EDUCATIONAL PEARL

A Nanopass Framework for Compiler Education[*]

Dipanwita Sarkar
Microsoft Corporation
dipas@microsoft.com

Oscar Waddell
Abstrax, Inc.
owaddell@abstrax.com

R. Kent Dybvig
Indiana University
dyb@cs.indiana.edu

Abstract

A compiler structured as a small number of monolithic passes is difficult to understand and difficult to maintain. The steep learning curve is daunting, and even experienced developers find that modifying existing passes is difficult and often introduces subtle and tenacious bugs. These problems are especially frustrating when the developer is a student in a compiler class. An attractive alternative is to structure a compiler as a collection of many fine-grained passes, each of which performs a single task. This structure aligns the implementation of a compiler with its logical organization, simplifying development, testing, and debugging. This paper describes the methodology and tools comprising a framework for constructing such compilers.

1 Introduction

Production compilers often exhibit a monolithic structure in which each pass performs several analyses, transformations, and optimizations. An attractive alternative, particularly in an educational setting, is to structure a compiler as a collection of many small passes, each of which performs a small part of the compilation process. This "micropass" structure aligns the implementation of a compiler with its logical organization, yielding a more readable and maintainable compiler. Bugs that arise are more easily isolated to a particular task, and adding new functionality is easier since new code need not be grafted onto existing passes nor wedged between two logical passes that would have been combined in a monolithic structure.

Each pass of a micropass compiler performs a single specific task to simplify, verify, convert, analyze, or improve the code. A simplification pass reduces the complexity of subsequent passes by translating its input into a simpler intermediate language, e.g., replacing pattern matching constructs with more primitive code. A verification pass checks compiler invariants that are not easily expressed within the grammar, e.g., that all bound variables are unique. A conversion pass makes explicit an abstraction that is not directly supported by the low-level target language, e.g., transforming higher order procedures into explicitly allocated closures. An analysis pass collects information from the input program and records that information with annotations in the output program, e.g., annotating each `lambda` expression with

∗ A preliminary version of this article was presented at the 2004 International Conference on Functional Programming.

```
expr  ⟶  constant | (quote datum) | var
      |  (set! var expr) | (if expr expr) | (if expr expr expr)
      |  (begin expr expr*) | (lambda (var*) expr expr*)
      |  (let ((var expr)*) expr expr*)
      |  (letrec ((var expr)*) expr expr*)
      |  (primitive expr*) | (expr expr*)
```

Fig. 1. A small but representative Scheme subset

its set of free variables. An improvement pass attempts to decrease the run time or resource utilization of the program.

While simplification and conversion passes alter the intermediate language in some way, each verification or improvement pass produces a program in the same intermediate language as its input program so that we may selectively enable or disable individual checks or optimizations simply by not running the corresponding passes. Verification passes are enabled only during compiler development, where they can help identify bugs in upstream passes. The ability to disable optimizations supports compiler switches that trade code quality for compile-time speed and is also useful for regression testing when optimizations may mask bugs in seldom-executed portions of other passes.

A few years ago we switched to the micropass structure in our senior- and graduate-level compiler courses. Students are supported in the writing of their compilers by several tools: a pattern matcher with convenient notations for recursion and mapping, macros for expanding the output of each pass into executable code, a reference implementation of the compiler, a suite of (terminating) test programs, and a driver. The driver runs the compiler on each of the programs in the test suite and evaluates the output of each pass to verify that it returns the same result as the reference implementation. Intermediate-language programs are all represented as s-expressions, which simplifies both the compiler passes and the driver.

The switch to the micropass methodology and the tools that support it have enabled our students to write more ambitious compilers. Each student in our one-semester compiler class builds a 50-pass compiler from the s-expression level to Sparc assembly code for the subset of Scheme shown in Figure 1. The compiler includes several optimizations as well as a graph-coloring register allocator. Students in the graduate course implement several additional optimizations. The passes included in the compiler for a recent semester are listed in Table 1. Due to space limitations, we cannot go into the details of each pass, but the pass names are suggestive of their roles in the compilation process.

The micropass methodology and tools are not without problems, however. The repetitive code for traversing and rewriting abstract syntax trees can obscure the meaningful transformations performed by individual passes. In essence, the sheer volume of code for each pass can cause the students to miss the forest for the trees. Furthermore, we have learned the importance of writing out grammars describing the output of each pass as documentation, but the grammars are not enforced. Thus, it is easy for an unhandled specific case to fall through to a more general

Week 1: simplification
verify-scheme[1]
rename-var
remove-implicit-begin[1]
remove-unquoted-constant
remove-one-armed-if
verify-a1-output[1]

Week 2: assignment conv.
remove-not
mark-assigned
optimize-letrec[2]
remove-impure-letrec
convert-assigned
verify-a2-output[1]

Week 3: closure conversion
optimize-direct-call
remove-anon-lambda
sanitize-binding-forms
uncover-free
convert-closure
optimize-known-call[3]
uncover-well-known[2]
optimize-free[2]
optimize-self-reference[2]
analyze-closure-size[1]
lift-letrec
verify-a3-output[1]

Week 4: canonicalization
introduce-closure-prims
remove-complex-constant
normalize-context
verify-a4-output[1]

Week 5: encoding/allocation
specify-immediate-repn
specify-nonimmediate-repn

Week 6: start of UIL compiler
verify-uil

Week 7: labels and temps
remove-complex-opera*
lift-letrec-body
introduce-return-point
verify-a7-output[1]

Week 8: virtual registerizing
remove-nonunary-let
uncover-local
the-return-of-set!
flatten-set!
verify-a8-output[1]

Week 9: brief digression
generate-C-code[4]

Week 10: register alloc. setup
uncover-call-live[2]
optimize-save-placement[2]
eliminate-redundant-saves[2]
rewrite-saves/restores[2]
impose-calling-convention
reveal-allocation-pointer
verify-a10-output[1]

Week 11: live analysis
uncover-live-1
uncover-frame-conflict
strip-live-1
uncover-frame-move
verify-a11-output[1]

Week 12: call frames
uncover-call-live-spills
assign-frame-1
assign-new-frame
optimize-fp-assignments[2]
verify-a12-output[1]

Week 13: spill code
finalize-frame-locations
eliminate-frame-var
introduce-unspillables
verify-a13-output[1]

Week 14: register assignment
uncover-live-2
uncover-register-conflict
verify-unspillables[1]
strip-live-2
uncover-register-move
assign-registers
assign-frame-2
finalize-register-locations
analyze-frame-traffic[1]
verify-a14-output[1]

Week 15: assembly
flatten-program
generate-Sparc-code

Table 1. *Passes assigned during a recent semester, given in running order and grouped roughly by week and primary task. Notes:* [1]*Supplied by the instructor.* [2]*Challenge passes required of graduate students, not necessarily in the week shown.* [3]*Actually written during Week 4.* [4]*Not included in final compiler. During Week 6, students also had an opportunity to turn in updated versions of earlier passes. Week 9 was a short week leading up to spring-break week. Most passes are run once, but the passes comprising the main part of the register and frame allocator are repeated until all variables have been given register or frame homes.*

case, resulting in either confusing errors or malformed output to trip up later passes. Finally, the resulting compiler is slow, which leaves students with a mistaken impression about the speed of a compiler and the importance thereof.

To address these problems, we have developed a "nanopass" methodology and a domain-specific language for writing nanopass compilers. A nanopass compiler differs from a micropass compiler in three ways: (1) the intermediate-language grammars are formally specified and enforced; (2) each pass needs to contain traversal code only for forms that undergo meaningful transformation; and (3) the intermediate code is represented more efficiently as records, although all interaction with the programmer is still via the s-expression syntax. We use the word "nanopass"

to indicate both the intended granularity of passes and the amount of source code required to implement each pass.

The remainder of this paper describes the nanopass methodology and supporting tools. Section 2 describes the tools that are used to build nanopass compilers. Section 3 presents an example set of language and pass definitions. Section 4 describes the implementation of nanopass tools. Section 5 discusses related work. Section 6 presents our conclusions and discusses future work.

2 Nanopass tools

This section describes tools for defining new intermediate languages and compiler passes. These tools comprise a domain-specific language for writing nanopass compilers and are implemented as extensions of the host language, Scheme, via the `syntax-case` macro system (Dybvig *et al.*, 1993). This approach provides access to the full host language for defining auxiliary procedures and data structures, which are particularly useful when writing complex passes, such as a register allocation pass.

2.1 Defining intermediate languages

Intermediate language definitions take the following form:

(**define-language** *name* { **over** *tspec*$^+$ } **where** *production*$^+$)

The optional *tspec* declarations specify the terminals of the language and introduce metavariables ranging over the various terminals. Each *tspec* is of the form

(*metavariable*$^+$ **in** *terminal*)

where the *terminal* categories are declared externally. A metavariable declaration for x implicitly specifies metavariables of the form xn, where n is a numeric suffix. Each *production* corresponds to a production in the grammar of the intermediate language.

A production pairs a nonterminal with one or more alternatives, with an optional set of metavariables ranging over the nonterminal.

({ *metavariable*$^+$ **in** } *nonterminal alternative*$^+$)

Productions may also specify elements that are common to all alternatives using the following syntax.

({ *metavariable*$^+$ **in** } (*nonterminal common*$^+$) *alternative*$^+$)

Common elements may be used to store annotations, e.g., source information or analysis byproducts, that are common to all subforms of the intermediate language.

Each *alternative* is a metavariable or parenthesized form declaring an intermediate language construct, followed by an optional set of production properties *property*$^+$. Parenthesized forms typically begin with a keyword and contain substructure with metavariables specifying the language category into which each subform falls. Since parenthesized forms are disambiguated by the beginning keyword,

```
(define-language L0 over
  (b in boolean)
  (n in integer)
  (x in variable)
 where
  (Program Expr)
  (e body in Expr
    b n x
    (if e1 e2 e3)
    (seq c1 e2) => (begin c1 e2)
    (lambda (x ...) body)
    (e0 e1 ...))
  (c in Cmd
    (set! x e)
    (seq c1 c2) => (begin c1 c2)))
```

L0	⟶	*Program*
Program	⟶	*Expr*
Expr	⟶	*boolean* | *integer* | *var*
	|	`(if` *Expr Expr Expr*`)`
	|	`(seq` *Cmd Expr*`)`
	|	`(lambda (`*var*`*)` *Expr*`)`
	|	`(`*Expr Expr*`*)`
Cmd	⟶	`(set!` *var Expr*`)`
	|	`(seq` *Cmd Cmd*`)`

Fig. 2. A simple language definition and the corresponding grammar

at most one alternative of a production may be a parenthesized form that does not begin with a keyword. This allows the intermediate language to include applications using the natural s-expression syntax. Each *property* is a *key*, *value* pair. Properties are used to specify semantic, type, and flow information for the associated alternative.

Figure 2 shows a simple language definition and the grammar it represents. It defines metavariables `x`, `b`, and `n` ranging over variables, booleans, and integers, and defines three sets of productions. The first set defines `Program` as an `Expr`. The second defines metavariables `e` and `body` ranging over `Expr` and declares that `Expr` is a boolean, integer, variable reference, `if` expression, `seq` expression, `lambda` expression, or application. The third defines metavariable `c` ranging over `Cmd` and declares that `Cmd` is a `set!` command or `seq` command.

The semantics of each intermediate language form may be specified implicitly via its natural translation into the host language, if one exists. In Figure 2, this implicit translation suffices for booleans, numbers, variable references, `lambda`, `set!`, `if`, and applications. For `seq` expressions, the translation is specified explicitly using the `=>` (translates-to) property. Implicit and explicit translation rules establish the meaning of an intermediate language program in terms of the host language. This is an aid to understanding intermediate language programs and provides a mechanism whereby the output of each pass can be verified to produce the same results as the original input program while a compiler is being debugged. Explicit translations can become complex, in which case we often express the translation in terms of a syntactic abstraction (macro) defined in the host language.

2.2 Language inheritance

Consecutive intermediate languages are often closely related due to the fine granularity of the intervening passes. To permit concise specification of these languages, the **define-language** construct supports a simple form of inheritance via the

extends keyword, which must be followed by the name of a base language, already defined:

(define-language *name* **extends** *base*
{ **over** { *mod tspec* }$^+$ }
{ **where** { *mod production* }$^+$ })

The terminals and productions of the base language are copied into the new language, subject to modifications in the **over** and **where** sections of the definition, either of which may be omitted if no modifications to that section are necessary. Each *mod* is either `+`, which adds new terminals and productions, or `-`, which removes the corresponding terminals and productions. The example below defines a new language `L1` derived from `L0` (Figure 2) by removing the `boolean` terminal and `Expr` alternative and replacing the `Expr if` alternative with an `Expr case` alternative.

```
(define-language L1 extends L0
  over
   - (b in boolean)
  where
   - (Expr b (if e1 e2 e3))
   + (default in Expr
       (case x (n1 e1) ... default)))
```

Language `L1` could serve as the output of a conversion pass that makes language-specific details explicit en route to a language-independent back end. For example, C treats zero as false, while Scheme provides a distinct boolean constant `#f` representing false. Conditional expressions of either language could be translated into `case` expressions in `L1` with language-specific encodings of false made explicit.

Language inheritance is merely a notational convenience. A complete language definition is generated for the new language, and this definition behaves exactly as if it had been written out fully.

2.3 Defining passes

Passes are specified using a pass definition construct that names the input and output languages and specifies transformation functions that map input-language forms to output-language forms:

(define-pass *name input-language* `->` *output-language transform****)**

Analysis passes are often run purely for effect, e.g., to collect and record information about variable usage. For such passes, the special output language `void` is used. Similarly, the special output language `datum` is used when a pass traverses an AST to compute some more general result, e.g., an estimate of object code size.

Each *transform* specifies the transformer's name, a signature describing the transformer's input and output types, and a set of clauses implementing the transformation.

(*name* `:` *nonterminal* *arg** `->` *val*$^+$
  { (*input-pattern* { *guard* } *output-expression*) }*)

The input portion of the signature lists a nonterminal of the input language followed by the types of any additional arguments expected by the transformer. The output portion lists one or more result types. Unless `void` or `datum` is specified in place of the output language, the first result type is expected to be an output-language nonterminal.

Each clause pairs an *input-pattern* and a host-language *output-expression* that together describe the transformation of a particular input-language form. Input patterns are specified using an s-expression syntax that extends the syntax of alternatives in the production for the corresponding input-language nonterminal. Subpatterns are introduced by commas, which indicate, by analogy to `quasiquote` and `unquote` (Kelsey *et al.*, 1998), portions of the input form that are not fixed. For example, `(seq (set! ,x ,n) ,e2)` introduces three subpatterns binding pattern variables `x`, `n`, and `e2`. Metavariables appearing within patterns impose further constraints on the matching process. Thus, the given pattern matches only those inputs consisting of a `seq` form whose first subform is a `set!` form that assigns a numeric constant to a variable, and whose second subform is an `Expr`.

Pattern variables are used within input patterns to constrain the matching of subforms of the input AST. They also establish variable bindings that may be used in output expressions to refer to input subforms or the results of structural recursion. The various forms that subpatterns may take are summarized below, where the metavariable a ranges over alternate forms of an input-language nonterminal A, and the metavariable b ranges over alternate forms of an output-language nonterminal B.

1. The subpattern `,`a matches if the corresponding input subform is a form of A and binds a to the matching subform.
2. The subpattern `,[`f `:` a `->` b`]` matches if the corresponding input subform is a form of A, binds a to the input subform, and binds b to the result of invoking f on a. An error is signaled if the result type is not a form of B.
3. The subpattern `,[`a `->` b`]` is equivalent to `,[`f `:` a `->` b`]` if f is the sole transformer mapping $A \rightarrow B$.
4. The subpattern `,[`b`]` is equivalent to the subpattern `,[`a `->` b`]` in contexts where the input form is constrained by the language definition to be a form of A, except that no variable is bound to the input subform.

Transformers may accept multiple arguments and return multiple values. To support these transformers, the syntax `,[`f `:` a x^* `->` b y^*`]` may be used to supply additional arguments x^* to f and bind program variables y^* to the additional values returned by f. The first argument must be an AST as must the first return value, unless `void` or `datum` is specified as the output type. Subpatterns 3 and 4 are extended in the same way to support the general forms `,[`a x^* `->` b y^*`]` and `,[`b y^*`]`. Pattern variables bound to input subforms may be referenced among the extra arguments x^* within structural-recursion patterns 2 and 3 above. Metavari-

ables are used within patterns to guide the selection of appropriate transformers for structural recursion.

When present, the optional guard expression imposes additional constraints on the matching of the input subform prior to any structural recursion specified by the subpattern. Pattern variables bound to input forms are visible within guard expressions. Output expressions may contain *templates* for constructing output-language forms using syntax that extends the syntax of alternatives in the production for the corresponding output-language nonterminal.

New abstract syntax trees are constructed via output templates specified using an overloaded `quasiquote` syntax that constructs record instances rather than list structure. Where commas, i.e., `unquote` forms, do not appear within an output template, the resulting AST has a fixed structure. An expression prefixed by a comma within an output template is a host-language expression that must be evaluated to obtain an AST to be inserted as the corresponding subform of the new AST being produced. For example, `‘(if (not ,e1) ,e2 ,e3)` constructs a record representing an `if` expression with an application of the primitive `not` as its test and the values of program variables `e1`, `e2`, and `e3` inserted where indicated. An error is signaled if a subform value is not a form of the appropriate output-language nonterminal.

Where ellipses follow an `unquote` form in an output template, the embedded host-language expressions must evaluate to lists of objects. For example, `‘(let ((,x ,e) ...) ,body)` requires that `x` be bound to a list of variables and `e` be bound to a list of `Expr` forms.

Often, a pass performs nontrivial transformation for just a few forms of the input language. In such cases, the two intermediate languages are closely related and the new language can be expressed using language inheritance. When two intermediate languages can be related by inheritance, a pass definition needs to specify transformers only for those forms that undergo meaningful change, leaving the implementation of other transformers to a *pass expander*. The pass expander completes the implementation of a pass by consulting the definitions of the input and output languages. Strong typing of passes, transformers, and intermediate languages helps the pass expander to automate these simple transformations. The pass expander is an important tool for keeping pass specifications concise. Intermediate language inheritance and the pass expander together provide a simple and expedient way to add new intermediate languages and corresponding passes to the compiler.

3 Example: Assignment Conversion

Assignment conversion is a transformation that replaces assigned variables with mutable data structures to make the locations of assigned variables explicit. For example, assignment conversion would translate the program

```
(lambda (a b) (seq (set! a b) a))
```

into the following.

```
; mark-assigned runs for effect over programs in language L1, marking each
; assigned variable by setting a flag in its record structure.

(define-pass mark-assigned L1 -> void
  (process-command : Command -> void
    [(set! ,x ,[e])
     (set-variable-assigned! x #t)]))

; convert-assigned produces a program in language L2 from a program in language
; L1, creating an explicit location (pair) for each assigned variable and rewriting
; references and assignments accordingly.

(define-pass convert-assigned L1 -> L2
  (process-expr : Expr -> Expr
    [,x (variable-assigned x) `(primapp car ,x)]
    [(lambda (,x ...) ,[body])
     (let-values ([(xi xa xr) (split-vars x)])
       `(lambda (,xi ...)
          (let ((,xa (primapp cons ,xr #f)) ...)
            ,body)))])
  (process-command : Command -> Command
    [(set! ,x ,[e]) `(primapp set-car! ,x ,e)]))

; split-vars is used by convert-assigned to introduce temporaries for assigned
; variables.

(define split-vars
  (lambda (vars)
    (if (null? vars)
        (values '() '() '())
        (let-values ([(ys xas xrs) (split-vars (cdr vars))])
          (if (variable-assigned (car vars))
              (let ([t (make-variable 'tmp)])
                (values (cons t ys) (cons (car vars) xas) (cons t xrs))
                (values (cons (car vars) ys) xas xrs)))))))
```

Fig. 3. Assignment conversion

```
(lambda (t b)
  (let ((a (cons t #f)))
    (seq (set-car! a b) (car a))))
```

Assignment conversion involves two passes. The first pass, `mark-assigned`, locates assigned variables i.e., those appearing on the left-hand side of an assignment, and marks them by setting a flag in one of the fields of the variable record structure. The second pass, `convert-assigned`, rewrites references and assignments to assigned variables as explicit structure accesses or mutations.

The `mark-assigned` pass runs for effect only over programs in a language `L1` that may be derived from `L0` (Figure 2) by adding the forms `let` and `primapp` and the terminal `primitive`:

```
(define-language L1 extends L0
  over
   + (pr in primitive)
  where
   + (Expr
       (let ((x e) ...) body)
       (primapp pr e ...))
   + (Command
       (primapp pr e ...)))
```

Since `convert-assigned` removes the `set!` form, its output language, `L2`, is derived from its input language `L1` to reflect this change.

```
(define-language L2 extends L1 where - (Command (set! x e)))
```

The code for both passes is shown in Figure 3. Each pass deals with just those language forms that undergo meaningful transformation and relies on the pass-expander to supply the code for the remaining cases.

Only one language form, `set!`, need be handled explicitly by the first of the two passes. If the input is a `set!` command, the pass simply sets the assigned flag in the record structure representing the assigned variable and recursively processes the right-hand-side expression via the `,[·]` syntax.

The second pass handles three forms explicitly: variable references, `lambda` expressions, and assignments. It binds each assigned variable to a pair whose `car` is the original value of the variable, replaces each reference to an assigned variable with a call to `car`, and replaces each assignment with a call to `set-car!`.

4 Implementation

From a student's perspective, a language definition specifies the structure of an intermediate language in terms of the familiar s-expression syntax. All of the student's interactions with the intermediate language occur via this syntax. Internally, however, intermediate language programs are represented more securely and efficiently as record structures (Illustration 1).

Intermediate language programs are also evaluable in the host language, using the translation properties attached to production alternatives. To support these differing views of intermediate language programs, a language defined via **define-language** implicitly defines the following items:

1. a set of record types representing the abstract syntax trees (ASTs) of intermediate-language programs,
2. a parser mapping s-expressions to record structure,
3. an unparser mapping record structure to s-expressions, and
4. a partial parser mapping s-expression patterns to record schema.

These products are packaged within a module and may be imported where they are needed. The remainder of this section describes these products and how they are used.

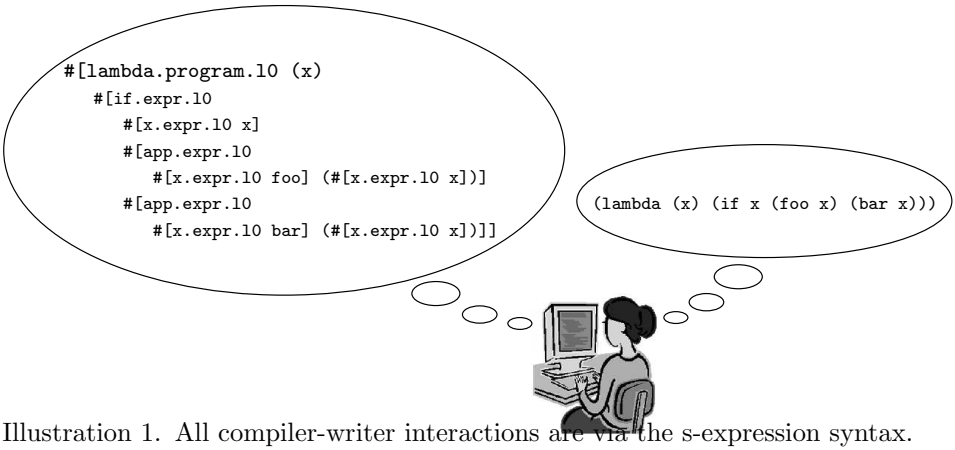

Illustration 1. All compiler-writer interactions are via the s-expression syntax.

4.1 Record-type definitions

A language definition automatically generates a set of record definitions as illustrated for language L0 in Figure 4. A base record type is constructed for the language along with a subtype for each nonterminal. The subtype for each nonterminal declares the common elements for that nonterminal. A new record type is also created for each alternative as a subtype of the corresponding nonterminal.

4.2 Parser

Each language definition produces a parser capable of transforming s-expressions to the corresponding record structure representing the same abstract syntax tree. The parser for the first input language serves as the first pass of the compiler (after lexical analysis and parsing) and provides the record-structured input required by subsequent passes of the compiler. The parsers for other intermediate languages simplify debugging and experimentation by making it easy to obtain inputs suitable for any given pass.

Figure 5 shows part of the code for the parser that is generated by the language definition for `L0` in Figure 2. The code for the parser mirrors the language definition. The language definition produces a set of mutually recursive procedures, `parse-program`, `parse-expr` and `parse-command`, each handling all the alternatives of the corresponding nonterminal. The nonterminal parsers operate by recursive descent on list structured input, following the grammar.

4.3 Unparser

The unparser converts the AST records to their corresponding host-language executable forms. Like the parser this also serves as a good debugging aid by allowing the student to view the output of any pass in host-language form. It enables the student to trace and manually translate programs, e.g., during the exploratory phase of the development of a new optimization. Each record-type definition stores the parenthesized form and the host-language form for the alternative.

```
(define-record L0 ())
(define-record program L0 ())
(define-record expr L0 ())
(define-record command L0 ())
(define-record b.expr expr (b))
(define-record n.expr expr (n))
(define-record x.expr expr (x))
(define-record if.expr expr (e1 e2 e3))
(define-record seq.expr expr (c1 c2))
(define-record lambda.expr expr (xs body))
(define-record app.expr expr (e0 es))
(define-record set!.command command (x e))
(define-record seq.command command (c1 c2))
```

Fig. 4. Record definitions generated for L0

```
(define (parser-lang.L0 s-exp)
  (define (parse-program s-exp) —)
  (define (parse-expr s-exp)
    (if (pair? s-exp)
        (cond
          —
          [(and (eq? 'seq (car s-exp)) (= 3 (length s-exp)))
           (make-seq.expr.L0.6
             (parse-command (cadr s-exp))
             (parse-expr (caddr s-exp)))]
          —
          [else
           (make-anon.7
             (parse-expr (car s-exp))
             (map parse-expr (cdr s-exp)))])
        (cond
          [(boolean? s-exp) (make-b.expr.L0.1 s-exp)]
          —
          [else (error —)])))
  (define (parse-command s-exp) —)
  (parse-program s-exp))
```

Fig. 5. Parser generated for L0

The host-language form, if different from the parenthesized form, is expressed as the translates-to production property in the language definition, as in the case of `(seq c1 e2) => (begin c1 e2)` in Figure 2. Each record type stores the information required to unparse instances of itself. As a result, all languages share one unparse procedure.

The unparser can also translate the record structures into their implied parenthesized forms, i.e., with no host-language translations, allowing the student to pretty-print intermediate language code.

4.4 Partial Parser

The partial parser is used to support input pattern matching and output construction, which are described in Section 4.5 and Section 4.6.

The partial parser translates s-expression syntax representing an input pattern or output template into its corresponding record schema. A record schema is similar to a record structure except that the variable parts of the s-expression pattern are converted to a special list representation that is later used to generate code for the pass. The structures produced by the partial parser are not visible to the student, but are used to generate the code for matching the given pattern and constructing the desired output.

4.5 Matching input

When invoked, a transformer matches its first argument, which must be an AST, against the input pattern of each clause until a match is found. Clauses are examined in order, with user-specified clauses preceding any clauses inserted by the pass expander. This process continues until the input pattern of some clause is found to match the AST and the additional constraints imposed by guard expressions and pattern variables are satisfied. When a match is found, any structural recursive processing is performed and the corresponding output expression is evaluated to produce a value of the expected result type. An error is signaled if no clause matches the input.

Input patterns are specified using an s-expression syntax that extends the syntax of alternatives for the corresponding nonterminal with support for pattern variables as described in Section 2.3.

4.6 Constructing output

When the input to a transformer matches the input pattern of one of the clauses, the corresponding output expression is evaluated in an environment that binds the subforms matched by pattern variables to like-named program variables. For example, if an input language record representing `(set! y (f 4))` matches the pattern `(set! ,x ,[e])`, the corresponding output expression is evaluated in an environment that binds the program variables `x` and `e` to the records representing `y` and the result of processing `(f 4)`. When a pattern variable is followed by an ellipsis (`...`) in the input pattern, the corresponding program variable is bound to a matching list of records.

The record constructors available within output templates are determined by the output language specified in the pass definition. Instantiating the output template must produce an AST representing a form of the output nonterminal for the transformer containing the clause. The syntax of output templates is described in Section 2.3.

5 Related work

Our automatically generated parsers and unparsers are similar to the language-specific functions that read and write the language-specific data structures generated by tree-like intermediate languages in the Zephyr Abstract Syntax Definition Language (ASDL). ASDL focuses solely on intermediate representation and therefore does not integrate support for defining passes.

The TIL compiler operates on a series of typed intermediate languages (Tarditi *et al.*, 1996), allowing the output of each pass to be statically type-checked. The unparsers produced by our language definitions for each intermediate language allow us to generate semanticaly equivalent host-language programs with which we can similarly verify static and dynamic properties, e.g., via control-flow analysis.

Polyglot (Nystrom *et al.*, 2003) ensures that the work required to add new passes or new AST node types is proportional to the number of node types or passes affected, doing so with some fairly involved OOP syntax and mechanics. Our pass expander and our support for language inheritance approach the same goal with less syntactic and conceptual overhead.

Tm is a macro processor in the spirit of `m4` that takes a source code template and a set of data structure definitions and generates source code (van Reeuwijk, 1992). Tree-walker and analyzer templates that resemble **define-pass** have been generated using Tm (van Reeuwijk, 2003). These templates are low-level relatives of **define-pass**, which provides convenient input pattern syntax for matching nested record structures and output template syntax constructing nested record structures.

The PFC compiler (Allen & Kennedy, 1982) uses macro expansion to fill in boilerplate transformations, while the object-oriented SUIF system (Aigner *et al.*, 2000a; Aigner *et al.*, 2000b), which operates on a single intermediate language, allows boilerplate transformations to be inherited. The nanopass approach achieves effects similar to these systems but extends them by formalizing the language definitions, including sufficient information in the language definitions to allow automated conversion to and from the host language, and separating traversal algorithms from intermediate language and pass definitions.

6 Conclusions

The nanopass methodology supports the decomposition of a compiler into many small pieces. This decomposition simplifies the task of understanding each piece and, therefore, the compiler as a whole. There is no need to "shoe-horn" a new analysis or transformation into an existing monolithic pass. The methodology also simplifies the testing and debugging of a compiler, since each task can be tested independently, and bugs are easily isolated to an individual task.

The nanopass tools enable a compiler student to focus on concepts rather than implementation details while having the experience of writing a complete and substantial compiler. While it is useful to have students write out all traversal and rewriting code for the first few passes to understand the process, the ability to focus only on meaningful transformations in later passes reduces the amount of

tedium and repetitive code. The code savings are significant for many passes. With our old tools, `remove-not` was the smallest pass at 25 lines; it is now 7 lines. Similarly, `convert-assigned` was 55 lines and is now 20 lines. On the other hand, the sizes of a few passes cannot be reduced. The code generator, for example, must explicitly handle every grammar element.

Our experience indicates that fine-grained passes work extremely well in an educational setting. We are also interested in using the nanopass technology to construct production compilers, where the overhead of many traversals of the code may be unacceptable. To address this potential problem, we plan to develop a pass combiner that can, when directed, fuse together a set of passes into a single pass, using deforestation techniques (Wadler, 1988) to eliminate rewriting overhead.

Acknowledgements. Dan Friedman suggested the use of small, single-purpose passes and contributed to an earlier compiler based on this principle. Erik Hilsdale designed and implemented the matcher we used to implement micropass compilers. Our input-pattern subpatterns were inspired by this matcher. Jordan Johnson implemented an early prototype of the nanopass framework. Comments by Matthias Felleisen led to several improvements in the presentation. Dipanwita Sarkar was funded by a gift from Microsoft Research University Relations.

References

Aigner, G., Diwan, A., Heine, D., Lam, M., Moore, D., Murphy, B., & Sapuntzakis, C. (2000a). *An overview of the SUIF2 compiler infrastructure.* Tech. rept. Stanford University.

Aigner, G., Diwan, A., Heine, D., Lam, M., Moore, D., Murphy, B., & Sapuntzakis, C. (2000b). *The SUIF2 compiler infrastructure.* Tech. rept. Stanford University.

Allen, J.R., & Kennedy, K. (1982). *Pfc: A program to convert fortran to parallel form.* Technical Report MASC-TR82-6. Rice University, Houston, TX.

Dybvig, R. Kent, Hieb, Robert, & Bruggeman, Carl. (1993). Syntactic abstraction in Scheme. *Lisp and symbolic computation*, **5**(4), 295–326.

Kelsey, Richard, Clinger, William, & (Editors), Jonathan A. Rees. (1998). Revised^5 report on the algorithmic language Scheme. *Sigplan notices*, **33**(9), 26–76.

Nystrom, N., Clarkson, M., & Myers, A. (2003). Polyglot: An extensible compiler framework for Java. *Pages 138–152 of: Proceedings of the 12th international conference on compiler construction.* Lecture Notes in Computer Science, vol. 2622. Springer-Verlag.

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., & Lee, P. (1996). TIL: a type-directed optimizing compiler for ML. *Pages 181–192 of: Proceedings of the acm sigplan 1996 conference on programming language design and implementation.*

van Reeuwijk, C. (1992). Tm: a code generator for recursive data structures. *Software practice and experience*, **22**(10), 899–908.

van Reeuwijk, C. (2003). Rapid and robust compiler construction using template-based metacompilation. *Pages 247–261 of: Proceedings of the 12th international conference on compiler construction.* Lecture Notes in Computer Science, vol. 2622. Springer-Verlag.

Wadler, P. (1988). Deforestation: Transforming programs to eliminate trees. *Pages 344–358 of: ESOP '88: European symposium on programming.* Lecture Notes in Computer Science, vol. 300. Springer-Verlag.