which is the first lexeme on a line is indented using the current indentation level for that line.

10. A comment that is not the first lexeme on a line is reformatted to begin in the *comment start column*, or two columns to the right of the previous lexeme, whichever is the least. By default, the comment start column is column 30, but this may be changed with the `-c` command line option.

## 9.3 Pretty-printer limitations

The conventions listed above are a useful start, but it turns out that there are some sequences of C statements that can result in ugly formatting. In its present form, `pretty_c` is good enough for everyday use (all of the `rdp` source has been formatted using it, for instance) but in this section we note a series of special cases that are handled poorly. In the next chapter we make some suggestions on how to extend the tool to cope with some more esoteric constructions.

### 9.3.1 Operators which may be monadic or diadic

Some language tokens serve a dual rôle. The `*` operator, for instance, is used to denote multiplication, pointer definition and pointer dereferencing. Ideally we should like to produce formatted output is the following form:

```
char *str;
int a;

a = *str * 4;
```

Since `pretty_c` only ever examines the token to be formatted and its immediate predecessor it is hard to distinguish between the monadic and diadic uses of `*`. In the present pretty-printer, `*` is always treated as a diadic operator with a space on both sides, resulting in output of the form

```
char * str;
int a;

a = * str * 4;
```

### 9.3.2 Consecutive indenting keywords

The convention for indenting keywords is that they should cause a temporary indentation of the following line. This is inadequate for the case of a sequence of indenting keywords on neighbouring lines. For instance, this piece of code

```
if (x != 0)
  do
    y += 3;
  while (y < x);
```

will be reformatted as

```
if (x != 0)
  do
  y += 3;
while (y < x);
```

This is because temporary indentations do not accumulate.

### 9.3.3 Continuation lines

Occasionally, a long expression or function call will be broken over several lines, with significant horizontal formatting. `pretty_c` does not preserve this formatting. Consider

```
x = 4 +
    long_function_call(first,
                       second,
                       third
                      );
```

Ideally we would like `pretty_c` to recognise that the open parenthesis marks the start of a new indentation level, but in fact `pretty_c` will simply reformat this as

```
x = 4 +
long_function_call(first,
second,
third
);
```

### 9.3.4 Embedded comments

A comment which is not the first lexeme on a line will be moved to the comment start column, if possible. This is undesirable if the comment is intended to be embedded within a line.

```
x = func(3 /* parameter width */, 45, 67);
```

will be reformatted as

```
x = func(3                    /* parameter width */, 45, 67);
```

### 9.3.5 Formatting of lexemes

In one place, `pretty_c` does not even follow its own conventions: a string or character literal containing an octal escape sequence such as

`'\03'` or `An embedded control \012 character`

will be output with the numerical escape sequence reformatted to use hexadecimal notation, as in

`'\X03'` or `An embedded control \X0A character`

This minor unpleasantness arises from a limitation of the `rdp` scanner which only returns the binary version of a string or character literal.

## 9.4 A grammar for a superset of ANSI-C

Our aim with the grammar for `pretty_c` is to accept all valid ANSI-C programs, but we are not limited to accepting *only* valid ANSI-C. The outline form of the grammar is:

`program ::=` { *any valid ANSI-C lexeme* }.

This will be a string of zero or more ANSI-C lexemes in whatever order, including sequences that are not syntactically correct ANSI-C, so for instance `pretty_c` would accept a 'program' of the form

```
int main(void)
{
  else 3 do case 16:
}
```

which should certainly be rejected by any real C compiler.

The full `pretty_c` grammar specification is shown in Figures 9.1 and 9.2. The top level rule `program` accepts zero or more matches against one of 16 subrules that between them generate the complete ANSI-C lexicon. Each of the 16 subrules defines a particular kind of token, and each kind has different spacing conventions — all of the diadic operators, for instance, are defined in rule `diadic`. Rule `program` receives (in the synthesized attribute `lexeme`) the string of characters matched by the scanner, and a local attribute `kind` is assigned one member of the *kind* enumeration which is defined in the auxiliary file `pr_c_aux.h` shown in Figure 9.3. Rule `program` also includes calls to the auxiliary functions `pretty_open()` and `pretty_close()` that control the file handling.

There are two cases where the string returned in `lexeme` is not necessarily the actual string matched by the scanner. In the case of both character and string literals (defined in lines 82 and 84) the scanner will process embedded escape sequences to produce a string which may contain binary characters. This is the source of the restriction noted in section 9.3.5 in which octal escape sequences will be rewritten as hexadecimal escape sequences on output: the scanner does not preserve information on whether a particular escape sequence was octal or hexadecimal so we have arbitrarily decided to output them all as hexadecimal.

Comments in `rdp` generated parsers are usually defined using one of the 'invisible' comment scanner primitives and quietly suppressed by the scanner. In this application, of course, we wish to pass comments from the parser into the pretty printer (otherwise the comments would be removed from the formatted output!) As a result, comments are defined in line 56 using the `COMMENT_VISIBLE` primitive.

Preprocessor directives in ANSI-C present the pretty printer with particular problems. Unusually for a high level language, spacing is critical in preprocessor definitions. These two commands have quite different meanings:

```
#define a(b) b=3;

#define a (b) b=3;
```

```
(*******************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* pretty_c.bnf - a pretty-printer for ANSI-C
*
* This file may be freely distributed. Please mail improvements to the author.
*
* This grammar illustrates a rather different approach to writing language
* parsers. Instead of trying to exactly define the language with the
* grammar we try and find a simple grammar that accepts the language, and
* also allow it to accept lots of incorrect strings. The rationale is that
* a pretty-printer does not need to check a program for syntax errors
* because a conventional compiler will be used subsequently to do that.
* As a result we end up with a very flat, loose grammar
*
*******************************************************************************)
TITLE("C pretty-printer V1.50 (c) Adrian Johnstone 1997")
SUFFIX("c")
PARSER(program)
OUTPUT_FILE("pretty.c")
TEXT_SIZE(100_000)
USES("pr_c_aux.h")

ARG_NUMERIC(i indent_size "Number of spaces per indent level: 0 means use tabs (default 2)")
ARG_NUMERIC(c comment_start "Preferred start column for trailing comments (default 30)")

program ::= [* enum kinds kind;
               long unsigned line, column;
               pretty_open(rdp_sourcefilename, rdp_outputfilename);
            *]
            {
              [* line = scan_line_number(); column = scan_column_number(); *]
              (
                comment: lexeme        [* kind = K_COMMENT;        *] |
                string: lexeme         [* kind = K_STRING;         *] |
                character: lexeme      [* kind = K_CHARACTER;      *] |
                block_open: lexeme     [* kind = K_BLOCK_OPEN;     *] |
                block_close: lexeme    [* kind = K_BLOCK_CLOSE;    *] |
                preprocessor: lexeme   [* kind = K_PREPROCESSOR;   *] |
                monadic: lexeme        [* kind = K_MONADIC;        *] |
                diadic: lexeme         [* kind = K_DIADIC;         *] |
                open_bracket: lexeme   [* kind = K_OPEN_BRACKET;   *] |
                close_bracket: lexeme  [* kind = K_CLOSE_BRACKET;  *] |
                item: lexeme           [* kind = K_ITEM;           *] |
                field_delim: lexeme    [* kind = K_FIELD_DELIM;    *] |
                punctuation: lexeme    [* kind = K_PUNCTUATION;    *] |
                keyword: lexeme        [* kind = K_KEYWORD;        *] |
                keyword_indent: lexeme [* kind = K_KEYWORD_INDENT; *] |
                EOLN: lexeme           [* kind = K_EOLN;           *]
              )
              [* pretty_print(lexeme, kind, column, line); *]
            }
            [* pretty_close(rdp_sourcefilename, rdp_outputfilename); *].

```

**Figure 9.1** `rdp` grammar for pretty-printer: part 1

```
comment: char* ::= COMMENT_VISIBLE('/*' '*/'):result.

preprocessor: char* ::= COMMENT_LINE_VISIBLE('#'):result.

monadic: char* ::= '!':result | '++':result | '--':result | '~':result .

diadic: char* ::= '&&':result | '&':result | '^':result | '|':result |
                  '||':result | '%':result | '*':result | '/':result |
                  '+':result | '-':result | '<<':result | '>>':result |
                  '<':result | '<=':result | '==':result | '>':result |
                  '>=':result | '?':result | '!=':result | '%=':result |
                  '&=':result | '*=':result | '+=':result | '-=':result |
                  '/=':result | '=':result | '^=':result | '|=':result |
                  '<<=':result | '>>=':result | '\\':result.

block_open: char* ::= '{':result.

block_close: char* ::= '}':result.

open_bracket: char* ::= '(':result | '[':result.

close_bracket: char* ::= ')':result | ']':result.

item: char* ::= ([* result = SCAN_CAST->id; *] (INTEGER | REAL)) |
                ID:result | '...':result .

string: char* ::= STRING_ESC('"''\\'):result.

character:char* ::= STRING_ESC('\''''\\'): result.

field_delim: char* ::= '->':result | '.':result.

punctuation: char* ::= ':':result | ';':result | ',': result.

keyword: char* ::= 'auto':result | 'break':result | 'case':result |
                   'char':result | 'const':result | 'continue':result |
                   'default':result | 'double':result | 'enum':result |
                   'extern':result | 'float':result | 'goto':result |
                   'int':result | 'long':result | 'register':result |
                   'return':result | 'short':result | 'signed':result |
                   'sizeof':result | 'static':result | 'struct':result |
                   'union':result | 'unsigned':result | 'void':result |
                   'volatile':result.

keyword_indent: char* ::= 'do':result | 'else':result | 'for':result |
                          'if':result | 'switch':result | 'while':result.
```

**Figure 9.2** rdp grammar for pretty-printer: part 2

The first defines a macro called `a` with a parameter `b` which has 3 assigned to it in the body of the macro. The second defines a parameterless macro `a` which expands to the string `(b) b=3;`. The absence or presence of the space between the macro name and the opening parenthesis is used to decided whether a macro has parameters or not. This is an immediate problem for the `rdp` scanner because spaces are discarded and must be reconstructed from the token stream. It *would* be possible to do this by keeping track of the column numbers for the tokens immediately after a `#define` token, but fortunately for us there is a simpler solution. In the C preprocessor, no line endings are allowed within preprocessor directives. As a result we can make use of the `rdp` scanner's `COMMENT_LINE_VISIBLE` primitive to define a 'comment' that opens with the token `#` and closes with the line end. This will cause the complete preprocessor directive to be handled in the parser as a single monolithic unit, just like a comment. In this way the spacing is preserved. Of course, a side effect of this is that preprocessor lines will never be 'prettified', but given the subtleties of parsing preprocessor directives this conservative design decision is perhaps justified.

The lexeme and its associated `kind` value are passed to the auxiliary function `pretty_print()` in line 52 along with the line and column numbers for the token. This function will be described in the next section: we simply note here that the lexeme will be printed (possibly with a preceding space) to the output file, and that line end tokens will be followed by a string of spaces corresponding to the indentation level.

## 9.5 Auxiliary routines

The auxiliary functions and the *kind* enumeration are defined in the auxiliary header file `pr_c_aux.h` shown in Figure 9.3. The two externally visible variables, `indent_size` and `comment_start` receive the values of the `-i` and `-c` command line arguments. The *kind* enumeration has 17 values: the first 16 correspond to the 16 subrules in the `pretty_c` grammar and the last one `K_TOP` is a dummy value that is set to the number of subrules.

The source code for the three auxiliary functions is shown in Figures 9.4–9.7. The data declarations are in Figure 9.4 and include variables to keep count of the number of line endings seen, the number of lexemes seen and the number of comments. We also remember the value of the last reported line number. This may be different to the number of line endings seen because a comment that spans a line ending will be parsed as a single comment lexeme, and so some line endings may be hidden.

The file handling routines `pretty_open()` and `pretty_close()` are shown in Figure 9.5 and are straightforward.

### 9.5.1 The space array

The $16 \times 16$ array of booleans `space_array` is used by the pretty printer to decide whether a space should precede the current lexeme before it is output.

```
/*******************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* pr_c_aux.h - pretty-printer semantic routines
*
* This file may be freely distributed. Please mail improvements to the author.
*
*******************************************************************************/
enum kinds
{
  K_BLOCK_CLOSE, K_BLOCK_OPEN, K_CHARACTER, K_CLOSE_BRACKET, K_COMMENT,
  K_DIADIC, K_EOLN, K_FIELD_DELIM, K_KEYWORD, K_KEYWORD_INDENT, K_ITEM,
  K_MONADIC, K_OPEN_BRACKET, K_PREPROCESSOR, K_PUNCTUATION, K_STRING, K_TOP
};

extern unsigned long indent_size;
extern unsigned long comment_start;

void pretty_close(char * sourcefilename, char * outputfilename);
void pretty_open(char * sourcefilename, char * outputfilename);
void pretty_print(char * lexeme, enum kinds kind, unsigned long column, unsigned long line);

/* End of pr_c_aux.h */
```

**Figure 9.3** Pretty-printer auxiliary functions: header file

The `pretty_print()` function remembers in a static variable the *token kind* of last lexeme seen, so at each stage it has access to the token kinds of the previous and current lexemes. Line 129 uses the space array to check whether a space should be output:

```
129:  if (space_table[last_kind][kind]) /* insert space if necessary */
130:    printed += fprintf(outputfile, " ");
```

Here we see that if there is a one at position (last, current) of the space array then a preceding space will be output. Line 54, for instance, specifies that a space shall always be output if the last kind was punctuation (such as a comma or semicolon). This effectively inserts a space after every punctuation character.

This lookup table mechanism is very flexible and almost sufficiently powerful but it does suffer from some limitations. In particular, it is not easy to decide whether a `*` token is a diadic multiplication or a monadic pointer dereference operator, especially in contexts such as `mytype *temp;` where `mytype` is a user defined type definition that has been created using a `typedef` statement. To be able to handle such cases we would need to keep track of all `typedef` statements which would require a much more detailed grammar. In fact, we would have to implement a complete C preprocessor to perform this task perfectly because it is conceivable that the user defined type `mytype` had been defined in a macro or in an included file. `pretty_c` simply ignores these complications and always treats `*` (and for that matter `&`) as a diadic operator with spaces on both sides. This is the source of the restrictions described in section 9.3.1.

```
/*****************************************************************************
*
* RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 20 December 1997
*
* pr_c_aux.c - pretty printer semantic routines
*
* This file may be freely distributed. Please mail improvements to the author.
*
*****************************************************************************/
#include <stdio.h>
#include "scan.h"
#include "textio.h"
#include "pr_c_aux.h"

static int lexeme_count = 0;
static int eoln_count = 0;
static int comment_count = 0;
static int last_line = 1;
static FILE * outputfile;
unsigned long indent_size = 2l;
unsigned long comment_start = 30l;


static int space_table[K_TOP][K_TOP]= {
/*                                               K
                               C                 E
                               L                 Y        O  P
                      B        O           F     W        P  R  P
                      L  B     S           I     O        E  E  U
                      O  L  C  E           E     R        N  P  N
                      C  O  H  _           L     D        _  R  C
                      K  C  A  B  C        D  K  _     M  B  O  T
                      _  K  R  R  O  D     _  E  I     O  R  C  U  S
                      C  _  A  A  M  I     D  Y  N     N  A  E  A  T
                      L  O  C  C  M  A  E  E  W  D  I  A  C  S  T  R
                      O  P  T  K  E  D  O  L  O  E  T  D  K  S  I  I
                      S  E  E  E  N  I  L  I  R  N  E  I  E  O  O  N
                      E  N  R  T  T  C  N  M  D  T  M  C  T  R  N  G
                      ---------------------------------------------- */
/* BLOCK_CLOSE    */ {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
/* BLOCK_OPEN     */ {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
/* CHARACTER      */ {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
/* CLOSE_BRACKET  */ {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0},
/* COMMENT        */ {1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1},
/* DIADIC         */ {1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1},
/* EOLN           */ {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
/* FIELD_DELIM    */ {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
/* KEYWORD        */ {0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0},
/* KEYWORD_INDENT */ {0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0},
/* ITEM           */ {0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1},
/* MONADIC        */ {1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1},
/* OPEN_BRACKET   */ {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
/* PREPROCSSOR    */ {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1},
/* PUNCTUATION    */ {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
/* STRING         */ {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1}
};

```

**Figure 9.4** Pretty-printer auxiliary functions: part 1

```
void pretty_open(char * sourcefilename, char * outputfilename)
{
  if (strcmp(sourcefilename, outputfilename)== 0)
    text_message(TEXT_FATAL, "temporary output filename is the same as the source filename");

  if (* outputfilename == '-')
    outputfile = stdout;
  else if ((outputfile = fopen(outputfilename, "w"))== NULL)
    text_message(TEXT_FATAL, "unable to open output file \'%s\'", outputfilename);
}

void pretty_close(char * sourcefilename, char * outputfilename)
{
  unsigned long useful_lexeme_count = lexeme_count - comment_count - eoln_count;
  char * backup_filename = text_force_filetype(sourcefilename, "bak");

  fclose(outputfile);

  remove(backup_filename);

  if (rename(sourcefilename, backup_filename)!= 0)
    text_message(TEXT_FATAL, "unable to rename \'%s\' to \'%s\'\n", sourcefilename, backup_filename);

  if (rename(outputfilename, sourcefilename)!= 0)
    text_message(TEXT_FATAL, "unable to rename \'%s\' to \'%s\'\n", outputfilename, sourcefilename);

  text_printf("%s,%lu,%lu,%.2lf\n", sourcefilename,
  last_line,
  useful_lexeme_count,
  (double) useful_lexeme_count /(double) last_line);
}

```

**Figure 9.5** Pretty-printer auxiliary functions: part 2

```
void pretty_print(char * lexeme, enum kinds kind, unsigned long column, unsigned long line)
{
  static int last_kind = K_EOLN;
  static int indentation = 0;
  static int temporary_indent = 0;
  static int printed = 0;

  lexeme_count++;             /* bump lexeme counter for statistics */
  last_line = line;           /* remember the highest line number seen */

  if (kind == K_BLOCK_CLOSE)
    indentation--;
  else if (last_kind == K_BLOCK_OPEN)
    indentation++;

  if (last_kind == K_EOLN)    /* do indentation */
  {
    int indent_count, space_count;

    if (temporary_indent && kind != K_BLOCK_OPEN) /* add an indent of we aren't opening a block */
      indentation++;

    for (indent_count = 0; indent_count < indentation; indent_count++)
      if (!((column == 1)&&(kind == K_COMMENT))) /* Don't indent comments that start in column 1 */
      if (indent_size == 0)
    {
      fprintf(outputfile, "\t");  /* indent using a tab */
      printed += text_get_tab_width();
    }
    else
      for (space_count = 0; space_count < indent_size; space_count++)
      printed += fprintf(outputfile, " ");

    if (temporary_indent && kind != K_BLOCK_OPEN) /* reset temporary indent */
      indentation--;

    temporary_indent = 0;
  }

  if (space_table[last_kind][kind]) /* insert space if necessary */
    printed += fprintf(outputfile, " ");

```

**Figure 9.6** Pretty-printer auxiliary functions: part 3

```
  /* Print the lexeme: some kinds need special actions */
  switch (kind)
  {
    case K_EOLN:
    fprintf(outputfile, "\n");
    eoln_count++;
    printed = 0;
    break;

    case K_COMMENT:
    comment_count++;
    if (last_kind != K_EOLN)  /* comments that aren't first on a line move to middle */
      do
      printed += fprintf(outputfile, " ");
    while (printed < comment_start);

    printed += fprintf(outputfile, "/*%s*/", lexeme);
    break;

    case K_STRING:
    printed += fprintf(outputfile, "\"");
    printed += text_print_C_string_file(outputfile, lexeme);
    printed += fprintf(outputfile, "\"");
    break;

    case K_CHARACTER:
    printed += fprintf(outputfile, "\'");
    printed += text_print_C_char_file(outputfile, lexeme);
    printed += fprintf(outputfile, "\'");
    break;

    case K_PREPROCESSOR:
    printed += fprintf(outputfile, "#%s", lexeme);
    break;

    default:
    printed += fprintf(outputfile, "%s", lexeme);
    break;
  }

  if (kind == K_KEYWORD_INDENT) /* Set an indent for next line */
    temporary_indent = 1;

  last_kind = kind;
}
/* End of pr_c_aux.c */
```

**Figure 9.7** Pretty-printer auxiliary functions: part 4

### 9.5.2 The pretty-print function

The pretty-print function is called after each lexeme read by the parser. It is defined in Figures 9.6 and 9.7, lines 90–176. The internal state of the pretty printer is maintained between calls in four static integer variables:

1. `last_kind` contains the token kind of the lexeme processed on the previous call to `pretty_print()`,

2. `indentation` contains the current indentation level,

3. `temporary_indent` is a boolean flag that is set after an indenting keyword such as `if` or `while` is seen, and

4. `printed` contains the number of characters output since the last new line character, or (equivalently) the current output column number.

In lines 97 and 98 the global variables `lexeme_count` and `last_line` are updated. These variables are used within function `pretty_close` to output the number of lines, number of lexemes and the average lexeme *per* line count at the end of a run.

The indentation of lines is performed by the code in lines 105-127, which is only executed if the last token seen was a new line so as to ensure that indentation is only performed at the beginning of a line. Lines 109-110 temporarily increment the indentation level if the `temporary_indent` flag is set and we are not processing a new block. Any temporary increment is reset in lines 123–124.

Line 113 detects comments that start in column 1 and suppresses their indentation. Lines 114–121 output tab characters or groups of spaces according to the value of the `indent_size` variable that is set using the `-i` command line option. In each case, the `printed` variable is updated to show the column number after printing. The routine `text_get_tab_width()` is used to get the current value of the tab setting as set using the `-t` command line option. Incidentally, note the ugly layout of this code which is a manifestation of the problem described in section 9.3.2.

The space table is accessed in lines 129–130 to control the output of a space character before the current lexeme is printed. You may dislike the spacing convention used here (some people like a space after an opening parenthesis and a space before a closing parenthesis, for instance) in which case you should experiment with modifications to the space table.

The lexemes are printed out under the control of the `switch` statement at lines 133–170. Most token kinds receive the `default` treatment of being simply printed out. However the following token kinds require special treatment:

1. `K_EOLN` must be output as a newline character, and the `eoln_count` and `printed` variables must be updated at the same time,

2. `K_COMMENT` lexemes do not contain the delimiting `/*` and `*/` brackets so these must be reinstated on output, and the comment must be placed as near to the comment start column as possible,

3. K_STRING and K_CHARACTER lexemes do not contain the delimiting quote marks so these must be reinstated on output, and

4. K_PREPROCESSOR lexemes do not contain the delimiting # token, so this must be inserted before output of the body of the preprocessor command.

The final actions of the pretty-print function are to set a temporary indent if an indentable keyword has been output and to update the last_kind variable ready for the next invocation.

# Chapter 10

# Design projects

In this chapter we list suggestions for enhancements to the `mini` languages described in earlier chapters that might reasonably be undertaken as exercises, as well as a larger project to build a subset C compiler.

1. Add block definition to `minicond` (hint: follow the syntax in `miniloop.bnf`).
2. Add left and right shift operators to `minicalc` and its descendent languages.
3. Add logical operators to `minicalc` and its descendent languages.
4. Add real arithmetic `minicalc` and its descendent languages.
5. Add a switch statement `minicond` and `miniloop`.
6. Add a for loop to `miniloop`.
7. Add a goto statement to `miniloop`.
8. Add function definition and call to `miniloop`.
9. Implement common mode subexpression elimination for `minitree`.
10. Add registers to MVM and a simple register allocator to `minitree`.
11. Add a graph colouring register allocator to `minitree`.
12. Add conditional assembly to `mvmasm`.
13. Add macro definition and call to `mvmasm`.
14. Implement a subset C compiler with `rdp`.

    This is an ambitious project which could build into a complete compiler for C targeted at a virtual machine of the MVM form. The suggestions here form a coherent path through the tasks but an experienced language implementor would probably coalesce some of the intermediate stages together.

(a) Define a language for the target machine. The MVM assembly language used in this manual is suitable as a basic language and has the advantage that an assembler and a simulator already exist for that language. An extremely ambitious choice would be to use the language of a real processor, although this is only recommended for readers that are very familiar with programming the chosen processor.

(b) Define a tree-like intermediate data structure to represent the result of parsing the source program. The student can either decide to build this structure using actions embedded in the parser specification or use the automatic tree building capability of `rdp`.

(c) Define a subset of C. A simple subset might correspond to a version of C that only allows integer operations, has no pointers, no user-defined types and no capability to define functions. Control structures might also be restricted: a simple `if-then-else` statement and a `while-do` statement would suffice in the first instance.

(d) Write an `rdp` specification that parses the chosen subset language and test it against a set of test examples illustrating both correct and incorrect usage.

(e) Enhance the `rdp` specification using either explicit semantic actions or the `rdp` tree operators to build the intermediate form.

(f) Write a `POST_PARSE` function that traverses the intermediate form emitting instructions for the target processor.

(g) Demonstrate correct compilation and execution of test programs using simulation or by direct execution on the target architecture.

(h) Add a full complement of C control structures, including `switch`, `break`, `goto` and `for`.

(i) Add support for floating point arithmetic.

(j) Add support for function definition.

(k) Add support for user defined type definition.

(l) Add support for pointers.

(m) Implement common subexpression elimination.

(n) Implement register allocation using graph colouring.

# 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


[JS97a] 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.

[JS97b] Adrian Johnstone and Elizabeth Scott. `rdp_supp` – support routines for the `rdp` compiler compiler version 1.5. Technical Report TR-97-26, Royal Holloway, University of London, Computer Science Department, December 1997.

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

[PD82a] S Pemberton and M C Daniels. *Pascal implementation: compiler and assembler/interpreter.* Ellis Horwood, 1982.

[PD82b] S Pemberton and M C Daniels. *Pascal implementation: the P4 compiler.* Ellis Horwood, 1982.