Nanopass Framework Users Guide[1]

Andrew W. Keep

January 12, 2014

[1] This documentation is largely extracted from Chapter 2 of my dissertation [1]. The user guide has been updated to reflect recent updates the nanopass framework. Several example passes and languages have also been replaced with a more recent, publicly available example compiler.

Chapter 1

Introduction

The nanopass framework is an embedded DSL for writing compilers. The framework provides two main syntactic forms: `define-language` and `define-pass`. The `define-language` form specifies the grammar of an intermediate language. The `define-pass` form specifies a pass that operates over an input language and produces another, possibly different, output language.

1.1 A Little Nanopass Framework History

The idea of writing a compiler as a series of small, single-purpose passes grew out of a course on compiler construction taught by Dan Friedman in 1999 at Indiana University. The following year, R. Kent Dybvig and Oscar Waddell joined Friedman to refine the idea of the *micropass compiler* into a set of assignments that could be used in a single semester to construct a compiler for a subset of Scheme. The micropass compiler uses an S-expression pattern matcher developed by Friedman to simplify the matching and rebuilding of language terms. Erik Hilsdale added a support for catamorphisms [2] that provides a more succinct syntax for recurring into sub-terms of the language, which further simplified pass development.

Passes in a micropass compiler are easy to understand, as each pass is responsible for just one transformation. The compiler is easier to debug when compared with a traditional compiler composed of a few, multi-task passes. The output from each pass can be inspected to ensure that it meets grammatical and extra-grammatical constraints. The output from each pass can also be tested in the host Scheme system to ensure that the output of each pass evaluates to the value of the initial expression. This makes it easier to isolate broken passes and identify bugs. The compiler is more flexible than a compiler composed of a few, multi-task passes. New passes can easily be added between existing passes, which allows experimentation with new optimizations. In an academic setting, writing compilers composed of many, single-task passes is useful for assigning extra compiler passes to advanced students who take the course.

Micropass compilers are not without drawbacks. First, efficiency can be a problem due to pattern-matching overhead and the need to rebuild large S-expressions. Second, passes often contain boilerplate code to recur through otherwise unchanging language forms. For instance, in a pass to remove one-armed `if` expressions, where only the `if` form changes, other forms in the language must be handled explicitly to locate embedded `if` expressions. Third, the representation lacks formal structure. The grammar of each intermediate language can be documented in comments, but the structure is not enforced.

The `define-language` and `define-pass` syntactic forms are used by the nanopass framework to address these problems. A `define-language` form formally specifies the grammar of an intermediate language. A `define-pass` form defines a pass that operates on one language and produces output in a possibly different language. Formally specifying the grammar of an intermediate language and writing passes based on these intermediate languages allows the nanopass framework to use a record-based representation of language terms that is more efficient than the S-expression representation, autogenerate boilerplate code to recur through otherwise unchanging language forms, and generate checks to verify that the output of each pass adheres to the output-language grammar.

The summer after Dybvig, Waddell, and Friedman taught their course, Jordan Johnson implemented an initial prototype of the nanopass framework to support the construction of micropass compilers. In 2004, Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig developed a more complete prototype nanopass framework for compiler construction and submitted a paper on it to ICFP [3]. The initial paper focused on the nanopass framework as a tool capable of developing both academic and commercial quality compilers. The paper was accepted but on the condition that it be refocused only on academic uses. The reviewers were not convinced that the framework or nanopass construction method was capable of supporting a commercial compiler. In retrospect, the reviewers were right. Sarkar implemented only a few of the passes from the compiler used in the course on compilers. This implementation showed that the nanopass framework was viable, but it did not support the claim that the nanopass framework could be used for a commercial compiler. In fact, because the class compiler was started but never completed, it is unclear whether the prototype was even up to the task of writing the full class compiler.

The nanopass framework described in this guide improves on the prototype developed by Sarkar. In this framework, language definitions are no longer restricted to top-level definitions. Additionally, passes can accept more than one argument and return zero or more values. Passes can be defined that operate on a subset of a language instead of being restricted to starting from the entry-point nonterminal of the language. Passes can also autogenerate nonterminal transformers not supplied by the compiler writer. The new nanopass framework also defines two new syntactic forms, `nanopass-case` and `with-output-language`, that allow language terms to be matched and constructed outside the context of a pass.

1.2 The Nanopass Framework Today

Although the nanopass framework defines just two primary syntactic forms, the macros that implement them are complex, with approximately 4600 lines of code. In both the prototype and the new version of the nanopass framework, the `define-language` macro parses a language definition and stores a representation of it in the compile-time environment. This representation can be used to guide the definition of derived languages and the construction of passes. Both also create a set of record types used to represent language terms at run time, along with an unparser for translating the record representation to an S-expression representation. Finally, both create meta-parsers to parse S-expression patterns and templates. An S-expression to record-form parser can also be created from the language using `define-parser`.[1]

The `define-pass` form, in both versions of the framework, operates over an input-language term and produces an output-language term. The input-language meta-parser generates code to match the specified pattern as records, as well as a set of bindings for the variables named in the pattern. The output-language meta-parser generates record constructors and grammar-checking code. Within a pass definition, a transformer is used to define a translation from an input nonterminal to an output nonterminal. Each transformer has a set of clauses that match an input-language term and construct an output-language term. The pattern matching also supports catamorphisms [2] for recurring into language sub-terms.

1.3 Examples using the Nanopass Framework

There are two, publicly available examples of the nanopass framework. The first is in the `tests` sub-directory of the nanopass framework git repository at github.com/akeep/nanopass-framework. This is part of a student compiler, originally included with the prototype nanopass framework developed by Sarkar et al. and updated to conform with the changes that have been made in the updated nanopass framework.

[1] In the prototype, this was part of the functionality of `define-language`, but in a commercial compiler we do not frequently need an S-expression parser, so we no longer autogenerate one.

The second example is available in the github.com/akeep/scheme-to-c repository. This compiler is better documented and provides a complete compiler example targeting fairly low-level C from a simplified Scheme dialect. It was developed to be presented at Clojure Conj 2013, just days before the Conj started, and compiles a small subset of Scheme to C. It is similar to the included example, but has the advantage of being a complete end-to-end compiler that can be run from a Scheme REPL. It uses `gcc`, targeting a 64-bit platform as the back-end, but I hope can be modified to target other platforms without too much trouble, or even moved off of C to target JavaScript, LLVM, or other back ends.

1.4 Other Uses of the Nanopass Frameowrk

The nanopass framework was used to replace the original Chez Scheme compiler [**?**] with a nanopass version of the compiler. The nanopass version has officially been released as Chez Scheme version 9.0. Chez Scheme is a closed-source commercial compiler.

The nanopass framework is also being used as part of the Harlan compiler. Harlan is a general purpose language for developing programs for running on the GPU. Harlan uses an S-expression format that is compiled into C++ using OpenCL to run computational kernels on the GPU. The source code for Harlan is publicly available at github.com/eholk/harlan.

Chapter 2

Defining Languages and Passes

The nanopass framework builds on the prototype, originally developed by Sarkar et al. The examples in this section are pulled from the Scheme to C compiler available at github.com/akeep/scheme-to-c.

2.1 Defining languages

The nanopass framework operates over a set of compiler-writer-defined languages. Languages defined in this way are similar to context-free grammars, in that they are composed of a set of terminals, a set of nonterminal symbols, a set of productions for each nonterminal, and a start symbol from the set of nonterminal symbols. We refer to the start symbol as the entry nonterminal of the language. An intermediate language definition for a simple variant of the Scheme programming language, post macro expansion, might look like:

```
(define-language Lsrc
  (terminals
    (symbol (x))
    (primitive (pr))
    (constant (c))
    (datum (d)))
  (Expr (e body)
    pr
    x
    c
    (quote d)
    (if e0 e1)
    (if e0 e1 e2)
    (or e* ...)
    (and e* ...)
    (not e)
    (begin e* ... e)
    (lambda (x* ...) body* ... body)
    (let ([x* e*] ...) body* ... body)
    (letrec ([x* e*] ...) body* ... body)
    (set! x e)
    (e e* ...)))
```

The `Lsrc` language defines a subset of Scheme suitable for our example compiler. It is the output language of a more general “parser” that parses S-expressions into `Lsrc` language forms. The `Lsrc` language consists of a set of terminals (listed in the `terminals` form) and a single nonterminal `Expr`. The terminals of the language are

- `symbol` (for variables),
- `primitive` (for the subset of Scheme primitives support by this language),
- `constant` (for the subset of Scheme constants, and
- `datum` (for the subset of Scheme datum supported by this language).

The compiler writer must supply a predicate corresponding to each terminal, lexically visible where the language is defined. The nanopass framework derives the predicate name from the terminal name by adding a `?` to the terminal name. In this case, the nanopass framework expects `symbol?`, `primitive?`, `constant?`, and `datum?` to be lexically visible where `Lsrc` is defined.

Each terminal clause lists one or more meta-variables, used to refer to the terminal in nonterminal productions. Here, `x` refers to a `symbol`, `pr` refers to a `primitive`, `c` refers to a `constant`, and `d` refers to a `datum`.

For our example compiler, the host Scheme system's `symbol?` is used to determine when an item is a variable.

The example compiler also selects a subset of primitives from Scheme and represents these primitives as symbols. A `primitive?` predicate like the following can be used to specify this terminal.[1]

```
(define primitive?
  (lambda (x)
    (memq x
      '(cons make-vector box car cdr vector-ref vector-length unbox
        + - * / pair? null? boolean? vector? box? = < <= > >= eq?
        vector-set! set-box!))))
```

Our example compiler also limits the constants that can be expressed to a subset of those allowed by Scheme. The `constant?` predicate limits these to booleans (`#t` and `#f`), null (`()`), and appropriately sized integers (between -2^60 and $2^60 - 1$).

```
(define target-fixnum?
  (lambda (x)
    (and (and (integer? x) (exact? x))
         (<= (- (expt 2 60)) x (- (expt 2 60) 1)))))

(define constant?
  (lambda (x)
    (or (target-fixnum? x) (boolean? x) (null? x))))
```

The example compiler limits the Scheme datum that can be represented to constants, pairs, vectors, and boxes. The `datum?` predicate can be defined as follows:

```
(define datum?
  (lambda (x)
    (or (constant? x)
        (and (box? x) (datum? (unbox x)))
        (and (pair? x) (datum? (car x)) (datum? (cdr x)))
        (and (vector? x)
             (let loop ([i (vector-length x)])
               (or (fx=? i 0)
```

[1] In the example compiler, the primitives are specified in separate association lists to capture the arity of each primitive and the place in the compiler is handled as it goes through the compiler process. This complexity has been eliminated for the dicussion here. Please reference the source code for a more complete discussion of primitive handling in the example compiler.

```
                (let ([i (fx- i 1)])
                  (and (datum? (vector-ref x i))
                       (loop i)))))))))
```

The Lsrc language also defines the nonterminal Expr. Nonterminals start with a name, followed by a list of meta-variables and a set of grammar productions. In this case, the name is Expr, and two meta-variables, e and body, are specified. Just like the meta-variables named in the terminals clause, nonterminal meta-variables are used to represent the nonterminal in nonterminal productions. Each production follows one of three forms. It is a single meta-variable, an S-expression that starts with a keyword, or an S-expression that does not start with a keyword (referred to as an *implicit* production). The S-expression forms cannot include keywords past the initial starting keyword. In Lsrc, the x, c, and pr productions are the single meta-variable productions and indicate that a stand-alone symbol, constant, or primitive are valid Exprs. The only implicit S-expression production is the (e e* ...) production, and it indicates a call that takes zero or more Exprs as arguments. (The * suffix on e is used by convention to indicate plurality and does not have any semantic meaning: It is the ... that indicates that the field can take zero or more Exprs.) The rest of the productions are S-expression productions with keywords that correspond to the Scheme syntax that they represent.

In addition to the star, *, suffix mentioned earlier in the call productions, meta-variable references can also use a numeric suffix (as in the productions for if), a question mark (?), or a caret (^). The ? suffix is intended for use with maybe meta-variables, and the ^ is used when expressing meta-variables with a more mathematical syntax than the numeric suffixes provide. Suffixes can also be used in combination. References to meta-variables in a production must be unique, and the suffixes allow the same root name to be used more than once.

Language definitions can also include more than one nonterminal, as the following language illustrates:

```
(define-language L8
  (terminals
    (symbol (x a))
    (constant (c))
    (void+primitive (pr)))
  (entry Expr)
  (Expr (e body)
    x
    le
    (quote c)
    (if e0 e1 e2)
    (begin e* ... e)
    (set! x e)
    (let ([x* e*] ...) abody)
    (letrec ([x* le*] ...) body)
    (primcall pr e* ...)
    (e e* ...))
  (AssignedBody (abody)
    (assigned (a* ...) body) => body)
  (LambdaExpr (le)
    (lambda (x* ...) abody)))
```

This language has three nonterminals, Expr, AssignedBody, and LambdaExpr. When more than one nonterminal is specified, one must be selected as the entry point. In language L8, the Expr nonterminal is selected as the entry nonterminal by the (entry Expr) clause. When the entry clause is not specified, the first nonterminal listed is implicitly selected as the entry point.

The L8 language uses a single terminal meta-variable production, x, to indicate that a stand-alone symbol is a valid Expr. In addition, the L8 language uses a single nonterminal meta-variable production, le, to indicate that any LambdaExpr production is also a valid Expr. The LambdaExpr is separated from Expr because the letrec production is now limited to binding symbols to LambdaExprs.

The assigned production of the AssignedBody nonterminal utilizes a the => syntax to indicate a pretty unparsing form. This allows the unparser that is automatically produced by define-language to generate an S-expression that can be evaluated in the host Scheme system. In this case, the assigned from is not a valid Scheme form, so we simply eliminated the assigned wrapper and list of assigned variables when unparsing.[2]

In addition to the nanopass framework providing a syntax for specifying list structures in a language production, it is also possible to indicate that a field of a language production might not contain a (useful) value. The following language has an example of this:

```
(define-language Lopt
  (terminals
    (uvar (x))
    (label (l))
    (constant (c))
    (primitive (pr)))
  (Expr (e body)
    x
    (quote c)
    (begin e* ... e)
    (lambda (x* ...) body)
    (let ([x* e*] ...) body)
    (letrec ([x* le*] ...) body)
    (pr e* ...)
    (call (maybe l) (maybe e) e* ...))
  (LambdaExpr (le)
    (lambda (x* ...) body)))
```

The (maybe l) field indicates that either a label, l, or #f will be provided. Here, #f is a stand-in for bottom, indicating that the value is not specified. The (maybe e) field indicates that either an Expr or #f will be provided.

Instead of using (maybe l) to indicate a label that might be provided, a maybe-label terminal that serves the same purpose could be added. It is also possible to eliminate the (maybe e) form, although it requires the creation of a separate nonterminal that has both an e production and a production to represent ⊥, when no Expr is available.

2.2 Extending languages

The first "pass" of the example compiler is a simple expander that produces Lsrc language forms from S-expressions. The next pass takes the Lsrc language and removes the one-armed-if expressions, replacing them with a two-armed-if that results in the void value being produced by the expression when the test clause is false. code appropriate to construct these constants. The output grammar of this pass changes just one production of the language, exchanging potentially complex quoted datum with quoted constants and making explicit the code to build the constant pairs and vectors when the program begins execution.

The compiler writer could specify the new language by rewriting the Lsrc language and replacing the appropriate terminal forms. Rewriting each language in its full form, however, can result in verbose source code, particularly in a compiler like the class compiler, which has nearly 30 different intermediate languages.

[2] Unparsers can also produce the non-pretty from by passing both the language form to be unparsed and a #f to indicate the pretty form should not be used.

Instead, the nanopass framework supports a language extension form. The output language can be specified as follows:

```
(define-language L1
  (extends Lsrc)
  (terminals
    (- (primitive (pr)))
    (+ (void+primitive (pr))))
  (Expr (e body)
    (- (if e0 e1))))
```

The `L1` language removes the `primitive` terminal and replaces it with the `void+primitive` terminal. It also removes the `(if e0 e1)` production. A language extension form is indicated by including the `extends` clause, in this case `(extends Lsrc)`, that indicates that this is an extension of the given base language. In a language extension, the `terminals` form now contains subtraction clauses, in this case `(- (primitive (pr)))`, and addition clauses, in this case `(+ (void+primitive (pr)))`. These addition and subtraction clauses can contain one or more terminal specifiers. The nonterminal syntax is similarly modified, with the subtraction clause, in this case `(- (if e0 e1))`, that indicates productions to be removed and an addition clause that indicates productions to be added, in this case no productions are added.

The list of meta-variables indicated for the nonterminal form is also updated to use the set in the extension language. It is important to include not only the meta-variables named in the language extension but also those for terminal and nonterminal forms that will be maintained from the base language. Otherwise, these meta-variables will be unbound in the extension language, leading to errors.

Nonterminals can be removed in an extended language by removing all of the productions of the nonterminal. New nonterminals can be added in an extended language by adding the productions of the new nonterminal. For instance, language `L15` removes the `x`, `(qoute c)`, and `(label l)` productions from the `Expr` nonterminal and adds the `SimpleExpr` nonterminal.

```
  (define-language L15
    (extends L14)
    (Expr (e body)
      (- x
         (quote c)
         (label l)
         (primcall pr e* ...)
         (e e* ...))
      (+ se
         (primcall pr se* ...) => (pr se* ...)
         (se se* ...)))
    (SimpleExpr (se)
      (+ x
         (label l)
         (quote c))))
```

2.2.1 The `define-language` form

The `define-language` syntax has two related forms. The first form fully specifies a new language. The second form uses the `extends` clause to indicate that the language is an extension of an existing base language.

Both forms of `define-language` start with the same basic syntax:

(define-language *language-name* *clause* ...)

where *clause* is an `extension` clause, an `entry` clause, a `terminals` clause, or a nonterminal clause.

Extension clause. The extension clause indicates that the new language is an extension of an existing language. This clause slightly changes the syntax of the `define-language` form and is described in Section 2.2.

Entry clause. The entry clause specifies which nonterminal is the starting point for this language. This information is used when generating passes to determine which nonterminal should be expected first by the pass. This default can be overridden in a pass definition, as described in Section 2.3.1. The entry clause has the following form:

```
(entry nonterminal-name)
```

where *nonterminal-name* corresponds to one of the nonterminals specified in this language. Only one entry clause can be specified in a language definition.

Terminals clause. The terminals clause specifies one or more terminals used by the language. For instance, in the `Lsrc` example language, the terminals clause specifies three terminal types: `uvar`, `primitive`, and `datum`. The terminals clause has the following form:

```
(terminals terminal-clause ...)
```

where *terminal-clause* has one of the following forms:

```
(terminal-name (meta-var ...))
(=> (terminal-name (meta-var ...)) prettifier)
(terminal-name (meta-var ...)) => prettifier
```

Here,

- *terminal-name* is the name of the terminal, and a corresponding *terminal-name*? predicate function exists to determine whether a Scheme object is of this type when checking the output of a pass,
- *meta-var* is the name of a meta-variable used for referring to this terminal type in language and pass definitions, and
- *prettifier* is a procedure expression of one argument used when the language unparser is called in “pretty” mode to produce a pretty, S-expression representation.

The final form is syntactic sugar for the form above it. When the *prettifier* is omitted, no processing is done on the terminal when the unparser runs.

Nonterminal clause. A nonterminal clause specifies the valid productions in a language. Each nonterminal clause has a name, a set of meta-variables, and a set of productions. A nonterminal clause has the following form:

```
(nonterminal-name (meta-var ...)
  production-clause
  ...)
```

where *nonterminal-name* is an identifier that names the nonterminal, *meta-var* is the name of a meta-variable used when referring to this nonterminal in language and pass definitions, and *production-clause* has one of the following forms:

```
terminal-meta-var
nonterminal-meta-var
production-s-expression
(keyword . production-s-expression)
```

Here,

- *terminal-meta-var* is a terminal meta-variable that is a stand-alone production for this nonterminal,
- *nonterminal-meta-var* is a nonterminal meta-variable that indicates that any form allowed by the specified nonterminal is also allowed by this nonterminal,
- *keyword* is an identifier that must be matched exactly when parsing an S-expression representation, language input pattern, or language output template, and
- *production-s-expression* is an S-expression that represents a pattern for production and has the following form:

```
meta-variable
(maybe meta-variable)
(production-s-expression ellipsis)
(production-s-expression ellipsis production-s-expression ... . production-s-expression)
(production-s-expression . production-s-expression)
()
```

Here,

- *meta-variable* is any terminal or nonterminal meta-variable extended with an arbitrary number of digits, followed by an arbitrary combination of `*`, `?`, or `^` characters; for example, if the meta-variable is `e`, then `e1`, `e*`, `e?`, and `e4*?` are all valid meta-variable expressions;
- `(maybe` *meta-variable*`)` indicates that an element in the production is either of the type of the meta-variable or bottom (represented by `#f`); and
- *ellipsis* is the literal `...` and indicates that a list of the *production-s-expression* that proceeds it is expected.

Thus, a Scheme language form such as `let` can be represented as a language production as:

```
(let ([x* e*] ...) body* ... body)
```

where `let` is the *keyword*, `x*` is a meta-variable that indicates a list of variables, `e*` and `body*` are meta-variables that each indicate a list of expressions, and `body` is a meta-variable that indicates a single expression.

Using the `maybe` form, something similar to the named-let form could be represented as follows:

```
(let (maybe x) ([x* e*] ...) body* ... body)
```

although this would be slightly different from the normal named-let form, in that the non-named form would then need an explicit `#f` to indicate that no name was specified.

2.2.2 Extensions with the `define-language` form

A language defined as an extension of an existing language has a slightly modified syntax to indicate what should be added to or removed from the base language to create the new language. A compiler writer indicates that a language is an extension by using an extension clause.

Extension clause. The extension clause has the following form:

```
(extends language-name)
```

where *language-name* is the name of an already defined language. Only one extension clause can be specified in a language definition.

Entry clause. The entry clause does not change syntactically in an extended language. It can, however, name a nonterminal from the base language that is retained in the extended language.

Terminals clause. When a language derives from a base language, the `terminals` clause has the following form:

```
(terminals extended-terminal-clause ...)
```

where *extended-terminal-clause* has one of the following forms:

```
(+ terminal-clause ...)
(- terminal-clause ...)
```

where the *terminal-clause* uses the syntax for terminals specified in the non-extended `terminals` form. The `+` form indicates terminals that should be added to the new language. The `-` form indicates terminals that should be removed from the list in the old language when producing the new language. Terminals not mentioned in a terminals clause will be copied unchanged into the new language. Note that adding and removing *meta-var*s from a terminal currently requires removing the terminal type and re-adding it. This can be done in the same step with a `terminals` clause, similar to the following:

```
(terminals
  (- (variable (x)))
  (+ (variable (x y))))
```

Nonterminal clause. When a language extends from a base language, a nonterminal clause has the following form:

```
(nonterminal-name (meta-var ...)
  extended-production-clause
  ...)
```

where *extended-production-clause* has one of the following forms:

```
(+ production-clause ...)
(- production-clause ...)
```

The `+` form indicates nonterminal productions that should be added to the nonterminal in the new language. The `-` form indicates nonterminal productions that should not be copied from the list of productions for this nonterminal in the base language when producing the new language. Productions not mentioned in a nonterminal clause will be copied unchanged into the nonterminal in the new language. If a nonterminal has all of its productions removed in a new language, the nonterminal will be dropped in the new language. Conversely, new nonterminals can be added by naming the new nonterminal and using the `+` form to specify the productions of the new nonterminal.

2.2.3 Products of `define-language`

The `define-language` form produces the following user-visible bindings:

- a language definition, bound to the specified *language-name*;
- an unparser (named `unparse-`*language-name*) that can be used to unparse a record-based representation back into an S-expression representation; and

- a set of predicates that can be used to identify a term of the language or a term from a specified nonterminal in the language.

It also produces the following internal bindings:

- a meta-parser that can be used by the `define-pass` macro to parse the patterns and templates used in passes and
- a set of record definitions that will be used to represent the language forms.

The `Lsrc` language, for example, will bind the identifier `Lsrc` to the language definition, produce an unparser named `unparse-Lsrc`, and create two predicates, `Lsrc?` and `Lsrc:Expr?`. The language definition is used when the *language-name* is specified as the base of a new language definition and in the definition of a pass.

The `define-parser` form can also be used to create a simple parser for parsing S-expressions into language forms as follows:

`(define-parser` *parser-name language-name*`)`

The parser does not support backtracking; thus, grammars must be specified, either by specifying a keyword or by having different length S-expressions so that the productions are unique.

For instance, the following language definition cannot be parsed because all four of the `set!` forms have the same keyword and are S-expressions of the same length:

```
(define-language Lunparsable
  (terminals
    (variable (x))
    (binop (binop))
    (integer-32 (int32))
    (integer-64 (int64)))
  (Program (prog)
    (begin stmt* ... stmt))
  (Statement (stmt)
    (set! x0 int64)
    (set! x0 x1)
    (set! x0 (binop x1 int32))
    (set! x0 (binop x1 x2))))
```

Instead, the `Statement` nonterminal must be broken into multiple nonterminals, as in the following language:

```
(define-language Lparsable
  (terminals
    (variable (x))
    (binop (binop))
    (integer-32 (int32))
    (integer-64 (int64)))
  (Program (prog)
    (begin stmt* ... stmt))
  (Statement (stmt)
    (set! x rhs))
  (Rhs (rhs)
    x
    int64
    (binop x arg))
  (Argument (arg)
    x
    int32))
```

2.3 Defining passes

Passes are used to specify transformations over languages defined by using define-language. Before going into the formal details of defining passes, we need to take a look at a simple pass to convert an input program from the Lsrc intermediate language to the L1 intermediate language. This pass removes the one-armed-if by making the result of the if expression explicit when the predicate is false.

We define a pass called remove-one-armed-if to accomplish this task, without using any of the catamorphism [2] or autogeneration features of the nanopass framework. Below, we can see how this feature helps eliminate boilerplate code.

```
(define-pass remove-one-armed-if : Lsrc (e) -> L1 ()
  (Expr : Expr (e) -> Expr ()
    [(if ,e0 ,e1) `(if ,(Expr e0) ,(Expr e1) (void))]
    [,pr pr]
    [,x x]
    [,c c]
    [(quote ,d) `(quote ,d)]
    [(if ,e0 ,e1 ,e2) `(if ,(Expr e0) ,(Expr e1) ,(Expr e2))]
    [(or ,e* ...) `(or ,(map Expr e*) ...)]
    [(and ,e* ...) `(and ,(map Expr e*) ...)]
    [(not ,e) `(not ,(Expr e))]
    [(begin ,e* ... ,e) `(begin ,(map Expr e*) ... ,(Expr e))]
    [(lambda (,x* ...) ,body* ... ,body)
     `(lambda (,x* ...) ,(map Expr body*) ... ,(Expr body))]
    [(let ([,x* ,e*] ...) ,body* ... ,body)
     `(let ([,x* ,(map Expr e*)] ...)
        ,(map Expr body*) ... ,(Expr body))]
    [(letrec ([,x* ,e*] ...) ,body* ... body)
     `(letrec ([,x* ,(map Expr e*)] ...)
        ,(map Expr body*) ... ,(Expr body))]
    [(set! ,x ,e) `(set! ,x ,(Expr e))]
    [(,e ,e* ...) `(,(Expr e) ,(map Expr e*) ...)])
  (Expr e))
```

The pass definition starts with a name (in this case, remove-one-armed-if) and a signature. The signature starts with an input-language specifier (e.g. Lsrc), along with a list of formals. Here, there is just one formal, e, for the input-language term. The second part of the signature has an output-language specifier (in this case, L1), as well as a list of extra return values (in this case, empty).

Following the name and signature, is an optional definitions clause, not used in this pass. The definitions clause can contain any Scheme expression valid in a definition context.

Next, a transformer from the input nonterminal Expr to the output nonterminal Expr is defined. The transformer is named Expr and has a signature similar to that of the pass, with an input-language nonterminal and list of formals followed by the output-language nonterminal and list of extra-return-value expressions.

The transformer has a clause that processes each production of the Expr nonterminal. Each clause consists of an input pattern, an optional guard clause, and one or more expressions that specify zero or more return values based on the signature. The input pattern is derived from the S-expression productions specified in the input language. Each variable in the pattern is denoted by unquote (,). For instance, the clause for the set! production matches the pattern (set! ,x ,e), binds x to the symbol specified by the set! and e to the Expr specified by the set!.

The variable names used in pattern bindings are based on the meta-variables listed in the language definition. This allows the pattern to be further restricted. For instance, if we wanted to match only set! forms that had a variable reference as the RHS, we could specify our pattern as (set! ,x0 ,x1), which would be equivalent of using our original pattern with the guard clause: (guard (symbol? e)).

The output-language expression is constructed using the `(set! ,x ,(Expr e)) quasiquoted template. Here, quasiquote, (`), is rebound to a form that can construct language forms based on the template, and unquote (,), is used to escape back into Scheme. The ,(Expr e) thus puts the result of the recursive call of Expr into the output-language (set! x e) form.

Following the Expr transformer is the body of the pass, which calls Expr to transform the Lsrc Expr term into an L1 Expr term and wraps the result in a let expression if any structured quoted datum are found in the program that is being compiled.

In place of the explicit recursive calls to Expr, the compiler writer can use the catamorphism syntax to indicate the recurrence, as in the following version of the pass.

```
(define-pass remove-one-armed-if : Lsrc (e) -> L1 ()
  (Expr : Expr (e) -> Expr ()
    [(if ,[e0] ,[e1]) `(if ,e0 ,e1 (void))]
    [,pr pr]
    [,x x]
    [,c c]
    [(quote ,d) `(quote ,d)]
    [(if ,[e0] ,[e1] ,[e2]) `(if ,e0 ,e1 ,e2)]
    [(or ,[e*] ...) `(or ,e* ...)]
    [(and ,[e*] ...) `(and ,e* ...)]
    [(not ,[e]) `(not ,e)]
    [(begin ,[e*] ... ,[e]) `(begin ,e* ... ,e)]
    [(lambda (,x* ...) ,[body*] ... ,[body])
     `(lambda (,x* ...) ,body* ... ,body)]
    [(let ([,x* ,[e*]] ...) ,[body*] ... ,[body])
     `(let ([,x* ,e*] ...)
       ,body* ... ,body)]
    [(letrec ([,x* ,[e*]] ...) ,[body*] ... ,[body])
     `(letrec ([,x* ,e*] ...)
        ,body* ... ,body)]
    [(set! ,x ,[e]) `(set! ,x ,e)]
    [(,[e] ,[e*] ...) `(,e ,e* ...)])
  (Expr e))
```

Here, the square brackets that wrap the unquoted variable expression in a pattern indicate that a catamorphism should be applied. For instance, in the set! clause, the ,e from the previous pass becomes ,[e]. When the catamorphism is included on an element that is followed by an ellipsis, map is used to process the elements of the list and to construct the output list.

Using a catamorphism changes, slightly, the meaning of the meta-variables used in the pattern matcher. Instead of indicatinng a input language restriction that must be met, it indicates an output type that is expected. In the set! clause example, we use e for both, because our input language and output language both use e to refer to their Expr nonterminal. The nanopass framwork uses the input type and the output type, along with any additional input values and extra expected return values to determine which transformer should be called. In some cases, specifically where a single input nonterminal form is transformed into an equivalent output nonterminal form, these transformers can be autogenerated by the framework.

Using catamorphisms helps to make the pass more succinct, but there is still boilerplate code in the pass that the framework can fill in for the compiler writer. Several clauses simply match the input-language production and generate a matching output-language production (modulo the catamorphisms for nested Expr forms). Because the input and output languages are defined, the define-pass macro can automatically generate these clauses. Thus, the same functionality can be expressed as follows:

```
(define-pass remove-one-armed-if : Lsrc (e) -> L1 ()
  (Expr : Expr (e) -> Expr ()
    [(if ,[e0] ,[e1]) `(if ,e0 ,e1 (void))]))
```

In this version of the pass, only the one-armed-`if` form is explicitly processed. The `define-pass` form automatically generates the other clauses. Although all three versions of this pass perform the same task, the final form is the closest to the initial intention of replacing just the one-armed-if form with a two-armed-if.

In addition to `define-pass` autogenerating the clauses of a transformer, `define-pass` can also autogenerate the transformers for nonterminals that must be traversed but are otherwise unchanged in a pass. For instance, one of the passes in the class compiler removes complex expressions from the right-hand side of the `set!` form. At this point in the compiler, the language has several nonterminals:

```
(define-language L18
  (entry Program)
  (terminals
    (integer-64 (i))
    (effect+internal-primitive (epr))
    (non-alloc-value-primitive (vpr))
    (symbol (x l))
    (predicate-primitive (ppr))
    (constant (c)))
  (Program (prog)
    (labels ([l* le*] ...) l))
  (SimpleExpr (se)
    x
    (label l)
    (quote c))
  (Value (v body)
    (alloc i se)
    se
    (if p0 v1 v2)
    (begin e* ... v)
    (primcall vpr se* ...)
    (se se* ...))
  (Effect (e)
    (set! x v)
    (nop)
    (if p0 e1 e2)
    (begin e* ... e)
    (primcall epr se* ...)
    (se se* ...))
  (Predicate (p)
    (true)
    (false)
    (if p0 p1 p2)
    (begin e* ... p)
    (primcall ppr se* ...))
  (LocalsBody (lbody)
    (locals (x* ...) body))
  (LambdaExpr (le)
    (lambda (x* ...) lbody)))
```

The pass, however, is only interested in the `set!` form and the `Value` form in the right-hand-side position of the `set!` form. Relying on the autogeneration of transformers, this pass can be written as:

```
(define-pass flatten-set! : L18 (e) -> L19 ()
  (SimpleExpr : SimpleExpr (se) -> SimpleExpr ())
  (Effect : Effect (e) -> Effect ()
    [(set! ,x ,v) (flatten v x)])
  (flatten : Value (v x) -> Effect ()
    [,se ‘(set! ,x ,(SimpleExpr se))]
    [(primcall ,vpr ,[se*] ...) `(set! ,x (primcall ,vpr ,se* ...))]
    [(alloc ,i ,[se]) `(set! ,x (alloc ,i ,se))]
    [(,[se] ,[se*] ...) `(set! ,x (,se ,se* ...))]))
```

Here, the `Effect` transformer has just one clause for matching the `set!` form. The `flatten` transformer is called to produce the final `Effect` form. The `flatten` transformer, in turn, pushes the `set!` form into the `if` and `begin` forms and processes the contents of these forms, which produces a final `Effect` form. Note that the `if` and `begin` forms do not need to be provided by the compiler writer. This is because the input and output language provide enough structure that the nanopass framework can automatically generate the appropriate clauses. In the case of `begin` it will push the `set!` form into the final, value producing, expression of the `begin` form. In the case of the `if` it will push the `set!` form into both the consquent and alternative of the if form, setting the variable at the final, value producing expression on both possible execution paths. The `define-pass` macro autogenerates transformers for `Program`, `LambdaExpr`, `LocalsBody`, `Value`, and `Predicate` that recur through the input-language forms and produce the output-language forms. The `SimpleExpr` transformer only needs to be written to give a name to the transformer so that it can be called by `flatten`.

It is sometimes necessary to pass more information than just the language term to a transformer. The transformer syntax allows extra formals to be named to support passing this information. For example, in the pass from the scheme to C compiler that converts the `closures` form into explicit calls to procedure primitives, the closure pointer, `cp`, and the list of free variables, `free*`, are passed to the `Expr` transformer.

```
(define-pass expose-closure-prims : L12 (e) -> L13 ()
  (Expr : Expr (e [cp #f] [free* '()]) -> Expr ()
    (definitions
      (define handle-closure-ref
        (lambda (x cp free*)
          (let loop ([free* free*] [i 0])
            (cond
              [(null? free*) x]
              [(eq? x (car free*)) `(primcall closure-ref ,cp (quote ,i))]
              [else (loop (cdr free*) (fx+ i 1))]))))
      (define build-closure-set*
        (lambda (x* l* f** cp free*)
          (fold-left
            (lambda (e* x l f*)
              (let loop ([f* f*] [i 0] [e* e*])
                (if (null? f*)
                    (cons `(primcall closure-code-set! ,x (label ,l)) e*)
                    (loop (cdr f*) (fx+ i 1)
                      (cons `(primcall closure-data-set! ,x (quote ,i)
                                  ,(handle-closure-ref (car f*) cp free*))
                        e*)))))
            '()
            x* l* f**))))
    [(closures ([,x* ,l* ,f** ...] ...)
       (labels ([,l2* ,[le*]] ...) ,[body]))
     (let ([size* (map length f**)])
       `(let ([,x* (primcall make-closure (quote ,size*))] ...)
          (labels ([,l2* ,le*] ...)
            (begin
              ,(build-closure-set* x* l* f** cp free*) ...
              ,body))))]
    [,x (handle-closure-ref x cp free*)]
    [((label ,l) ,[e*] ...) `((label ,l) ,e* ...)]
    [(,[e] ,[e*] ...) `((primcall closure-code ,e) ,e* ...)])
 (LabelsBody : LabelsBody (lbody) -> Expr ())
 (LambdaExpr : LambdaExpr (le) -> LambdaExpr ()
   [(lambda (,x ,x* ...) (free (,f* ...) ,[body x f* -> body]))
    `(lambda (,x ,x* ...) ,body)]))
```

The catamorphism and clause autogeneration facilities are also aware of the extra formals expected by transformers. In a catamorphism, this means that extra arguments need not be specified in the catamorphism, if the formals are available in the transformer. For instance, in the `Expr` transformer, the catamorphism specifies only the binding of the output `Expr` form, and `define-pass` matches the name of the formal to the transformer with the expected argument. In the `LambdaExpr` transformer, the extra arguments need to be specified, both because they are not available as a formal of the transformer and because the values change at the `LambdaExpr` boundary. Autogenerated clauses in `Expr` also call the `Expr` transformer with the extra arguments from the formals.

The `expose-closure-prims` pass also specifies default values for the extra arguments passed to the `Expr` transformer. It defaults the `cp` variable to `#f` and the `free*` variable to the empty list. The default values will only be used in calls to the `Expr` transformer when the no other value is available. In this case, this happen only when the `Expr` transformer is first called in the body of the pass. This is consistent with the body of the program, which cannot contain any free variables and hence does not need a closure pointer. Once we begin processing within the body of a `lambda` we then have a closure pointer, with the list of free variables, if any.

Sometimes it is also necessary for a pass to return more than one value. The nanopass framework relies upon Scheme's built-in functionality for dealing with returning of multiple return values. To inform the nanopass framework that a given transformer is returning more than one value, we use the signature to tell the framework both how many values we are expecting to return, and what the default values should be when a clause is autogenerated. For instance, the `uncover-free` pass returns two values, the language form and the list of free variables.

```
(define-pass uncover-free : L10 (e) -> L11 ()
  (Expr : Expr (e) -> Expr (free*)
    [(quote ,c) (values `(quote ,c) '())]
    [,x (values x (list x))]
    [(let ([,x* ,[e* free**]] ...) ,[e free*])
     (values `(let ([,x* ,e*] ...) ,e)
       (apply union (difference free* x*) free**))]
    [(letrec ([,x* ,[le* free**]] ...) ,[body free*])
     (values `(letrec ([,x* ,le*] ...) ,body)
       (difference (apply union free* free**) x*))]
    [(if ,[e0 free0*] ,[e1 free1*] ,[e2 free2*])
     (values `(if ,e0 ,e1 ,e2) (union free0* free1* free2*))]
    [(begin ,[e* free**] ... ,[e free*])
     (values `(begin ,e* ... ,e) (apply union free* free**))]
    [(primcall ,pr ,[e* free**]...)
     (values `(primcall ,pr ,e* ...) (apply union free**))]
    [(,[e free*] ,[e* free**] ...)
     (values `(,e ,e* ...) (apply union free* free**))]])
  (LambdaExpr : LambdaExpr (le) -> LambdaExpr (free*)
    [(lambda (,x* ...) ,[body free*])
     (let ([free* (difference free* x*)])
       (values `(lambda (,x* ...) (free (,free* ...) ,body)) free*))])
  (let-values ([(e free*) (Expr e)])
    (unless (null? free*) (error who "found unbound variables" free*))
    e))
```

Transformers can also be written that handle terminals instead of nonterminals. Because terminals have no structure, the body of such transformers is simply a Scheme expression. The Scheme to C compiler does not make use of this feature, but we could imagine a pass where references to variables are replaced with already specified locations, such as the following pass:

```
(define-pass replace-variable-refereces : L23 (x) -> L24 ()
  (uvar-reg-fv : symbol (x env) -> location ()
    (cond [(and (uvar? x) (assq x env)) => cdr] [else x]))
  (SimpleExpr : SimpleExpr (x env) -> Triv ())
  (Rhs : Rhs (x env) -> Rhs ())
  (Pred : Pred (x env) -> Pred ())
  (Effect : Effect (x env) -> Effect ())
  (Value : Value (x env) -> Value ())
  (LocalsBody : LocalsBody (x) -> Value ()
    [(finished ([,x* ,loc*] ...) ,vbody) (Value vbody (map cons x* loc*))]))
```

The two interesting parts of this pass are the `LocalsBody` transformer that creates the environment that maps variables to locations and the `uvar-reg-fv` transformer that replaces variables with the appropriate location. In this pass, transformers cannot be autogenerated because extra arguments are needed, and the nanopass framework only autogenerates transformers without extra arguments or return values. The autogeneration is limited to help reign in some of the unpredictable behavior that can result from autogenerated transformers.

Passes can also be written that do not take a language form but that produce a language form. The initial parser for the Scheme to C compiler is a good example of this. It expects an S-expression that conforms to an input grammar for the subset of Scheme supported by the compiler.

```
(define-pass parse-and-rename : * (e) -> Lsrc ()
  (definitions
    (define process-body
      (lambda (who env body* f)
        (when (null? body*) (error who "invalid empty body"))
        (let loop ([body (car body*)] [body* (cdr body*)] [rbody* '()])
          (if (null? body*)
              (f (reverse rbody*) (Expr body env))
              (loop (car body*) (cdr body*)
                (cons (Expr body env) rbody*))))))
    (define vars-unique?
      (lambda (fmls)
        (let loop ([fmls fmls])
          (or (null? fmls)
              (and (not (memq (car fmls) (cdr fmls)))
                   (loop (cdr fmls)))))))
    (define unique-vars
      (lambda (env fmls f)
        (unless (vars-unique? fmls)
          (error 'unique-vars "invalid formals" fmls))
        (let loop ([fmls fmls] [env env] [rufmls '()])
          (if (null? fmls)
              (f env (reverse rufmls))
              (let* ([fml (car fmls)] [ufml (unique-var fml)])
                (loop (cdr fmls) (cons (cons fml ufml) env)
                  (cons ufml rufmls)))))))
    (define process-bindings
      (lambda (rec? env bindings f)
        (let loop ([bindings bindings] [rfml* '()] [re* '()])
          (if (null? bindings)
              (unique-vars env rfml*
                (lambda (new-env rufml*)
                  (let ([env (if rec? new-env env)])
                    (let loop ([rufml* rufml*]
                               [re* re*]
```

```
                            [ufml* '()]
                            [e* '()])
                  (if (null? rufml*)
                      (f new-env ufml* e*)
                      (loop (cdr rufml*) (cdr re*)
                        (cons (car rufml*) ufml*)
                        (cons (Expr (car re*) env) e*)))))))
          (let ([binding (car bindings)])
            (loop (cdr bindings) (cons (car binding) rfml*)
              (cons (cadr binding) re*)))))))
(define Expr*
  (lambda (e* env)
    (map (lambda (e) (Expr e env)) e*)))
(with-output-language (Lsrc Expr)
  (define build-primitive
    (lambda (as)
      (let ([name (car as)] [argc (cdr as)])
        (cons name
          (if (< argc 0)
              (error who
                "primitives with arbitrary counts are not currently supported"
                name)
              (lambda (env . e*)
                (if (= (length e*) argc)
                    `(,name ,(Expr* e* env) ...)
                    (error name "invalid argument count"
                      (cons name e*)))))))))
  (define initial-env
    (cons*
      (cons 'quote (lambda (env d)
                     (unless (datum? d)
                       (error 'quote "invalid datum" d))
                     `(quote ,d)))
      (cons 'if (case-lambda
                  [(env e0 e1) `(if ,(Expr e0 env) ,(Expr e1 env))]
                  [(env e0 e1 e2)
                    `(if ,(Expr e0 env) ,(Expr e1 env) ,(Expr e2 env))]
                  [x (error 'if (if (< (length x) 3)
                                    "too few arguments"
                                    "too many arguments")
                       x)]))
      (cons 'or (lambda (env . e*) `(or ,(Expr* e* env) ...)))
      (cons 'and (lambda (env . e*) `(and ,(Expr* e* env) ...)))
      (cons 'not (lambda (env e) `(not ,(Expr e env))))
      (cons 'begin (lambda (env . e*)
                     (process-body env e*
                       (lambda (e* e)
                         `(begin ,e* ... ,e)))))
      (cons 'lambda (lambda (env fmls . body*)
                      (unique-vars env fmls
                        (lambda (env fmls)
                          (process-body 'lambda env body*
                            (lambda (body* body)
                              `(lambda (,fmls ...)
                                 ,body* ... ,body)))))))
      (cons 'let (lambda (env bindings . body*)
                   (process-bindings #f env bindings
                     (lambda (env x* e*)
                         (process-body ’let env body*
                           (lambda (body* body)
                             ‘(let ([,x* ,e*] ...) ,body* ... ,body))))))
        (cons ’letrec (lambda (env bindings . body*)
                        (process-bindings #t env bindings
                          (lambda (env x* e*)
                            (process-body ’letrec env body*
                              (lambda (body* body)
                                ‘(letrec ([,x* ,e*] ...)
                                   ,body* ... ,body)))))))
        (cons ’set! (lambda (env x e)
                      (cond
                        [(assq x env) =>
                          (lambda (as)
                            (let ([v (cdr as)])
                              (if (symbol? v)
                                  ‘(set! ,v ,(Expr e env))
                                  (error ’set! "invalid syntax"
                                    (list ’set! x e)))))]
                        [else (error ’set! "set to unbound variable"
                                (list ’set! x e))])))
        (map build-primitive user-prims)))
    ;;; App - helper for handling applications.
    (define App
      (lambda (e env)
        (let ([e (car e)] [e* (cdr e)])
          ‘(,(Expr e env) ,(Expr* e* env) ...)))))
(Expr : * (e env) -> Expr ()
  (cond
    [(pair? e)
     (cond
       [(assq (car e) env) =>
        (lambda (as)
          (let ([v (cdr as)])
            (if (procedure? v)
                (apply v env (cdr e))
                (App e env))))]
       [else (App e env)])]
    [(symbol? e)
     (cond
       [(assq e env) =>
        (lambda (as)
          (let ([v (cdr as)])
            (cond
              [(symbol? v) v]
              [(primitive? e) e]
              [else (error who "invalid syntax" e)])))]
       [else (error who "unbound variable" e)])]
    [(constant? e) e]
    [else (error who "invalid expression" e)]))
(Expr e initial-env))
```

The `parse-and-rename` pass is structured similarly to a simple expander with keywords and primitives.[3] It also performs syntax checking to ensure that the input grammar conforms to the expected input grammar. Finally, it produces an `Lsrc` language term that represents the Scheme program to be compiled.

[3] It could easily be extended to handle simple macros, in this case, just the fixed `and` macro, `or` macro, and `not` macro would be available.

In the pass syntax, the * in place of the input-language name indicates that no input-language term should be expected. The Expr and Application transformers do not have pattern matching clauses, as the input could be of any form. The quasiquote is, however, rebound because an output language is specified.

It can also be useful to create passes without an output language. The final pass of the Scheme to C compiler is the code generator that emits C code.

```
(define-pass generate-c : L22 (e) -> * ()
  (definitions
    (define string-join
      (lambda (str* jstr)
        (cond
          [(null? str*) ""]
          [(null? (cdr str*)) (car str*)]
          [else (string-append (car str*) jstr (string-join (cdr str*) jstr))])))
    (define symbol->c-id
      (lambda (sym)
        (let ([ls (string->list (symbol->string sym))])
          (if (null? ls)
              "_"
              (let ([fst (car ls)])
                (list->string
                  (cons
                    (if (char-alphabetic? fst) fst #\_)
                    (map (lambda (c)
                           (if (or (char-alphabetic? c)
                                   (char-numeric? c))
                               c
                               #\_))
                      (cdr ls)))))))))
    (define format-function-header
      (lambda (l x*)
        (format "ptr ~a(~a)" l
          (string-join
            (map
              (lambda (x)
                (format "ptr ~a" (symbol->c-id x)))
              x*)
            ", "))))
    (define format-label-call
      (lambda (l se*)
        (format "  ~a(~a)" (symbol->c-id l)
          (string-join
            (map (lambda (se)
                   (format "(ptr)~a" (format-simple-expr se)))
              se*)
            ", "))))
    (define format-general-call
      (lambda (se se*)
        (format "((ptr (*)(~a))~a)(~a)"
          (string-join (make-list (length se*) "ptr") ", ")
          (format-simple-expr se)
          (string-join
            (map (lambda (se)
                   (format "(ptr)~a" (format-simple-expr se)))
              se*)
            ", "))))
    (define format-binop
      (lambda (op se0 se1)
        (format "((long)~a ~a (long)~a)"
          (format-simple-expr se0)
          op
          (format-simple-expr se1))))
    (define format-set!
      (lambda (x rhs)
        (format "~a = (ptr)~a" (symbol->c-id x) (format-rhs rhs)))))
  (emit-function-decl : LambdaExpr (le l) -> * ()
    [(lambda (,x* ...) ,lbody)
     (printf "~a;~%" (format-function-header l x*))])
  (emit-function-def : LambdaExpr (le l) -> * ()
    [(lambda (,x* ...) ,lbody)
     (printf "~a {~%" (format-function-header l x*))
     (emit-function-body lbody)
      (printf "}~%~%")])
  (emit-function-body : LocalsBody (lbody) -> * ()
    [(locals (,x* ...) ,body)
     (for-each (lambda (x) (printf "  ptr ~a;~%" (symbol->c-id x))) x*)
     (emit-value body x*)])
  (emit-value : Value (v locals*) -> * ()
    [(if ,p0 ,v1 ,v2)
     (printf "  if (~a) {~%" (format-predicate p0))
     (emit-value v1 locals*)
     (printf "  } else {~%")
     (emit-value v2 locals*)
     (printf "  }~%")]
    [(begin ,e* ... ,v)
     (for-each emit-effect e*)
     (emit-value v locals*)]
    [,rhs (printf "  return (ptr)~a;\n" (format-rhs rhs))])
  (format-predicate : Predicate (p) -> * (str)
    [(if ,p0 ,p1 ,p2)
     (format "((~a) ? (~a) : (~a))"
       (format-predicate p0)
       (format-predicate p1)
       (format-predicate p2))]
    [(<= ,se0 ,se1) (format-binop "<=" se0 se1)]
    [(< ,se0 ,se1) (format-binop "<" se0 se1)]
    [(= ,se0 ,se1) (format-binop "==" se0 se1)]
    [(true) "1"]
    [(false) "0"]
    [(begin ,e* ... ,p)
     (string-join
       (fold-right (lambda (e s*) (cons (format-effect e) s*))
         (list (format-predicate p)) e*)
       ", ")])
  (format-effect : Effect (e) -> * (str)
    [(if ,p0 ,e1 ,e2)
     (format "((~a) ? (~a) : (~a))"
       (format-predicate p0)
       (format-effect e1)
       (format-effect e2))]
    [((label ,l) ,se* ...) (format-label-call l se*)]
    [(,se ,se* ...) (format-general-call se se*)]
    [(set! ,x ,rhs) (format-set! x rhs)]
    [(nop) "0"]
    [(begin ,e* ... ,e)
```

```
   (string-join
     (fold-right (lambda (e s*) (cons (format-effect e) s*))
       (list (format-effect e)) e*)
     ", ")]
  [(mset! ,se0 ,se1? ,i ,se2)
   (if se1?
       (format "((*((ptr*)((long)~a + (long)~a + ~d))) = (ptr)~a)"
         (format-simple-expr se0) (format-simple-expr se1?)
         i (format-simple-expr se2))
       (format "((*((ptr*)((long)~a + ~d))) = (ptr)~a)"
         (format-simple-expr se0) i (format-simple-expr se2)))])
(format-simple-expr : SimpleExpr (se) -> * (str)
  [,x (symbol->c-id x)]
  [,i (number->string i)]
  [(label ,l) (format "(*~a)" (symbol->c-id l))]
  [(logand ,se0 ,se1) (format-binop "&" se0 se1)]
  [(shift-right ,se0 ,se1) (format-binop ">>" se0 se1)]
  [(shift-left ,se0 ,se1) (format-binop "<<" se0 se1)]
  [(divide ,se0 ,se1) (format-binop "/" se0 se1)]
  [(multiply ,se0 ,se1) (format-binop "*" se0 se1)]
  [(subtract ,se0 ,se1) (format-binop "-" se0 se1)]
  [(add ,se0 ,se1) (format-binop "+" se0 se1)]
  [(mref ,se0 ,se1? ,i)
   (if se1?
       (format "(*((ptr)((long)~a + (long)~a + ~d)))"
         (format-simple-expr se0)
         (format-simple-expr se1?) i)
       (format "(*((ptr)((long)~a + ~d)))" (format-simple-expr se0) i))])
;; prints expressions in effect position into C statements
(emit-effect : Effect (e) -> * ()
  [(if ,p0 ,e1 ,e2)
   (printf "  if (~a) {~%" (format-predicate p0))
   (emit-effect e1)
   (printf "  } else {~%")
   (emit-effect e2)
   (printf "  }~%")]
  [((label ,l) ,se* ...) (printf "  ~a;\n" (format-label-call l se*))]
  [(,se ,se* ...) (printf "  ~a;\n" (format-general-call se se*))]
  [(set! ,x ,rhs) (printf "  ~a;\n" (format-set! x rhs))]
  [(nop) (if #f #f)]
  [(begin ,e* ... ,e)
   (for-each emit-effect e*)
   (emit-effect e)]
  [(mset! ,se0 ,se1? ,i ,se2)
   (if se1?
       (printf "(*((ptr*)((long)~a + (long)~a + ~d))) = (ptr)~a;\n"
         (format-simple-expr se0) (format-simple-expr se1?)
         i (format-simple-expr se2))
       (printf "(*((ptr*)((long)~a + ~d))) = (ptr)~a;\n"
         (format-simple-expr se0) i (format-simple-expr se2)))])
;; formats the right-hand side of a set! into a C expression
(format-rhs : Rhs (rhs) -> * (str)
  [((label ,l) ,se* ...) (format-label-call l se*)]
  [(,se ,se* ...) (format-general-call se se*)]
  [(alloc ,i ,se)
   (if (use-boehm?)
       (format "(ptr)((long)GC_MALLOC(~a) + ~dl)"
         (format-simple-expr se) i)
```

```
      (format "(ptr)((long)malloc(~a) + ~dl)"
       (format-simple-expr se) i))]
   [,se (format-simple-expr se)])
  ;; emits a C program for our progam expression
  (Program : Program (p) -> * ()
   [(labels ([,l* ,le*] ...) ,l)
    (let ([l (symbol->c-id l)] [l* (map symbol->c-id l*)])
     (define-syntax emit-include
      (syntax-rules ()
       [(_ name) (printf "#include <~s>\n" 'name)]))
     (define-syntax emit-predicate
      (syntax-rules ()
       [(_ PRED_P mask tag)
        (emit-c-macro PRED_P (x) "(((long)x & ~d) == ~d)" mask tag)]))
     (define-syntax emit-eq-predicate
      (syntax-rules ()
       [(_ PRED_P rep)
        (emit-c-macro PRED_P (x) "((long)x == ~d)" rep)]))
     (define-syntax emit-c-macro
      (lambda (x)
       (syntax-case x()
        [(_ NAME (x* ...) fmt args ...)
         #'(printf "#define ~s(~a) ~a\n" 'NAME
             (string-join (map symbol->string '(x* ...)) ", ")
             (format fmt args ...))])))
     ;; the following printfs output the tiny C runtime we are using
     ;; to wrap the result of our compiled Scheme program.
     (emit-include stdio.h)
     (if (use-boehm?)
       (emit-include gc.h)
       (emit-include stdlib.h))
     (emit-predicate FIXNUM_P fixnum-mask fixnum-tag)
     (emit-predicate PAIR_P pair-mask pair-tag)
     (emit-predicate BOX_P box-mask box-tag)
     (emit-predicate VECTOR_P vector-mask vector-tag)
     (emit-predicate PROCEDURE_P closure-mask closure-tag)
     (emit-eq-predicate TRUE_P true-rep)
     (emit-eq-predicate FALSE_P false-rep)
     (emit-eq-predicate NULL_P null-rep)
     (emit-eq-predicate VOID_P void-rep)
     (printf "typedef long* ptr;\n")
     (emit-c-macro FIX (x) "((long)x << ~d)" fixnum-shift)
     (emit-c-macro UNFIX (x) "((long)x >> ~d)" fixnum-shift)
     (emit-c-macro UNBOX (x) "((ptr)*((ptr)((long)x - ~d)))" box-tag)
     (emit-c-macro VECTOR_LENGTH_S (x) "((ptr)*((ptr)((long)x - ~d)))" vector-tag)
     (emit-c-macro VECTOR_LENGTH_C (x) "UNFIX(VECTOR_LENGTH_S(x))")
     (emit-c-macro VECTOR_REF (x i) "((ptr)*((ptr)((long)x - ~d + ((i+1) * ~d))))"
      vector-tag word-size)
     (emit-c-macro CAR (x) "((ptr)*((ptr)((long)x - ~d)))" pair-tag)
     (emit-c-macro CDR (x) "((ptr)*((ptr)((long)x - ~d + ~d)))" pair-tag word-size)
     (printf "void print_scheme_value(ptr x) {\n")
     (printf "  long i, veclen;\n")
     (printf "  ptr p;\n")
     (printf "  if (TRUE_P(x)) {\n")
     (printf "    printf(\"#t\");\n")
     (printf "  } else if (FALSE_P(x)) {\n")
     (printf "    printf(\"#f\");\n")
     (printf "  } else if (NULL_P(x)) {\n")
```

```
       (printf "    printf(\"()\");\n")
       (printf "  } else if (VOID_P(x)) {\n")
       (printf "    printf(\"(void)\");\n")
       (printf "  } else if (FIXNUM_P(x)) {\n")
       (printf "    printf(\"%ld\", UNFIX(x));\n")
       (printf "  } else if (PAIR_P(x)) {\n")
       (printf "    printf(\"(\");\n")
       (printf "    for (p = x; PAIR_P(p); p = CDR(p)) {\n")
       (printf "      print_scheme_value(CAR(p));\n")
       (printf "      if (PAIR_P(CDR(p))) { printf(\" \"); }\n")
       (printf "    }\n")
       (printf "    if (NULL_P(p)) {\n")
       (printf "      printf(\")\");\n")
       (printf "    } else {\n")
       (printf "      printf(\" . \");\n")
       (printf "      print_scheme_value(p);\n")
       (printf "      printf(\")\");\n")
       (printf "    }\n")
       (printf "  } else if (BOX_P(x)) {\n")
       (printf "    printf(\"#(box \");\n")
       (printf "    print_scheme_value(UNBOX(x));\n")
       (printf "    printf(\")\");\n")
       (printf "  } else if (VECTOR_P(x)) {\n")
       (printf "    veclen = VECTOR_LENGTH_C(x);\n")
       (printf "    printf(\"#(\");\n")
       (printf "    for (i = 0; i < veclen; i += 1) {\n")
       (printf "      print_scheme_value(VECTOR_REF(x,i));\n")
       (printf "      if (i < veclen) { printf(\" \"); } \n")
       (printf "    }\n")
       (printf "    printf(\")\");\n")
       (printf "  } else if (PROCEDURE_P(x)) {\n")
       (printf "    printf(\"#(procedure)\");\n")
       (printf "  }\n")
       (printf "}\n")
       (map emit-function-decl le* l*)
       (map emit-function-def le* l*)
       (printf "int main(int argc, char * argv[]) {\n")
       (printf "  print_scheme_value(~a());\n" l)
       (printf "  printf(\"\\n\");\n")
       (printf "  return 0;\n")
       (printf "}\n"))]))
```

Again, a * is used to indicate that there is no language form in this case for the output language. The C code is printed to the standard output port. Thus, there is no need for any return value from this pass.

Passes can also return a value that is not a language form. For instance, we could write the `simple?` predicate from `purify-letrec` pass as its own pass, rather than using the `nanopass-case` form. It would look something like the following:

```
(define-pass simple? : (L8 Expr) (e bound* assigned*) -> * (bool)
  (simple? : Expr (e) -> * (bool)
    [(quote ,c) #t]
    [,x (not (or (memq x bound*) (memq x assigned*)))]
    [(primcall ,pr ,e* ...)
     (and (effect-free-prim? pr) (for-all simple? e*))]
    [(begin ,e* ... ,e) (and (for-all simple? e*) (simple? e))]
    [(if ,e0 ,e1 ,e2) (and (simple? e0) (simple? e1) (simple? e2))]
    [else #f])
  (simple? e))
```

Here, the extra return value is indicated as `bool`. The `bool` here is used to indicate to `define-pass` that an extra value is being returned. Any expression can be used in this position. In this case, the `bool` identifier will simply be an unbound variable if it is ever manifested. It is not manifested in this case, however, because the body is explicitly specified; thus, no code will be autogenerated for the body of the pass.

2.3.1 The `define-pass` syntactic form

The `define-pass` form has the following syntax.

```
(define-pass name : lang-specifier (fml ...) -> lang-specifier (extra-return-val-expr ...)
  definitions-clause
  transformer-clause ...
  body-expr ...)
```

where *name* is an identifier to use as the name for the procedure definition. The *lang-specifier* has one of the following forms:

```
*
lang-name
(lang-name nonterminal-name)
```

where

- *lang-name* refers to a language defined with the `define-language` form, and
- *nonterminal-name* refers to a nonterminal named within the language definition.

When the `*` form is used as the input *lang-specifier*, it indicates that the pass does not expect an input-language term. When there is no input language, the transformers within the pass do not have clauses with pattern matches because, without an input language, the `define-pass` macro does not know what the structure of the input term will be. When the `*` form is used as the output *lang-specifier*, it indicates that the pass does not produce an output-language term and should not be checked. When there is no output language, the transformers within the pass do not bind `quasiquote`, and there are no templates on the right-hand side of the transformer matches. It is possible to use the `*` specifier for both the input and output *lang-specifier*. This effectively turns the pass, and the transformers contained within it, into an ordinary Scheme function.

When the *lang-name* form is used as the input *lang-specifier*, it indicates that the pass expects an input-language term that is one of the productions from the entry nonterminal. When the *lang-name* form is used as the output *lang-specifier*, it indicates that the pass expects that an output-language term will be produced and checked to be one of the records that represents a production of the entry nonterminal.

When the (*lang-name nonterminal-name*) form is used as the input-language specifier, it indicates that the input-language term will be a production from the specified nonterminal in the specified input language. When the (*lang-name nonterminal-name*) form is used as the output-language specifier, it indicates that the pass will produce an output production from the specified nonterminal of the specified output language.

The *fml* is a Scheme identifier, and if the input *lang-specifier* is not `*`, the first *fml* refers to the input-language term.

The *extra-return-val-expr* is any valid Scheme expression that is valid in value context. These expressions are scoped within the binding of the identifiers named as *fml*s.

The optional *definitions-clause* has the following form:

```
(definitions scheme-definition ...)
```

where *scheme-definition* is any Scheme expression that can be used in definition context. Definitions in the *definitions-clause* are in the same lexical scope as the transformers, which means that procedures and macros defined in the *definitions-clause* can refer to any transformer named in a *transformer-clause*.

The *definitions-clause* is followed by zero or more *transformer-clauses*s of the following form:

```
(name : nt-specifier (fml-expr ...) -> nt-specifier (extra-return-val-expr ...)
  definitions-clause?
  transformer-body)
```

where *name* is a Scheme identifier that can be used to refer to the transformer within the pass. The input *nt-specifier* is one of the following two forms:

```
*
nonterminal-name
```

When the * form is used as the input nonterminal, it indicates that no input nonterminal form is expected and that the body of the *transformer-body* will not contain pattern matching clauses. When the * form is used as the output nonterminal, `quasiquote` will not be rebound, and no output-language templates are available. When both the input and output *nt-specifier* are *, the transformer is effectively an ordinary Scheme procedure.

The *fml-expr* has one of the following two forms:

```
fml
[fml default-val-expr]
```

where *fml* is a Scheme identifier and *default-val-expr* is a Scheme expression. The *default-val-expr* is used when an argument is not specified in a catamorphism or when a matching `fml` is not available in the calling transformer. All arguments must be explicitly provided when the transformer is called as an ordinary Scheme procedure. Using the catamorphism syntax, the arguments can be explicitly supplied, using the syntax discussed on page 31. It can also be specified implicitly. Arguments are filled in implicitly in catamorphisms that do not explicitly provide the arguments and in autogenerated clauses when the nonterminal elements of a production are processed. These implicitly supplied formals are handled by looking for a formal in the calling transformer that has the same name as the formal expected by the target transformer. If no matching formal is found, and the target transformer specifies a default value, the default value will be used in the call; otherwise, another target transformer must be found, a new transformer must be autogenerated, or an exception must be raised to indicate that no transformer was found and none can be autogenerated.

The *extra-return-val-expr* can be any Scheme expression. These expressions are scoped within the *fml*s bound by the transformer. This allows an input formal to be returned as an extra return value, implicitly in the autogenerated clauses. This can be useful for threading values through a transformer.

The optional *definitions-clause* can include any Scheme expression that can be placed in a definition context. These definitions are scoped within the transformer. When an output nonterminal is specified, the `quasiquote` is also bound within the body of the `definitions` clause to allow language term templates to be included in the body of the definitions.

When the input *nt-specifier* is not *, the *transformer-body* has one of the following forms:

```
[pattern guard-clause body* ... body]
[pattern body* ... body]
[else body* ... body]
```

where the `else` clause must be the last one listed in a transformer and prevents autogeneration of missing clauses (because the `else` clause is used in place of the autogenerated clauses). The *pattern* is an S-expression pattern, based on the S-expression productions used in the language definition. Patterns can be arbitrarily nested. Variables bound by the pattern are preceded by an `unquote` and are named based on the meta-variables named in the language definition. The variable name can be used to restrict the pattern by using a meta-variable that is more specific than the one specified in the language definition. The *pattern* can also contain catamorphisms that have one of the following forms:

```
[Proc-expr : input-fml arg ... -> output-fml extra-rv-fml  ...]
[Transformer-name : output-fml extra-rv-fml ...]
[input-fml arg ... -> output-fml extra-rv-fml ...]
[output-fml extra-rv-fml ...]
```

In the first form, the *Proc-expr* is an explicitly specified procedure expression, the *input-fml* and all arguments to the procedure are explicitly specified, and the results of calling the *Proc-expr* are bound by the *output-fml* and *extra-rv-fml*s. Note that the *Proc-expr* may be a *Transformer-name*. In the second form, the *Transformer-name* is an identifier that refers to a transformer named in this pass. The `define-pass` macro determines, based on the signature of the transformer referred to by the *Transformer-name*, what arguments should be supplied to the transformer. In the last two forms, the transformer is determined automatically. In the third form, the nonterminal type associated with the *input-fml*, the *arg*s, the output nonterminal type based on the *output-fml*, and the *extra-rv-fml*s are used to determine the transformer to call. In the final form, the nonterminal type for the field within the production, along with the formals to the calling transformer, the output nonterminal type based on the *output-fml*, and the *extra-rv-fml*s are used to determine the transformer to call. In the two forms where the transformer is not explicitly named, a new transformer can be autogenerated when no *arg*s and no *extra-rv-fml*s are specified. This limitation is in place to avoid creating a transformer with extra formals whose use is unspecified and extra return values with potentially dubious return-value expressions.

The *input-fml* is a Scheme identifier with a name based on the meta-variables named in the input-language definition. The specification of a more restrictive meta-variable name can be used to further restrict the pattern. The *output-fml* is a Scheme identifier with a name based on the meta-variables named in the output-language definition. The *extra-rv-fml* is a Scheme identifier. The *input-fml*s named in the fields of a pattern must be unique. The *output-fml*s and *extra-rv-fml*s must also be unique, although they can overlap with the *input-fml*s that are shadowed in the body by the *output-fml* or *extra-rv-fml* with the same name.

Only the *input-fml*s are visible within the optional *guard-clause*. This is because the *guard-clause* is evaluated before the catamorphisms recur on the fields of a production. The *guard-clause* has the following form:

```
(guard guard-expr ...)
```

where *guard-expr* is a Scheme expression. The *guard-clause* has the same semantics as `and`.

The *body** and *body* are any Scheme expression. When the output *nt-specifier* is not `*`, `quasiquote` is rebound to a macro that interprets `quasiquote` expressions as templates for productions in the output nonterminal. Additionally, `in-context` is a macro that can be used to rebind `quasiquote` to a different nonterminal. Templates are specified as S-expressions based on the productions specified by the output language. In templates, `unquote` is used to indicate that the expression in the `unquote` should be used to fill in the given field of the production. Within an `unquote` expression, `quasiquote` is rebound to the appropriate nonterminal based on the expected type of the field in the production. If the template includes items that are not `unquote`d where a field value is expected, the expression found there is automatically quoted. This allows self-evaluating items such as symbols, booleans, and numbers to be more easily specified in templates. A list of items can be specified in a field that expects a list, using an ellipsis.

Although the syntax of a language production is specified as an S-expression, the record representation used for the language term separates each variable specified into a separate field. This means that the template syntax expects a separate value or list of values for each field in the record. For instance, in the `(letrec ([x* e*] ...) body)` production, a template of the form `(letrec (,bindings ...) ,body)` cannot be used because the nanopass framework will not attempt to break up the `bindings` list into its `x*` and `e*` component parts. The template `(letrec ([,(map car bindings) ,(map cadr bindings)] ...) ,body)` accomplishes the same goal, explicitly separating the variables from the expressions. It is possible that the nanopass framework could be extended to perform the task of splitting up the `binding*` list automatically, but it is not done currently, partially to avoid hiding the cost of deconstructing the `binding*` list and constructing the `x*` and `e*` lists.

The `in-context` expression within the body has the following form:

`(in-context` *nonterminal-name body** `...` *body*`)`

The `in-context` form rebinds the `quasiquote` to allow productions from the named nonterminal to be constructed in a context where they are not otherwise expected.

Chapter 3

Working with language forms

3.1 Constructing language forms outside of a pass

In addition to creating language forms using a parser defined with `define-parser` or through a pass defined with `define-pass`, language forms can also be created using the `with-output-language` form. The `with-output-language` form binds the `in-context` transformer for the language specified and, if a nonterminal is also specified, binds the `quasiquote` form. This allows the same template syntax used in the body of a transformer to be used outside of the context of a pass. In a commercial compiler, such as Chez Scheme, it is often convenient to use functional abstraction to centralize the creation of a language term.

For instance, in the `convert-assignments` pass, the `with-output-languge` form is wrapped around the `make-boxes` and `build-let` procedures. This is done so that primitive calls to `box` along with the `let` form of the `L10` language can be constructed with quasiquoted expressions.

```
(with-output-language (L10 Expr)
  (define make-boxes
    (lambda (t*)
      (map (lambda (t) `(primcall box ,t)) t*)))
  (define build-let
    (lambda (x* e* body)
      (if (null? x*)
          body
          `(let ([,x* ,e*] ...) ,body)))))
```

This rebinds both the `quasiquote` keyword and the `in-context` keyword.

The `with-output-language` form has one of the following forms:

(`with-output-language` *lang-name expr** `...` *expr*)
(`with-output-language` (*lang-name nonterminal-name*) *expr** `...` *expr*)

In the first form, the `in-context` form is bound and can be used to specify a *nonterminal-name*, as described at the end of Section 2.3. In the second form, both `in-context` and `quasiquote` are bound. The `quasiquote` form is bound in the context of the specified *nonterminal-name*, and templates can be defined just as they are on the right-hand side of a transformer clause.

The `with-output-language` form is a splicing form, similar to `begin` or `let-syntax`, allowing multiple definitions or expressions that are all at the same scoping level as the `with-output-language` form to be contained within the form. This is convenient when writing a set of definitions that all construct some piece of a language term from the same nonterminal. This flexibility means that the `with-output-language` form cannot be defined as syntactic sugar for the `define-pass` form.

3.2 Matching language forms outside of a pass

In addition to the `define-pass` form, it is possible to match a language term using the `nanopass-case` form. This can be useful when creating functional abstractions, such as predicates that ask a question based on matching a language form. For instance, suppose we write a `lambda?` predicate for the `L8` language as follows:

```
(define lambda?
  (lambda (e)
    (nanopass-case (L8 Expr) e
      [(lambda (,x* ...) ,abody) #t]
      [else #f])))
```

The `nanopass-case` form has the following syntax:

```
(nanopass-case (lang-name nonterminal-name) expr
  matching-clause ...)
```

where *matching-clause* has one of the following forms:

```
[pattern guard-clause expr* ... expr]
[pattern expr* ... expr]
[else expr* ... expr]
```

where the *pattern* and *guard-clause* forms have the same syntax as in the *transformer-body* of a pass.

Similar to `with-output-language`, `nanopass-case` provides a more succinct syntax for matching a language form than does the general `define-pass` form. Unlike the `with-output-language` form, however, the `nanopass-case` form can be implemented in terms of the `define-pass` form. For example, the `lambda?` predicate also could have been written as:

```
(define-pass lambda? : (L8 Expr) (e) -> * (bool)
  (Expr : Expr (e) -> * (bool)
    [(lambda (,x* ...) ,abody) #t]
    [else #f])
  (Expr e))
```

This is, in fact, how the `nanopass-case` macro is implemented.

Chapter 4

Working with languages

4.1 Displaying languages

The `language->s-expression` form can be used to print the full definition of a language by supplying it the language name to be printed. This can be helpful when working with extended languages, such as in the case of `L1`:

```
(language->s-expression L1)
```

which returns:

```
(define-language L1
  (entry Expr)
  (terminals
    (void+primitive (pr))
    (symbol (x))
    (constant (c))
    (datum (d)))
  (Expr (e body)
    pr
    x
    c
    (quote d)
    (if e0 e1 e2)
    (or e* ...)
    (and e* ...)
    (not e)
    (begin e* ... e)
    (lambda (x* ...) body* ... body)
    (let ([x* e*] ...) body* ... body)
    (letrec ([x* e*] ...) body* ... body)
    (set! x e)
    (e e* ...)))
```

4.2 Differencing languages

The extension form can also be derived between any two languages by using the `diff-languages` form. For instance, we can get the differences between the `Lsrc` and `L1` language (giving us back the extension) with:

```
(diff-languages Lsrc L1)
```

which returns:

```
(define-language L1
  (extends Lsrc)
  (entry Expr)
  (terminals
    (- (primitive (pr)))
    (+ (void+primitive (pr))))
  (Expr (e body)
    (- (if e0 e1))))
```

4.3 Viewing the expansion of passes and transformers

The `define-pass` form autogenerates both transformers and clauses within transformers. In simple passes, these are generally straightforward to reason about, but in more complex passes, particularly those that make use of different arguments for different transformers or include extra return values, it can become more difficult to determine what code will be generated. In particular, the experience of developing a full commercial compiler has shown that the `define-pass` form can autogenerate transformers that shadow those defined by the compiler writer. To help the compiler writer determine what code is being generated, there is a variation of the `define-pass` form, called `echo-define-pass`, that will echo the expansion of `define-pass`.

For instance, we can echo the `remove-one-armed-if` pass to get the following:

```
(echo-define-pass remove-one-armed-if : Lsrc (e) -> L1 ()
  (Expr : Expr (e) -> Expr ()
    [(if ,[e0] ,[e1]) `(if ,e0 ,e1 (void))]))
```

⇒

```
pass remove-one-armed-if expanded into:
(define remove-one-armed-if
  (lambda (e)
    (define who 'remove-one-armed-if)
    (define-nanopass-record)
    (define Expr
      (lambda (e)
        (let ([g221.159 e])
          (let-syntax ([quasiquote '#<procedure tmp>]
                       [in-context '#<procedure>])
            (begin
              (let ([rhs.160 (lambda (e0 e1) `(if ,e0 ,e1 (void)))])
                (cond
                  [(primitive? g221.159) g221.159]
                  [(symbol? g221.159) g221.159]
                  [(constant? g221.159) g221.159]
                  [else
                   (let ([tag (nanopass-record-tag g221.159)])
                     (cond
                       [(eqv? tag 4)
                        (let* ([g222.161 (Lsrc:if:Expr.387-e0 g221.159)]
                               [g223.162 (Lsrc:if:Expr.387-e1 g221.159)])
                          (let-values ([(e0) (Expr g222.161)]
                                       [(e1) (Expr g223.162)])
```

```
     (rhs.160 e0 e1)))]
[(eqv? tag 2)
 (make-L1:quote:Expr.400
   'remove-one-armed-if
   (Lsrc:quote:Expr.386-d g221.159)
   "d")]
[(eqv? tag 6)
 (make-L1:if:Expr.401 'remove-one-armed-if
   (Expr (Lsrc:if:Expr.388-e0 g221.159))
   (Expr (Lsrc:if:Expr.388-e1 g221.159))
   (Expr (Lsrc:if:Expr.388-e2 g221.159)) "e0" "e1"
   "e2")]
[(eqv? tag 8)
 (make-L1:or:Expr.402
   'remove-one-armed-if
   (map (lambda (m) (Expr m))
        (Lsrc:or:Expr.389-e* g221.159))
   "e*")]
[(eqv? tag 10)
 (make-L1:and:Expr.403
   'remove-one-armed-if
   (map (lambda (m) (Expr m))
        (Lsrc:and:Expr.390-e* g221.159))
   "e*")]
[(eqv? tag 12)
 (make-L1:not:Expr.404
   'remove-one-armed-if
   (Expr (Lsrc:not:Expr.391-e g221.159))
   "e")]
[(eqv? tag 14)
 (make-L1:begin:Expr.405 'remove-one-armed-if
   (map (lambda (m) (Expr m))
        (Lsrc:begin:Expr.392-e* g221.159))
   (Expr (Lsrc:begin:Expr.392-e g221.159)) "e*"
   "e")]
[(eqv? tag 16)
 (make-L1:lambda:Expr.406 'remove-one-armed-if
   (Lsrc:lambda:Expr.393-x* g221.159)
   (map (lambda (m) (Expr m))
        (Lsrc:lambda:Expr.393-body* g221.159))
   (Expr (Lsrc:lambda:Expr.393-body g221.159)) "x*"
   "body*" "body")]
[(eqv? tag 18)
 (make-L1:let:Expr.407 'remove-one-armed-if
   (Lsrc:let:Expr.394-x* g221.159)
   (map (lambda (m) (Expr m))
        (Lsrc:let:Expr.394-e* g221.159))
   (map (lambda (m) (Expr m))
        (Lsrc:let:Expr.394-body* g221.159))
   (Expr (Lsrc:let:Expr.394-body g221.159)) "x*"
   "e*" "body*" "body")]
[(eqv? tag 20)
 (make-L1:letrec:Expr.408 'remove-one-armed-if
   (Lsrc:letrec:Expr.395-x* g221.159)
   (map (lambda (m) (Expr m))
        (Lsrc:letrec:Expr.395-e* g221.159))
   (map (lambda (m) (Expr m))
        (Lsrc:letrec:Expr.395-body* g221.159))
```

```
                          (Expr (Lsrc:letrec:Expr.395-body g221.159)) "x*"
                          "e*" "body*" "body")]
                       [(eqv? tag 22)
                        (make-L1:set!:Expr.409 'remove-one-armed-if
                          (Lsrc:set!:Expr.396-x g221.159)
                          (Expr (Lsrc:set!:Expr.396-e g221.159)) "x" "e")]
                       [(eqv? tag 24)
                        (make-L1:e:Expr.410 'remove-one-armed-if
                          (Expr (Lsrc:e:Expr.397-e g221.159))
                          (map (lambda (m) (Expr m))
                               (Lsrc:e:Expr.397-e* g221.159))
                          "e" "e*")]
                       [else
                        (error 'remove-one-armed-if
                          "unexpected Expr"
                          g221.159)]))])))))))
    (let ([x (Expr e)])
      (unless ((lambda (x)
                 (or (L1:Expr.399? x)
                     (constant? x)
                     (symbol? x)
                     (void+primitive? x)))
                x)
        (error 'remove-one-armed-if
          (format "expected ~s but got ~s" 'Expr x)))
      x)))
```

This exposes the code generated by `define-pass` but does not expand the language form construction templates. The autogenerated clauses, such as the one that handles `set!`, have a form like the following:

```
[(eqv? tag 7)
 (make-L1:set!:Expr.18
   (Lsrc:set!:Expr.8-x g0.14)
   (Expr (Lsrc:set!:Expr.8-e g0.14)))]
```

Here, the tag of the record is checked and a new output-language record constructed, after recurring to the `Expr` transformer on the `e` field.

The body code also changes slightly, so that the output of the pass can be checked to make sure that it is a valid `L1` `Expr`.

In addition to echoing the output of the entire pass, it is also possible to echo just the expansion of a single transformer by prefixing the transformer with the `echo` keyword.

```
(define-pass remove-one-armed-if : Lsrc (e) -> L1 ()
  (echo Expr : Expr (e) -> Expr ()
    [(if ,[e0] ,[e1]) `(if ,e0 ,e1 (void))]))
```

⇒

```
Expr in pass remove-one-armed-if expanded into:
(define Expr
  (lambda (e)
    (let ([g442.303 e])
      (let-syntax ([quasiquote '#<procedure tmp>]
                   [in-context '#<procedure>])
        (begin
          (let ([rhs.304 (lambda (e0 e1) `(if ,e0 ,e1 (void)))])
            (cond
              [(primitive? g442.303) g442.303]
              [(symbol? g442.303) g442.303]
              [(constant? g442.303) g442.303]
              [else
               (let ([tag (nanopass-record-tag g442.303)])
                 (cond
                   [(eqv? tag 4)
                    (let* ([g443.305 (Lsrc:if:Expr.770-e0 g442.303)]
                           [g444.306 (Lsrc:if:Expr.770-e1 g442.303)])
                      (let-values ([(e0) (Expr g443.305)]
                                   [(e1) (Expr g444.306)])
                        (rhs.304 e0 e1)))]
                   [(eqv? tag 2)
                    (make-L1:quote:Expr.783
                      'remove-one-armed-if
                      (Lsrc:quote:Expr.769-d g442.303)
                      "d")]
                   [(eqv? tag 6)
                    (make-L1:if:Expr.784 'remove-one-armed-if
                      (Expr (Lsrc:if:Expr.771-e0 g442.303))
                      (Expr (Lsrc:if:Expr.771-e1 g442.303))
                      (Expr (Lsrc:if:Expr.771-e2 g442.303)) "e0" "e1"
                      "e2")]
                   [(eqv? tag 8)
                    (make-L1:or:Expr.785
                      'remove-one-armed-if
                      (map (lambda (m) (Expr m))
                           (Lsrc:or:Expr.772-e* g442.303))
                      "e*")]
                   [(eqv? tag 10)
                    (make-L1:and:Expr.786
                      'remove-one-armed-if
                      (map (lambda (m) (Expr m))
                           (Lsrc:and:Expr.773-e* g442.303))
                      "e*")]
                   [(eqv? tag 12)
                    (make-L1:not:Expr.787
                      'remove-one-armed-if
                      (Expr (Lsrc:not:Expr.774-e g442.303))
                      "e")]
                   [(eqv? tag 14)
                    (make-L1:begin:Expr.788 'remove-one-armed-if
                      (map (lambda (m) (Expr m))
                           (Lsrc:begin:Expr.775-e* g442.303))
                      (Expr (Lsrc:begin:Expr.775-e g442.303)) "e*" "e")]
                   [(eqv? tag 16)
                    (make-L1:lambda:Expr.789 'remove-one-armed-if
                      (Lsrc:lambda:Expr.776-x* g442.303)
                      (map (lambda (m) (Expr m))
                           (Lsrc:lambda:Expr.776-body* g442.303))
                      (Expr (Lsrc:lambda:Expr.776-body g442.303)) "x*"
                      "body*" "body")]
                   [(eqv? tag 18)
                    (make-L1:let:Expr.790 'remove-one-armed-if (Lsrc:let:Expr.777-x* g442.303)
                      (map (lambda (m) (Expr m))
                           (Lsrc:let:Expr.777-e* g442.303))
                      (map (lambda (m) (Expr m))
                           (Lsrc:let:Expr.777-body* g442.303))
```

```
                       (Expr (Lsrc:let:Expr.777-body g442.303)) "x*" "e*"
                       "body*" "body")]
                    [(eqv? tag 20)
                     (make-L1:letrec:Expr.791 'remove-one-armed-if
                       (Lsrc:letrec:Expr.778-x* g442.303)
                       (map (lambda (m) (Expr m))
                            (Lsrc:letrec:Expr.778-e* g442.303))
                       (map (lambda (m) (Expr m))
                            (Lsrc:letrec:Expr.778-body* g442.303))
                       (Expr (Lsrc:letrec:Expr.778-body g442.303)) "x*" "e*"
                       "body*" "body")]
                    [(eqv? tag 22)
                     (make-L1:set!:Expr.792 'remove-one-armed-if (Lsrc:set!:Expr.779-x g442.303)
                       (Expr (Lsrc:set!:Expr.779-e g442.303)) "x" "e")]
                    [(eqv? tag 24)
                     (make-L1:e:Expr.793 'remove-one-armed-if
                       (Expr (Lsrc:e:Expr.780-e g442.303))
                       (map (lambda (m) (Expr m))
                            (Lsrc:e:Expr.780-e* g442.303))
                       "e" "e*")]
                    [else
                     (error 'remove-one-armed-if
                       "unexpected Expr"
                       g442.303)]))])))))))
```

4.4 Tracing passes and transformers

Echoing the code generated by define-pass can help compiler writers to understand what is happening at expansion time, but it does not help in determining what is happening at run time. To facilitate this type of debugging, passes and transformers can be traced at run time. The tracing system, similar to Chez Scheme's trace-define-syntax, unparses the input-language term and output-language term of the pass using the language unparsers to provide the S-expression representation of the language term that is being transformed.

The trace-define-pass form works just like the define-pass form but adds tracing for the input-language term and output-language term of the pass. For instance, if we want to trace the processing of the input:

```
(let ([x 10])
  (if (= (* (/ x 2) 2) x) (set! x (/ x 2)))
  (* x 3))
```

the pass can be defined as a tracing pass, as follows:

```
(trace-define-pass remove-one-armed-if : Lsrc (e) -> L1 ()
  (Expr : Expr (e) -> Expr ()
    [(if ,[e0] ,[e1]) `(if ,e0 ,e1 (void))]))
```

Running the class compiler with remove-one-armed-if traced produces the following:

```
> (my-tiny-compile
    '(let ([x 10])
       (if (= (* (/ x 2) 2) x) (set! x (/ x 2)))
       (* x 3)))
|(remove-one-armed-if
   (let ([x.12 10])
     (if (= (* (/ x.12 2) 2) x.12) (set! x.12 (/ x.12 2)))
     (* x.12 3)))
|(let ([x.12 10])
   (if (= (* (/ x.12 2) 2) x.12) (set! x.12 (/ x.12 2)) (void))
   (* x.12 3))
15
```

The tracer prints the *pretty* (i.e., S-expression) form of the language, rather than the record representation, to allow the compiler writer to read the terms more easily. This does not trace the internal transformations that happen within the transformers of the pass. Transformers can be traced by adding the `trace` keyword in front of the transformer definition. We can run the same test with a `remove-one-armed-if` that traces the `Expr` transformer, as follows:

```
> (my-tiny-compile
      '(let ([x 10])
         (if (= (* (/ x 2) 2) x) (set! x (/ x 2)))
         (* x 3)))
|(Expr
   (let ([x.0 10]) (if (= (* (/ x.0 2) 2) x.0) (set! x.0 (/ x.0 2))) (* x.0 3)))
| (Expr (* x.0 3))
| |(Expr x.0)
| |x.0
| |(Expr 3)
| |3
| |(Expr *)
| |*
| (* x.0 3)
| (Expr (if (= (* (/ x.0 2) 2) x.0) (set! x.0 (/ x.0 2))))
| |(Expr (= (* (/ x.0 2) 2) x.0))
| | (Expr (* (/ x.0 2) 2))
| | |(Expr (/ x.0 2))
| | | (Expr x.0)
| | | x.0
| | | (Expr 2)
| | | 2
| | | (Expr /)
| | | /
| | |(/ x.0 2)
| | |(Expr 2)
| | |2
| | |(Expr *)
| | |*
| | (* (/ x.0 2) 2)
| | (Expr x.0)
| | x.0
| | (Expr =)
| | =
| |(= (* (/ x.0 2) 2) x.0)
| |(Expr (set! x.0 (/ x.0 2)))
| | (Expr (/ x.0 2))
| | |(Expr x.0)
| | |x.0
| | |(Expr 2)
| | |2
| | |(Expr /)
| | |/
| | (/ x.0 2)
| |(set! x.0 (/ x.0 2))
| (if (= (* (/ x.0 2) 2) x.0) (set! x.0 (/ x.0 2)) (void))
| (Expr 10)
| 10
|(let ([x.0 10])
   (if (= (* (/ x.0 2) 2) x.0) (set! x.0 (/ x.0 2)) (void))
   (* x.0 3))
15
```

Here, too, the traced forms are the pretty representation and not the record representation.

Bibliography

[1] A. W. Keep. *A Nanopass Framework for Commercial Compiler Development.* PhD thesis, Indiana University, Indianapolis, IN, USA, Feb. 2013.

[2] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, London, UK, 1991. Springer-Verlag.

[3] D. Sarkar, O. Waddell, and R. K. Dybvig. A nanopass infrastructure for compiler education. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 201–212, New York, NY, USA, 2004. ACM.