New Jersey Machine-Code Toolkit
Reference Manual
Version 0.1

Norman Ramsey Mary Fernandez

October 18, 1994

Contents

1 Introduction **5**
- 1.1 Multiple-language support . . . 5
- 1.2 Caveat user . . . 6
- 1.3 Known bugs . . . 6

2 The specification language **9**
- 2.1 Tokens and fields . . . 9
- 2.2 Patterns . . . 10
  - 2.2.1 Simple patterns . . . 11
  - 2.2.2 Combining patterns . . . 12
  - 2.2.3 Internal form of patterns . . . 12
  - 2.2.4 Pattern matching and the meanings of the operators . . . 13
  - 2.2.5 Pattern shapes and the ellipsis . . . 13
- 2.3 Pattern bindings . . . 13
- 2.4 Constructors . . . 14
- 2.5 Equations and expressions . . . 17
- 2.6 Miscellaneous . . . 18
  - 2.6.1 Addressing units for the program counter . . . 18
  - 2.6.2 Placeholders . . . 18
  - 2.6.3 Word size . . . 18
  - 2.6.4 Discarding constructors . . . 19
- 2.7 Lexical details . . . 19

3 Decoding applications, matching statements **21**
- 3.1 Matching statements . . . 22
- 3.2 Access to instruction streams during decoding . . . 23

4 Encoding applications and library **25**
- 4.1 Building an encoding application . . . 25
- 4.2 Labels and relocatable blocks . . . 26
  - 4.2.1 Operations on relocatable blocks . . . 28
  - 4.2.2 Operations on labels . . . 29
- 4.3 Emitting information . . . 30
- 4.4 Manipulating the location counter . . . 31

4.5 Relocation and closures 31
4.6 Application's obligations 33
4.7 Accessing contents of relocatable blocks 34
4.8 Relocatable Addresses 34
4.9 Excerpts from `mld`, an encoding application 35
4.9.1 Using encoding procedures 35
4.9.2 Using relocatable blocks 36
4.9.3 Using relocatable addresses 37
4.9.4 Relocation 38
4.9.5 Example makefile 38

5 The generator and translator **41**
5.1 E: Encoding options 41
5.2 D: Decoding options 44
5.3 d: Options for debugging decoders 45
5.4 V: Verbosity options 46
5.5 A: Analysis options 46
5.6 Example execution of `tools` 47
5.6.1 Running the generator 47
5.6.2 Running the translator 48

A Vaporware **49**
A.1 Dictionaries 49
A.2 Pickling and Unpickling 50
A.3 C language support for pickles 52

References **55**

Chapter 1

Introduction

This is the reference manual for version 0.1 of the New Jersey Machine-Code toolkit. New users should read our overview (Ramsey and Fernandez 1995) and study our annotated descriptions of real machines (Ramsey and Fernandez 1994) before tackling this manual. Here you will find:

Chap. 1	Promises, cautions, and warnings.
Chap. 2	A grammar for the specification language, with a detailed description of each construct.
Chap. 3	How to write decoding applications. Describes matching statements.
Chap. 4	How to write encoding applications. Describes the toolkit library, which provides support for emitting data into streams and relocating instructions.
Chap. 5	How to run the generator and translator. Describes all their command-line options.

1.1 Multiple-language support

The toolkit currently has rudimentary support for multiple languages. For decoding applications, this support actually works; we use it both for C and for Modula-3. The support for encoding applications is only partly in place; there are lots of little C-isms lurking around the declarations of encoding procedures and the results of typed constructors, and there is only one version of the toolkit's library, which is written in C. We would be pleased to try to support users interested in other languages. We would probably ask such users to develop the library support, since that is a fairly straightforward and separable task. We would tackle the details of making the generator truly support multiple languages.

1.2 Caveat user

This is a preliminary, prototype reference manual, and it documents preliminary, prototype software. Up to this point, we've had no users other than ourselves, so we're sure both the manual and the code are chock-full of errors. We can't justify more effort on the toolkit or its documentation unless we have users. But who will use the toolkit if it's not robust and well-documented? But we can't make it robust or well-documented unless we have users! But... you get the idea.

We can do this: *if you have a problem with the toolkit, or you don't understand its documentation, we'll make it right.* Send email to `mff@princeton.edu` and `norman@bellcore.com`.

1.3 Known bugs

We already know about these bugs in the toolkit. Pester us if you want them fixed.

- If the placeholder `_` is used multiple times on the left of a matching statement, each instance should be distinct, but at the moment the instances all stand for the same variable.
- The toolkit generates lots of dead code in decoders. For efficiency, we need to assume the compiler does dead-code elimination and copy propagation.
- If you write a constraint in which the right-hand side is not an integer literal, the toolkit may create a field binding even if you didn't write `=`. For example, the constraint `offset < n` may create a field binding equating `offset` to `n`.
- Field-width constraints don't show up in closures.
- The toolkit should choose alternatives based on width conditions, but it doesn't.
- Generated decoders assume that input streams go on forever—they have no way of coping with (or even indicating!) end of stream.
- We don't implement the obvious optimization that avoids creating closures for branches into the current relocatable block.[1]
- The equation solver is shaky, particularly around bit slices. You should be OK if you make sure your slices partition the bit space—don't allow overlaps or gaps.
- We don't solve the MIPS `li` constructor on decoding. We're not sure why yet.

[1] OK, a missing feature is not a bug. But it should be in there.

- The toolkit dies on the first error, and our error messages are terrible. In particular, all messages are treated using the syntax-error mechanism, which means that semantic errors point to the end of the specification file in the most misleading way. Using the `-impossible` option sometimes gives enough information to help track down the real source of an error.

  Real improvement in error handling will probably await re-implementation in a language better suited to the problem (like one that has exceptions!).

Chapter 2

The specification language

This chapter describes the language that the toolkit uses to describe the encodings of machine instructions. We write a grammar in which *nonterminals* appear in italics, *terminals* appear slanted, and `keywords` and literal text appear in typewriter font. Terminals ending in *-name* are identifiers. Terminals with no special names are integer literals unless otherwise stated.

To describe the specification language, we use an EBNF grammar. We use the standard EBNF metasymbols for

$$\{\text{sequences}\},\ [\text{optional constructs}],\ \text{and}\ (\text{alternative} \mid \text{choices}).$$

We use large metasymbols to make them easy to distinguish from literals.

The start symbol for the grammar describing a specification is *Specification*. Definitions must appear before uses, but otherwise the parts of a specification may appear in any order. The grammar for a complete specification is therefore

Specification ⇒ {*spec*}

2.1 Tokens and fields

The specification

spec ⇒ `fields of` *token-name* `(` *width* `)` *field-specs*

field-specs ⇒ {*field-name low-bit*:*high-bit*}

defines a token named *token-name* that is *width* bits wide, as well as all the fields that may be used to refer to parts of that token. Bits are numbered from 0, with 0 denoting the least significant bit.

A `fieldinfo` specification gives auxiliary information about a field or list of fields. (It is useful to describe several fields simultaneously when, for example, they all refer to the same set of registers.) Such auxiliary information might include special names used to refer to values of the fields (as when they describe

data formats or registers) or information about what kind of checking should be done when an application supplies a putative value for a field.

The syntax of `fieldinfo` specifications is:

spec ⇒ `fieldinfo` {*fieldinfo-spec*}

fieldinfo-spec ⇒ *field-specifier* `is [` {*field-item*} `]`

field-specifier ⇒ *field-name* | `[` {*field-name*} `]`

field-item ⇒ `checked`
| `unchecked`
| `guaranteed`
| `sparse [` *Binding* { `,` *Binding*} `]`
| `names [` {*Ident* | *String*} `]`

binding ⇒ (*Ident* | *String*) `=` *integer*

The terms `checked`, `unchecked`, and `guaranteed` denote three possible levels of checking in generated encoding procedures:

- If f is a checked field, the encoding procedure checks to make sure its value falls within range. If not, it calls an error handler.
- If f is an unchecked field, the encoding procedure does not check its value, but it does mask out high bits, ensuring that other parts of the token can't be corrupted by a bad value of f.
- If f is a guaranteed field, the encoding procedure simply uses its value without any checking or masking at all.[1] This option is useful mainly for situations in which f denotes a register, and the register allocator promises never to use a bogus register.

`sparse` and `names` are both ways of specifying names of fields. `names` is used when all the names are known; `sparse` is used otherwise. Identifiers can be used to specify names on most targets, but names containing special characters can only be specified using string literals. Note that the toolkit accepts the C literal syntax for characters in place of integer literals; this property may be useful when writing `sparse` bindings. The use of commas to separate lists of bindings is not consistent with our other representations of lists, but they look funny without the commas.

2.2 Patterns

Patterns can be defined both in terms of themselves or in terms of simpler elements. We discuss the base cases of the recursion first.

[1] Signed field parameters must always be masked, but most field parameters are unsigned.

2.2.1 Simple patterns

Simple patterns can be described by any of the following syntaxes:

pattern ⇒	*name*	*name* of pattern, field, or constructor type
	| *opcode* `(` *arguments* `)`	Constructor application
	| *field-binding*	Binds field to variable
	| *constraint*	Constrains field
	| `epsilon`	The empty sequence.
	| `some` *token-name*	Matches a single token of the class named.
	| `[` {name} `]`	List of patterns

field-binding ⇒ *field-name* `=` *variable-name*

constraint ⇒ *field-name relational-operator* (*integer* | *generating-expression*)

relational-operator ⇒ `<` | `<=` | `=` | `!=` | `>` | `>=`

Unless a list of pattern names or a generating expression is used, the pattern denotes a single pattern; otherwise it denotes a list. Lists are ordinarily used to describe opcode tables. In a *field-binding*, the variable may be an operand or it may be computed by the solver.[2]

When an identifier is used as a pattern, its meaning depends on context. Pattern names defined with the `patterns` pattern-binding statement can always be used in later patterns. On the right-hand side of a constructor definition, the names of operands that represent fields or constructor types may also be used as patterns. The name of a field operand `f` is short for

`f` = *the value of the operand* `f`.

Similarly, the name of a constructor-type operand `T` is short for its value, which must be a pattern because the result of applying a constructor is always a pattern.

The *opcode* used in a constructor application may denote a single constructor or a group of constructors with identical operands. It works the same way in a constructor specification, as described in Section 2.4. Strictly speaking, applying a constructor to create a pattern is not "simple," because patterns themselves are used to define constructors. It is actually part of a mutual recursion between the definitions of patterns and constructors.

A *generating expression* can be used instead of an integer in field constraints. The resulting constraint denotes a list of patterns, one for each value of the generating expression. There are three forms of generating expression:

generating-expression ⇒	`{` *lo* `to` *hi* `}`
	| `{` *lo* `to` *hi* `columns` *n* `}`
	| `[` {*integer*} `]`

The first form generates the integers from *lo* to *hi* inclusive, in order. The second form generates the same integers, but in an order suited to an *n*-column layout, e.g. $lo, lo+n, lo+2n, \ldots$, wrapping around as needed. The third form simply lists the integers generated; it is suitable for sparse tables.

[2] We would like to arrange for fields to be bound to expressions, not just variables, but we're not there yet.

2.2.2 Combining patterns

The recursive part of the definition of patterns makes it possible to combine patterns using the following operators:

pattern ⇒ *pattern* ...	Loosens rules restricting conjunction.
| ... *pattern*	Loosens rules restricting conjunction.
| *pattern* `&` *pattern*	Conjunction.
| *pattern* `|` *pattern*	Disjunction.
| *pattern* `;` *pattern*	Sequence.

Operators listed first have higher precedence.

2.2.3 Internal form of patterns

Patterns constrain both the way in which streams are divided into tokens and the values of the fields in those tokens. Internally, patterns are composed from *range constraints* and *field bindings*. A range constraint limits the acceptable values of a field to a range known at compile time. Often this range contains only a single element, e.g., as fixed by the constraint `op = 1`. A field binding forces a field to have a particular value, but that value may not be known at compile time—in general, it depends on the values of operands passed to a constructor. The binding `offset = (target - $pc - 4)[2:17]` is an example.

The toolkit represents range constraints in the form $lo \leq f < hi$. The toolkit uses its knowledge of field widths to map relational expressions into range constraints. For example, `f = 9` maps into $9 \leq f < 10$, and `f > 3` maps into $4 \leq f < 2^{width(f)}$.

The toolkit represents field bindings in the form $f = expression$. Expressions are computed by the toolkit's equation solver. Depending on whether an application does encoding or decoding, the solver may compute field values as a function of constructor operands or vice versa.

Range constraints and field bindings form the basis for patterns, which are represented internally in "disjunctive" normal form. Patterns are disjunctions. Each disjunct is a sequence. Each sequent applies to tokens of a particular class, and it is a (possibly empty) collection of constraints of fields in that class.[3] Many interesting patterns are single disjuncts; for example, `epsilon` is the empty sequence, and `some` *class-name* is a sequence with one element (sequent), that element containing no constraints. The pattern `f = 0` is a sequence with one element, that element containing the single range constraint $0 \leq f < 1$. The inequality operator is an exception to this happy rule. In general, an inequality can't be represented as a single range constraint, so `f != n` is syntactic sugar for a disjunction:

$$0 \leq \mathtt{f} < \mathtt{n} \mid \mathtt{n} + 1 \leq \mathtt{f} < 2^{width(\mathtt{f})},$$

Since a disjunction almost never makes sense when encoding, using inequality constraints is asking for trouble. We get away with it in a few cases where $n = 0$, in which case one of the disjuncts is vacuous and is discarded.

[3] By "constraint" we mean either a range constraint or a field binding.

2.2.4 Pattern matching and the meanings of the operators

The basic patterns and pattern operators can be described in terms of their effect on the normal form, but it is easier to consider how they affect matching. The constraint $lo \le$ `f` $< hi$ matches an input token if the `f` field of that token falls in the range defined by *lo* and *hi*. `f` is declared as a field of a particular class of tokens, which determines the size of the token matched. The wild-card constraint "`some` *class*" matches any token of class *class*, for example, on the SPARC, "`some itoken`" matches any 32-bit token. A conjunction "`p & q`" matches if both `p` and `q` match. A concatenation "`p; q`" matches if `p` matches an initial sequence of input tokens and `q` matches the following tokens. A disjunction "`p | q`" matches if either `p` or `q` matches.

2.2.5 Pattern shapes and the ellipsis

Concatenation and conjunction distribute over disjunction, so it is usually convenient to talk about operations on disjuncts (sequences). Each disjunct (sequence) has a unique *shape*, which is obtained by taking the class of each element of the sequence. So, for example, on the Intel 486, a 32-bit displacement addressing mode defined by the pattern

```
mod = 2 & r_m != 4; i32 = d
```
[4]

has the shape `modr_m; I32`.

The toolkit uses shapes to restrict conjunctions; two patterns may be conjoined if and only if they have the same shape. For example, on the 486, the pattern `mod = 0 & r_m = 5` is permitted, but the pattern `mod = 0 & index = 2` is not. The latter conjunction attempts to combine patterns of shape `modr_m` and `sib`.

The ellipsis operator provides a way to loosen this restriction. "`p ...`" creates a pattern that is equivalent to `p`, except it is OK to write `p ... & q` whenever `p`'s shape is a prefix of `q`'s shape. Examples can be found in the description of the Intel 486 by Ramsey and Fernandez (1994). The ellipsis may also be used as a prefix operator on patterns, in which case `... p & q` is permissible whenever `p`'s shape is a suffix of `q`'s shape. We haven't had occasion to use the prefix ellipsis in machine descriptions, because most hardware decodes complex instructions from left to right.

Be careful to surround the ellipsis operator with whitespace. The dot is just like a letter for purposes of lexical analysis, so

```
p...
```

is a four-character identifier name, not the pattern `p` followed by the ellipsis.

2.3 Pattern bindings

Patterns can be named using pattern-binding declarations:

[4] The `!=` relational operator is a cheat. Don't use it.

spec ⇒ `patterns` {*pattern-binding*}

pattern-binding ⇒ *pattern-name* `is` *pattern*
| `[` {*pattern-name*} `]` `is` *pattern*
| *pattern-name* `is any of [` {*pattern-name*} `], which is` *pattern*

The first form names a single pattern.

The second, most common form binds a list of names on the left to a list of patterns on the right. The number of names must match the number of patterns; the special name "_" can be used to match uninteresting patterns on the right. This form requires the use of generating expressions (or an explicit list of patterns) to produce a list of patterns. It is typically used to create opcode tables. In this situation, the name "_" simply indicates an unused opcode.

The third form extends the second to handle a common idiom. The sequence of pattern names and the pattern on the far right act just as they do in the second form, but in addition, the single name on the left is bound to the disjunction of all the patterns on the right that are bound to names other than "_". Examples can be found in the SPARC and 486 specifications given by Ramsey and Fernandez (1994).

The name to which a pattern is bound remains associated with that pattern, and that name may be used in the formation of opcode names. When a disjunction is bound, the individual disjuncts retain their names, if any. This behavior can be exploited when writing constructor specifications, in which disjuncts are enumerated when patterns appear in opcodes.

2.4 Constructors

The constructor specifications are the most complicated in the toolkit; they can use compound opcodes, equations, assembly-language syntax, and conditional assembly. Constructor specifications are the only ones in which newlines are significant. We make newlines significant so that we can use them in the very common case in which the output pattern of the constructor is implicit.

The syntax of constructor specifications is

spec ⇒ `constructors` {*constructor*}

constructor ⇒ *opcode* {*operand*} [: *type-name*] ⋆ [*branches*]

Because both type and branches are optional, we use the newline to separate the last operand of the current constructor from the opcode of the following constructor. The newline is significant only in the position marked ⋆, and only the first newline is significant. It is OK to include the newline even when explicit branches are present.

Constructors specified by the toolkit are *typed* or *untyped*, depending on whether a *type-name* appears. All constructors are applied to operands to produce patterns, but the encoding procedures that the toolkit generates for a constructor have different effects depending on typing. Untyped constructors

emit tokens into the current instruction stream, but typed constructors produce opaque values encapsulating patterns. The values are opaque in the sense that their only use is to be passed as operands to other constructors.[5] A new constructor type is defined simply by using a new *type-name* in a specification. All constructors of a type must be defined before the type is used, for example, as an operand to another constructor. This restriction prevents circular definitions.[6]

The opcode and operand notations used in constructor specifications are designed to enable concise specifications that resemble assembly language. As noted above, the constructor specification begins with opcode and operands. The list of operands is terminated by the first newline, or by any of the tokens that introduce later constructs.[7]

Opcodes are formed from three kinds of units: literal strings, pattern names, and field names. Literal strings simply contribute to the opcode name, but patterns and fields introduce an implicit iteration: patterns iterate over disjuncts, and fields iterate over named values. Each iteration results in the specification of a separate constructor. For a pattern, the name of the disjunct is substituted for the pattern name in the opcode, and the disjunct is used to stand for the pattern on the right-hand side. For a field, the name of the field's value[8] is substituted for the the field name on the opcode, and the pattern binding the field to that value is used to stand for the field on the right-hand side. Any pattern so introduced on the right-hand side is conjoined into the *implicit output pattern*, which is the one used in the constructor specification when branches are omitted. For examples, see Ramsey and Fernandez (1995). The basic idea is for any pattern or field on the left to be re-used on the right — typically it will be conjoined in, in which case it is used implicitly.

The elements of an opcode are joined with a hat (^). The syntax for an opcode is

$$\textit{opcode} \Rightarrow \textit{opname} \left\{ \hat{} \ \textit{opname} \right\}$$

$$\textit{opname} \Rightarrow \textit{string} \mid \textit{pattern-name} \mid \textit{field-name}$$

Most opcodes use just a single name. It's permissible to use an unbound identifier as an *opname*, which has the same effect as a literal string.[9]

[5] We recognize that this restriction is a defect in the toolkit, and that application writers might like to match on or manipulate the opaque values produced by typed constructors. We welcome suggestions about how such a facility should be designed, implemented, or made available to applications.

[6] It also prevents recursive constructor types. We avoid recursive constructors for two reasons. For efficiency, the toolkit's generator "inlines" constructor operands by enumerating all possible patterns, a technique that works only when the enumeration is finite. When decoding, the toolkit can use a simple recognition algorithm instead of having to create a parser, which might be ambiguous.

[7] Computers are better than I at computing follow sets, but certainly the colon that introduces a constructor type, the brace that introduces equations, and the keyword `is` all terminate lists of operands.

[8] As given by `names` or `sparse` in a `fieldinfo` declaration.

[9] This is hokey language design, but it makes the specifications look nice. It's unlikely to lead to errors, because a misspelling, which would result in an unbound identifier, would also

Our operand syntax is unusual in that we permit arbitrary "noise words" to enable the specification writer to mimic assembly-language syntax. (We have an ulterior motive in that we hope one day to generate disassmblers, and maybe even assemblers, from this syntax.) We call such noise words *literal*s. For encoding and decoding, literals are ignored completely. We permit any integer or quoted string, plus a variety of special characters that are common in assembly languages but not otherwise used in our specifications:

literal ⇒ *string* | *integer*
| any of these characters: `[]()+*/&@#$%,;|`

The real operands of a constructor are all given by name. Constructor operands may have several different types, but we don't want to clutter up our notation with type information, so we use special names shamefully:

operand ⇒ *field-name* [!]	Field, possibly signed
| *constructor-type-name*	Result of applying typed constructor
| `reloc`	Relocatable address
| *unbound-name*	Integer operand
| *literal*	Ignored for encoding and decoding

Both `reloc` and constructor inputs may have underscores and digits appended; they can be used to distinguish multiple inputs of the same type. Field and constructor inputs are available to be used as output patterns; fields stand for the pattern binding that field to the value of the operands, and constructor inputs stand for themselves—the result of applying a constructor is always a pattern. Relocatable addresses and integers can be used in equations, in field constraints, or both.

The toolkit always treats field values as unsigned. When a field is actually signed, specification writers must use an exclamation point as an explicit sign-extension operator. Using the exclamation point after a field name in an operand list indicates that the parameter to an encoding procedure is a signed integer, which should be narrowed to produce the value of the field.

The branches given with a constructor determine its output pattern. If branches are omitted, all the field names, pattern names, and constructor-type names in the opcode and operands are conjoined to produce an *implicit* output pattern. We use this idiom heavily in our MIPS and SPARC descriptions.

Most commonly, each constructor produces a unique sequence of tokens when applied, so its right-hand side is specified by one pattern containing a single disjunct:

branches ⇒ [`{` *equations* `}`] `is` *pattern*

The equations (Section 2.5) are solved to produce a set of *conditions*, which are predicates sufficient to guarantee that the equations have a solution, and

result in that identifier's being unbound on the right-hand side, which would be flagged as an error. (We don't expect users to write specifications in which pattern or field names are used on the left without also being used on the right, and we issue a warning message if they do.)

to produce *bindings* for the variables solved for. When encoding, the field- and integer-valued operands to the constructor provide the inputs to the equation solver; when decoding, it is the variables bound to fields in *pattern* which perform that function.

Sometimes we want to choose an output pattern based on some properties of the operands (conditional assembly). In that case, we can give multiple branches, each with its own set of equations.

branches ⇒ {`when {` *equations* `} is` *pattern*} [`otherwise is` *pattern*]

The sequence of `when` branches may not be empty.

When using this constructor in encoding, the toolkit chooses the first branch whose conditions are known to be satisfied. This conservative approach means that the toolkit can always produce correct code in a single pass, but that, for example, branches to unknown labels are always encoded in the most general way. It would be pleasant if we got around to incorporating standard multi-pass algorithms for conditional assembly, but don't look for it soon.

2.5 Equations and expressions

Equations are written by relating two expressions using the relational operators. Multiple equations are separated by commas.

equations ⇒ *equation* {`,` *equation*}
equation ⇒ *expr relational-operator expr*

Only the `=` operator truly contributes information to a set of equations to be solved. All inequalities in a list of equations are simply added to the list of *conditions* that must be satisfied if the equations are to have a solution.

The special identifier `$pc` may be used in the equations of a constructor specification. When so used, it stands for the address at which the tokens of that constructor begin.[10] During encoding, it is a relocatable address, which can be treated as an integer by the usual mechanism. During decoding, it has to be converted to an integer by a special template (see Section 3.2).

The expression syntax uses the standard binary operators, plus special postfix syntax for bit slicing and sign extension:

expr ⇒ *integer*
| *identifier* [*bitrange*] [`!`]
| *expr binary-operator expr*
| `(` *expr* `)`

binary-operator ⇒ `+` | `-` | `*` | `/`

bitrange ⇒ `[` *lo-bit* [`:` *hi-bit*] `]`

[10] This would be the address of the first token, except that a constructor doesn't have to include any tokens at all.

The binary operators have the standard precedence. Using explicit division is almost certain to cause the solver to complain. Multiplying two unknowns will also break the solver—you should only multiply by integer literals.

A bit slice denotes the range of bits from *lo-bit* to *hi-bit*, inclusive. Bit 0 is the least significant bit. If *hi-bit* is omitted, it defaults to *lo-bit*, that is, the slice is one bit wide.[11] Any quantity may be sliced. Only slices and fields may be sign-extended, because they are the only quantities whose widths are known.

The toolkit's equation solver is very simple-minded, and it gives terrible error messages.

2.6 Miscellaneous

2.6.1 Addressing units for the program counter

The specification

spec ⇒ `pc_unit_bits` *width*

says that the program counter measures units of *width* bits. For example, if `pc_unit_bits` is 8 (the default), a 16-bit token in a sequence causes the program counter to be advanced by 2. The size of any token that appears in a sequence must be exactly divisible by `pc_unit_bits`.

2.6.2 Placeholders

The specification

spec ⇒ `placeholder for` *token-name* `is` *pattern*

Tells the toolkit's generator to use *pattern* when it needs to emit a placeholder for the named token. Placeholders are emitted whenever a relocatable address is needed but not yet known.

2.6.3 Word size

There are a few situations in which we use the word size of the machine for which we are generating code—mostly to see whether certain values will fit. It defaults to 32 bits, but it can be specified by

spec ⇒ `wordsize` *width*

where *width* is the width in bits.

[11] It is a defect in the toolkit that there is no way to notate a slice from bit k up to the most significant bit, whatever that may be (e.g., 31 or 64). If you want this defect remedied, suggest a suitable notation.

2.6.4 Discarding constructors

You may want to delete constructors from a specification, reducing the number of encoding procedures generated. You may say what to keep or what to throw away.

spec ⇒ `keep` {*opcode*}
| `discard` {*opcode*}

Constructors (or groups thereof) are named by their opcode specifications.

2.7 Lexical details

The lexical structure used in specifications is similar to that of C, except that dots as well as underscores are treated as letters for purposes of finding identifiers. Strings are delimited with double quotes, but the C "backslash escapes" aren't used—instead, a double quote within a string is represented by two consecutive double quotes, as in Pascal. The `0x` notation is available to give integer literals in hexadecimal. Integer literals may also be specified as C-style character constants. For reasons that make no sense whatever, these constants *do* use the C "backslash escapes."

The toolkit reserves the following words as keywords:

```
...           discard    fields   otherwise     to
_             else       for      patterns      type
address       endmatch   is       pc_unit_bits  using
any           epsilon    keep     placeholder   when
columns       fetch      names    some          which
constructors  fieldinfo  of       sparse        wordsize
```

These words cannot be used to name constructors, fields, patterns, etc.

Chapter 3

Decoding applications and matching statements

Decoding applications are written using ordinary C or Modula-3 with embedded *matching statements*. The matching statements are like case statements, except that the arms are labelled with patterns, not with values. The toolkit's translator transforms the matching statement into ordinary code, which is then compiled as usual. Supposing that the file `decode.m` is part of a decoding application and contains one or more matching statements, this transformation can be specified in a Makefile as follows:

```
decode.c: decode.m
        tools -decoder decode.c -matcher decode.m m.spec i.spec
```

where `m.spec` contains a machine description and `i.spec` contains instruction-stream templates (see Section 3.2). Typically the `m.spec` file can be shared among a variety of applications, but different applications may use different representations of instruction streams, and therefore different `i.spec` files.

The toolkit works by computing a *decision tree* for each matching statement. Each internal node of the tree tests a particular field of a token from the input stream. No field is tested more than once, and no token is fetched more than once. The leaves of the tree contain single arms of the matching statement.[1] At each leaf, the toolkit solves equations to compute the values of free variables in the matching pattern; the inputs to these equations are the values of the fields tested. The toolkit turns the tree into a dag by combining identical leaves; this process can eliminate unnecessary tests. Finally, the toolkit generates C or Modula-3 code from the dag.

The rest of this chapter explains the syntax of matching statements and the way to specify the representation of instruction streams using templates.

[1] If two arms match the same pattern, but with different equations, they are both assigned to the same leaf, and the equations have to be tested to distinguish them.

3.1 Matching statements

The toolkit's matching statements are inspired by pattern matching in Standard ML (Milner, Tofte, and Harper 1990). Matching statements are embedded directly in C or Modula-3 code. A matching statement begins with

`match` [`[` *pointer-name* `]`] *code* `to`

on a line by itself. The *code* is an expression that computes a value of type "address," as specified by the `address type` template. This address points into an instruction stream at the point where decoding is done. The optional *pointer-name* names another address variable, which is set to point to the location following the match. The body of a matching statement contains a series of arms, and the matching statement ends with `endmatch` on a line by itself.

The first line of an arm begins with a `|` and has the following syntax:

`|` *pattern* [`{` *equations* `}`] [`[` *name* `]`] `=>` *code*

Free variables of *pattern* or *equations* are available to be used in the *code* on the right-hand side. Free variables from *pattern* can come from field constraints or from constructor applications; in either case, all such variables represent binding instances. The special variable `_` can be used in constructor applications to stand for operands whose values are uninteresting. The optional *name*, if present, is an identifier that is bound to the name of the *pattern*. This name may be a name introduced by a `patterns` declaration, or it may be the name of a constructor that was applied to produce the pattern. The *code* may occupy multiple lines; the toolkit takes all the text between the `=>` and the next initial `|`, or the end of the matching statement.

The last arm of a matching statement may be written

`else` *code*

which is syntactic sugar for

`|` `epsilon` `=>` *code*

Indulging in wild abuse of notation, we could summarize the matching statement as follows:

`match` [`[` *pointer-name* `]`] *code* `to`
{ `|` *pattern* [`{` *equations* `}`] [`[` *name* `]`] `=>` *code* }
[`else` *code*]
`endmatch`

The decoder generated by the toolkit has the effect of checking each arm of the matching statement in turn. As soon as it finds an arm whose pattern matches and whose equations have a solution,[2] it binds all the free identifiers

[2] Note that patterns obtained by constructor application may carry along some equations implicitly.

on the left-hand side of the arm, then executes the code on the right-hand side. The decoders the toolkit generates are actually much more efficient, testing inputs against all arms at once, but the sequential model accurately describes the semantics.

3.2 Access to instruction streams during decoding

Decoders generated by the toolkit's translator instantiate code templates to get access to the tokens of the instruction stream being decoded. The toolkit treats addresses and instruction streams as abstract data types; these templates give the operations.

spec ⇒ `fetch` (*width* | `any`) `using` *template*
| `address type is` *template*
| `address add using` *template*
| `address to pc using` *template*

Each *template* is a string. `fetch` templates are used to fetch words from an instruction stream. Some applications may be able to give a single `fetch` templates that works with any width; others may have to give a different `fetch` template for each width used in the specification. `address type` identifies the type the application uses to represent a location in an instruction stream; the `address add` template is used for address arithmetic, which moves through the stream. `address to pc` is used to convert an address to an integer; it is used to assign a value to `$pc`. In the templates, `%a` stands for an address, `%o` for an offset, and `%w` for a width. Offsets are measured in increments of `pc_unit_bits`, but widths are measured in bits. `fetch` templates may use `%a` and `%w`. The `address add` template may use `%a` and `%o`. The `address to pc` template may use only `%a`. The `address type` template doesn't use any escapes.

Chapter 4

Encoding applications and the toolkit's library

Encoding applications generate instructions in binary form by calling two kinds of toolkit procedures: machine-dependent encoding procedures that are generated from a machine description, and machine-independent procedures that are part of the toolkit's library. These procedures require some support from the application, for example, to allocate space for the relocatable blocks into which instructions are emitted. This support is explained in Section 4.6.

This chapter specifies the library functions that are used by the toolkit-generated encoding procedures and by the application program. The library includes support for instruction and data streams, in the form of relocatable blocks, and for reading and writing the streams. If the application writer chooses to use his own implementation of streams, the implementation must conform to the specifications given here.

The library is written in C, we specify it by writing ANSI C prototypes for its procedures. Some of the library "procedures" are actually implemented as macros; for such macros, we provide prototypes that describe the equivalent procedures.

4.1 Building an encoding application

The toolkit generator transforms a machine description `m.spec` into a `noweb` file (Ramsey 1994) that contains the encoding procedures and their interfaces. The following rules for a Makefile specify how to create and use encoding procedures. The first rule emits encoding procedures into the file `encoder.nw`.

```
encoder.nw: m.spec
        tools -encoder encoder.nw m.spec
```

The encoding procedures and their interfaces are extracted from `encoder.nw` using `notangle`. The encoding procedures are obtained from the default root

chunk named `*`; the interfaces are obtained from the root chunk named `header`. The `-L` option specifies that line number directives will appear in the output so that any errors can be cross-referenced to the source in `encoder.nw`.

```
encoder.c: encoder.nw
        notangle -L encoder.nw >encoder.c
encoder.h: encoder.nw
        notangle -L -Rheader encoder.nw >encoder.h
```

Application source that includes calls to the encoding procedures must include the encoding procedures' header file as well as `mclib.h`, the header file for the toolkit library.

```
app.o: app.c encoder.h mclib.h
        cc -c app.c
```

Users may have to add an `-I` option so the C preprocessor can find `mclib.h`.

The application executable must include the encoding procedures and the library itself. The last three rules compile the encoding procedures and library source and link the complete application.

```
encoder.o: encoder.c encoder.h mclib.h
        cc -c encoder.c
mclib.o: mclib.c mclib.h
        cc -c mclib.c
app: app.o encoder.o mclib.o
        cc  app.o encoder.o mclib.o -o app
```

The rest of this chapter describes the types, functions, and macros that appear in `mclib.h`.

4.2 Labels and relocatable blocks

A *relocatable block* is the smallest unit of information relocatable by the library and is used to represent a stream of instructions or data. Relocatable blocks can represent information of various types. A traditional linker, for example, might associate one relocatable block with each section in an object file. An incremental linker might associate one relocatable block with each procedure, which would facilitate relocation of individual procedures at link time.

A relocatable block `rb` has the following fields:

`rb.lc`	the current location counter (offset) in `rb`.
`rb.size`	the size of the relocatable block.
`rb.low`	the lowest value of `rb.lc` for which the contents of `rb` are known, or null if nothing is known of the contents of `rb`.
`rb.address`	the absolute address of `rb` if that address is known, otherwise null.
`rb.contents`	a buffer storing the contents of `rb`. Clients may refer to the elements `0..rb.size-1` of `rb.contents`. The values of `rb.contents[0..rb.low-1]` are undefined, and it is illegal to attempt to write these values. (This restriction enables an implementation that handles completely uninitialized blocks without allocating any memory.)
`rb.label`	an unnamed label that always refers to the location of the beginning of (`rb.contents[0]`).

Because of performance optimizations within the library, it is not safe for applications to read or write these fields directly. Most fields may be read using one of these macros:

⟨*library types*⟩≡
```
typedef struct reloc_block *RBlock;
```
⟨*faux declarations of library macros*⟩≡
```
extern unsigned block_lc(RBlock rb);
extern unsigned block_size(RBlock rb);
extern unsigned block_low(RBlock rb);
extern int      block_address_known(RBlock);
extern unsigned block_address(RBlock);
extern RLabel   block_label(RBlock);
```

The `contents` aren't directly accessible as values; instead, we provide procedures to copy them into a buffer or a file (see Section 4.7). Except for the `address` field, this interface provides no way to tell whether fields are `null`. These macros return implementation-dependent values instead of `null`.

The fields of a relocatable block can be written using the procedures defined below.

A *label* is a location in a relocatable block. All undefined labels are located in a distinguished relocatable block, `undef`, whose `address` field is always null. A label `l` has the following fields:

`l.block`	the relocatable block in which `l` is located
`l.offset`	`l`'s offset from the beginning of `l.block`
`l.name`	a string associated with `l`

The name field may be used by the application to associate labels with names of important objects, e.g., procedures and global symbols.

⟨*library types*⟩+≡
```
typedef struct label *RLabel;
```

4.2.1 Operations on relocatable blocks

`block_new` creates a new relocatable block, `rb`, with `rb.lc = rb.size = 0`, null `rb.address` and `rb.low`, and empty `rb.contents`. If the argument `size` passed to `block_new` is non-zero, the library uses it as a hint about the maximum size the relocatable block is expected to reach. If the hint is accurate, the library will make more efficient use of memory. `block_new` adds each newly created block to `defined_rbs`, the set of all relocatable blocks created, excluding `undef`. `block_defined` succeeds if the given block is defined, i.e. it is not `undef`.

⟨*library procedures*⟩≡
```
extern RBlock block_new(unsigned size);
```
⟨*faux declarations of library macros*⟩+≡
```
extern int block_defined(RBlock rb);
```

`set_lc` sets the value of `rb.lc` and also updates `rb.size` if it is lower than `rb.lc`; `rb.size` is the "high-water mark" of `rb.lc`.

⟨*library procedures*⟩+≡
```
extern void set_lc(RBlock rb, unsigned lc);
```

In order to accomodate traditional assembly languages, the library maintains a current relocatable block `crb`. Many of the procedures in this interface have side effects on `crb`. Initially, `crb` is undefined. `set_block` sets the current relocatable block to `rb` and `crb` produces the current relocatable block.

⟨*library procedures*⟩+≡
```
extern void set_block(RBlock rb);
```
⟨*faux declarations of library macros*⟩+≡
```
extern RBlock crb(void);
```

Relocation occurs when an absolute address is assigned to a relocatable block. `set_address` sets the address of a block. It can be called multiple times to to change the address.[1] `block_address` produces the address. Its value is meaningful if and only if `block_address_known` returns nonzero; otherwise the address is null.

⟨*library procedures*⟩+≡
```
extern void set_address(RBlock, unsigned);
```
⟨*faux declarations of library macros*⟩+≡
```
extern int      block_address_known(RBlock);
extern unsigned block_address      (RBlock);
```

An application may want to reserve a register to point into a relocatable block. For example, the MIPS short-data (`.sdata`) section could be treated as a relocatable block, in which case the reserved register is the global pointer

[1] For efficiency, some applications may want to discard relocation information associated with a relocatable block `rb`. That would require the ability to mark `address` read-only. Let us know if you are interested.

(gp). If an application wants to reserve more than one register, it should split the relocatable block into multiple blocks. The following two fields make it possible to reserve registers:

`rb.reg`	if non-null, it identifies the register that points to `rb.contents[rb.reglc]`.
`rb.reglc`	the offset within the block to which `rb.reg` points.

The application notifies the library that a register points into a relocatable block by calling `set_register`. The application or toolkit can query the value of the register or its absolute address by calling `block_reg` or `block_regaddr`. Note that the value of `reg` is undefined if `set_register` has not been called; `regaddr` is similarly undefined if `reg` or `address` is undefined. It might be desirable to have some error checking to avoid binding two different registers to the same relocatable block; an `unset_register` procedure might help but we don't have one right now.

⟨*library procedures*⟩+≡
```
extern void set_register(RBlock rb, unsigned reg, unsigned regaddr);
extern unsigned block_reg(RBlock rb);
extern unsigned block_regaddr(RBlock rb);
```

4.2.2 Operations on labels

`label_new` creates a new label located in `undef` and assigns `name` to `l.name`. `label_define` binds a label to the given `offset` in the current relocatable block. Once a label is defined, it can't be redefined.[2]

⟨*library procedures*⟩+≡
```
extern RLabel label_new(char *name);
extern void label_define(RLabel lbl, int offset);
```

Because `label_define` uses an offset, the first pass of an assembler can maintain its own location counter in a register, using that counter to define labels. This trick means one library call per label instead of one per instruction.

The absolute address of a label is returned by `label_location`. Its value is defined only if `label_location_known` is true.

⟨*faux declarations of library macros*⟩+≡
```
extern int label_location_known(RLabel lbl);
extern unsigned label_location (RLabel lbl);
```

[2] For most applications (e.g., an assembler, `mld`), it is wrong to redefine a label, but an incremental linker might want to redefine a label when a relocatable block is replaced. We're willing to reconsider this decision.

4.3 Emitting information

The library is set up to make it easy and efficient to emit instructions or data at the current location in the current relocatable block. Such emission automatically advances the current relocatable block's location counter (`crb.lc`).

Most applications emit information one word at a time, not one byte at a time, raising the spectre of byte order. Byte order could be made part of the global state of the library, or it could be selected by supplying a parameter to an emitter procedure, but we have chosen to provide different emitter procedures for different byte orders. Applications that use only one byte order enjoy maximum efficiency; others can get reasonable efficiency by using procedure variables.

We provide two procedures that use fixed byte orders, plus a pointer to a third procedure that detects the byte order of the host machine and selects the matching procedure. `*emitm`, `emitl`, and `emitb` convert the least significant `8n` bits of `value` to a string of bytes, using the endianness requested (`machine`, `little`, or `big`). The `contents` of the current relocatable block is extended if it is not big enough to hold the new bits, and the bits are written to that `contents` starting at the current location. The location counter is incremented by `n`.

⟨*library procedures*⟩+≡
```
extern void (*emitm)(unsigned long value, unsigned n);
extern void emitl(unsigned long value, unsigned n);
extern void emitb(unsigned long value, unsigned n);
```

Emitting bytes sets the low-water mark of the current relocatable block if it isn't set already. It's illegal to emit bytes below the low-water mark.

Generated encoding procedures call one of these procedures to emit instructions, as selected by the generator's `-byteorder` option.

In some applications, the overhead required by the emitting procedures can be significant. This overhead includes not only the cost of a procedure call, but also the costs of manipulating bytes in a well-defined order. For example, it is in general unsafe to use a full-word store to place a full word into a relocatable block, even when the target byte order matches that of the host machine, because the alignment may not be correct. Some applications (e.g., RISC code generators) always maintain proper alignment. Such applications can achieve significant speedups by using the `emitfast` macro, which has the same effect as `emitm`, provided that the location counter is properly aligned to the size of the token being emitted. When the location counter is not properly aligned, the effect of `emitfast` is undefined, so it is in general unsafe.

⟨*faux declarations of library macros*⟩+≡
```
extern void emitfast(unsigned long value, unsigned n);
```

For convenience, we provide macros that emit full and half words and individual bytes using the byte order of the machine. They are simply abbreviations for calls to `emitm`.

⟨*faux declarations of library macros*⟩+≡
```
extern void emitu(unsigned value);
extern void emitus(unsigned short value);
extern void emituc(unsigned char value);
```

It would also be convenient to have a procedure for emitting relocatable addresses, but that one would be hard to write. Luckily, users can generate it from a specification like this one:

⟨*specification for generation* `emit_raddr`⟩≡
```
fields of addrword (32) addr32 0:31
placeholder for addrword is addr32 = 0xbadbad
constructors emit_raddr reloc is addr32 = reloc
```

4.4 Manipulating the location counter

The current location counter can be changed by `align`, `org`, and `addlc`. `align` aligns the current location counter on an `n`-byte boundary, `org` sets the location counter to `lc`, and and `addlc` adds `n` bytes to the location counter's current value.

⟨*library procedures*⟩+≡
```
extern void align(unsigned n);
```
⟨*faux declarations of library macros*⟩+≡
```
extern void org(unsigned lc);
extern void addlc(unsigned n);
```

Applications may also change the location counter of any relocatable block using `set_lc`, as described above.

The application can read `crb`'s location counter using `lc`.

⟨*faux declarations of library macros*⟩+≡
```
extern unsigned lc(void);
```

The toolkit is optimized for advancing location counters, so it is more efficient to use `addlc(n)` than to use `set_lc(crb(), lc()+n)`, because `set_lc` must handle the general case.

4.5 Relocation and closures

Computation involving relocatable addresses may be delayed until the location of a relocatable block is fixed. For example, the `emit_raddr` constructor specified above emits a 32-bit relocatable address into the current relocatable block. If `reloc` is unknown when `emit_raddr` is applied, the toolkit generator emits the place-holder pattern `addr32 = 0xbadbad` and creates a *relocation closure* (`RClosure`). In general, the toolkit generator emits a place-holder and creates a relocation closure for any instruction that refers to a relocatable address whose

value is unknown. A closure captures the information necessary for delayed evaluation of expressions involving relocatable addresses. When a relocation closure is applied, it overwrites the place-holder instruction with the relocated instruction.

Each relocation closure writes a sequence of tokens starting at a single location in a single relocatable block. Every closure uses, or *depends on*, at least one relocatable address. A relocation closure `cl` has the following fields:

`cl.dest_block`	the relocatable block to be written by `cl`
`cl.dest_lc`	the location written
`cl.depends_on`	the set of relocatable addresses on which the closure depends
`cl.apply`	a function that writes to `cl.dest_lc`, using the values of all the addresses in `cl.depends_on`

Applications may apply closures individually, as the relocatable addresses on which they depend become known, or all at once, after all relocatable blocks have known addresses. Because time of relocation is application-dependent, the library relies upon the application to save relocation closures in whatever manner it finds most convenient and then to apply the closures when ready. The toolkit requires the application to implement `save_closure`, which saves a closure when it is created; `save_closure`'s specification is given in Section 4.6. The library provides the application with `apply_closure`, a closure-independent procedure that applies the closure `cl`.

⟨*library types*⟩+≡
```
typedef struct rclosure *RClosure;
typedef void (*Emitter)
    (RBlock, unsigned dest_lc, unsigned long bits, unsigned size);
typedef void (*FailCont)(char *fmt, ...);
```
⟨*library procedures*⟩+≡
```
extern void apply_closure(RClosure cl, Emitter emitter, FailCont failc);
```

If any of the addresses on which `cl` depends is unknown, the failure continuation `failc` is applied. It should not return (see Section 4.6). The `Emitter` procedure performs the same functions as the `emit` procedures specified above, but instead of writing the current relocatable block, it updates the relocatable block `dest_block` at the location `dest_lc`. The library provides `Emitter` procedures in three byte orders: machine order, little-endian, and big-endian.

⟨*library procedures*⟩+≡
```
extern void (*cl_emitm)(RBlock dest_block, unsigned dest_lc,
                        unsigned long bits, unsigned size);
extern void cl_emitl (RBlock dest_block, unsigned dest_lc,
                        unsigned long bits, unsigned size);
extern void cl_emitb (RBlock dest_block, unsigned dest_lc,
                        unsigned long bits, unsigned size);
```

If the application writer provides an `Emitter` procedure to `apply_closure`, its action must not depend on the values of the current relocatable block or current location counter used by the library's `emit` procedures.

4.6 Application's obligations

The library relies on the application to allocate and store relocation closures, apply closures once the relocatable addresses on which they depend are known, and provide memory-management services to the library, which needs to allocate relocatable blocks, labels, relocatable addresses (fixed size) and contents of blocks (variable size).

The application must provide an allocator named `mc_alloc`, which accepts two parameters. The first is the size of the object to be allocated, and the second indicates what is being allocated. The second parameter may be used by the application to allocate objects from different memory pools; it tells the application whether memory is being allocated for a relocatable block, the contents of a relocatable block, a label, or a relocatable address.

⟨*library types*⟩+≡
```
typedef enum mc_alloc_pool {
    RBlock_pool=1, RBlock_data_pool, RLabel_pool, RAddr_pool
} Mc_alloc_pool;
```
⟨*declarations of procedures provided by the application*⟩≡
```
extern void *mc_alloc(int size, Mc_alloc_pool pool);
```

Applications must also provide `mc_alloc_closure` procedure to allocate memory for a new closure. We use a separate procedure because applications might want to use the destination block and location as hints to determine from what memory to allocate the closure:

⟨*declarations of procedures provided by the application*⟩+≡
```
extern RClosure mc_alloc_closure(unsigned size,
                                 RBlock dest_block, unsigned dest_lc);
```

`mc_alloc_closure` is called by generated encoding procedures, which fill in the fields of the new closure.

Generated encoding procedures call `*fail` if they are called upon to do something they can't, i.e., if their equations can't be solved. For example, the MIPS encoding procedures call `*fail` if asked to branch to an address that is not aligned to a 4-byte boundary. Applications must initialize `fail` to point to a suitable procedure (e.g., one that prints an error message, and maybe exits or `longjmp`s).

⟨*exported data declarations*⟩≡
```
extern void (*fail)(char *fmt, ...);
```

`*fail` may be used as a failure continuation when calling `apply_closure`.

4.7 Accessing contents of relocatable blocks

To access data already emitted in a relocatable block, applications may call `block_write` or `block_copy`. `block_write` writes the `n` bytes of `rb`'s `contents` beginning at offset `start` to the file descriptor `fd`; `block_copy` is similar but copies into the buffer `buf` instead of writing to `fd`; the application must allocate memory for `buf` before calling `block_copy`.

⟨*library procedures*⟩+≡
```
extern void block_write(int fd, RBlock rb, unsigned start, unsigned n);
extern void block_copy(unsigned char *buf,
                       RBlock rb, unsigned start, unsigned n);
```

4.8 Relocatable Addresses

Encoding applications often want to work with addresses whose values aren't known. Common examples include calls to unknown procedures, or forward branches within a procedures. This kind of work can be done with labels alone, but it is more convenient to use a more general abstraction: the *relocatable address*. In its most general form, a relocatable address is an expression whose value depends on one or more labels.[3] Most of the toolkit treats relocatable addresses simply as an abstract data type:

⟨*library types*⟩≡
```
typedef struct relocatable_address *RAddr;
```

The operations available on a relocatable address are to see whether it has a known value, and to extract that value:

⟨*faux declarations of library macros*⟩≡
```
extern int location_known(RAddr addr);
extern unsigned location (RAddr addr);
```

`location_known(addr)` will be true whenever all the labels on which `addr` depends have known values. If an application writer wants to change our representation of relocatable addresses, his only obligation is to provide `location_known` and `location`.

It is often convenient to refer to the "program counter" (`pc`), i.e., the address of the current location in the current relocatable block, as if it were a relocatable address. In our current scheme, it is not representable as a relocatable address, but we provide a similar interface through macros:

⟨*faux declarations of library macros*⟩+≡
```
extern int cur_pc_known(void);
extern unsigned cur_pc(void);
```

[3] Note that our definition is different from the more traditional definition, in which, for example, the difference of two labels is taken to be an absolute address, not a reloctable address. This traditional definition works only if the two labels are in the same compilation unit.

`cur_pc` returns `pc`'s value if `cur_pc_known` is true.

The representation we have chosen for relocatable addresses does not support the fully general form defined above, because we would need a little language for representing and interpreting addressing expressions, and we despair of devising a language whose definition and implementation would be acceptable to all application writers. Instead, the toolkit provides relocatable addresses of the form $L + n$, where L is a label and n is an integer offset. A relocatable address `ra` therefore has two fields:

```
ra.label
ra.offset
```

and one can be created by calling the following procedure:

⟨*library procedures*⟩≡
```
extern RAddr addr_new(RLabel label, int offset);
```

4.9 Excerpts from `mld`, an encoding application

`mld` is an optimizing, retargetable linker that links an intermediate code before generating object code for a target architecture. It performs the functions of a conventional assembler, i.e., it generates object code, and of a conventional linker, i.e., it assigns absolute addresses to data and text segments, places data and procedures in the appropriate segments, and relocates references to external symbols. In this section, we provide some snippets of code from `mld` to illustrate how the library is used in an encoding application. Many of the examples are target independent and are used by `mld` for each target architecture. The target-dependent excerpts are for the MIPS.

4.9.1 Using encoding procedures

`mld`'s code generators are based on those used in the `lcc` compiler, which emit assembly code (Fraser and Hanson 1991). Much of each code generator is generated from a BURG specification (Fraser, Henry, and Proebsting 1992), which contains rules for rewriting intermediate-code subtrees to assembly-language templates; the rest of the code generator is written by hand. Adapting a code generator means modifying both the BURG specification and the hand-written parts. The toolkit simplifies those modifications. Code that prints assembly language is replaced with code that calls the encoding procedures generated by the toolkit; the following three rules are extracted from `mld`'s set of rules that map assembly-language templates to calls to encoding procedures. The format arguments `%c` and `%0` refer to the registers chosen by the code generator.

```
"cvt.d.s $f%c,$f%0"     cvt_d_s(%c, %0), nop()
"mov.d $f%c,$f%0"       mov_d(%c, %0)
"not $%c,$%0"           nor(%c, 0, %0)
```

Hand-written assembly statements are replaced in a similar manner; for example,

```
print("fadds %f1, %f2, %f3");
```

is replaced by

```
fadds(f1, f2, f3);
```

a call to a toolkit-generated encoding procedure. The rules for mapping assembly to binary must be written for each target architecture.

4.9.2 Using relocatable blocks

The encoding procedures use the toolkit's library to write the generated code into relocatable blocks in `mld`'s memory; `mld` calls the library to create relocatable blocks and to set the current relocatable block. At startup, `mld` assigns one relocatable block to each data section by calling `block_new`. `mld` uses four of the MIPS data sections: read only (`rdata`), short data (`sdata`), initialized data (`data`) and uninitialized data (`bss`). `mld` pre-computes the sizes of each data section and provides these sizes to `block_new` as hints of the maximum size in bytes of each relocatable block. The following snippet initializes the blocks.

⟨`mld` *example*⟩≡
```
rdata = block_new(rdata_size);
sdata = block_new(sdata_size);
data  = block_new(data_size);
bss   = block_new(bss_size);
```

`mld` assigns a relocatable block to each procedure for which it generates object code by calling target-independent `new_procedure`. In `new_procedure`, `text` is a global variable that refers to the relocatable block of the current procedure. `set_block` is called first to set the current relocatable block to the current procedure. `block_new` is then called to create a new relocatable block; it uses `size` as a hint of the procedure's maximum size in bytes. The procedure's absolute address in the text segment is set using the current `pc` as its start address; `align(4)` aligns the `pc` on a 4-byte word [4] If the current `pc` is unknown, the base address of the code segment is used. Finally, `set_block` sets the current relocatable block to the new text block. All subsequent calls to encoding procedures will emit instructions in the new relocatable block.

⟨`mld` *example*⟩+≡
```
static void new_procedure(size, name) char *name; {
    set_block(text);
    text = block_new(size);
    text->label->name = name;
```

[4] The current location in any text block should already be aligned on a 4-byte word. This is just a sanity check.

```
    align(4);
    set_address(text, cur_pc_known() ?
                      cur_pc() : segments[CODE].baseaddr);
    set_block(text);
}
```

The adapted code generators also call the library directly, e.g., to emit data. The target-dependent `segment` procedure sets the segment in which data is emitted; the current relocatable block is set to the appropriate segment.

⟨`mld` *example*⟩+≡
```
static void segment(int s) {
  switch(s) {
    case LIT:  set_block(rdata); break;
    case DATA: set_block(data);  break;
    case BSS:  set_block(bss);   break;
    default:   assert(0);
  }
}
```

4.9.3 Using relocatable addresses

`mld` associates a relocatable address with each label and global symbol. The target-independent procedure `global` defines a global symbol in the current data segment. If the symbol does not have a relocatable address, `mld` calls `addr_new` to create and initialize its address. An error message is issued if the symbol is already defined.

⟨`mld` *example*⟩+≡
```
static void global(p) Symbol p; {
    ⟨assign p a relocatable address⟩
    if (!block_defined(p->x.raddr->label->block))
        label_define(p->x.raddr->label, p->x.raddr->offset);
    else  error("Attempt to redefine %s\n", p->name);
}
```
⟨*assign* `p` *a relocatable address*⟩≡
```
if (p->x.raddr == (RAddr)0)
    p->x.raddr = addr_new(label_new(p->name), 0);
```

References to global symbols often appear in data segments, e.g., as the initial values of other global symbols. `defaddress` emits the relocatable address of a symbol into the current data segment by calling `emit_raddr`, the constructor defined in Section 4.3 that emits a relocatable address.

⟨`mld` *example*⟩+≡
```
static void defaddress(p) Symbol p; {
    ⟨assign p a relocatable address⟩
    emit_raddr(p->x.raddr);
}
```

4.9.4 Relocation

When an encoding procedure emits an instruction that refers to an unknown relocatable address, it creates a closure to save the information necessary for relocating the instruction. `mld` supplies the procedure `save_closure`, which saves each closure in a table associated with the current relocatable block in the current segment.

⟨`mld` *example*⟩+≡
```
void save_closure(cl, size) RClosure cl; unsigned size; {
    addtotable(RClosure *, segments[cur_seg].closures[cur_rb], cl);
}
```

The choice of when to perform relocation is application dependent. `mld` performs relocation after it has assigned an absolute address to all of the data segments and procedures and just before emitting an executable program. The toolkit provides most of the support for performing relocation in `apply_closure`. `mld`'s `dispatch` procedure applies `apply_closure` to every closure it has saved. If any relocatable address is unknown, `apply_closure` calls `error` to issue an error message. The implementations of `save_closure` and `dispatch` are target independent.

⟨`mld` *example*⟩+≡
```
void dispatch() {
    int seg, rb, i;
    RClosure cl;

    for (seg = 0; seg < NSEGS; seg++)
        for (rb = 0; segments[seg].relocblocks[rb] != 0; rb++)
            gentable(segments[seg].closures[rb], i, 0) {
                cl = tableentry(RClosure *, segments[seg].closures[rb], i);
                apply_closure(cl, cl_emitm, &error);
            }
}
```

4.9.5 Example makefile

The following example makefile shows how `tk.o`, an object file containing all of the encoding procedures needed by `mld`, is created. `mips.tk` is created by `tools` with `mips.spec`, the specification provided with the toolkit, and `mld-mips.spec`, a specification file specific to `mld`, are provided as inputs. The implementation of the encoding procedures are extracted from `mips.tk` into `tk.c` using `notangle`; the interface prototypes are extracted into `tk.h`. The library interface file, `mclib.h`, is included by `tk.c`.

```
mips.tk: mips.spec mld-mips.spec
    tools -byteorder fast -lc-cons-names -encoder mips.tk \
```

```
        mips.spec mld-mips.spec

tk.c: mips.tk
    notangle -t8 -L mips.tk | cpif tk.c

tk.h: mips.tk
    notangle -t8 -L -Rheader mips.tk | cpif tk.h

tk.o: tk.c tk.h mclib.h mldtk.h
    cc -c -I./ tk.c
```

Chapter 5

The generator and translator

Although we describe the generator and translator as separate entities, they actually are implemented by a single program, named `tools`. `tools` parses one or more files containing specifications and reports any syntactic or semantic errors. It is not unusual to decompose a machine specification into multiple files. `mld`, for example, generates encoding procedures using a "core" specification, a specification of "synthetic" instructions, and a specification of constructors used only by `mld`. Command-line options are used to ask `tools` to run the generator or translator or to perform other functions. Given no options, `tools` simply looks for syntactic and simple semantic errors in a specification.

The options recognized by `tools` are grouped into five categories:

E	Encoding
D	Decoding
d	Debugging decoding
V	Verbosity
A	Analysis

Table 5.1 shows all the options in alphabetical order.

5.1 E: Encoding options

`-byteorder` *string*

This misleadingly named option actually determines the name of the function that the encoding procedures use to emit tokens (`emit`*string*). Since the toolkit library provides `emitb`, `emitl`, and `emitm` procedures, which emit tokens using big-endian, little-endian, or machine-dependent byte orders, the values `b`, `l`, and `m` are good choices. My favorite trick, however, is to use `-byteorder X`, then use the C compiler to pick a byte order later on, for example, by compiling with `cc -DemitX=emitfast`. This trick avoids

Option	Category	Usage
`-alt`	d	Show alternates at decision-tree leaves
`-ascii-dag` *file*	d	write an ascii representation of decision DAGs used in matching statements
`-ascii-tree` *file*	d	write an ascii representation of decision trees used to build matching DAGs
`-byteorder` *string*	E	choose emission procedure
`-c`	DE	generate C code
`-count`	E	Count invocations of encoding procedures
`-debug-heur` *file*	d	write information about heuristic scores for tree-building
`-debug-match` *file*	d	write other information about building matching statements
`-debug-solver`	V	turn on solver debugging (to `stderr`)
`-debug-split` *file*	d	write debugging info when splitting tree nodes
`-decoder` *file*	D	generate code on *file*
`-dot` *file*	d	write `dot` commands showing matching statements
`-emit-bits` n	E	unit of emission is n bits
`-encoder` *file*	E	generate encoding procedures on *file*
`-expand-spec` *file*	A	write expanded specification on *file*
`-fieldnames` *file*	A	write arrays of filed names on *file*
`-foldemit`	E	fold constants in calls to emit procedure
`-impossible`	V	make errors force stop with stack trace
`-lc-cons-names`	E	use all lower case for constructor names
`-lc-pat-names`	D	force pattern names to lower case
`-ledger`	d	Use ledger paper with `-dot`
`-matcher` *file*	D	transform pattern-matching statements in *file*
`-m3`	D	generate Modula-3 code
`-nowarn`	V	suppress warnings (not recommended)
`-tryall`	d	try all heuristics on every field in tree
`-verbose`	V	print extra warnings and information

Table 5.1: Alphabetical list of all the options

running the generator multiple times, which is good, because running the generator can be expensive.

`-c`

Tells the generator to generate C code (the only option), or tells the translator that the code emitted for the `-decoder` file should be C code.

`-count`

Tells the generator to include code in encoding procedures that counts the invocations of such procedures. This is useful for determining the static frequency of an instruction in a program, e.g., how many loads are generated. The C procedure

```
void reset_cons_counters(void);
```

resets the counters; the procedure

```
typedef void (*Printer)(void *closure, char *fmt, ...);
void dump_cons_counters(Printer p, void *closure);
```

dumps the values of all the counters. The user must provide a function of type `Printer`, a variadic function which accepts a closure, a `printf`-style format, and additional arguments. The closure encapsulates any information needed by the user's printing function. For example, the standard C function `fprintf` may be used as a `Printer`, in which case a file pointer acts as a closure.

`-emit-bits` *n*

The toolkit's generator works with bits, but it is more convenient for the toolkit's library to work with larger units. In particular, the second argument to the `emit` functions tells the library how wide the emitted token is, in units of *n* bits. The default is 8, which means that the `emit` functions work with bytes, since these are how the library distributed with the toolkit works. The width of every emitted token must be a multiple of *n* or the generator will complain.

`-encoder` *file*

Invokes the generator, which writes encoding procedures on *file*. *file* is a source file in `noweb` format (Ramsey 1994) from which the interface and implementation can be extracted. The root chunk for the interface is named `header`; the root chunk for the implementation is the default, i.e. `*`.

`-foldemit`

Normally, the toolkit refrains from folding constants when generating the code that emits a token. This behavior makes it easy to examine the generated code to double-check that it really looks right. The `-foldemit` option tells the toolkit to fold constants aggressively. Its only real use is to make the generated code smaller, because modern compilers fold constants themselves, don't they?

`-lc-cons-names`

Ensures that only lower-case letters are used in the names of generated encoding procedures.

5.2 D: Decoding options

`-c`

Tells the generator to generate C code (the only option), or tells the translator that the code emitted for the `-decoder` file should be C code.

`-decoder` *file*

Invoke the toolkit translator, which translates embedding matching statements from the `-matcher` file and writes the results on the `-decoder` file. To use the translator, both `-matcher` and `-decoder` must be given exactly once.

`-lc-pat-names`

Pattern names are accessible using the [*name*] syntax in matching statements; this option forces all such names to use lower-case letters only.

`-matcher` *file*

Invoke the toolkit translator, which translates embedding matching statements from the `-matcher` file and writes the results on the `-decoder` file. To use the translator, both `-matcher` and `-decoder` must be given exactly once.

`-m3`

Tells the translator that the code emitted for the `-decoder` file should be Modula-3 code. (The toolkit generator and library do not yet support Modula-3.)

5.3 d: Options for debugging decoders

These options provide different kinds of diagnostic information that shows how the toolkit is building a decoder.

`-alt`

Prints something to do with alternatives at the leaves of decision trees? I'm no longer sure...

`-ascii-dag` *file*

Writes on *file* an ASCII representation of the decision trees (actually DAGs) used to implement matching statements. Useful for debugging, especially performance debugging. *Probably broken. In any case, it should be rewritten to use the new prettyprinter.*

`-ascii-tree` *file*

Writes on *file* an ASCII representation of the decision trees before they are converted to DAGs. This option is really here only in case something goes badly wrong with the dagging process (or in case you're curious). *This, too, is probably broken.*

`-debug-heur` *file*

When building the decision tree, the toolkit uses about a half dozen heuristics to choose the field to test next. Each heuristic assigns a score to each field. This option writes information about heuristic scoring onto *file*.

`-debug-match` *file*

During tree building, writes information about building matching statements onto *file*.

`-debug-split` *file*

During tree building, writes information about splitting internal nodes onto *file*.

`-dot` *file*

Creates output for the `dot` graph-drawing program, which can be used to create a graphical representation of the decision tree (actually a DAG) used to implement matching statements. The output is written to *file*. *This option probably doesn't work. It certainly doesn't understand multiple branches with identical patterns but different conditions.*

`-ledger`

Insert a PostScript incantation that might make the `-dot` option use ledger paper instead of regular paper. *Almost certainly broken.*

`-tryall`

Show what split would be given by each heuristic. Useful only with the option `-debug-heur`.

5.4 V: Verbosity options

These options control the verbosity of the messages printed by the toolkit.

`-debug-solver`

Makes the equation solver (and a few other parts of the toolkit) print lots of debugging information on standard error.

`-impossible`

Arranges for all error messages to be treated as "impossible," which causes "this can't happen" to be printed and a stack trace to be generated. It is intended primarily for debugging the toolkit.

`-nowarn`

Suppresses warnings. It's not a good idea to use this option routinely. If a warning annoys you, let us know—maybe we can make it appear only when `-verbose`.

`-verbose`

Make the toolkit print extra warnings and informational messages. (We haven't done as much with this as we might.)

5.5 A: Analysis options

These options support neither encoding nor decoding; instead, they offer various kinds of analysis of what's in the toolkit specification. In fact, it's even possible to run `tools` without any options at all, just to check the syntax of a specification.

`-expand-spec` *file*

Rewrites the specification onto *file*, but with all patterns in normal form. Can be useful for seeing the meanings of complex opcodes. *This option predates the introduction of constructors, which it does not treat, and it may break horribly. Pester us if you want it brought up to date, so that it emits a complete specification in some canonical form.*

`-fieldnames` *file*

For each field in the specification, write to *file* an array of string literals representing the names of values of that field. Such arrays may be useful in writing disassemblers, for example.

5.6 Example execution of `tools`

The `tools` program is a prototype and has not been used by anyone other than its authors. We believe we have excised all uninteresting diagnostic messages; let us know if you think otherwise. The line numbers used in error messages reflect the state of the parser when the error is detected, not where the error actually occurred. In particular, many semantic errors are not detected until the whole specification is parsed, in which case the line number is worse than meaningless; it is misleading. We apologize for this behavior, which we hope to correct in a future release.

5.6.1 Running the generator

The following command uses the specification in `mips.spec` and produces `noweb` source for an encoder in the file `mips.out`. Because the `-m3` option is not used, the toolkit generates C code. Commands typed by the user are shown in *`typewriter italic`*; the toolkit's responses are shown in `typewriter`. Wide lines have been broken by hand.

```
% tools -byteorder X -foldemit -encoder mips.out mips.spec
Parsing...
"mips.nw", line 368:
  Warning -- Ignoring duplicate definition of constructor bc1f
"mips.nw", line 368:
  Warning -- Ignoring duplicate definition of constructor bc1t
Done parsing.
Warning: toolkit should check that fieldname literals introduced
  in fieldinfo statements are never used as names of constructor
  operands or as ids in equations --- but it doesn't
```

The duplicate definitions of `bc1f` and `bc1t` arise because these instructions are underspecified in the MIPS architecture manual, which is faithfully reflected in

`mips.spec`. The final warning warns users about a potential showstopper we haven't fixed yet.

5.6.2 Running the translator

To generate a decoder, we use two specifications: `mips.spec`, as in the generator command, and `fetch.spec`, which specifies how to fetch words from an instruction stream. `dis.m` includes the matching statement for the decoding application; the decoding code is emitted in C to `dis.c`. The last warning indicates that there are inputs that aren't matched by any arm of the matching statement. If such an input is encountered, the decoder will halt with a checked run-time error.

```
% tools -c -decoder dis.c -matcher dis.m mips.spec fetch.spec
Parsing...
"mips.nw", line 368:
  Warning -- Ignoring duplicate definition of constructor bc1f
"mips.nw", line 368:
  Warning -- Ignoring duplicate definition of constructor bc1t
Done parsing.
Warning: toolkit should check that fieldname literals introduced
  in fieldinfo statements are never used as names of constructor
  operands or as ids in equations --- but it doesn't
"./dis.m", line 8:
  Warning -- Matching statement at "./dis.m", line 8
  doesn't cover all cases
```

Appendix A

Vaporware

This appendix describes procedures that we would like to have in the library but that we haven't had time to implement. These procedures are geared to supporting *toolkit object code*, viz., a disk-based representation of the data structures used in encoding applications. We need *dictionaries* so that labels and such can be referred to by name, and *pickles* to be able to read and write relocatable blocks, relocation closures, and so on.

A.1 Dictionaries

Applications will need dictionaries for binding names of objects to their values.

⟨*library*⟩≡
```
Dictionary newdictionary(void)
    return an empty dictionary

Any lookup(Dictionary dict, char *name)
    return dict[name]

procedure insert(Dictionary dict, char *name, Any value)
    dict[name] := value
```

The library provides two predefined, global dictionaries:

`labeldict`	Binds the names of global labels to the labels themselves
`sectiondict`	Binds the names of sections to lists of relocatable blocks

An assembler may use `labeldict` to store labels defined in assembly text and `sectiondict` to store relocatable blocks that represent data sections or procedure text. The application may use the procedures defined above to create additional dictionaries.

The application must tell the library what section names it understands. `newsection` creates a new section if it has not already been created. An assembler, for example, might call `newsection` when assembling a pseudo-op such as ".data" for the first time.

⟨*library*⟩+≡
```
procedure newsection(char *name)
    /sectiondict[name] := [] | fail(multiply defined sections)
```

Each entry of `sectiondict` is a list. Relocatable blocks are added to a section by simply concatenating the block on the appropriate list. A C implementation of the library will provide the C equivalents of the Icon code `l := [], put(l,x), every x := !l do ...`.

A.2 Pickling and Unpickling

The application can externalize and internalize all of the data structures provided by the library using the following "pickling" and "unpickling" interface.

The following EBNF grammar specifies the layout of pickled objects on disk.[1] The smallest unit of reading and writing is a word. Within the grammar, four-letter strings, integer literals, and symbols in all caps are terminal symbols standing for single words. Symbols in mixed case are nonterminals.

The word represented by a four-character string is computed by putting the first character of the string in the most significant byte, and so on. For example, the string `"dict"`, used to tag dictionaries, forms the type tag

```
'd' << 24 + 'i' << 16 + 'c' << 8 + 't'.
```

This type tag *cannot* always be formed by casting the string `"dict"` to an integer using the native byte order of the machine.

The first word in a pickle is always `"picl"`. By checking this word, a program reading the pickle can tell whether the words in the pickle were written in the native or swapped byte order. A pickle always represents a single object. To provide a modicum of error checking, the object is always tagged. Most applications pickle their own data structures, which may point to more than one object of the kind defined in this library.

⟨*pickle grammar*⟩≡
```
Pickle  => "picl" Tagged
```

Every object within a pickle is identified by a unique signed integer or *UID*; UIDs make it possible to represent cycles and pointer sharing. The UID 0 always represents the `NULL` or `NIL` pointer.

Under some circumstances, a UID may be followed by a *type tag*, which identifies the type of the object. For example, every element of a heterogeneous

[1] In EBNF, | denotes alternation, `[...]` optional fields, and `{...}` Kleene closure.

list must carry a type tag. This library reserves the type tags in the range 0–32767 for use by applications.

The following productions show the type tags reserved for use with the toolkit types. The library supports only homogeneous dictionaries and lists.

⟨*pickle grammar*⟩+≡
```
Any    => [ Dictionary | List | RLabel | RAddr | RBlock
          | RClosure | INTEGER | UserDef
          ]
Tagged => [ "dict" Dictionary | "list" List | "rlbl" Rlabel
          | "radr" RAddr | "rblk" RBlock | "rclo" RClosure
          | "chrs" String | "int\0" INTEGER | TAG UserDef
          ]
```

A pickled homogeneous dictionary includes a `TAG` denoting the type of the dictionary's elements, a `Count` denoting the total number of dictionary elements, and a sequence of key-value pairs. The values are untagged because the single `TAG` at the beginning describes all of the values.

⟨*pickle grammar*⟩+≡
```
Dictionary => UID [TAG Count {String Any}]
```

A `List` is just like a dictionary but does not include any keys.

⟨*pickle grammar*⟩+≡
```
List       => UID [TAG Count {Any}]
```

Of the four basic data types provided by the library, `RAddr`, `RLabel`, `RBlock`, and `RClosure`, the first two are completely straightforward:

⟨*pickle grammar*⟩+≡
```
RAddr   => UID [OFFSET RLabel]
RLabel  => UID [OFFSET RBlock]
```

An `RBlock` contains two optional fields `ADDRESS` and `REG OFFSET`. The value of the `Flags` determines the presence of these fields. If `Flags & 1`, `ADDRESS` is the value of `rb.address`; otherwise, `ADDRESS` is omitted and `rb.address` is null. *and similarly for other fields...* `String` denotes `rb.contents`.

⟨*pickle grammar*⟩+≡
```
RBlock  => UID [LOW RLabel FLAGS [ADDRESS] [REG OFFSET] String]
```

An `RClosure` includes a `Function`, which represents the closure's `apply` method. `COUNT` denotes the number of relocatable addresses in the closure. Since a closure is created only when a relocatable address is unknown, there must be at least one `RAddr` in the closure.

⟨*pickle grammar*⟩+≡
```
RClosure => UID [OFFSET Function RBlock COUNT {RAddr}]
```

A function is always represented by its UID alone. The assembler tool reserves the negative UIDs for library functions and functions it generates.

⟨*pickle grammar*⟩+≡
```
Function => UID
```

Strings are an exception to the rule that all reading and writing is word-oriented. The disk representation of a string is a word indicating the number of bytes in the string, followed by the bytes themselves. The bytes are followed by zero or more zero bytes, used to pad the string so its length is a multiple of the word length. This representation is *not* equivalent to the C representation; for example, strings may contain embedded null characters. The C version of the library provides a procedure that pickles and unpickles strings in this format.

⟨*pickle grammar*⟩+≡
```
String => UID [COUNT bytes padding]
```

A.3 C language support for pickles

To pickle and unpickle a type, the library needs a tag, a pickling procedure, and an unpickling procedure:

⟨*C library*⟩≡
```
typedef struct pklstream *PklStream;
unsigned getword(PklStream s);                  /* probably a macro */
procedure putword(PklStream s, unsigned n);     /* probably a macro */
procedure read_bytes(char *buf, int size, PklStream s);

typedef struct pkltag {
    unsigned tagword;   /* TAG */
    void (*write)(PklStream s, void *data, ...);
    void *(*read)(PklStream s);
} *PklTag;
```

All calls to write procedures must be bracketed between `pickleBegin` and `pickleEnd`.

A `PklTag` is provided for each library data structure. The write procedures for homogeneous dictionaries and lists take an extra argument, which is a tag describing the element type.[2] Pickling procedures that are part of the application may use this extra argument to pickle dictionaries and lists of the application's own data structures. This library provides no support for pickling and unpickling heterogeneous dictionaries and lists.

⟨*C library*⟩+≡
```
PklTag pickleDict, pickleList, pickleRAddr, pickleRBlock,
       pickleRClosure, pickleRLabel;
```

[2] This admittedly ugly scheme is a poor man's substitute for real closures.

As an example, here is the code used to pickle a dictionary `d` in which the elements are labels:

⟨*example use*⟩≡
```
pickleDict->write(s, d, pickleRLabel);
```

`pickleRBlockList` is provided as a tag that simplifies pickling of `sectiondict`. To pickle `sectiondict`, the application would call

```
pickleDict(sectiondict, pickleRBlockList).
```

⟨*library*⟩+≡
```
PklTag pickleRBlockList;
    pickleRBlockList->write(s, l) == pickleList->write(s, l, pickleRBlock)
```

Applications use the tags to write and read pickles. `pickleWrite(fd, p, tag)` pickles the object pointed to by `p`, pointers within the object are followed and their referents pickled, and so on transitively. Pointer cycles and sharing are correctly represented. *flag for tagging every single object?*

⟨*C library*⟩+≡
```
procedure pickleWrite(int fd, void *p, PklTag tag);
    write disk representation of p to descriptor fd.
```

The library uses `int fd` instead of `FILE *fp` for input and output because some applications (like `lcc` and `mld`) don't tolerate `FILE *`.
`pickleRead` reads a pickled object from disk. *Support for pickles in which all objects are tagged?*

⟨*library*⟩+≡
```
procedure *pickleRead(int fd, PklTag type_expected, Arena *p,
                  (char *)(*pickleGetmem)(int size), int initialSize);
    return pointer to object in pickle
```

Allocation of pickles is arena-based, and all objects in a pickle are allocated in the same arena. If `*p` is nonzero, it is used as the arena; otherwise a new arena is created and *p is set to it. If `getmem` is nonzero, it is used to add space to the arena. If `pickleSizeHint` is nonzero, it is used as the initial size of the arena when an arena must be created.

References

Fraser, Christopher W. and David R. Hanson. 1991 (October).
A retargetable compiler for ANSI C.
SIGPLAN Notices, 26(10):29–43.

Fraser, Christopher W., Robert R. Henry, and Todd A. Proebsting. 1992 (April).
BURG—fast optimal instruction selection and tree parsing.
SIGPLAN Notices, 27(4):68–76.

Milner, Robin, Mads Tofte, and Robert W. Harper. 1990.
The Definition of Standard ML.
Cambridge, Massachusetts: MIT Press.

Ramsey, Norman and Mary F. Fernandez. 1994 (September).
New Jersey Machine-Code Toolkit architecture specifications.
Technical Report TR-470-94, Department of Computer Science, Princeton University.

———. 1995 (January).
The New Jersey machine-code toolkit.
In *Proceedings of the Winter 1995 USENIX Conference*, New Orleans, LA.
To appear. Preliminary version available as Technical Report TR-469-94, Department of Computer Science, Princeton University.

Ramsey, Norman. 1994 (September).
Literate programming simplified.
IEEE Software, 11(5):97–105.

Index

`addlc`, 31
`addr_new`, 35
`address add`, 23
`address to pc`, 23
`address type`, 23
`align`, 31
`-alt`, 45
`apply_closure`, 32
`-ascii-dag` *file*, 45
`-ascii-tree` *file*, 45

binary-operator, 17
binding, 10
bitrange, 17
`block_address`, 27, 28
`block_address_known`, 27, 28
`block_copy`, 34
`block_defined`, 28
`block_label`, 27
`block_lc`, 27
`block_low`, 27
`block_new`, 28
`block_reg`, 29
`block_regaddr`, 29
`block_size`, 27
`block_write`, 34
branches, 16, 17
byte order, 30
`-byteorder` *string*, 41

`-c`, 43, 44
`checked`, 10
`cl_emitb`, 32
`cl_emitl`, 32
`cl_emitm`, 32
constraint, 11
constructor, 14
`constructors`, 14
`-count`, 43
`crb`, 28
`cur_pc`, 34
`cur_pc_known`, 34

`-debug-heur` *file*, 45
`-debug-match` *file*, 45
`-debug-solver`, 46
`-debug-split` *file*, 45
`-decoder` *file*, 44
`discard`, 19
`-dot` *file*, 45

ellipsis
    use of to permit conjunction, 13
`-emit-bits` *n*, 43
`emitb`, 30
`emitfast`, 30
`emitl`, 30
`emitm`, 30
`Emitter`, 32
`emitu`, 30
`emituc`, 30
`emitus`, 30
`-encoder` *file*, 43
`epsilon`, 11
equation, 17
equation solver
    apology for, 18
equations, 17
error messages
    apology for, 47
    quality of, 18
`-expand-spec` *file*, 47
expr, 17

`fail`, 33
`FailCont`, 32
`fetch`, 23
field-binding, 11
field-item, 10
field-specifier, 10
field-specs, 9
`fieldinfo`, 9
fieldinfo-spec, 10
`-fieldnames` *file*, 47
`fields`, 9
`-foldemit`, 44

generating expression, 11
generating-expression, 11
`guaranteed`, 10

`-impossible`, 46
`insert`, 49

`keep`, 19

`label_define`, 29
`label_location`, 29
`label_location_known`, 29
`label_new`, 29
`lc`, 31
`-lc-cons-names`, 44
`-lc-pat-names`, 44
`-ledger`, 46
literal, 16
`location`, 34
`location_known`, 34
`lookup`, 49

`-m3`, 44
`-matcher` *file*, 44
matching rules, 13
`mc_alloc`, 33
`mc_alloc_closure`, 33
`Mc_alloc_pool`, 33

`names`, 10
`newdictionary`, 49
`newsection`, 50
`-nowarn`, 46

opcode, 15
operand, 16
opname, 15
`org`, 31

pattern, 11, 12
pattern bindings, 13
pattern operators, 12
    meaning of, 13
pattern-binding, 14
patterns
    complex, 12
    matching rules for, 13
    normal form of, 12
    simple, 11
`$pc`, 17
    referring to in library, 35
    template for conversion, 23
`pc_unit_bits`, 18
`placeholder`, 18

`RAddr`, 34
relational-operator, 11

`set_address`, 28
`set_block`, 28
`set_lc`, 28
`set_register`, 29
solver, *see* equation solver
`some`, 11
`sparse`, 10
spec, 9, 10, 14, 18, 19, 23
Specification, 9

`-tryall`, 46

`unchecked`, 10

`-verbose`, 46

`wordsize`, 18