Preface

Working programmers engineer software systems. The typical engineering process consists of a number of phases, including the collection of requirements, the creation of a specification, the design of the architecture, the design of the modules, the maintenance of the system, and so on. In many cases, the engineering process involves iterative approaches, with repetitions across various phases in the process.

Working semanticists engineer models of programming languages. Such models almost always include at least one semantics, often several. Like the engineering of a software system, the creation of a semantics requires many different steps, including the formulation of a model, its exploration with test cases, example-based validation of invariants, theorem-proving, typesetting, and more. Also like the engineering of software systems, the engineering of a semantics is an iterative process, not a linear one.

This text is addressed to the working semantics engineer who needs a lightweight modeling tool. Typically these are graduate students and working programmers who find themselves engineering the semantics of a programming language or, more likely, an extension of a programming language. An ideal reader knows the functional programming language Scheme and has significant experience with representing information via S-expressions. A regular reader has a working knowledge of some functional language (Erlang, Haskell, ML) or has extensive experience programming functionally in a conventional language. In our experience, the key is a familiarity with programming over algebraic data types.

Organization. We have organized the text into three parts. Part I presents a framework for the formulation of language models, focusing on *equational calculi* and *abstract machines*. Roughly speaking, a minimal model of a language requires the specification of the core of the syntax and its meaning. This book uses a *syntactic approach* to the formulation of meaning. Put differently, the meaning of the core syntax is mostly formulated through binary relations on itself. We illustrate the approach and its major ideas with several small programming languages inspired by Church's λ-calculus.

Part II introduces *software tools* for expressing the models from part I as PLT Redex programs. In principle, Redex is just a small domain-specific programming language, hosted in PLT Scheme. A typical Redex model consists of two parts: a grammar for the language syntax and a binary relation on

the syntax. Put differently, formulating a language model in Redex requires as little effort as writing it down with paper and pencil. One immediate advantage of Redex programs is that these models are executable. It is thus possible to explore the models with examples; to create and run test suites that are also useful for an eventual implementation; to check whether examples satisfy some invariant; to typeset the model. Because Redex is an embedded programming language, semantics engineers may also use plain Scheme functions for the model and may even import their own tools. Part II explains how to do all this with examples from part I and more.

Part III collects seven contributions on a wide range of models, all formulated in Redex. In the first contribution, Carl Eastlund explains an intermediate point in the design of a variant of ACL2 with modules; the Redex model is used to explore how to map modules to ACL2's evaluator and, separately, to the theorem prover. Martin Gasblicher, in the second contributed article, presents the first rewriting model of macro expansion for the R^5RS version of Scheme; the Redex model allows programmers to step through the expansion of macro expressions. For the third article, Kathryn Gray extends her work on interoperability between untyped programming languages and typed ones; her Redex model combines the semantics of two distinct languages in one in order to investigate soundness issues. Joe Hallet, Eric Allen, and Sukyoung Ryu use a Redex model to explain one intricate part of Fortress's type system. The fifth article continues George Kuan's series of articles on type checking as a reduction system; thus, his Redex model uses reductions for modeling both the type checking process and for the evaluation process. With the penultimate contribution, Jacob Matthews describes his development of a web-based interview specification language. Last but not least, Mike Rainey demonstrates how Redex models are helpful for developing scheduling algorithms for a multi-core world.

To the Instructor. We have used the first two parts in graduate courses on topics in programming languages. A typical course covers the first two parts in four to eight weeks, depending on the familiarity of students with Scheme and/or functional programming. While we assign weekly homework projects for this period, we also ask students to work out Redex models for papers we pick. Students present the results of these projects in class. Equipped with this background, the course tackles additional topics. Sometimes students develop Redex models for papers of their choice; at other times, we jointly investigate historical developments of ideas.

To the Student. In addition, we have used the notes to train individual PhD students with a self-study plan. These students typically work

through parts I and II in parallel so that they can immediately use the theoretical lessons to build practical models. Since most of our students end up extending a programming language, and since most of these designs are long-term efforts with many iterations, we consider it critical that students become intimately familiar with Redex (and other lightweight modeling tools).

Many times, our students iterate the model development to the point where we can reuse the test suites from the model for an actual implementation of a language extension. Our paper at the 2007 International Conference on Functional Programming on engineering functional and delimited continuations for a realistic programming system is a particularly good example of this kind of language development. We conjecture that all students of programming languages will benefit from such work flows.

Software. PLT Redex comes with the PLT Scheme implementation, which is available for free at

```
http://www.plt-scheme.org/
```

A Redex program is just a Scheme module that imports the Redex language. We urge you to download the software before you start reading so that you can experiment with Redex as you work your way through the book.

Programmers do not have to know much about Scheme to use Redex. Programmers who do know Scheme, especially PLT Scheme, can easily escape into the host language and build their own extensions. Jacob Matthew's contribution, for example, is the specification of a domain-specific web programming language; his model's reduction relations open a web browser to interact with the user.

Like all software, Redex isn't perfect. For that reason, DrScheme comes with a Help Desk facility and an error reporting interface. The former explains our intent and contains details that are beyond the scope of a text like this one. We intend to continue the development of Redex, and you will be able to find the documentation for these extensions and revisions in Help Desk. The error reporting interface is useful both for contacting us concerning failures and for requesting additional features.

In addition to Redex, our web site also hosts the models in this book, additional examples, and future contributions. Visit

```
http://redex.plt-scheme.org/
```

and also use it to share your favorite Redex models with others.

Acknowledgments. We thank Daniel P. Friedman for inspiring the exploration of reduction semantics and for pushing us to implement our ideas. Lambda, lambda, and once more, lambda.

Robert Hieb (1953–1992) and Matthias jointly worked out the style of reduction semantics used in part I in 1989 and 1990. Their collaboration was tragically shortened by a traffic accident; Bob is dearly missed.

Thanks to Robby's Tutu for the use of her lānai, where he built the first version of Redex. And thanks to Jacob Matthews and Casey Klein for numerous hours of work on Redex.

Over the past decade, many people have read drafts of part I and have used early versions of Redex. Three stand out for their use of our work and their feedback: Olivier Danvy, Johnathan Edwards, and Bill Richter. In addition, the book has benefited from the feedback of several generations of students who took our courses. They are too numerous to enumerate here, but we certainly we owe them a big "thank you."

Finally, we wish to thank our editors at the Press. Bob Prior encouraged us for years to turn these notes into a book, and Ada Brunstein took us over the final hurdles to make it happen. Mel Goldsipe helped us polish our writing and that of our contributors.

Of course, the remaining errors are all ours.

Matthias Felleisen, Robby Findler, Matthew Flatt