JFP **20** (1): 3–18, 2010. © Cambridge University Press 2010
doi:10.1017/S0956796809990281

EDUCATIONAL PEARL

"Little language" project modules

JOHN CLEMENTS
California Polytechnic State University, San Luis Obispo, CA, USA
(*e-mail:* clements@brinckerhoff.org)

KATHI FISLER
Worcester Polytechnic Institute
(*e-mail:* kfisler@cs.wpi.edu)

Abstract

Many computer science departments are debating the role of programming languages in the curriculum. These discussions often question the relevance and appeal of programming-languages content for today's students. In our experience, domain-specific, "little languages" projects provide a compelling illustration of the importance of programming-language concepts. This paper describes projects that prototype mainstream applications such as PowerPoint, TurboTax, and animation scripting. We have used these exercises as modules in non-programming languages courses, including courses for first year students. Such modules both encourage students to study linguistic topics in more depth and provide linguistic perspective to students who might not otherwise be exposed to the area.

1 Studying the design of (domain-specific) languages

Students (and some faculty!) sometimes wonder why the standard computing curriculum includes a course on programming languages. In their eyes, most employers have settled on a small, slowly changing pool of general-purpose languages that share a common programming idiom. Languages that do not fit this model, even very popular ones, are often regarded as "scripting" languages which students can learn on their own as needed. From this perspective, studying the building blocks and design principles that underlie programming languages seems unnecessary to students once they have command of one language. In many computer science departments (at least in America), programming languages is losing its position as core content (Fisher & Krintz 2008).

Programming-languages researchers cite many benefits of linguistic training: flexibility as a programmer, ability to design effective abstractions, and appreciation for different computational models. In our experience, these points are too abstract to resonate with many mainstream students. To make these points more concrete, we have developed a series of software construction projects within which students design and implement domain-specific languages (Bentley 1986; Deursen *et al.* 2000). Each project has students build a prototype of a mainstream software

application such as PowerPoint, TurboTax, or an animations-scripting platform such as Flash. Each of these applications processes complex and context-dependent data in well-specified ways: PowerPoint displays user-specified slide decks; TurboTax prompts end-users to fill in forms and fields required by local tax code; animations platforms provide constructs for coordinating interactions between graphical objects. Robust implementations of these packages decouple the data specification from the engine that processes it. In other words, software engineering concerns motivate the construction of domain-specific languages for these projects.

Grounding language design in software engineering enables using these projects as modules in courses beyond programming languages. We have used these projects successfully in two such settings: a software construction course for students in their third college year, and perhaps more surprisingly, an honors-level programming course for students in their first college year. In both settings, students confront core linguistic material: they must define syntax and semantics, identify program errors to flag, and provide an implementation via an interpreter, compiler, or embedding into a host language. Our modules therefore support many uses, from providing basic linguistic content to students who will not take a languages course, to advertising for languages courses, to providing real-world applications within languages courses. Given that domain-specific languages are often declarative in nature (Sabry 1999), these projects also provide a framework for exposing students to functional and declarative programming.

The paper presents these projects using tax-preparation software as a running example. Section 2 describes the tax-form problem. Section 3 discusses two implementation techniques and their pedagogic implications. Section 4 discusses advanced tax-form features and their linguistic consequences. Section 5 describes other domain-specific language examples that we have used effectively in classes. Section 6 describes our experience using these exercises as modules outside of programming-languages courses. Section 7 offers concluding remarks.

2 Tax forms: a running example

The task is to design and implement a tax assistant that helps a taxpayer fill out tax forms. For this example, we use the United States of America's federal tax forms (which residents complete annually); fragments of these forms appear in Figure 1. The tax assistant program should query the taxpayer for user-supplied fields (such as wages earned), compute the value of fields that are derived from other fields (such as total income), and produce the amount of tax. Some fields require completion of auxiliary forms (called *schedules* in the U.S. tax code) whose fields are referenced in computations on other forms; the arrows in Figure 1 show a reference to schedule C from both form 1040 (the main form) and another schedule (SE). The tax assistant should prompt for the completion of each form or schedule at most once. As tax laws can change from year to year, the program should be designed to adapt easily to different tax forms and to variations of the same form.

Several extensions to this basic problem offer richer features to tax-form authors and taxpayers using the software:

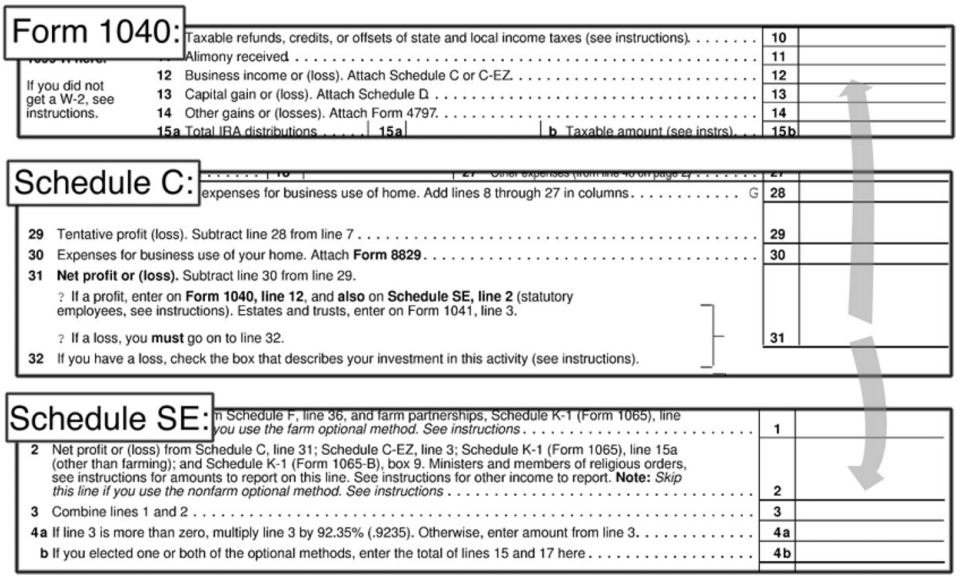

Fig. 1. Sample tax forms and the flow of information between them.

1 Print the completed tax forms when the user is finished.
2 Include error checking to catch form-specification errors.
3 Capture and check form invariants. For example, taxpayers who earn more than $1500 in dividends must fill out schedule B.
4 Allow users to prepare their taxes over multiple sessions, each of which resumes where the previous session ended.

We first discuss implementations of the basic problem, then address the advanced features.

3 Implementing a basic tax assistant

Different implementation approaches raise different pedagogic issues. This section discusses two approaches – interpreters and language embeddings via macros – in detail. Section 7 contrasts these approaches pedagogically in the context of our projects.

3.1 An interpreter-based approach

Writing an interpreter follows the style of many programming-languages curricula (Kamin 1989; Friedman *et al.* 2001; Krishnamurthi 2007). An interpreter-based implementation typically requires three artifacts: a *concrete syntax* in which a tax expert would describe a tax form, an *abstract syntax* representation (i.e., a data structure) that the tax assistant processes, and an *interpreter* which executes programs written in the abstract syntax. For simplicity, we assume that the concrete syntax is textual. Figure 2 shows sample concrete syntax for a portion of the 1040

```
(form form-1040 (export total-income)
      (field wages (prompt "Wages earned"))
      (field big-dividends (prompt-checkbox "are dividends > $1500?"))
      (field dividends (if big-dividends
                           (form-ref schedule-b total-dividends)
                           (prompt "enter total dividends")))
      (field total-income (calculated (+ wages dividends))))
```

```
(define-struct form (name exports lines))
(define-struct line (label instructions))

(make-form "form-1040" (list "total-income")
           (list (make-line "wages" (make-prompt "Wages earned"))
                 (make-line "dividends"
                            (make-if (make-prompt "are dividends > $1500?")
                                     (make-form-ref "schedule B" "total dividends")
                                     (make-prompt "enter total dividends")))
                 (make-line "total-income" (make-calculated (make-plus "wages" "dividends")))))
```

Fig. 2. Possible concrete (above) and abstract (below) syntaxes for a simple tax form.

tax form, including named fields containing both prompted and computed values, along with a corresponding abstract syntax for the same fragment. The abstract syntax, as with all code in the paper, is presented in PLT Scheme (Flatt & PLT 2009); each *make-* operator is a typed record-constructor.

Although it might seem obvious that students should design their concrete syntax before a corresponding abstract syntax, designing the abstract syntax first nearly always makes more sense. The abstract syntax is simply a careful definition of the set of possible inputs to the evaluator. When prototyping a known software package, students have many examples (such as actual tax forms) from which to identify an appropriate data structure for the abstract syntax. The concrete syntax, by contrast, is designed to simplify the task of the specifier by providing syntactic short-cuts; its design is far more open ended and thus harder as a starting point for novices.

Designing Abstract Syntax: Students encounter certain common problems while designing abstract syntax, including:

1 The tax-form specification is separate from the data that the tax assistant gathers from an end-user. Many students include dummy placeholders or default values in the abstract syntax for the data to be requested on each line, (*e.g.* (*make-prompt* "enter wages" *false*) in the abstract syntax of Figure 2). The *false* serves as a placeholder for the actual wages value that a user will enter when running the tax program. Students who do this generally cite principles about keeping related data together, without realizing the competing principle of separating instructions from data. This difficulty is compounded by the existence of the tax form's physical artifact, wherein tax-form instructions appear next to blank spaces for a user to fill in. This is a nontrivial step for some students as they learn to think about languages.

2 Meta-language identifiers are not inherently visible in the object language. Consider this incorrect program fragment:

```
(define line1 (make-prompt "Enter your wages"))
(define line2 (make-prompt "Enter your dividends"))
(define line3 (make-calculated (make-plus line1 line2)))
```

This code's author imagines that *line1* refers at run-time to a number representing the end-user's wage, when in fact it refers at compile-time to a piece of a tax-form abstract syntax.[1]

3 Languages must be more general than the artifacts they capture. In the section of tax forms capturing dividend amounts, for example, the printed form provides a fixed number of lines. The tax assistant, however, should use an arbitrary-length list rather than expect the number of entries printed on the form.

4 Arbitrary-size data structures require more interesting naming schemes. Each entry in a list of dividend amounts, for example, involves several pieces of information; it is effectively a *table* with a fixed number of columns and variable number of rows. Referencing data within tables introduces the need for addressing schemes, since students cannot introduce fixed names in instances of the abstract syntax.

Starting with abstract syntax helps students confront these issues directly. Asking students to define the data structures for their abstract syntax, express a representative subset of an actual tax form in those structures, and load that expression in the meta-language reveals many of these problems to the students. Ironically, then, the abstract syntax turns out to be more concrete!

Defining an Interpreter: Conceptually, a tax-form interpreter must process each line of the form by either prompting the user for data or computing a value based on previously-entered data. Figure 3 shows a possible fragment of the tax-form interpreter corresponding to the abstract syntax from Figure 2. Each line in the form, whether prompted or calculated, constitutes a binding which other lines may reference. Implementing bindings correctly tends to be the biggest conceptual hurdle for students: they struggle to separate bindings in the tax language from bindings in the interpreter, and to devise a run-time data structure to manage the bindings. Their attempts can generally be grouped into three categories:

1 Define a meta-language variable for each entered datum and reference those variables as values in other pieces of abstract syntax (as shown in item 2 at top of this page): Inability to rule out this option reflects fundamental misunderstandings about the idea that programs are data and the differences between object and meta-language.

[1] Curiously, this can be made to work as part of a macro solution, but the use of the host-language binding mechanism destroys the ability to, for example, compute the number of lines in a form or print out the completed form.

```
;; process-line : line env → env
;; requests or computes data for one tax-form line, storing it for future reference
(define (process-line aline env)
  (let ([instructions (line-instructions aline)])
    (add-to-env
     env
     (line-label aline)
     (cond [(prompt? instructions)
            (begin (printf (prompt-text instructions))
                   (read-line))]
           [(calculated? instructions)
            (evaluate-computed instructions)]))))
```

Fig. 3. Fragment of tax-form interpreter corresponding to the line structure.

2 Extend the abstract syntax with a placeholder for the answer, and mutate the abstract syntax as line values are determined: this approach admits fewer data structure definitions, but fails to separate abstract syntax from run-time data and requires repeated searches over the abstract syntax to look up previously-determined values.
3 Maintain a separate data structure linking forms and line numbers to values: this is the approach that most successful students eventually settle on.

Defining Concrete Syntax: Making the tax-form language available to tax-form specifiers requires a specification of concrete syntax and a mechanism to convert the concrete syntax into the abstract syntax. The specification and conversion method are closely linked, in that adopting constraints on the form of the concrete syntax enables different conversion methods. At one extreme, students could choose an arbitrary concrete syntax, but would need to write a parser to produce the abstract syntax. At the other extreme, students could adopt the meta-language's concrete syntax and avoid writing any conversion method: the bottom half of Figure 2, for example, is concrete syntax in Scheme that corresponds to the abstract syntax. The first approach requires more time than a project module might allow (especially if students aren't already familiar with parsing). The second is neither compelling nor motivating: students don't see data structures as languages.

Hygienic macros can provide a gentle solution between these extremes. The macro system's pattern language constrains the shape of the concrete syntax in exchange for producing abstract syntax via rewrite rules. There are many hygienic macro systems for functional languages (Kohlbecker *et al.* 1986; Dybvig *et al.* 1988; Clinger & Rees 1991; Mauny & de Rauglaudre 1992; Cardelli *et al.* 1994; Herman & Wand 2008), some of which are syntactically quite complex. We find that simple Scheme macros in the **syntax-rules** system pose little difficulty for students, especially given the abstract syntax to guide development of the rules. Indeed, our experience is that students find macros amusing and even fascinating; some try to create more elaborate compilations that require richer macro systems (such as **syntax-case**). Even within **syntax-rules**, we augment the macros with checks for syntax errors to inspire students to use macros in more sophisticated ways.

```
(define-syntax form
  (syntax-rules (export)
    [(form name (export var ...) entry ...)          ;; ⇒
     (define name (form-internals (export var ...) entry ...))]))

(define-syntax form-internals
  (syntax-rules (export field)
    [(form-internals (export var ...) (field name content) entry ...) ;; ⇒
     (let ([name content]) (form-internals (export var ...) entry ...))]

    [(form-internals (export var ...))               ;; ⇒
     (list (list 'var var) ...)]))
```

Fig. 4. Form expansion macros.

3.2 Language embedding

Tax-form evaluators can be implemented without interpreters. For advanced students, we instead espouse reuse: build the tax form as an embedded language atop an existing one (Hudak 1996; Clements *et al.* 2001, 2004). Rather than specifying an abstract syntax and an interpreter for it, students define a concrete syntax that is an extension of a host language, and use hygienic macros to map this syntax onto the native abstract syntax of the existing evaluator.[2] They give up control over the surface syntax of the language, and in return they get to reuse the existing evaluator.

In a design such as this, there is no need to specify abstract expression forms (*e.g. make-plus*) or to define their meanings; tax-form specifiers can simply reuse the + of the host language. This approach concretely demonstrates the advantages of reuse, not just in an implementation but also in a design sense. Reuse is particularly important in prototyping, where the goal is to produce a working program in a limited time.

Figure 4 shows a simplified set of macros for expanding forms. For brevity, these examples omit error-checking clauses. The tax-form expansion is implemented as a simple two-level macro. Each tax-form expands into a single (**define** ...) statement, containing a cascaded (**let** ...). Each form binds its name to an association list containing pairs of exported field names and their associated values.[3]

The macros do not show the (**calculated** ...) terms from the language in Figure 2, but handling them is literally trivial: (**calculated** *exp*) expands into *exp*. The syntax of these terms is simply that of Scheme itself. This is the most vivid illustration of the work saved by working with macros; rather than implementing a heavyweight interpreter for calculations in dozens of lines, a two-line macro suffices.

[2] There are competing definitions of the term "domain-specific embedded language". We use it to refer to a simple extension of the host-language semantics using macros.

[3] Using host-language variables for form values is appropriate in the macro context, whereas it constituted an error in the interpreter context. This reinforces the distinction between meta and object language, which is blurred in the macro context.

Students confront several issues in writing the macro-based embedding:

- The seeming lack of work involved: Macros can confound students at first, because students expect programs as complex as a tax assistant to require a substantial amount of code. We frequently see students steer themselves away from elegant solutions in macros that seem too easy. This experience is useful for helping students think differently about languages; many gain new appreciation for programs as data through writing macros for nontrivial tasks.
- Tradeoffs between exposing implementation details and complicating the macros: as an example, consider tables (item 4 on page 7). Scheme has a well-developed set of operations on lists. The student could represent a table as a list, and simply expose this list to the author of the form. Exposing the Scheme list (and its accompanying library functions) in the domain-specific language makes expansion simple, but requires more host-language knowledge on the part of those programming in the domain-specific language. It would also be difficult to prevent errors such as "car of null" in such an implementation. A student might alternatively choose to build a set of custom table-access functions for each table that a form contains. This expansion would require more work, but might discover certain form errors more quickly, and would probably lead to more readable forms. For instance, this style might allow named references to tables:

  (**field** ... (**calculated** (*sum-up* (*table-ref foreign-taxes tax-paid*))))

  In practice, such tradeoffs expose students to subtle challenges of language design.
- Restricting the language: clever students realize that their tax-form language can contain arbitrary source code, thus allowing tax-form specifiers to embed hostile code with no relationship to a tax-form computation. Most students do not address this issue, but the questions it raises are educational for those who discover it.

Graham's example of embedding a database query language in Lisp through macros (Graham 1994) provides another interesting example of this approach. Of course, macros are not the only way to build a domain-specific language by linking the meaning of the new language to the meaning of an existing one. Two competing approaches – one offering less control over the details of the language, one offering more – are presented by Haskell and Ziggurat (Fisher & Shivers 2008). Using a "combinator library" approach in Haskell (Hudak 1996) leverages laziness, type classes, and monads to make it possible to dramatically extend the language with new values, operations, and pattern-matching forms (Rhiger 2009) without adding a macro-like transformation system. At the other extreme, systems such as Ziggurat promise the ability to equip each language extension with its own semantics, analysis, and other tools. In Ziggurat, a language consists of a tower of languages, where each additional layer expands and compiles into the next one down, and static analyses may be inherited and extended. Students would have great flexibility in designing language extensions, and the corresponding responsibility for implementing specialized extensions to the existing analysis tools.

4 Advanced features and concepts

Whenever we assign students large projects (such as a language implementation), we espouse iterative refinement, building towards the final system. We first implement a core of the system, then augment the system's features. For the tax-forms project, each of the extensions for type checking, assertions checking, and multiple sessions requires modifications with interesting linguistic content.

4.1 Type and error checking

A basic tax assistant probably performs little to no error checking, neither for program-specification errors nor for run-time errors. At run time, a taxpayer could enter non-numeric data in response to a prompt for numeric data. At specification time, tax-form authors could insert unbound references or circular data dependencies. Tax forms are also subject to type errors, particularly with respect to units of measure (so-called *unit-checking* (Kennedy 1997; Allen *et al.* 2004; Antoniu *et al.* 2004)). Consider the following example. A taxpayer can claim a reduction in taxable income for himself, his spouse, and all of his dependents. The total deduction for dependents in 2008 is calculated by multiplying the number of dependents by US$3500. Multiplying a simple number by a dollar amount is fine, and the resulting unit is the dollar. If the developer of the tax form were to mistakenly add these numbers rather than multiplying them, the resulting total would be a nonsensical combination of dollars and people.

Extending the language to support error checking is a natural next step once the core tax system is working. Unbound-reference errors can be checked using macros, or as a separate phase in the interpreter. We find the former particularly instructive, as it shows macros doing work beyond rewriting.

To handle run-time data-entry errors, we ask students to extend their syntax with type declarations for **prompt** expressions:

(**prompt** "Enter wages" *number*)

Their implementations then check that entered data conforms to those types, re-prompting the taxpayer if necessary.

For unit-checking, we show students how to extend their syntax with type labelers and their implementations with variants of standard operators that are aware of the unit types. This is much simpler than an approach such as Kennedy's (1997), which addresses unit-checking problems via the addition of relational parametricity to an existing type system. Tax forms, for example, need numeric values representing both dollar amounts and scalars. We therefore introduce the following pair of macros per unit type:

(**units** <*type*> *exp*) ⇒ (*make-*<*type*> *exp*)
(**prompt** *query-string* <*numeric-type*>) ⇒
(**units** <*numeric-type*> (*num-prompt query-string*))

We then expand tax-form programs to label constants appropriately, as in

(**field** *deductions* (**calculated** (* *num-dependents* (**units dollars** 3500))))

then rewrite multiplication in terms of these units

```
(define (tax-* a b)
  (match (list a b)
    [(list (struct dollarv (d)) (struct scalarv (s)))   (make-dollarv (* d s))]
    [(list (struct scalarv (s)) (struct dollarv (d)))   (make-dollarv (* d s))]
    [(list (struct scalarv (s1)) (struct scalarv (s2)))  (make-scalarv (* s1 s2))]))
```

Topics such as unit checking are most appropriate for senior students. With younger students, we cover only unbound-reference checking. Covering some form of error checking with these students has proven extremely useful, however. Once we reduce language implementation to writing and processing data structures for programs, many students begin to ask what distinguishes a language from a library (recalling Gosper's quote that "A data structure is nothing more than a stupid programming language." (Hewitt *et al.* 1973)). Ideally, languages include both static and dynamic constraints on well-formed programs. This idea, that languages embody principles of use as well as computation, only starts to take root when students implement language-specific error handling.

4.2 Form invariants

Invariant-checking, like unit checking, requires language extensions. Students who wish to add assertion checking may add a tax-form construct similar to the following (building off the identifiers in Figure 2):

```
(assertion (if big-dividends
               (> dividends (units dollars 1500))
               (<= dividends (units dollars 1500))))
```

Assertion checking poses a nice contrast to unbound-identifier checking, since it typically requires dynamic, rather than static, checks.

4.3 Multiple sessions, revisions, and out-of-order evaluation

A simple evaluator would force users to work through a tax form in order in a single session. More sophisticated tools could allow users to edit previous answers, save and resume previous sessions, or complete sections of the form in an order of their choosing. These features raise advanced linguistic topics, such as dataflow programming (to automatically propagate edits), continuations (to save and resume session state), or laziness (to compute form data as needed in other forms). While we sometimes use these features as motivators for these topics in full-fledged programming languages courses, we do not introduce them when using these projects as modules in other courses.

5 Other domain-specific language problems

In addition to tax forms, we have used several other domain-specific languages in a similar manner. Each highlights different language-design issues.

Slideshows (a.k.a. PowerPoint): Slideshow presentations provide a compelling language design example for two main reasons: first, students are extraordinarily familiar with PowerPoint; second, PowerPoint's limited abstraction mechanisms and control flow operators motivate the benefits of building products around domain-specific languages. Concretely, our slideshow language covers the following linguistic topics:

- Conditionals: we introduce a feature that bases slide sequencing on the time that has elapsed since the start of the talk.
- Variables: we compute some slide data dynamically (such as example numbering – this is useful once conditionals allow us to skip slides). We contrast a narrow construct for dynamic example numbering with a more general implementation of program variables.

The second point helps students understand that language design is about tradeoffs: what we choose to exclude is often as important as what we choose to include.

An Automated Testing Service: Computerized testing systems administer exams in which different questions may be posed to different people based on their performance so far. During exams, people may also receive feedback about their performance on particular topics. Specifications of alternative questions, question sequencing, and when to provide feedback constitute a domain-specific language. Our version of this problem features multiple question styles (multiple choice and free-response) and optional hints, as well as question sequencing and feedback. This example highlights the following linguistic concepts:

- Conditionals: these arise from sequencing questions and their alternatives.
- Structuring program data for querying: giving feedback to users requires tabulating a user's performance on different categories of questions. Students contrast tabulating questions on a per-section basis with tagging data and allowing queries over those tags (which engenders another language-design question).
- Separation of model and view: should layout information for multiple-choice questions be built into the abstract-syntax data structure or customized externally?
- Web-based control flow: if the interpreter uses the web to display questions and process answers, does the system work properly if the test-taker uses the back-button during the exam? This is fundamentally a question of environments versus stores.

Animations Scripting: Students create a language for scripting basic animations over interacting objects. Our objects are basic shapes (circles and rectangles) that can move across the screen, change size, jump to new locations, collide with one another, and appear or disappear during the animation. This example highlights the following linguistic concepts:

- Conditionals and defining predicates: animated objects change behavior when they collide. Do we implement general predicates to capture collisions or special-purpose constructs for pre-defined types of collisions?
- Variables: animated objects may have attributes defined in terms of variables that change during the animation (such as a circle and square with dimensions computed from a shared program variable).
- Parallel versus sequential operations: some animations are easier to express through independent operations executing in parallel rather than purely sequential execution.

State Machine Simulation: A simple state-machine simulator consumes the description of a state machine and a list of input symbols. The simulator produces either a trace of corresponding outputs or a simple flag indicating whether the input list is recognized by the state machine. This example highlights the following linguistic concepts:

- Interpretation versus compilation: this example is small enough that students can implement two versions of it in a short time frame. One version converts the state-machine syntax to a data structure of states and transitions which an interpreter must then simulate against inputs. Another version, due to Krishnamurthi, compiles state machines into mutually-recursive functions, each of which consumes the remaining inputs (Krishnamurthi 2006). While this may seem less like a domain-specific application than the others, we have found it resonates well with engineering students.
- Error checking: typos can abound in state-machine descriptions. A useful language would check for errors such as unbound state names in transitions. This raises questions of what languages should do for programmers, thus distinguishing languages from mere data structures.

6 Experience

Both authors have used domain-specific language design exercises in the classroom, but in different kinds of courses with different levels of students.

In an Advanced Freshman-Level Course: The second author has used all of these examples in an accelerated introductory Computer Science course at the college level. Her course, aimed at first-semester college students with prior programming experience (usually a year of Java in high school), spends roughly 10 lecture hours introducing functional programming (including lists, trees, and higher-order functions), then another 10 hours on domain-specific language design and implementation. The lectures work through the slide show example to introduce ASTs as a data structure, writing interpreters, and macros.[4]

[4] Notes, pacing, and exercise descriptions are at `http://www.cs.wpi.edu/~cs1102/a08/`.

Students do assignments based on two of the remaining examples. One example – often the automated tester – becomes two assignments: a homework in which pairs of students design abstract syntax for the full example, and a lab in which students write an interpreter for a core of the example. Another example – the tax form or the animations language – serves as the course project: students individually design the AST, implement an interpreter, and optionally provide a clean concrete syntax via macros.

Analysis of student projects and grades across four offerings of this course show that almost all of the students understood the concepts well enough to design a basic AST and interpreter. In fact, instructor experience suggests that even those who receive poor grades can attribute their failures to late starts and poor planning, rather than difficulty with the material. Additionally, some students go beyond the stated assignment, embellishing their languages in interesting and original ways.

In-class reviews of AST designs contribute to our success with these assignments. For each AST-design assignment, we conduct a class-wide critique of three or four different styles of designs. Students are very engaged in these design reviews. Typical issues raised by students include whether to model certain constructs in the meta-language or the object language, whether multiple constructs could be abstracted into common core constructs, and whether a construct design is sufficiently flexible to accommodate reasonable extensions to the language. Analysis of grade data shows that a significant fraction of students had poor designs at the AST phase, but got C-grade or better implementations working by the final deadline. We find many students understand the concept of an interpreter more readily than the concept of capturing programs as data. These students benefit from seeing multiple examples of plausible ASTs during the design reviews. We encounter relatively few students who are able to write ASTs and not able to produce simple interpreters. That we are able to achieve these results with students in the first semester of college speaks to the power of domain-specific languages as a project topic.

We also attribute success with this audience to our choice of functional programming curriculum and meta-language. This course uses *How to Design Programs* (Felleisen *et al.* 2001), which teaches students to design programs by first defining their data then deriving the program structure from the data. Using this approach, abstract syntax leads directly to the structure of the interpreter. PLT's pedagogic Scheme programming environment (Findler & PLT 2009) provides a hygienic, source-correlating macro system that supports both **syntax-rules** and **syntax-case** forms. PLT Scheme also supports images as first-class values (Felleisen *et al.* 2009), which simplifies the animations-language exercise.

In a Junior-Level Software Construction Course: The first author (jointly with Matthias Felleisen) used the tax-form example in a junior-level software construction class at Rice University in 1998. Students had a choice of implementation style, host/meta-language, and final feature set. The scope and quality of the students' projects varied widely. Some students were content to produce a bare-bones interpreter that was little more than a spreadsheet, while others undertook pretty-printers and sophisticated input mechanisms.

To assess whether the students' languages could accommodate changes in the tax forms, teams exchanged projects late in the semester. Each team was charged with updating another team's project to accommodate the changes occasioned by the annual update to the tax forms. Each receiving team assessed the ease with which they could adapt their given code base; this feedback had substantial weight in the authoring team's project grade.

7 Perspective and conclusion

Domain-specific languages for concrete software applications provide fertile ground for teaching students about programming language design. Implementing a domain-specific language forces students to confront many of the same issues that arise when implementing the core of a more general-purpose language. Focusing on domain-specific languages that underlie recognizable software products, however, casts the problem in software engineering terms that students find compelling. Moreover, these applications provide concrete artifacts that help students get off the ground with language design.

Our approach contrasts most directly with that of teaching programming languages via interpreters. That approach tends to have students implement either a small abstract language core or fragments of real languages (sometimes using the same language as both meta and object language). While we use this approach successfully in our own upper-level programming languages courses, the domain-specific examples have allowed us to bring this material into nonstandard courses with a broader range of students. In our experience, many first-year students simply don't understand the concept of meta-circular interpreters (à la SICP (Abelson *et al.* 1996)) at all, while finding domain-specific language interpreters an exciting challenge.

Both the interpreter- and macro-based implementations offer pedagogic strengths. On the one hand, the macro solution emphasizes reuse at many levels: the host-language syntax, implementation, and the programming environment tools. As a result, the solution is approximately one tenth the size of the full interpreter. The time savings are similar. This implementation style teaches students the importance of reuse, as well as language features to look for in a potential host language. The coolness factor of doing a complicated task with macros is also appealing and motivating to many students.

On the other hand, the interpreter lies on a clear conceptual path from other programming problems, and is thus easier for less experienced students to grasp. We also find the interpreter forces students to confront certain fundamental issues that are easy to overlook in a macro-based solution (such as identifiers in the object language). In the interpreter, students have to manage their own mapping between identifiers and values. In the macro-based solution, students can fall through to treating identifiers in the host language with simple binding forms as shown in Figure 4; while elegant, this masks the core issues of namespace separation.

Our positive experiences with domain-specific languages projects should be widely replicable. Hopefully, this report can guide others in designing their own projects.

Naturally, there are many other real-world applications of our ideas, and we are hoping that the community will create a repository of similar case studies for the benefit of all language instructors.

References

Abelson, H., Sussman, G. J. & Sussman, J. (1996) *Structure and Interpretation of Computer Programs*. 2nd ed. McGraw-Hill.

Allen, E., Chase, D., Luchangco, V., Maessen, J.-W. & Jr., Guy, L. S. (2004) Object-oriented units of measurement. *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, pp. 384–403.

Antoniu, T., Steckler, P. A., Krishnamurthi, S., Neuwirth, E. & Felleisen, M. (2004) Validating the unit correctness of spreadsheet programs. *International Conference on Software Engineering*. ACM, pp. 439–448.

Bentley, J. (1986) Little languages. *Communications of the ACM*, August, ACM, 711–721.

Cardelli, L., Matthes, F. & Abadi, M. (1994) *Extensible syntax with lexical scoping*. Research Report 121. Digital SRC.

Clements, J., Graunke, P., Krishnamurthi, S. & Felleisen, M. (2001) Little languages and their programming environments. *Proceedings of the Monterey Workshop on Engineering Automation for Software Intensive System Integration*. ACM, pp. 1–18.

Clements, J., Felleisen, M., Findler, R., Flatt, M. & Krishnamurthi, S. (2004) Fostering little languages, *Dr. Dobb's Journal*, March, 29 (3): 16–24.

Clinger, W. & Rees, J. (1991) Macros that work. *Pages 155–162 of: ACM SIGPLAN Conference on Principles of Programming Languages*, ACM.

Deursen, A. v., Klint, P. & Visser, J. (2000) Domain-specific languages: An annotated bibliography, *ACM SIGPLAN Notices*, 35 (6): 26–36.

Dybvig, R. K., Friedman, D. P. & Haynes, C. T. (1988) Expansion-passing style: A general macro mechanism, *LISP and Symbolic Comput.*, 1 (1): 53–75.

Felleisen, M., Findler, R. B., Flatt, M. & Krishnamurthi, S. (2001) *How to Design Programs*. MIT Press.

Felleisen, M., Findler, R. B., Flatt, M. & Krishnamurthi, S. (2009) A functional I/O System or, fun for freshman kids. *ACM SIGPLAN International Conference on Functional Programming*. ACM, pp. 47–58.

Findler, R. B. & PLT. (2009 July) DrScheme: PLT programming environment. Reference Manual PLT-TR2009-drscheme-v4.2.1. PLT Scheme Inc. Available at: `http://plt-scheme.org/techreports/` Accessed 4 January 2010.

Fisher, D., & Shivers, O. (2008) Building language towers with Ziggurat, *J. Funct. Program.*, 18 (5–6): 707–780.

Fisher, K. & Krintz, C. (2008) 2008 SIGPLAN programming language curriculum workshop report, *ACM SIGPLAN notices*, 43 (11), 29–30.

Flatt, M. & PLT. (2009 July) *Reference: PLT scheme*. Reference Manual PLT-TR2009-reference-v4.2.1. PLT Scheme Inc. `http://plt-scheme.org/techreports/` Accessed 4 January 2010.

Friedman, D. P., Wand, M. & Haynes, C. T. (2001) *Essentials of Programming Languages*. 2nd ed. MIT Press.

Graham, P. (1994) *On Lisp: Advanced Techniques for Common Lisp*. Englewood Cliffs, NJ: Prentice-Hall.

Herman, D. & Wand, M. (2008) A theory of hygienic macros. *Lect. Notes Comput. Sci.*, 4960: 48–62.

Hewitt, C., Bishop, P. & Steiger, R. (1973) A universal modular actor formalism for artificial intelligence. *International Joint Conference on Artificial Intelligence*. pp. 235–245.

Hudak, P. (1996). Building domain-specific embedded languages, *ACM Comput. Surv.*, 28 (4es): 196.

Kamin, S. N. (1989) *Programming Languages: An Interepreter-Based Approach*. Addison-Wesley.

Kennedy, A. J. (1997) Relational parametricity and units of measure. *ACM SIGPLAN Conference on Principles of Programming Languages*. ACM, pp. 442–455.

Kohlbecker, E. E., Friedman, D. P., Felleisen, M. & Duba, B. F. (1986) Hygienic macro expansion. *ACM Symposium on Lisp and Functional Programming*. pp. 151–161.

Krishnamurthi, S. (2006) Automata via macros, *J. Funct. Program.*, 16 (3): 253–267.

Krishnamurthi, S. (2007) *Programming Languages: Application and Interpretation*. Self-published.

Mauny, M. & de Rauglaudre, D. (1992) Parsers in ML. *ACM Symposium on Lisp and Functional Programming*. ACM, pp. 76–85.

Rhiger, M. (2009) Type-safe pattern combinators, *J. Funct. Program.*, 19 (2): 145–156.

Sabry, A. (1999) Declarative programming across the undergraduate curriculum. *Workshop on functional and declarative programming in education*. Rice University TR99-346, online at `http://www.ccs.neu.edu/home/matthias/FDPE99/` Accessed 4 January 2010.