STATIC SEMANTICS FOR SYNTAX OBJECTS

A dissertation presented by

David Fisher

to the College of Computer and Information Science
of Northeastern University
Boston, Massachusetts

in partial fulfillment of the requirements for
Doctor of Philosophy

April 22, 2010

This material is based upon work supported by the National Science Foundation under Grant No. 0757025.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
Ph.D. THESIS COMPLETION APPROVAL FORM

THESIS TITLE: Static Semantics for Syntax Objects

AUTHOR: David Fisher

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Thesis Advisor	*Date* 2010/6/24
Thesis Reader	*Date* 6/24/10
Thesis Reader	*Date* 6/16/2010
Thesis Reader	*Date* 2 June 2010

GRADUATE SCHOOL APPROVAL:

Director, Graduate School	*Date* 6/24/10

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Recipient's Signature	*Date*

Distribution: *One Copy to Thesis Advisor*
One Copy to Each Member of Thesis Committee
One Copy to Director of Graduate School
One Copy to Graduate School Office (with signature approval sheet, including all signatures)

Acknowledgments

I am indebted to numerous people for helping and supporting me through the development of this dissertation. First and foremost, I would like to thank my committee, Mitch Wand, Riccardo Pucella and Alan Bawden, who have provided indispensable insight and suggestions. My advisor, Olin Shivers, has guided me for the majority of my education, and has provided the teaching without which none of this would have been possible.

I have had the privilege of working with numerous professors at Georgia Tech and Northeastern throughout my graduate career. I would like to particularly single out and thank Pete Manolios, Yannis Smaragdakis and Mary-Jean Harrold at Georgia Tech, and Matthias Felleisen and Will Clinger at Northeastern University, in addition to my advisor and thesis committee. In addition, I would like to acknowledge Andrew Appel at Princeton University for teaching me early in my graduate career, as well as writing the work that originally got me interested in language theory, *Compiling with Continuations.*

In addition, I have had the pleasure of learning and working alongside some of the most brilliant students I have ever known. At Georgia Tech, I had the pleasure of working with Daron Vroon, Shan Shan Huang and Matt Might. At Northeastern, I was welcomed into the Programming Research Laboratory, where I met Dave Herman, Richard Cobbe, Theo Skotiniosis, Sam Tobin-Hochstadt, Alex Heller and Christos Dimoulas.

I could not overstate the importance of my family in my academic and personal growth. My mother, father and sister, Elizabeth, Josh and Dora Fisher have provided the unconditional love and support that have made me into the person I am today.

Finally, but most importantly, I would like to thank my wife, Allison Fisher, and my son, Ari Alexander Fisher. Allison, you have truly supported me in good times and bad, and your love has given me the strength to complete this. Ari, you were born during my thesis work, and watching and helping you grow has inspired me just when I needed it most.

Contents

Chapter 1

Thesis

Designing custom notations is essential to designing advanced systems. Although an electrical engineer could write an English description of a circuit board, explaining which components were used and how each was connected, it is not very effective for him to do so; he would do much better to draw a circuit diagram. This not only streamlines the process of circuit architecture, allowing commonly-used design idioms to be expressed concisely, but it also allows a reasoning engine, such as an automated design tool or the electrical engineer himself, to reason about the circuit, identifying redundancies, determine its behavior and find potential problem points.

In programming, the custom notations we use are programming languages. Languages can target a specific computer architecture, can be intended for general-purpose programming, or can be tailored to a particular problem domain. The pre-testing reasoning power of the language is expressed as its *static semantics,* and are of vital importance in efficient compilation and in preventing bugs.

I have constructed a design framework that allows a language designer to:

- design a custom notation for a particular domain,
- associate rich static semantics with this notation,
- provide a realization of this notation by translating programs written in it into a "lower-level" language, and
- link the static semantics of the custom notation with those of the lower-level language.

The framework allows this by providing a macro system with two major features:

- syntax is represented as a tree of objects with classes, and
- methods are dispatched on these objects through a system called *lazy delegation.*

The thesis of this work is: **Object-oriented abstract syntax and lazy delegation allow language designers to build custom notations with rich static semantics, and to link these semantics with those of lower-level languages.**

Chapter 2

Introduction

Implementing languages is an arduous and tedious task. A well-known method of accelerating the process is by extending an existing language via macros, or by embedding the new language through rewriting it completely. However, newly-designed languages frequently include new static semantics, which may be lost in the translation process. My goal in this work is not only to add static semantics to macros, but show that it is necessary and possible to link these semantics across language levels. This work describes an object-oriented approach to doing so, called lazy delegation.

2.1 Language extension

A programming language is an embodiment of a way of thinking about computation, and different languages are best suited to different tasks. If one were to come up with a new model useful for programming in a certain domain, say, for querying databases, or for searching for patterns in DNA strings, it would be natural to express this abstraction as its own language. A logic-programming language might consist of a set of boolean equations, while a pattern-matching language might consist of syntax used to build regular expressions.

However, expressing abstractions as languages comes with a rather large drawback: designing and implementing a language is a big task. The designer of the database-query language is not concerned with the details of specifying a type system for calling functions, or of optimizing function closures, nor should he be: these details will merely distract from the task at hand. So, the natural solution is to take an existing language and extend its specification and implementation. To take the database example, perhaps the language designer wishes to embed this language in Java.

This method of language development presents its own implementation challenges. Since the designer is extending an existing language, a direct way to do so is to modify an existing implementation of that language, say, a Java compiler. But compilers are notoriously complex pieces of software. We would rather avoid tinkering with compiler internals if at all possible. Often, the best way to do this is by rewriting programs written in a specialized notation into another user-level language. For the database

example, we might translate the database query into Java primitives. In addition to allowing the designer to use an existing implementation, doing so allows him to use all of the language-analysis tools available for the language, including debuggers, interactive development environments, and verification tools. This approach was used to build an implementation of Java called ProfessorJ [37] out of the PLT Scheme implementation and tool suite. Rewriting removes the need to deal with code already in the original language; Java compiles to Java. This advantage was used to great effect in an early implementation of C++ as an extension to C [95], as well as in writing a number of special-purpose languages, such as the Yacc [44] parser generator, and the Lex [59] lexer generator.

A powerful method of language extension via rewriting in widespread use today is the macro system included in the LISP family of languages, notably Scheme [47]. If the language designer were to implement the database language as an extension to Scheme, he would translate a query in the specialized notation into pure Scheme code, either with a pattern-matching language (these are known as "high-level" macros) or with pure Scheme code (these are "low-level" macros). Scheme macros are very versatile, and have been used to implement object-oriented languages [73], Unix shell scripting tools [87], database query languages [112], and logic programming languages [13].

2.2 Extending a language with static semantics

Since Scheme's macro system has proven itself useful for extending Scheme, a natural experiment is to take that macro system, apply it to other languages, and determine whether it is adequate for extending those languages. In order to determine this, I have taken a version of the `syntax-case` algorithm [38], and used it to build a custom pre-processor for C [43]. There were a few barriers to doing so.

- Scheme's macro system is designed to work with an s-expression based grammar. This was solved by making the input to the pre-processor be an s-expression based variant of C, called C/sexp.
- C has several namespaces, while Scheme only has one. Additionally, C defines a number of different kinds of scope for identifiers, while Scheme only has nested lexical scope. This complicates the way identifiers are disambiguated in the `syntax-case` algorithm; however, this was fixable with minor modifications to the algorithm, which will be covered in the chapter on parsing.
- C has several non-terminals the language designer would want to extend, while Scheme macros only allow the programmer to design new expressions. This was solved by including syntax into C/sexp to allow the macro writer to extend several sorts of non-terminals.

Is this naïve approach adequate for language extension? This system suffices for simple macros. For example, a language feature that C lacks is a `cond` statement similar to Scheme, also known as a multi-armed if statement. With this macro system, it is possible to add that to the language. In a simple implementation of this approach, such a macro might look like:

```
(statement cond
  (lambda (env form)
    (syntax-case form (else)
      ((cond (else action))
       (lambda (env form)
         (parse-csexp-stm: env (syntax action))))
      ((cond (cnd action))
       (lambda (env form)
         (parse-csexp-stm: env (syntax (if cnd action)))))
      ((cond (cnd action) . more)
       (lambda (env form)
         (parse-csexp-stm:
          env (syntax (if cnd action (cond . more)))))))))
```

In a context where this macro is defined, the C/sexp statement

```
(cond
  ((== x 1) (return "one"))
  ((== x 2) (return "two"))
  (else (return "many")))
```

would expand into

```
(if (== x 1) (return "one")
    (if (== x 2) (return "two")
        (return "many")))
```

which, in turn, generates the C code

```
if ((x_0)==(1)) {
return "one";
}
else {
  if ((x_0)==(2)) {
    return "two";
  }
  else {
    return "many";
  }
}
```

where `x_0` is a new identifier corresponding to `x` in the input program.

When writing more complicated macros, problems begin to arise. Consider adding list types to C/sexp. We would like to be able to write a function to take the sum of a list of integers:

```
(fun sum-list int ((l (list int)))
  (if (null? l)
      (return 0)
      (return (+ (head l) (sum-list (tail l))))))
```

We cannot implement these operators as functions, because C does not provide polymorphic types; for example, the function `head` would have to have type `(list` α`)` $\rightarrow$ α, which cannot be specified in C. [1]

There are a number of ways to implement this as a macro. One is to implement the type `(list` α`)` as a pointer to the structure `(struct` i `((car` α`) (cdr (*` `(struct` i`)))))`, where i is a fresh identifier. The function `sum-list` above thus expands to

```
(fun sum-list int
     ((list (* (struct g100 ((car int)
                             (cdr (* (struct g100))))))))
  (if (== list NULL)
      (return 0)
      (return (+ (-> list car) (sum-list (-> list cdr))))))
```

This works reasonably well. However, there is a problem in using the `sum-list` function. The type `(struct g100 ((car int) (cdr (* (struct g100)))))` only is in scope for the declaration of `sum-list`. If we use `sum-list` somewhere else in the program, such as in

```
(var foo (list int))
(sum-list foo)
```

this expands to

```
(var foo (* (struct g100 ((car int)
                          (cdr (* (struct g100))))))
(sum-list foo)
```

The use of `struct g100` in the variable declaration of `foo` refers to a distinct structure from `struct g100` in the declaration of `sum-list`. This causes an implicit typecast of the variable `foo` upon passing it as an argument to `sum-list`. This implicit typecast makes it very easy to slip in bad code; C's type system cannot distinguish between this use-case and

```
(var foo (* int))
(sum-list foo)
```

What we would like for all uses of `(list int)` to expand into uses of the same structure, declared at the program top-level. In this scheme, the function `sum-list` would expand into

[1] We could work around this by using `void` types, but in that case, type-checking would not catch erroneous uses of the list type.

```
(struct g100 ((car int) (cdr (* (struct g100)))))

(fun sum-list int ((list (* (struct g100))))
  (if (== list NULL)
      (return 0)
      (return (+ (-> list car) (sum-list (-> list cdr))))))
```

This cannot be implemented with the `syntax-case` system mentioned above. The problem is that macro expansion needs to know what list types the program will require, and that information is not available until type analysis. This information is part of the static semantics of the language.

2.3 Ziggurat: Macros with static semantics

I have developed a language and macro system, called Ziggurat, that allows macros to modify the static semantics of a language as well as add new syntax. Ziggurat is a object-oriented variant of Scheme with lazy delegation, intended to be used as a metalanguage to implement extensible languages.

In Ziggurat, a language is built out of three components:

- A concrete syntax. This specifies how programs are written in a source file. In Ziggurat, the only concrete syntax we will consider will be s-expression-based.
- An abstract syntax. This specifies the abstract structure of a program. This consists of a set of classes from which we will be able to instantiate abstract syntax trees.
- A static semantics. This consists of a fixed set of questions that can be asked of a program as represented as an object tree. This static semantics will be a language client's only interface to a program.

2.4 Language towers

Once a language designer has the ability to describe one language in terms of another, there is nothing to stop him from building a third language on top of the second. Once the language designer has built a context-free grammar description language, another language designer can design a regular-expression language to compile into context-free grammars, which in turn compile into an underlying procedural language. This tower can be extended as far down as the language designer needs, even to platform-specific assembly languages, or to register-transfer languages. Additionally, there is no need for this tower to mimic a traditional compiler stack. For example, a machine language can be embedded in a procedural language in order to simulate a machine architecture, or a language can be implemented in itself, in order to insert instrumentation for debugging.

The model to keep in mind when designing a language with Ziggurat is that of building a tower. In this dissertation, we will build such a tower, and we will start with

a basic foundation—a "low-level" language where we define the most basic static and dynamic semantics. We then build the language up, step by step, feature by feature. We use the extension method to add new features, and the embedding method to implement new languages and provide a clean interface.

On top of the assembly language, we build a mini-tower of compiler intermediate languages. On top of this mini-tower we build a high-level language, and on top of this high-level language we build a domain-specific language. By designing languages this way, the implementor can effectively separate the concerns of language design.

2.5 Connecting semantics across the tower

Each of these steps in the language tower introduces some piece of static semantics. These semantics are frequently as interesting for the sorts of computation they forbid as for the computation they permit. For example, a language that has a notion of structured data in its semantics might forbid arbitrary accesses to the store. For this reason, higher-level languages have much to say in terms of the semantics of lower-level languages. An example of this is the parser-generator language mentioned above, that can make static guarantees about the code a macro implementation would generate, such as dependency chains.

2.6 Object-oriented syntax and lazy delegation

Ziggurat allows language semantics to be connected across levels by *not rewriting code unless it is necessary.* By preserving the original syntactic structure of the program, semantic questions can be intercepted at a higher language level. In Ziggurat, an abstract syntax tree representing a piece of code is built using an object-oriented model. Each node is an object, and the tower can ask semantic questions of that node by sending messages to it. Thus, there are messages to represent "Do you terminate?" or "What is your type?" and so forth. These questions are always asked at the highest language level available, but if that language level has no answer to the question, it is *delegated* to the rewritten code. The expectation is that the rewritten code will usually answer the question, but if a piece of syntax at a higher language level has more precise information, it has an opportunity to intercept the message. This model of computation is called *lazy delegation.*

2.7 A taxonomy of language extension

In Ziggurat, there are several means of extending a language. Breaking language extensions down by category allows us not only to apply different methodologies, and know which method applies in what situation, but also allows us to determine what guarantees can be provided by a particular extension, which is important when combining language extensions.

- A language can be *extended* with macros. New syntactic forms can be added to the language, and defined by rewriting the new syntax into existing forms. The new syntax has the option of returning its own answers to semantic messages, but can not define new messages. This will turn out to be very important in ensuring that different language extensions compose.

- A language can be *modified* by having new syntax added to it that does not rewrite to the base language, or by having new static semantics added to it. Such modifications are not guaranteed to compose.

- One language can be *embedded* within another. An embedded language takes the form of a macro whose syntax contains an entirely different language. Examples of this include SQL query languages and regular expression languages. Since these are implemented as macros, these embeddings compose. This is the primary means of building up a language tower— since the embedded language might have its own static semantics, these static semantics can be linked with the base language through the semantic messages mentioned above.

Language extension and language embedding provide a guarantee that language modification does not: different extensions are guaranteed to be composeable. Language modification is thus not as useful a tool; it is only to be used when concepts need to be added to the language that cannot be expressed through rewriting: for example, if a language designer wanted to embed an assembly language within a high-level language.

Chapter 3

Lazy delegation

Ziggurat is a variant of Scheme intended to provide a metalanguage for implementing extensible languages. Programs in these language are expressed in concrete syntaxes that are parsed into an abstract syntax, as will be expanded upon in chapter 5. Abstract syntax trees are represented in Ziggurat languages with a unique object-oriented programming style, called *lazy delegation.*

Lazy delegation is based on Friedman's object-oriented style [32], a programming style for languages in which the primary features of object-oriented programming (*e.g.*dynamic dispatch, method inheritance) are not provided directly. Lazy delegation is similar to other delegation-based object systems, such as Self [97], with a twist: the delegate of an object, instead of being provided at object-creation time, is calculated only when it is needed. Ziggurat makes additions to the standard Scheme grammar as presented in figure 3.1.

3.1 Classes

In Ziggurat, each object has a unique class. A class defines (among other things) the fields that an object has. Objects are defined by the `define-class` meta-language syntax.

`(define-class` c `(`f `...)` $[e_{deleg}]$`)`

This defines a class named c with fields f .... The optional delegation method, e_{deleg}, will be covered in section 3.4.

For example, if we wanted to define a class of object to represent floating-point real numbers, we would use the code

```
(define-class real% (mantissa^ exponent^))
```

This declares that objects of class `real%` have two fields: `mantissa^` and `exponent^`. The non-alphanumeric characters at the ends of identifiers are part of a naming convention, that will be spelled out explicitly in section 3.7.

$i \in$ Identifier ::= `x, a, foo, ...`
$d \in$ Definition ::= `...`
| `(define-class` c `(`f `...))`
| `(define-class` c `(`f `...)` e_{deleg}`)`
| `(define-generic (`m i `...))`
| `(define-generic (`m i `...)` e_{default}`)`
| `(define-attribute` m`)`
| `(define-attribute` m e_{default}`)`
| `(define-method` c m e`)`
$e \in$ Expression ::= `...`
| `(object` c e `...)`
| `(pass)`
$c \in$ Class ::= i
| `(`i i`)`
$f \in$ Field ::= i
| `(`i i`)`
| `(`i i i`)`
$m \in$ Method ::= i

Figure 3.1: Ziggurat extends Scheme with lazy-delegation objects

3.2 Objects

Objects are created using the `object` syntax.

```
(object i e_init ...)
```

This creates an object of class i with its fields initialized to e_{init} .... Classes are not first-class objects in Ziggurat, and thus the class must be a name, not an expression. It is a static error for the number of field initializers to be different from the number of fields.

Ziggurat does not have constructor methods like other object-oriented programming systems. The way to provide the functionality of constructor is by wrapping the `object` syntax in a function definition.

```
(define (make-real mantissa exponent)
  (object real% mantissa exponent))
```

The reason for making object constructors separate functions is to ensure the object system interacts well with module systems. In this object system, having access to a class name allows the programmer to perform primitive operations on it, such as adding methods, precisely the sort of functionality we would want to allow to be restricted by module boundaries. Therefore, by exporting a constructor function but not exporting the actual class, the programmer can ensure that users of the module can create objects of that class but cannot add new methods to it.

3.3 Generic functions

A generic function is a function whose behavior is dependent on the class of its first argument. Generic functions are introduced with the form

`(define-generic (`*m i* `...)` [$e_{default}$]`)`

which defines a generic function named *m* with $i \ldots$ providing the argument parameters. The optional body $e_{default}$, if it is present, defines the default behavior of the function. If the default expression is missing, then the default behavior of the generic function is to raise an error. Generic functions must have at least one argument, which is the "this" object of the function invocation. Generic functions are called as if they were ordinary Scheme functions, and can be passed around and used as such. Thus, the following code defines a generic function of one argument, `num`, whose default behavior is to return the string `"<object>"`.

```
(define-generic (object-number->string: num)
  "<object>")
```

In order to give the generic function more interesting behavior, we must override it with class-specific method definitions:

```
(define-method real% object-number->string:
  (lambda (this)
    (string-append (number->string mantissa^) "e"
                   (number->string exponent^))))
```

This code defines the behavior of the `object-number->string` generic function if its first argument is an object of class `real%`. Since we know that the first argument of this method is an object of class `real%`, we know that it has fields `mantissa^` and `exponent^`. These fields are available in the method body, as if they were ordinary Scheme variables. One consequence of this is that classes cannot be first-class values, since we must be able to tell statically what is a field of the object, and what is an ordinary Scheme variable. Objects and generic functions are first-class values, though, and can be passed around and manipulated just like any other Scheme value.

3.4 Delegation

Objects have a special method, known as the delegate-instantiation method, that is defined and invoked differently from other methods. Unlike other methods, delegate-instantiation methods have no arguments, and must return an object or `#f`. This object is the *delegate* of the current object, and is used in case a generic function is applied to the object and it has no corresponding defined method. If the delegate-instantiation method returns `#f`, the object has no delegate, and any method the object attempts to delegate will fall through to its default behavior. Delegates are thus created on demand

and then cached; after an object has invoked its delegate-instantiation method, future method lookups will automatically be passed to the delegate object when needed.

The delegate-instantiation method is provided as part of the `define-class` form. For example, to define an `int%` class that delegates to a `real%` object:

```
(define-class int% (value^)
  (if (= value^ 0)
      (object real% 0 0)
      (let* ((exponent (inexact->exact
                        (floor (/ (log (abs value^))
                                  (log 10)))))
             (mantissa (exact->inexact
                        (/ value^ (expt 10 exponent)))))
        (object real% mantissa exponent))))
```

What if we called `(object-number->string (object int% 4007))`? Since the `int` class does not have its own method for `object-number->string:`, it delegates the generic function to the `real%` object produced by its lazy-delegation method, which in turn produces `"4.007e3"`. If we wanted integers to have a specialized `object-number->string:` method, we would simply define one, *e.g.*

```
(define-method int% object-number->string:
  (lambda (this) (number->string value^)))
```

Now `(object-number->string: (object int% 4007))` will return `"4007"`.

It's worth noting that the first argument to the method `object-number->string:`, `this`, will be bound to the original object, not the delegate. This is important if, for example, further generic functions are to be invoked on the original object.

Ziggurat also allows the programmer to delegate explicitly through the function `pass`. The return value of the function `pass` is equivalent to what it would have been if the generic function had been delegated. For example, if we wanted an `int` object to use scientific notation when its value is less than zero, we would define the method `object-number->string:` on objects of class `int` to be

```
(define-method int% object-number->string:
  (lambda (this) (if (< value 0)
                     (pass)
                     (number->string value))))
```

3.5 Attributes

An attribute in Ziggurat is a memoized generic function of one argument. Attributes are declared with the form:

```
(define-attribute m)
```

Or, with a default method:

(define-attribute *m* e_{deleg})

Other than memoizing its result, an attribute behaves exactly like a method of a single argument. For example, if we wanted to define an attribute that returns the number of characters an object would take to represent, we could do so:

```
(define-attribute number-width:)

(define-method int% number-width:
  (lambda (this)
    (if (= value 0) 1
        (let ((exponent (inexact->exact
                         (ceiling (/ (log (abs value))
                                     (log 10))))))
          (if (< value 0)
              (+ exponent 1)
              exponent)))))
```

This has the advantage that the calculation only occurs once. Attributes can also be used for methods with side-effects that the language designer only wants to occur once. For example, if we wanted to associate a unique ID to every object, we could do so:

```
(define uid-counter 0)

(define-attribute object-unique-id:
  (lambda (this)
    (set! uid-counter (+ uid-counter 1))
    uid-counter))
```

3.6 Automatic methods

Certain methods are so common that Ziggurat provides syntactic sugar for creating them during class declaration. The full syntax for class declaration is:

(define-class *c* (*f* ...) [$e_{\texttt{deleg}}$])

The class name declaration *c* can take one of two forms.

i
(i_1 i_2)

The first form simply declares a class with the name i. The second form declares a class with name i_1, and a predicate named i_2 that tests whether an object has class i_1, or delegates to an object of class i_1. The declaration

```
(define-class (real% is-real?:) (mantissa^ exponent^))
```

is equivalent to the series of declarations

```
(define-class real% (mantissa^ exponent^))

(define-generic is-real?:
  (lambda (this) #f))

(define-method real% is-real?:
  (lambda (this) #t))
```

It's worth noting that `is-real?:` is a generic function. Consequently, objects that delegate to class `real%`, such as those of class `int%` will return `#t` to `is-real?:`. This is done to provide a clean abstraction to the programmer: if a piece of code that knows about the class `real%` but not `int%` is given an object of class `int%`, it will deal with that object as if it were of class `real%`.

While specifying fields, it is possible to specify accessor and mutator methods. The field declaration *f* can take one of three forms.

i
(i_1 i_2)
(i_1 i_2 i_3)

The first form simply declares a field with name i. The second form declares a field named i_1 and an accessor named i_2. The third form declares a field named i_1, an accessor named i_2, and a mutator named i_3. The declaration

```
(define-class real% ((mantissa^ real-mantissa:
                                set-real-mantissa!:)
                     (exponent^ real-exponent:
                                set-real-exponent!:)))
```

is equivalent to the series of declarations

```
(define-class real% (mantissa^ exponent^))


(define-generic real-mantissa:)


(define-method real% real-mantissa:
  (lambda (this) mantissa^))


(define-generic set-real-mantissa!:)
```

```
(define-method real% set-real-mantissa!:
  (lambda (this val) (set! mantissa^ val)))


(define-generic real-exponent:)


(define-method real% real-exponent:
  (lambda (this) exponent^))


(define-generic set-real-exponent!:)


(define-method real% set-real-exponent!:
  (lambda (this val) (set! exponent^ val)))
```

These are again generic functions. As a consequence, if a piece of code has a reference to an object of class `int%`, it can treat it as if it were of class `real%` via `real-mantissa:` and so forth.[1]

3.7 Naming conventions

Since Scheme is syntactically simple, it can be difficult to determine at a casual glance the roles of various identifiers, particularly in the case of Ziggurat, where there are multiple namespaces and varieties of identifier scope. The naming conventions presented in this section help disambiguate these roles.

3.7.1 Classes

Identifiers representing classes are signified by the `%` character at the end of the identifier. For example, in the arithmetic example above, the class representing floating-point numbers is named `real%`.

3.7.2 Fields

Identifiers representing fields are signified by the `^` character at the end of the identifier. For example in the arithmetic example above, the field representing the mantissa of a floating-point number is named `mantissa^`.

It is especially important to visually differentiate between object fields and ordinary variables, since these two appear alongside one another, but have different scope rules. For example, in the code

[1] It's worth noting here that `set-real-mantissa!:` will mutate the delegate object of class `real%`, not the original of class `int%`. If this behavior is not desired, there are two possible solutions: either (1) the implementor of `real%` can not provide a mutator for `mantissa`, or (2) the implementor of the `int%` class can provide a method for `set-real-mantissa!:`.

```
(define-method real% object-number->string:
  (lambda (this)
    (string-append (number->string mantissa^) "e"
                   (number->string exponent^))))
```

if one does not realize that `mantissa^` is a field, its nature is quite mysterious: where is it bound? What is its scope? Is it, perhaps, a global variable? The `^` character provides a visual clue to the programmer that it is a field present in the `this` object, bound when that object was defined and in scope for the current method declaration.

3.7.3 Methods

Identifiers representing generic functions are signified by the `:` character at the end of the identifier. For example, the method used to convert a number object to a string is named `object-number->string:`.

Accessor methods are denoted by names of the form *class-field*`:`. For example, the method used to extract the mantissa field of a floating-point object is named `real-mantissa:`. The characters `%` and `^` for class and field names are omitted for aesthetic reasons; it's all too easy for these names to become a train wreck of non-alphanumeric characters.

Mutator methods are denoted by names of the form `set-`*class-field*`!:`. For example, the method used to mutate the mantissa field of a floating-point number is named `set-real-mantissa!:`. Once again, internal signifier characters are omitted.

Class predicate methods are denoted by names of the form `is-`*class*`?:`. For example, the method used to test whether an object is of the floating-point class is `is-real?:`. Once again, internal signifier characters are omitted.

Chapter 4

Case study: Arith language

In this chapter, we will see through an example how to use Ziggurat's object system in order to build a language. The goal of this section is to demonstrate how a language is defined in Ziggurat, how lazy delegation leads to a natural implementation of this definition, and how it allows the language to be extensible, both in syntax and static semantics, giving us the properties promised by the thesis.

4.1 Language structure

In this dissertation, we view a language implemented in Ziggurat as consisting of three parts.

- A *concrete syntax.* This specifies how programs are written in a source file. In Ziggurat, the only concrete syntax we will consider will be s-expression-based.
- An *abstract syntax.* This is a set of classes, which will be used to instantiate objects in order to construct the abstract syntax tree representation of programs in the language.

  This particular terminology is unique to Ziggurat, and reflects the particular way languages are meant to be designed using the system. In fact, as introduced by Knuth [52], the term *abstract syntax* is used to refer to what in Ziggurat are class predicates, which are considered part of the semantics. Nonetheless, this terminology was chosen in order to make the translation from design into object-oriented implementation as straightforward as possible, and hence, there is a simple equivalence: syntax = classes, and semantics = generic functions.
- A *static semantics.* This is a fixed set of generic functions guaranteed to be defined on programs in the language. This includes all of the class predicates, field accessors and field mutators introduced by the abstract syntax.

In order to see how this works in practice, let us consider a very simple language.

```
i ∈ Identifier
c ∈ Const ::= ...  -1 | 0 | 1 ...
e ∈ Expr  ::= i
            | c
            | (+ e e)
            | (* e e)
            | (/ e e)
            | (- e e)
            | (input c)
            | (let (i e) e)
```

Figure 4.1: Grammar of an arithmetic-expression language.

```
(define-class arith-var% (name^))
(define-class arith-const% (value^))
(define-class arith-let% (var^ binding-expr^ expr^))
(define-class arith-add% (left^ right^))
(define-class arith-mul% (left^ right^))
(define-class arith-div% (left^ right^))
(define-class arith-sub% (left^ right^))
(define-class arith-input% (index^))
```

Figure 4.2: Abstract syntax for the arithmetic-expression language

4.2 The Arith language

Consider a language for expressing simple arithmetic calculations, with a concrete syntax presented in figure 4.1. This language has one basic syntax category: the expression. An expression is either a variable, a constant, a primitive operation such as multiplication or addition, a `let` form to bind variables, or a user-input form, which represents the dynamic component of the language. User input is provided by the `input` keyword. Input is modeled as an infinite vector mapping the natural numbers to values, and `(input` n`)` pulls the nth element from that vector.

In order to implement the abstract syntax, each syntax node is represented by an object. The classes of the syntax nodes correspond to the productions of the grammar, and the fields of the syntax node correspond to the sub-terms of the production. Implementing this involves directly transcribing the grammar into code, as seen in figure 4.2. Parsing, the process of turning a piece of concrete syntax into abstract syntax, will be covered in chapter 5.

This language is pretty basic, though, and there are any number of extensions we might want to add. For example, we might want to add a `(sqr` e`)` form, that squares the value of an expression. With our AST represented by lazy-delegation objects, this is easy to do:

```
;;; (sqr <exp>) delegates to (let (<x> <exp>) (* <x> <x>))
(define-class arith-sqr% (expr^)
  (lambda ()
    (let ((temp-var-name (gensym))) ; create fresh var
      (object arith-let% (object arith-var% temp-var-name)
                         expr^
              (object arith-mul%
                      (object arith-var% temp-var-name)
                      (object arith-var% temp-var-name))))))
```

Suppose that we have a generic function (`execute` *expr ctx inp*), defined on the basic language nodes, that calculates the numerical value of the arithmetic expression *expr* in some variable-binding context *ctx* with some input vector *inp*. Evaluating

```
(execute (object arith-mul% (object arith-const% 3)
                            (object arith-const% 7))
         empty-arith-context '())
```

would return the value `21`. Now, what about running `execute` on a syntax node of class `arith-sqr`? Consider what would happen if we attempted to evaluate

```
(execute (object arith-sqr (object arith-const 5))
         empty-arith-context '())
```

Since `arith-sqr` does not define a method for the `execute` generic function, it gets delegated to the expanded syntax tree:

```
(object arith-let% (object arith-var% 'g300)
                   (object arith-const% 5)
        (object arith-mul% (object arith-var% 'g300)
                           (object arith-var% 'g300)))
```

Here, `'g300` is the fresh symbol generated by `gensym`. When `execute` is delegated to this AST, the result is the expected `25`. Although we did not explicitly write an execute method for `arith-sqr`, that piece of semantics was handled though rewriting.

4.3 Extending static semantics

Let's consider adding to the Arith language a generic function on syntax nodes that calculates the possible signs of an expression. This is done by defining a generic function, `arith-sign:`, that returns a subset of `'(- zero +)`. Besides the `this` object, `arith-sign:` takes one argument, an environment that maps variables (represented as identifiers) to a set of possible signs. Environments of this form are covered in more detail in section 5.3.

The default behavior of `arith-sign:` is to give the most conservative result: `'(- zero +)`.

```
(define-generic (arith-sign: this env)
  (lambda (this env)
    '(- zero +)))
```

The behavior of `arith-sign:` at the basic syntax nodes follows the rules of elementary arithmetic.

```
(define-method arith-var% arith-sign:
  (lambda (this env)
    (arith-env-get: env name^ (lambda () '(+ - zero)))))

(define-method arith-const% arith-sign:
  (lambda (this env)
    (cond ((< 0 value^) '(+))
          ((> 0 value^) '(-))
          (else '(zero)))))

(define (add-sign sign1 sign2)
  (case sign1
    ((+) (if (symbol=? sign2 '-)
             '(+ - zero) '(+)))
    ((-) (if (symbol=? sign2 '+)
             '(+ - zero) '(-)))
    ((zero) sign2)))

(define-method arith-add% arith-sign:
  (lambda (self env)
    (let ((left-sign (arith-sign: left^ env))
          (right-sign (arith-sign: right^ env)))
      (fold (lambda (sign1 so-far1)
              (fold (lambda (sign2 so-far2)
                      (lset-union eq? so-far2
                                  (add-sign sign1 sign2)))
                    so-far1 right-sign))
            '() left-sign))))

...
```

Now, what happens if we apply `arith-sign:` to objects of class `arith-sqr%`? There is no method for `arith-sign:` at `arith-sqr%`. Therefore, the generic function `arith-sign:` would fall through to the delegate. For example, if we were to apply `arith-sign:` to `(sqr (input 1))`, it would rewrite as `(let (g100 (input 1)) (* g100 g100))`, which would give the conservative answer `'(- zero +)`. However, we know that no real number squared produces a negative result. So, we can do better.

```
(define-method arith-sqr% arith-sign:
 (lambda (self env)
   (let ((val-sign (arith-sign: expr^ env)))
     (lset-union (if (memq '+ val-sign) '(+) '())
                 (if (memq '- val-sign) '(+) '())
                 (if (memq 'zero val-sign) '(zero) '())))))
```

Now applying `arith-sign:` to `(sqr (input 1))` gives the much more precise result `'(zero +)`.

4.4 Recap

At the beginning of this chapter, we introduced an architecture for designing languages. In the remainder of the chapter, we saw a language designed using this architecture, and then extended it. The intention of the Ziggurat design architecture is to allow such extensions, and so it's worthwhile to step back and see how the design of Arith enabled its extension.

The description of Arith has three parts.

- A concrete syntax. The syntax of Arith is, as all Ziggurat languages, s-expression based. We will explore parsing of the concrete syntax in greater detail in the next chapter.
- An abstract syntax. Arith has eight classes, `arith-var%`, `arith-mul%` and so forth, that constitute its abstract syntax.
- A static semantics. Arith has a fixed set of generic methods that constitute its static semantics. This includes simple field accessors for the basic abstract syntax, such as `arith-mul-left:`, as well as the single more complex generic function, `arith-sign:`.

The `sqr` macro is an example of language extension, as presented in section 2.7. This means that we add a new piece of abstract syntax, `arith-sqr%`, that delegates to one of the base classes of the language. We also have the option of adding a method to any of the generic functions that constitute the static semantics of the language. When we write a specialized method for the new piece of syntax, we add to the computational power of our static semantics. In this case, we have given a more precise method for `arith-sign:`. This is a small example of the semantic flexibility we will exploit to a greater degree when we use Ziggurat to build more complex languages; the semantic functions we will use will follow the same basic structure as `arith-sign:`, and thus will benefit from the same extensibility.

When we define abstract syntax for `arith-sqr%`, we will want to define a corresponding piece of concrete syntax. We will see how to do this in the next chapter.

Chapter 5

Parsing

A language in Ziggurat has two forms of syntax: a *concrete syntax* and an *abstract syntax*. The abstract syntax of a language consists of the set of classes used to build programs, as in the introduction of the Arith language in section 4.2. However, it will be rare that we will want to program directly in the abstract syntax. Therefore, we provide a concrete syntax, which is much terser and clearer. The means of taking a program written in a concrete syntax and producing the equivalent program in abstract syntax is called *parsing*.

Since Ziggurat is a general-purpose programming language, the language implementor could implement any methodology of parsing. However, the topic of parsing extensible languages is complex, even in the context of s-expression languages. Therefore, Ziggurat provides a parsing engine based on the hygienic macro systems of languages such as Scheme and Dylan, modified to work with languages with multiple namespaces.

This parsing engine is closely tied to the structure of the metalanguage, and in fact drives some aspects of it. Some of this presentation will be illustrated by example; we will build a parser for the Arith language. The concrete syntax we will be imple-

$i \in$ Identifier
$c \in$ Const ::= `...  -1 | 0 | 1 ...`
$e \in$ Expr ::= i
| c
| `(let (`v e`)` e`)`
| `(+` e e`)`
| `(*` e e`)`
| `(/` e e`)`
| `(-` e e`)`
| `(input` c`)`

Figure 5.1: Grammar of an arithmetic-expression language.

menting (without macro definitions or uses) is presented in figure 4.1, and repeated for convenience in figure 5.1.

The parsing algorithm presented in this section will be built in stages, to illustrate several facets of the design.

- The first design will be a short implementation of a parser for a non-extensible version of the language.
- Next, we will alter the parser to make the language extensible.
- Finally, we will make the language hygienic.

The first stage, a parser for the fixed language, introduces many of the underlying mechanisms that allow the language designer to write parse functions.

5.1 Syntax-case

In our s-expression setting, we have a reader that takes in a stream of characters, and produces a tree of raw data which corresponds to the nested list structure of the unparsed code. For example, the reader takes the string "`(* (+ 1 2) 3)`", and produces a data structure[1] equivalent to the one constructed by:

```
(cons '* (cons (cons '+ (cons 1 (cons 2 '())))
               (cons 3 '())))
```

This raw tree is called a *form.* If we assume a set of symbols and a set of constants, we can define forms by the grammar

$s \in$ Symbols
$c \in$ Constants
$f \in$ Form $::= s \mid c \mid (f \ \ldots)$

The implementation of the reader in Ziggurat is unremarkable; in fact, it is taken directly from the reference reader for R6RS Scheme.

If our only goal were to parse the language presented in figure 5.1, we could accomplish this by writing a function that takes as an argument an input form and produces an abstract syntax tree. Ziggurat provides, as with many other macro systems, two convenient keywords for manipulating forms: `syntax` and `syntax-case`. These are similar (though not identical to) their Scheme counterparts [25]. The `syntax` keyword introduces a form for building concrete syntax, and `syntax-case` matches and deconstructs concrete syntax.

[1] In fact, the trees returned by the reader are not unadorned Scheme data: they are lazy-delegation objects, known as *annotated data.* Annotated data adorns Scheme objects with source-location information, to aid in analysis and debugging. The primitives for constructing and deconstructing annotated data are different from those of unadorned Scheme objects, but since the programmer will mainly be using `syntax` and `syntax-case` to manipulate forms, this distinction is mostly irrelevant.

The Ziggurat expression `(syntax-case` *f* `(`*k* `...) (`*p e*`) ...)` will attempt to match a form against one or more patterns, and evaluate the corresponding expression. The sub-expression *f* should be a Ziggurat expression that evaluates to a form. The `syntax-case` expression attempts to match this form against the pattern *p*. If the form matches the pattern, `syntax-case` binds the syntax variables found in *p*, and evaluates the Ziggurat expression *e* in scope of these syntax variables.

Patterns are themselves forms. The rules for pattern matching are straightforward; more so even than in Scheme.

- If the pattern *p* is a pair, `(`p_1 `.` p_2`)`, then `syntax-case` only matches if *f* evaluates to a form, `(`f_1 `.` f_2`)`, p_1 matches f_1 and p_2 matches f_2.
- If *p* is a symbol that appears in the keyword list `(`*k* `...)`, then `syntax-case` only matches if *f* evaluates to the same symbol.
- If *p* is a symbol that does not appear in `(`*k* `...)`, then it is a syntax variable. It always matches, and binds *f* to the syntax variable *p*. Syntax variables are covered below.

Syntax variables are not ordinary Ziggurat variables, and can only be accessed in `syntax` expressions. They do have scope; in the `syntax-case` expression above, the variables bound in pattern *p* are in scope in expression *e*.

Ziggurat `syntax` expressions build syntax forms. In the expression `(syntax` *t*`)`, the sub-form *t* is known as a *template.* The behavior of the `syntax` expression depends on this template.

- If *t* is the form `(unsyntax` *e*`)`, then evaluating the `syntax` expression evaluates the Ziggurat sub-expression *e*, and passes along its value.[2]
- If *t* is a pair `(`t_1 `.` t_2`)`, the `syntax` expression builds a pair form from `(syntax` t_1`)` and `(syntax` t_2`)`.
- If *t* is a constant, `syntax` builds a constant form.
- If *t* is a syntax variable, then `syntax` evaluates to the syntax bound to the variable *t*.
- If *t* is another symbol, then `syntax` evaluates to this symbolic form.[3]

Using these primitives, we can write a naïve parser. The atoms of our concrete syntax will be integers and identifiers. Identifiers are what is used to represent variables and keywords in a program; for now, it suffices to think of identifiers as symbols. If we assume functions `parse-identifier` and `parse-integer`, that respectively take annotated symbols and constants and return their unannotated counterparts, such a parser might look like:

[2] Due to this behavior, Ziggurat `syntax` is analogous to Scheme `quasisyntax`, not Scheme `syntax`.

[3] The actual behavior of `syntax` in this case is a little more complicated, but this will be expanded on once hygiene is introduced.

```
(define (naive-parse form)
  (syntax-case form (+ * - / let input)
    ((+ e1 e2)
     (object arith-add%
             (naive-parse (syntax e1))
             (naive-parse (syntax e2))))
    ((- e1 e2)
     (object arith-sub%
             (naive-parse (syntax e1))
             (naive-parse (syntax e2))))
    ((* e1 e2)
     (object arith-mul%
             (naive-parse (syntax e1))
             (naive-parse (syntax e2))))
    ((/ e1 e2)
     (object arith-div%
             (naive-parse (syntax e1))
             (naive-parse (syntax e2))))
    ((let (v e1) e2)
     (object arith-let%
             (parse-identifier (syntax v))
             (naive-parse (syntax e1))
             (naive-parse (syntax e2))))
    ((input c1)
     (object arith-input%
             (parse-integer (syntax c1))))
    (f
     (cond ((identifier? (syntax f))
            (object arith-var%
                    (parse-identifier (syntax f))))
           ((integer? (syntax f))
            (object arith-const%
                    (parse-integer (syntax f))))
           (else
            (parse-error "Could not parse expression"))))))
```

There are at least two problems with this parser.

- It does not allow for easy syntactic extension. In order to allow new syntactic forms, `naive-parse` would have to be replaced.
- Its handling of Arith variables is too simplistic. Eventually, we will have to resolve variables, to identify variables with their binding expression, and to signal an error if it is not in scope. We can, in fact, do this at parse time, but `naive-parse` does no such thing.

In order to solve these problems, we introduce syntactic environments.

5.2 Syntactic environments

The naïve parser presented in the previous section is not extensible. A first step to fixing this problem is making the parser more modular. Since Arith expressions are identified by keyword, we can store parse functions for individual expression types in a table, indexed by keyword. We use a lazy-delegation object for this. This object is called a *syntactic environment.*

```
(define-class arith-env% (table^))

(define (enter-arith-env)
  (object arith-env% (make-identifier-table)))
```

The field `table^` is a kind of mapping called an *identifier table.* An identifier table has identifiers as its keys. The primitives for identifier tables are

`(make-identifier-table)`	create table
`(identifier-table-put!:` *table key val*`)`	insert into table
`(identifier-table-defines?:` *table key*`)`	test inclusion in table
`(identifier-table-get:` *table key*`)`	access table entry

We define the parse function as a generic function which looks up a specialized function in the field `table^`.

```
(define-generic (parse-arith-expr: env form)
  (cond ((identifier? form)
         ;; variable reference
         (object arith-var%
                 (parse-identifier env form)))
        ;; number
        ((integer? form)
         (object arith-const%
                 (parse-constant: env form)))
        ;; error
        (else (parse-error "Could not parse expression"))))

(define-method arith-env% parse-arith-expr:
  (lambda (env form)
    (syntax-case form ()
      ((keyword . rest)
       (if (and (identifier? (syntax keyword))
                (identifier-table-defines?:
                 table^ (syntax keyword)))
           ((identifier-table-get: table^ (syntax keyword))
            env form)
           (pass)))
      (f (pass)))))
```

To add parse functions to a syntactic environment, we merely add them to the enclosed table.

```
(define-generic (add-keyword-parser!: env keyword parser))

(define-method arith-env% add-keyword-parser!:
  (lambda (env keyword parser)
    (identifier-table-put!: table^ keyword parser)))
```

If we define a syntactic environment to be the "top-level" Arith environment, adding specialized parse functions is now straightforward.

```
(define top-arith-env (enter-arith-env))

(add-keyword-parser!: top-arith-env (syntax +)
  (lambda (env form)
    (syntax-case form ()
      ((+ e1 e2)
       (object arith-add%
               (parse-arith-expr: env (syntax e1))
               (parse-arith-expr: env (syntax e2))))
      (f (parse-error form "Could not parse addition")))))

...
```

Now, if we want to add a new syntactic form, we simply add a new parse function to the top-level Arith environment.

```
(add-keyword-parser!: top-arith-env (syntax sqr)
  (lambda (env form)
    (syntax-case form ()
      ((sqr e)
       (object arith-sqr%
               (parse-arith-expr: env (syntax e)))
      (f (parse-error form "Could not parse sqr")))))
```

This is a rather heavyweight means of extending the language. What we would like would be a target-language form for extending the language in a local scope. Something akin to `let-syntax` in Scheme. Let's add a similar form to Arith.

$e \in$ Expr ::= ...
| `(let-syntax (`s z`)` e`)`

This `let-syntax` form defines a new keyword s, with parse function z (which is a Ziggurat form), with a scope of the expression e. With this form defined, we can define `sqr` in the target language as

```
(let-syntax
  (sqr (lambda (env form)
         (syntax-case form ()
           ((sqr e)
            (object arith-sqr%
                    (parse-arith-expr: env (syntax e)))
           (f (parse-error form "Could not parse sqr")))))
  ...))
```

How to define `let-syntax` presents us a problem: where to put this parse function? We don't want to put it in `top-arith-env`, since we want the new keyword to have local scope. So, when parsing `let-syntax`, we build a new syntactic environment to be used for parsing its body. However, since we want this new environment to include all of the parse functions of its enclosing scope, we make syntactic environments delegate. The `arith-env%` class now looks like:

```
(define-class arith-env% (table^ enclosing-env^)
  (lambda () enclosing-env^))

(define (enter-arith-env env)
  (object arith-env% (make-identifier-table) env))
```

Each syntactic environment now has an enclosing environment. If a local environment can not parse a form, it delegates to its enclosing environment. This form of delegation, where the delegate is pre-computed, is called *strict delegation,* and as we see above, it is implemented as a special case of lazy delegation. When an object of class `arith-env%` is instantiated, it is passed its delegate, rather than in lazy delegation, where the delegate is instantiated on the fly.

Now that we have layerable syntactic environments, we can implement the Arith parser for `let-syntax`.

```
(add-keyword-parser!: top-arith-env (syntax let-syntax)
  (lambda (env form)
    (syntax-case form ()
      ((let-syntax (name transformer) exp)
       (let ((new-env (enter-arith-env env))
             (transformer
              (eval (syntax transformer))))
         (add-keyword-parser!: new-env (syntax name)
                               transformer)
         (parse-arith-expr: new-env (syntax exp)))))))
```

This makes the language extensible, which was the first problem of the naïve approach. However, there is still the second problem: we still need to resolve variables and identify them with their binding expressions.

```
(define-class arith-var-namespace% (table^ enclosing-env^)
  (lambda () enclosing-env^))

(define (enter-arith-var-namespace env)
  (object arith-var-namespace% (make-identifier-table) env))

(define-method arith-var-namespace% namespace-define:
  (lambda (env name value)
    (identifier-table-put!: table^ name value)))

(define-generic (arith-var-namespace-lookup: env name)
  (parse-error name "Undefined variable"))

(define-method arith-var-namespace%
               arith-var-namespace-lookup:
  (lambda (env form)
    (if (and (identifier? form)
             (identifier-table-defines?: table^ form))
        (identifier-table-get: table^ form)
        (pass))))
```

Figure 5.2: The implementation of a namespace for variables in the Arith language.

5.3 Namespaces

A symbol can play many roles, based on its context: in Arith, for example, a symbol can either be a keyword or a variable. Ziggurat reifies these roles as *names.* Taking a symbol and finding the name it represents is called *name resolution.* Name resolution is done via syntactic environments called *namespaces.*

In Arith, we parse variables by first ensuring that each one is defined, and then replacing it with a unique symbol in the abstract syntax tree. This ensures that we will not have to deal with problems of scope after parse-time. In order to do this, we need what amounts to a large, extensible lookup table.

A namespace is a multidimensional mapping, extensible by both entries and dimensions. The implementation of namespaces for Arith variables is in figure 5.2. If we have a namespace in the Ziggurat variable `env`, and we wish to look up the identifier `x` as an Arith variable, we evaluate `(arith-var-namespace-lookup: env (syntax x))`. If we had another namespace for Arith, for, say, functions, we could define another generic function, `arith-function-namespace-lookup:`, that can look up identifiers as functions, *even in the same syntactic environment.* If we wish to look up the identifier `x` as a (hypothetical) Arith function, we evaluate the Ziggurat expression `(arith-function-namespace-lookup: env (syntax x))`, *even if* `env` *represents the same syntactic environment.*

We achieve this multidimensionality via the strict delegation introduced in the last section. In order to add new Arith-variable entries to a syntactic environment, we call `enter-arith-var-namespace`, creating a new namespace which delegates to the old syntactic environment. As a result, a syntactic environment has several "ribs": namespaces that respond to lookups of a particular type of identifier. These namespaces can map the same identifier to completely different names, but since they are looked up using different generic functions, there is no conflict. Indeed, there is no need for the implementor of one namespace to consider the implications for other namespaces. This will come in handy when we start using this mechanism in embedding languages.

Ziggurat provides syntactic sugar for defining these functions, generics and methods. The definitions in figure 5.2 can be abbreviated:

```
(define-namespace arith-var)
```

Variable namespaces are used while parsing `let` expressions and variables. Parsing a `let` introduces a variable namespace, while parsing a variable looks up a symbol in a variable namespace.

```
(define (parse-arith-var env form)
  (if (identifier? form)
      (object arith-var% (arith-var-namespace-lookup:
                          env form))
      (parse-error form "Expected a symbol")))
```

```
(define (parse-arith-let env form)
  (syntax-case form ()
    ((let (var val) exp)
     (let ((new-env (enter-arith-var-namespace env))
           (id (parse-identifier env (syntax var)))
           (new-symbol (gensym)))
       (namespace-define: new-env id new-symbol)
       (object arith-let%
               new-symbol
               (parse-arith-expr: env (syntax val))
               (parse-arith-expr: new-env
                                  (syntax exp)))))))
```

There are two kinds of name in Arith: variables and keywords. Since keywords are simply a form of name, we can and do implement syntactic environments as introduced in the last section as a special case of namespaces.

```
(define-namespace arith-keyword)

(define (parse-arith-expr env form)
  (syntax-case form ()
    ((k . rest)
     (if (identifier? (syntax k))
         ((namespace-lookup: env (syntax k)) env form)
         (parse-error form "Could not parse expression")))
    (f
     (cond ((identifier? form)
            (parse-arith-var env form))
           ((constant? form)
            (parse-arith-constant env form))
           (else
            (parse-error "Could not parse expression"))))))

(define-method arith-keyword-namespace% add-keyword-parser:
  (lambda (env keyword parser)
    (arith-keyword-namespace-put!: env keyword parser)))

(add-keyword-parser: top-arith-env (syntax let)
                     parse-arith-let)
...
```

These mechanics make parsing straightforward, and complete the definition of the language presented in the last chapter. In order to make defining lazy-delegation syntax classes a bit simpler, we introduce syntax objects.

5.4 Syntax objects

When defining classes to represent abstract syntax, it's convenient to keep around certain information from parse-time. The Ziggurat code

```
(define-syntax-class name (field ...) ...)
```

expands into

```
(define-class name (source^ env^ field ...) ...)
```

where the fields `form^` and `env^` are in scope for the methods defined on the class *name*.

The purpose of the field `form^` is to record the annotated data that was parsed to produce this particular syntax object. For example, for a type error, we will want to be able to invoke

```
(type-error source^ "Type error")
```

The `env^` field is for defining the delegate of new abstract syntax. It is useful to reuse the parsing functions of the language to define the delegate.

```
(define-syntax-class arith-sqr% (expr^)
  (lambda ()
    (parse-arith-expr
      env^ (syntax (let (x #,(object pre-parsed-datum%
                                      expr^))
                          (* x x))))))
```

The class `pre-parsed-datum%` is a wrapper, which is guaranteed always to parse to its first field. This greatly simplifies the definition of delegate instantiation functions.

5.5 A brief discussion of hygiene

A significant problem of macro programming in naïve macro systems such as in Common Lisp and C is that they do not respect lexical scoping. When a macro introduces an identifier during expansion, it can be confused with identifiers present at the expansion site. The parsing algorithm presented thus far also suffers from this problem.

Let us look at an example of this sort of error. We can define a macro `add-three` that adds three to the result of an expression:

```
(let-syntax
  (add-three
   (lambda (env form)
     (syntax-case form ()
       ((add-three x)
        (parse-arith-expr env
          (syntax (let (y 3)
                    (+ x y)))))))))
  (let (y 4)
    (add-three y)))
```

If we were to invoke this macro:

```
(let (y 4)
  (add-three y))
```

The result would be `6`, instead of the expected `7`.

The problem here is that, after expansion, the expression which gets parsed is:

```
(let (y 4)
  (let (y 3)
    (+ y y)))
```

The `add-three` macro did more than advertised: not only does invoking it add three to the value computed by the enclosed expression, it evaluates this expression in scope of a new variable `y`. This can cause problems if, for example, this new variable scope overrides the scope of a previously defined variable `y`; as in this case, the macro user would expect the symbol `y` to resolve to the previously defined variable, when in fact it would resolve to the new variable.

To avoid this, we can implement the macro system of Hieb, Dybvig and Bruggeman [38] to ensure that variables introduced at a macro definition do not have scope that includes code at the macro invocation, and vice-versa. Thanks to this macro technology, we can do this transparently to the macro writer; with the parsing algorithm altered appropriately, the `add-three` macro defined above will not introduce a new variable with scope that includes code at its invocation.

5.6 Identifiers

In the parse algorithm we've been looking at, name resolution is a process that maps an identifier to a variable. Up until now, we've considered identifiers to be synonymous with symbols, but the example presented in the previous section demonstrates that such unadorned symbols are not adequate. Name resolution needs to have some information about the expansion context in which the object it is resolving exists. We will need to redefine an *identifier* as a symbol adorned with this expansion information.

Name resolution thus has two steps. The first step involves resolving symbols to identifiers. This is done by the underlying macro machinery, and is for the most part invisible to the macro writer. The second step involves resolving identifiers to names, and uses the machinery we have already seen. In this way, the macro writer is shielded from the complicated bookkeeping involved in preserving hygiene.

The expansion context information takes the form of a set of *marks.* A mark is an arbitrary object "painted" on concrete syntax when macros are expanded, so that identifiers which are introduced at different expansion steps are distinguishable. In Ziggurat, we implement marks as integers. Mark sets are represented as sorted lists of integers, since the sets we will use will be fairly small.

In order to mark concrete syntax, we introduce a new kind of concrete form.

$m \in$ Mark
$f \in$ Form ::= ...
| *mark*(*m*, *f*)

We assume that *mark* forms are special, distinct from other forms. The concrete syntax *mark*(*m*, *f*) is inserted and parsed by the underlying macro machinery – there is no need for the macro writer to ever generate or observe a mark. We can use delegation here to make mark syntax completely transparent. By having mark syntax delegate to the enclosed expression, predicates and field accessors completely hide the fact that the syntax is marked. The syntax class of a piece of marked datum is `mark-datum-wrapper%`.

```
(define-class mark-datum-wrapper% (mark^ datum^)
  (lambda ()
    (cond
      ((pair?: datum^)
       (cons (object mark-datum-wrapper% mark^
                     (car: datum^))
             (object mark-datum-wrapper% mark^
                     (cdr: datum^))))
      ((identifier? datum^)
       (toggle-mark: datum^ mark^))
      ((symbol?: datum^)
       (toggle-mark: (symbol->identifier: datum^) mark^))
      (else datum^))))
```

An identifier consists of a symbol and a set of these marks. The marks of an identifier indicate the marking syntax that encloses the identifier in the program source, with the nuance that marking an identifier an even number of times with the same mark results in the mark not being added to the identifier's mark set. A mark "cancels itself out." Parsing an identifier thus consists of adorning the symbol being parsed with its enclosing mark set. Syntactic environments must therefore keep track of the "current" mark environment.

We thus introduce a new form of syntactic environment, the mark environment. A mark environment is entered when we parse a mark in concrete syntax.

```
(define-class (mark-env% is-mark-env?:)
  ((mark^ mark-env-mark:)
   (enclosing-env^ mark-env-enclosing-env:))
  (lambda () enclosing-env^))
```

It is possible to calculate the mark set represented by a syntactic environment by taking the mark set of the enclosing environment, and finding the symmetric difference of that set and a singleton set containing the mark. If we assume some set primitives defined on mark sets, we can implement this operation:

```
(define-attribute current-mark-set: empty-mark-set)

(define-method mark-env% current-mark-set:
  (lambda (env) (symmetric-difference:
                 (pass) (mark-set mark^))))
```

Parsing a form requires special behavior in case it is given a marked form. We must therefore modify `parse-arith-expr` to reflect this special behavior.

```
(define (parse-arith-expr env form)
  (cond
    ((mark-datum-wrapper?: form)
     (parse-arith-expr
      (enter-mark-env env (mark-datum-wrapper-mark: form))
      (mark-datum-wrapper-datum: form))
     (else ...))))
```

An identifier is a piece of abstract syntax, consisting of a symbol and a set of marks.

```
(define-class identifier% (symbol^ mark-set^))
```

Parsing a symbol as an identifier thus involves taking the input symbol and pairing it with the current mark set.

```
(define (parse-identifier env form)
  (cond
    ((mark-datum-wrapper?: form)
     (parse-identifier
      (enter-mark-env env (mark-datum-wrapper-mark: form))
      (mark-datum-wrapper-datum: form))
    ((symbol?: form)
     (object identifier% form (current-mark-set: env)))
    (else
     (parse-error form "Expected an identifier"))))
```

Marks are introduced into a program during macro expansion. Since the burden of maintaining hygiene should not fall on the language designer, and the language designer is responsible for implementing `let-syntax`, the code to introduce marks does not appear in the parsing of `let-syntax`. Instead, the language designer calls a function, `parse-transformer`, which parses a macro transformer and inserts the necessary mark machinery.

```
(add-keyword-parser!: top-arith-env (syntax let-syntax)
  (lambda (env form)
    (syntax-case form ()
      ((let-syntax (name transformer) exp)
       (let ((new-env (enter-arith-keyword-env env))
             (transformer (parse-transformer
                           (syntax transformer))))
         (add-keyword-parser!: new-env (syntax name)
                               transformer)
         (parse-arith-expr: new-env (syntax exp)))))))
```

The function `parse-transformer` is given Ziggurat syntax which evaluates into a parse function. The value returned by `parse-transformer` is a parse function that

wraps this function with macro machinery: it generates a mark, wraps the input form with this mark, and enters a mark environment.

```
(define (parse-transformer parser)
  (let ((parser (eval parser)))
    (lambda (env form)
      (let* ((mark (make-mark))
             (new-form (object mark-datum-wrapper%
                               mark form))
             (new-env (enter-mark-env env mark)))
        (parser new-env new-form)))))
```

In Dybvig, Hieb and Bruggeman's original algorithm, a macro transformer would have a mark placed on its input, and the same mark placed on its output. However, in Ziggurat, macros are not source-to-source transformers, but rather, parse functions, and thus their output is abstract syntax and can not be wrapped with a mark. For our purposes, entering a mark environment achieves the same effect: parse functions in that syntactic environment behave as though the form were wrapped in the mark.

Marks have the property that marking a piece of concrete syntax an even number of times with the same mark has the same effect as no mark being applied. The result of this two-stage marking is that identifiers introduced in the input to the macro do not have the new mark attached to it, and identifiers introduced by the macro do. Since names are associated with identifiers, this ensures that variables and other program objects introduced by macros are distinguishable from those introduced in the context of that macro's invocation.

This might be best shown through an example. Let's consider the case we had above.

```
(let-syntax
  (add-three
   (lambda (env form)
     (syntax-case form ()
       ((add-three x)
        (parse-arith-expr env
          (syntax (let (y 3)
                    (+ x y))))))))
  (let (y 4)
    (add-three y))
```

Let's trace through the execution of the Arith expression parse function. To do so, let's keep track of two parts of the syntactic environment: the current mark environment, and the contents of the variable namespace. Let's also keep track of the concrete syntax to be parsed, and the abstract syntax that has already been parsed.

```
mark environment: {}

variable namespace: {}

concrete syntax:
(let (y 4)
  (add-three y))

abstract syntax:
□
```

In the above figure, we use a □ to represent the part of the abstract syntax that is yet to be parsed. The first piece of concrete syntax to be parsed would be the let expression. The symbol `y` would be parsed as the identifier `y` with no marks, and bound to a new symbol, which we'll say is `g100`.

```
mark environment: {}

variable namespace: {y ↦ g100}

concrete syntax:
(add-three y)

abstract syntax:
(object arith-let%
        'g100
        (object arith-const% 4)
        □)
```

The definition of `add-three` is a macro, so we enter a mark environment, and parse the syntax inside the macro definition.

```
mark environment: {1}

variable namespace: {y ↦ g100}

concrete syntax:
(let (y 3) (+ (mark 1 y) y))

abstract syntax:
(object arith-let%
        'g100
        (object arith-const% 4)
        □)
```

Parsing this new `let` expression is similar to parsing the previous expression, with a caveat: since the current mark environment is {1}, the symbol `y` is parsed as the iden-

tifier `(mark 1 y)`. Therefore, the variable environment has two entries, each mapping a different identifier to a different value.

```
mark environment: {1}

variable namespace: {y ↦ g100, (mark 1 y) ↦ g101}

concrete syntax:
(+ (mark 1 y) y)

abstract syntax:
(object arith-let%
        'g100
        (object arith-const% 4)
        (object arith-let%
                'g101
                (object arith-const% 3)
                □))
```

Parsing the addition involves recursively parsing the two sub-expressions.

```
mark environment: {1}

variable namespace: {y ↦ g100, (mark 1 y) ↦ g101}

concrete syntax:
(mark 1 y)

abstract syntax:
(object arith-let%
        'g100
        (object arith-const% 4)
        (object arith-let%
                'g101
                (object arith-const% 3)
                (object arith-add% □ □))
```

Since this is a marked symbol, it parses as an identifier. The mark environment contains only the mark `1`, which cancels out the mark `1` on the symbol being parsed, so the resulting identifier is simply `y`, which parses as the variable `g100`.

```
mark environment: {1}

variable namespace: {y ↦ g100, (mark 1 y) ↦ g101}

concrete syntax:
y

abstract syntax:
(object arith-let%
        'g100
        (object arith-const% 4)
        (object arith-let%
                'g101
                (object arith-const% 3)
                (object arith-add%
                        (object arith-var% 'g100)
                        □))
```

Since the symbol `y` is unmarked, but is being parsed in the mark environment $\{1\}$, it is parsed as the identifier `(mark 1 y)`, which, when looked up in the variable namespace, resolves to the variable `g101`. The result is:

```
(object arith-let%
        'g100
        (object arith-const% 4)
        (object arith-let%
                'g101
                (object arith-const% 3)
                (object arith-add%
                        (object arith-var% 'g100)
                        (object arith-var% 'g101)))
```

This resulting abstract syntax adds the two distinct variables, as we intended. Unwanted variable capture has been avoided.

5.6.1 Symbols, identifiers and names

The major goal of this hygiene algorithm is to associate referring source-program terminals (*i.e.*, the symbols `x`, `y`, `let`, `+`) to their appropriate referents (*i.e.*, the variables "x" and "y", and the parse functions for the keywords "let" and "+".) As we've seen, there are three ways that referers are distinguished.

- The first are *symbols*. Symbols are terminals in the concrete grammar. Two symbols are typically equal if and only if their written representations are the same.

- The second are *identifiers*. Identifiers are used to distinguish between different stages of expansion in which referents can be introduced.

  To see how identifiers are distinguished, consider the Arith code:

  ```
  (let-syntax
    (foo
      (syntax-case ()
        ((foo x)
         (lambda (env form)
           (parse-arith-expr env
             (syntax (let (x 3) (let (y 4) x))))))))))
    (foo y))
  ```

  If we were to expand this without distinguishing identifiers, we would get:

  ```
  (let (y 3) (let (y 4) y))
  ```

  There are three occurrences of the symbol `y`: in the variable binding occurrence `(let (y 3) ...)`, the variable binding occurrence `(let (y 4) ...)`, and in the variable use. Ideally, we want the two variable bindings to bind different variables, and we want the variable use to refer to the variable in the outermost binding. The distinction that we draw between the two variable bindings is where the symbol `y` comes from: in the `(let (y 4) ...)` binding, the symbol `y` was introduced during macro expansion, whereas in the `(let (y 3) ...)` binding, the symbol `y` was introduced as an argument to the macro. Since the variable use `y` was also introduced as an argument to the macro, it should refer to the outermost binding. We say that these two uses of the symbol `y` are the same identifier.

- The third are *names*. Names are language- and context-dependent interpretation of identifiers. For example, in the Arith program:

  ```
  (let (+ 3) (+ + +))
  ```

  All instances of `+` represent the same identifier. However, in `(+ + +)` there are represented two names: the identifier `+` interpreted as a syntactic keyword, and the identifier `+` interpreted as the Arith variable bound to 3. Because the language draws this distinction, the above program evaluates to 6.

This is why the algorithm demonstrated above proceeds in two phases: first it adds marks to symbols to produce identifiers, and then it resolves identifiers to names. However, there is still a deficiency in this algorithm.

5.7 Namespace linking during expansion

We would like bound identifiers visible at the macro definition site to be available during macro expansion. This is a simple matter in the algorithms Ziggurat is built

on. However, these underlying algorithms rely on a facet of Scheme that we cannot use in the general case: that all identifiers are looked up in the same namespace. Thus, these algorithms can do namespace resolution during the macro definition, whereas in Ziggurat, we cannot do resolution until the macro is expanded, and we know more about the syntactic context of the expansion.

In order to ensure this property, we must modify the parse algorithm, For example, if we were to define an `add-x` macro:

```
(let (x 3)
  (let-syntax (add-x (lambda (env form)
                       (syntax-case form ()
                         ((add-x y)
                          (parse-arith-expr
                           env (syntax (+ x y))))))
   ...))
```

we would want the occurrence of `x` in `(+ x y)` to be bound to the variable `x` in scope where `add-x` was declared, `(let (x 3) ...)`. So, if we filled in the rest of this expression:

```
(let (x 3)
  (let-syntax add-x
    (lambda (env form)
      (syntax-case form ()
        ((add-x y)
         (parse-arith-expr env (syntax (+ x y))))))
    (let (x 13)
      (add-x 7))))
```

we would want the result to be `10`. As the parse algorithm currently stands, the occurrence of `x` in `(+ x y)` is not captured by the declaration `(let (x 13) ...)`, as would be undesirable, but neither is it captured by the declaration `(let (x 3) ...)`, as we want. Currently, this use of `x` is free.

This is a problem in the syntax-object framework that the Ziggurat approach is based on. [38]. Our solution is similar, but involves the Ziggurat machinery. This can be fixed by adding a new kind of namespace. When the macro `add-x` is expanded, the effect we want can be achieved by taking all identifiers present in the syntactic environment at the time `add-x` was defined, and duplicating their entries, marking them with the mark introduced by the current invocation of `add-x`. This would be quite inefficient, and could potentially cause namespaces to grow exponentially. However, we can achieve the same effect by introducing a new kind of syntactic environment: the namespace-duplicating syntactic environment.

```
(define-class namespace-dup-env% (marks^ enclosing-env^)
  (lambda () enclosing-env^))
```

Looking up any identifier `i` in a `namespace-dup-env%` has the effect of stripping

all of the marks in the table `marks^` from `i`, and then delegating the search to the enclosing the environment. The effect of this is to duplicate the enclosing environment for each of the marks in the mark table.

```
(define-method namespace-dup-env%
               arith-var-namespace-lookup:
  (lambda (env form)
    (arith-var-namespace-lookup:
     enclosing-env^ (strip-marks: form marks^))))
```

Now, each time we define a macro, we insert a `namespace-dup-env%`. If we recall the definition of `let-syntax`, we can add this functionality with minimal change to `let-syntax`, so as to again insulate the language designer from the bookkeeping required. We instead modify the definitions of the generic functions `enter-arith-env` and `parse-transformer`. For this to work, we must also pass the new syntactic environment into `parse-transformer`.

```
(add-keyword-parser!: top-arith-env (syntax let-syntax)
  (lambda (env form)
    (syntax-case form ()
      ((let-syntax (name transformer) exp)
       (let* ((new-env (enter-arith-env env))
              (transformer
               (parse-transformer (syntax transformer)
                                  new-env)))
         (add-keyword-parser!: new-env (syntax name)
                               transformer)
         (parse-arith-expr: new-env (syntax exp)))))))
```

The new `enter-arith-env` must additionally enter a `namespace-dup-env%`.

```
(define (enter-arith-env env)
  (object namespace-dup-env% (make-hash-table)
          (enter-arith-keyword-namespace env)))
```

The new `parse-transformer` must return a transformer that likewise adds its mark to the `namespace-dup-env%`.

```
(define (parse-transformer parser dup-env)
  (let ((parser (eval parser)))
    (lambda (env form)
      (let* ((mark (make-mark))
             (add-dup-mark: dup-env mark)
             (new-form (object mark-datum-wrapper%
                               mark form))
             (new-env (enter-mark-env env mark)))
        (parser new-env new-form)))))
```

This new definition enables the namespace duplication we need.

5.8 Macro-defining macros

Since the hygiene mechanism in Ziggurat is based on an established algorithm, we can provide previously-developed features with a minimum of hassle. For example, it is possible to write a macro that defines another macro. To illustrate this, we can write a macro `define-plus-three-macro` that takes one argument: an identifier that it binds to a macro that adds three to its argument.

```
(let-syntax
  (define-plus-three-macro
   (lambda (outer-env form)
     (syntax-case form ()
      ((define-plus-three-macro id exp)
       (parse-arith-expr outer-env
         (syntax
          (let-syntax
            (id (lambda (inner-env form)
                  (syntax-case form ()
                    ((add-three x)
                     (parse-arith-expr outer-env
                       (syntax (let (y 3)
                                 (+ x y))))))))
            exp)))))))
  ...)
```

Since keywords are parsed as identifiers and bound in namespaces, they can be used like any other arguments to a macro. Thus, defining a macro that defines a macro is semantically as straightforward as writing a macro that binds a variable.

There is a subtle point that differentiates macro-defining macros in this Ziggurat from macro-defining macros in other macro systems. In the example above, there are two languages represented: the Ziggurat meta-language, and the Arith language. These languages are nested several levels deep: the Arith expression `(+ x y)`, for example, is within the `add-three` macro definition (which is defined in the Ziggurat language), which is being bound with an Arith `let-syntax` expression, which is within the `define-add-three` macro definition (again defined in the Ziggurat language), which is again bound with an Arith `let-syntax` expression. This multi-layer nesting is common to all macro-defining macros, and causes an associated namespace issue that must be dealt with.

Namespace resolution in Arith is handled nicely by the hygiene mechanism covered in this chapter. However, there is still an issue with hygiene in the Ziggurat meta-language. Ziggurat is built on the existing Scheme implementation PLT Scheme, and inherits its hygiene mechanism from that. However, in order to mix Ziggurat code with Arith code as in the above example, each piece of Ziggurat code is considered to

be in its own namespace. Therefore, Ziggurat metavariables bound in the outer scope are not visible in the inner scope. This works well enough in practice, but it may be that a macro writer would want to have such a variable visible. In order to do this, it would be necessary to link the hygiene mechanism of the metalanguage with the hygiene mechanism of the language being defined, so that marks in the metalanguage correspond to marks in the language being defined.

Chapter 6

The importance of a fixed semantics

In section 4.1, we noted that it is important that the set of semantic methods remain fixed, and be executed in a phased order, and that language extensions not alter the set or order of semantics. The reason for this is somewhat subtle, and can best be seen by an example.

Let's consider an unusual syntactic extension for the Arith language. Ordinary Arith has the property that variable bindings dominate uses. With macros, we can choose to add a new form of variable that has its own scope story, and does not obey this property. Let's create **constants**, which are introduced by the `(with-constants` *e*`)` macro. Constants are defined with `(define-constant (`*i c*`)` *e*`)`, which defines the named constant *i* to be the value *c*, and continues with the evaluation of *e*. The form `(constant` *i*`)` will evaluate to the value of the constant *i*, *even if the constant use is not dominated by its definition.* Named constants thus have a flat namespace, delineated only by the `with-constants` form. An example of using constants is:

```
(with-constants
  (* (+ (constant x) (constant y))
     (define-constant (x 3)
       (define-constant (y 4)
         5))))
```

In the delegated abstract syntax, we would like there to be an outermost let-binding for each constant being defined. This let-binding defines a generated symbol, unique for each constant.[1]

The above code would look like:

[1]Other translations are possible, of course. We could, for example, simply replace all references to a named constant with the actual integer. This would not be a good strategy if our constants were "large" values such as strings, whose replication could result in space blowup. Additionally, the translation strategy we are using here is intended to illustrate issues in phase-ordering.

```
(let (g100 3)
  (let (g101 4)
    (* (+ g100 g101) 5)))
```

To implement this, we define three new syntax classes: `arith-with-constants%`, `arith-define-constant%` and `arith-constant%`. The outermost syntax class `arith-with-constants%` delegates to a tree of `arith-let%` objects, while the `arith-define-constant%` class delegates to the enclosed expression, and the `arith-constant%` delegates to the relevant variable. Implementing this presents a difficulty: when the `arith-with-constants%` object is instantiated, the set of constants defined is not yet known. A simple pass over the code will not reveal all of the named constants in the code, the body of the `with-constants` form may include macros that expand to named constants. There needs to be a way for constant definitions to communicate with the enclosing `with-constant` forms. There are two ways to do this: the naïve way and the correct way. Let's see the naïve way first.

We define a new generic function, `defined-constants:`, on the primitives of Arith and on the extensions we define. This generic function returns an association list of triples, containing (1) the constant being defined, (2) the value to which it is bound, and (3) a generated symbol to represent it in the generated code. For example, invoking `defined-constants:` on

```
(define-constant (x 3)
  (define-constant (y 4)
    5))
```

would return the association list `((x 3 g100) (y 4 g101))`.

Next, we define `arith-constant%` in such a way that it contains a field to hold the generated symbol that represents the variable to which the constant will delegate.

```
(define-class arith-constant% (name^ new-id^)
  (lambda ()
    (if new-id^
        (object arith-var% new-id^)
        (error "Premature delegation of constant"))))
```

The field `new-id` is initially undefined, so we define a generic function `set-constant-new-id!:` to set it. This generic function would take an association list of the kind returned by `defined-constants:`.

```
(define-generic-function (set-constant-new-id!: exp id))

(define-method arith-constant% set-constant-new-id!:
  (lambda (this constants)
    (let ((record (assoc name^ constants)))
      (if record (set! new-id^ (caddr record))))))
```

The method for `set-constant-new-id!:` on Arith nodes (except `arith-constant%`) would simply pass the generic function on to its enclosed nodes.

Then, the delegation method of `arith-with-constants%` would proceed in two phases:

- it would perform a pass over the inner expression, gathering the defined constants,
- it would perform a second pass to set the new-ids of each of the used constants.

After this is done, it would delegate to the appropriate let-nodes.

```
(define-class arith-with-constants% (inner-exp^)
  (lambda ()
    (let* ((constants (define-constants: inner-exp^))
      (set-constant-new-id!: inner-exp^ constants))
      (let loop ((constants constants))
        (if (pair? constants)
            (let ((record (car constants)))
              (object arith-let% (caddr record)
                                 (cadr record)
                      (loop (cdr constants))))
            inner-exp^)))))
```

This adds new static semantics to the language. A fundamental part of a language definition in Ziggurat is a fixed static semantics, and thus this is not the correct way to implement this. The reason that this is incorrect is subtle, since, after all, the example above works. The problem is in how it combines with other macros. Let's consider what happens when this macro is used in combination with an identical macro, with different names. For example, we could define a `with-konstants` macro, that introduces **konstants**, bound with `define-konstant` and referenced with `konstant`.[2] We could use both macros in the same program, which might look like:

```
(with-constants
  (with-konstants
    (* (constant x) (konstant y)
       (define-constant (x 3)
         (define-konstant (y 4)
           5)))))
```

Both constants and konstants define new static semantics, in the form of new generic functions, `defined-constants:` for constants, and `defined-konstants:` for konstants. Calculating the delegate of a `with-constants` node requires invoking the `defined-constants:` generic function on the enclosed syntax. If the enclosed syntax contains a `konstant` node, we run into a problem: the delegation method will signal an error. In order to see this, let's consider the method invocations that take place.

[2]This example may seem a little contrived, but this problem will also emerge in cases where the macros are not exactly the same, but merely each define new static semantics. The simplest demonstration of this is in the case where the two macros are functionally identical.

- The `with-constants` node calls `defined-constants:` on its enclosed expressions.
- Since the `with-konstants` node does not define a `defined-constants:` method, it must delegate.
- Calculating the delegate of a `with-konstants` node requires invoking the `defined-konstants:` method of the sub-expressions of the `with-konstants` node.
- This requires calculating the `defined-konstants:` of enclosed expressions, and specifically, the `constant` node.
- Since the `constant` node does not define a `defined-konstants:` method, it must delegate.
- Calculating the delegate of a `constant` node requires the `new-id` field to be set. However, this field will not be set until later in the delegation method of `with-constants`.

This circular dependency is the reason that the set of static semantics of the language is fixed. This requires us to implement constants somewhat differently. Fortunately, in Ziggurat, we have the entire Scheme-based language at our disposal. The solution, thus, is simple: we merely define a top-level Ziggurat variable, `constants`, and each time we parse a `define-constant` expression, we add that constant to `constants` by side-effect.[3]

```
(define constants '())

(define-class arith-with-constants% (inner-exp^)
  (lambda ()
    (let loop ((constants constants))
      (if (pair? constants)
        (let ((record (car constants)))
              (object arith-let% (caddr record)
                                 (cadr record)
                         (loop (cdr constants))))
            inner-exp^))))


(define-class arith-constant% (name^)
  (lambda ()
    (let ((record (assoc name^ constants)))
      (if record (caddr record)
          (error "Unknown constant")))))
```

[3]The use here of a global constant is due to the simplicity of the macro system. When we define macro systems for more complex languages such as C/sexp, we will have the ability to declare a variable local to a set of macro definitions. However, here all procedural macros have Ziggurat code defined within a single global scope, and thus, if we want to declare a variable within the scope of multiple macro definitions, we must make it global.

```
(add-keyword-parser!: top-arith-env (syntax constant)
  (lambda (env form)
    (syntax-case form ()
      ((constant c n)
       (let ((c-id (parse-identifier env (syntax c)))
             (n-val (parse-integer env (syntax n)))
             (new-od (genid (syntax c))))
         (set! constants `((,c-id ,new-id ,n-val) .
                           constants))
         (syntax-object arith-constant%
                        form env new-id))))))
```

This gets around the problem presented earlier. However, it does come with its own nuance: any macro expanding into a `constant` or `define-constant` form must either (1) expand during parsing, or (2) register the constant to `constants` itself.

6.1 Phase ordering

Simply having a fixed static semantics is not enough to prevent a circular dependency. In order to ensure that there are no cycles in the graph of method calls, the language designer must adhere to a set of rules.

- A language extension may not add new semantic functions to the language.
- Each generic function must be associated with a phase.
- If the delegate instantiation function of a syntactic class invokes a generic function of phase n, then that class must define methods for all semantic functions of phase n and below.

These rules are enough to ensure that an delegate instantiation method will not cause a dependency cycle. Right now, these conditions are merely up to the language designer to enforce. It is possible that these could be ensured at the language level; this would be an interesting avenue for future work.

Chapter 7

C/sexp

7.1 Motivation

A language tower must have a first floor, and C makes an excellent example of such a base, for a number of reasons:

- *Feature-poor.* Language-oriented programming builds up a language, feature by feature, until it is adequate for its intended tasks. Logically, then, the base language should not have many specialized features by itself. C fits this bill well. It has some basic features: function call, simple record and union syntax for structured data, and a weak type system.[1] However, it lacks many of the features of other higher-level language: for example, although C has function pointers, these are merely code pointers, not true closures, as we get in functional languages. Although C has records and record types, there is no notion of subtyping or inheritance as in a full object system. There is no exception mechanism, no polymorphism save ad-hoc conversion, and no module system. By adding an advanced macro system, these features can be added to the language selectively.

- *Weak type system.* When implementing language extensions, we will often have static checks we will want to implement. For example, if we were to implement a class-and-method-based object system, *à la* C++ or Java, we might want to ensure that a class implements the methods of its superclass correctly, that the subtype graph is cycle-free, and so forth. We can implement these checks as an extension to the base type system: we define a class definition as a new top-level declaration, and define a method for the type-checking generic function that only accepts the declaration if it passes all of these requirements. By defining this method, we extend the type system of C/sexp to encompass new ideas of program correctness.

- *Proliferation of implementations.* Since the macro system rewrites into plain C

[1] C's type system is both inexpressive and subvertable. This is what we mean by a "weak" type system: it provides no guarantees about the code.

code, a C-with-macros front-end can be pipelined into any conforming C compiler. Thus, a single macro system can be used on almost any platform.

The basic language of this system, then, is a variant of C that has nearly exactly the same abstract syntax, but an s-expression-based concrete syntax, and does not have C's simplistic, substitution-based macro system. Instead, it uses Ziggurat's hygienic macro system. Ziggurat's semantic components are used to implement C's type system and various source-level analyses, and allow them to be integrated with analyses of embedded languages.

7.2 Design of C/sexp

C/sexp's concrete syntax is essentially a direct s-expression encoding of the abstract syntax of C99 [43]. This s-expression encoding influences the design in a few ways.

- There is no need to specify operator precedence in an s-expression-based language, greatly simplifying the syntax.
- C's macro system has been completely removed. In most cases, the C99 standard separates the macro system from the unadorned language; however, in some cases, the macro system is used in the specification of the language, which causes some difficulty. For example, the identifier `NULL` is specified to be a CPP macro that expands to an implementation-dependent null-pointer constant. In C/sexp, `NULL` can be defined as a constant of type `void *`.
- In C, the basic types are specified by arbitrary combinations of the keywords `char`, `short`, `int`, `long`, `float`, `double`, `signed`, `unsigned`, `_Bool`, and `_Complex`. However, many of these combinations are invalid, and many indicate the same type. In fact, there are exactly eighteen valid basic types. In C/sexp, these eighteen types are pulled out as syntactic atoms. For example, the basic type that C allows to be specified `long long int` and `signed long long int` in C/sexp is specified `long-long-int`.
- In some cases, C/sexp provides implicit bracketing and sequencing. For example, the C function definition

```
int f() {
  foo = foo + 1;
  return foo;
}
```

in C/sexp can be written

```
(fun f int ()
  (:= foo (+ foo 1))
  (return foo))
```

- In Ziggurat, for macro writing, wherever a sequence of syntactic nodes appear, it helps to have a construct to introduce such a sequence as a single element. For example, the `block` statement performs this task for sequences of statements. However, there is no equivalent grouping construct for declarations, so, C/sexp allows declarations also to be grouped with the `block` keyword.

A possible alternate design might be to port only a core subset of the C syntax, and to implement the remaining C primitives as macros that rewrite to those primitives. This would be possible and straightforward to implement in Ziggurat. However, an expectation underlying the design of C/sexp is that the C compilers used as back-ends may rely on analyses that take advantage of the full language, and thus it makes sense for the C/sexp tool to implement the full range of C primitives as primitives, and pass them to back-end unmodified.

7.3 Compilation phases

Compilation of a C/sexp program occurs in four phases. Each of these phases consists of a set of static semantic functions, which will be used to compile a program. Making sure that only those generic functions that are defined to occur in a certain phase actually occur in that phase is key to preventing macros from delegating prematurely.

- Parsing: turning annotated data into syntax objects.
- Static analysis: performing decoration of the syntax tree, and rejecting those programs that violate the static conditions of the language, such as required type conditions.
- Additional analysis: performing additional (debugging) analyses on the C/sexp code.
- Code generation: turning the syntax tree into a C program, suitable for being passed to a standard C compiler.

Generic functions and attributes associated with one phase of compilation must not be invoked before their phase. In this fashion, syntax objects will not delegate prematurely, which could cause macros to perform in unexpected ways.

7.4 Concrete syntax

The concrete syntax of C/sexp is given in figure 7.1.

7.5 Examples

To make the definition of C/sexp less abstract, this section presents some simple C/sexp programs, without macros.

$i \in$ Ident
$n \in$ Constant
$p \in$ Program ::= (*d* ...)
$d \in$ Declaration ::= (fun *i t* ((*i t*) ...))
| (fun *i t* ((*i t*) ...) *s* ...)
| (var *i t*)
| (var *i t e*)
| (type *i t*)
| (block *d* ...)
| *t*

$t \in$ Type ::= *bt*
| *i*
| (struct *i*)
| (struct *i* ((*i t*) ...))
| (union *i*)
| (union *i* ((*i t*) ...))
| (enum *i* (*i* ...))
| (* *t*)
| (array *t*)
| (array *n t*)
| (-> *t* (*t* ...))

bt ∈ BasicType ::= char
| signed-char
| short-int
| int
| long-int
| long-long-int
| unsigned-char
| unsigned-short-int
| unsigned-int
| unsigned-long-int
| unsigned-long-long-int
| bool
| float
| double
| long-double
| complex-float
| complex-double
| complex-long-double
| void

$s \in$ Statement ::= *d*
| *e*
| (label *i s*)
| (switch *e s*)
| (case *s s*)
| (default *s*)
| (block *s* ...)
| (if *e s*)
| (if *e s s*)
| (while *e s* ...)
| (do *e s* ...)
| (for *s e s s* ...)
| (goto *i*)
| (continue)
| (break)
| (return *e*)
| (return)

$e \in$ Expression ::= *i*
| *n*
| (aref *e e*)
| (fld *e i*)
| (-> *e i*)
| (init *t e* ...)
| (sizeof *t*)
| (cast *t e*)
| (?: *e e e*)
| (*uso e*)
| (*uo e*)
| (*bo e e*)
| (*ro e e*)
| (*ao e e*)
| (*e e* ...)

uso ∈ UnarySEOp ::= pre+ | pre-
| post+ | post-
uo ∈ UnaryOp ::= * | & | ~ | !
bo ∈ BinaryOp ::= + | - | *
| / | % | <<
| >> | ^ | '|'
| & | && | '||'
ro ∈ RelationOp ::= < | > | <=
| >= | == | !=
ao ∈ AssignOp ::= := | =* | =/
| =% | =+ | =-
| =<< | =>>
| =& | =| | =^

Figure 7.1: The grammar of C/sexp.

7.5.1 Hello world

The canonical first example program is the classic toy program that, upon execution, outputs "Hello World!" The C/sexp variant should be reasonably clear to anyone familiar with both regular C syntax and s-expression programming languages.

```
(include "stdio.clib")

(fun main int ((argc int) (argv (* (* char))))
  (printf "Hello World!")
  (return 0))
```

It's worth noting here that the `include` form is actually a macro, that will be covered later in this chapter.

7.5.2 Binary tree search

To demonstrate C/sexp in some more depth, this section defines types for binary trees and a function to search a tree. This demonstrates a couple of features of C/sexp:

- type definitions
- function types and function objects

The implementation of this is in figure 7.2. The first step of this program is to import `stdlib.clib`, which contains some basic utilities for the language, particularly the constant `NULL`.

Next, we define the type `binary-tree`. Objects of this type consist of a pointer to a structure with the tag `binary-tree`.[2] This structure has three fields: a value field, with the essentially polymorphic type `(* void)`, and a left and a right field, which contain pointers to a `binary-tree` structure. Here, the type `(struct binary-tree)` is incomplete until the final right parenthesis of its definition. References to this type can be used, to allow such recursive types, but fields of the type cannot be used, to disallow structures of unbounded size.

Finally, we define a search function over these binary trees. The search function takes three arguments: a tree to search, an object to search for, and a comparison function. The comparison function takes two arguments to compare, which should both be of polymorphic type `(* void)`. It should return an integer, which should be less than `0` to indicate its first argument is less than its second, greater than `0` to indicate its first argument is greater than its second, or `0` to indicate that they are equal.

The first thing that this function does is check whether the tree to search contains an element. Next, it applies the comparison function and assigns the result to the variable `comp-val`. Finally, it tests the result of this comparison and either recurses, or returns the value at the root of the tree in the case of equality.

[2] Here, the identifier `binary-tree` is being parsed in two separate namespaces: as a type name, and as a structure tag. The fact that the identifier for both the type name and structure tag is the same is irrelevant to the parser.

```
(include "stdlib.clib")

(type binary-tree
      (* (struct binary-tree
                 ((value (* void))
                  (left (* (struct binary-tree)))
                  (right (* (struct binary-tree)))))))

(fun search (* void) ((tree binary-tree)
                      (key (* void))
                      (compare (-> int ((* void) (* void))))))

  (if (== NULL tree) (return NULL)) ; empty tree

  (var comp int (compare key (-> tree value)))

  (if (! comp) (return (-> tree value)))

  (if (< comp 0)
      (return (search (-> tree left)  key compare))
      (return (search (-> tree right) key compare))))
```

Figure 7.2: Searching a binary tree in C/sexp.

An interesting thing to note here is that C/sexp's type system, since it mirrors C's type system, allows one to express such advanced concepts as function types and polymorphic types, but provides no type-safety guarantees when these features are used.

7.6 Abstract syntax

The abstract syntax presented in this section does not include macro definitions or uses. C/sexp macros will be introduced in section 7.8. Furthermore, as this abstract syntax mirrors the syntax of C, this section will not cover every single production in the language, but instead will consider these productions by category.

C/sexp has four basic types of syntax node: declarations, statements, expressions and types. Declarations and expressions can also be used as statements, and types can be used as declarations. There are three effective namespaces used in the language: `csexp-var-namespace`, used for variables, functions, and enumeration constants; `csexp-tag-namespace`, which is used for struct and union tags; and `csexp-type-namespace`, used for named types.

These namespaces do not cover all of the possible identifiers in C/sexp. There are two kinds of identifiers not handled by namespaces: labels and fields. Labels are used by goto statements and labeled statements, and are declared on first use. Due to this,

there is no need to store labels in a namespace; labels are parsed simply as identifiers. Record and union fields have rather more complex scope: they are declared in a type declaration, and thus, resolution cannot occur until type-check time. So, fields are parsed as identifiers as well.

7.6.1 Declarations

The top level of a C/sexp program consists of a sequence of declarations. Putting aside syntax declarations for the moment, there are five types of declarations that can appear at the top level: function prototypes, declaration blocks, and declarations for functions, variables, and types.

```
(define-syntax-class (fun-csexp-decl%
                      is-fun-csexp-decl?:)
  (id^ name^ type^ args^ arg-types^
   stm^ args-extensible?^))
```

All computation in a C/sexp program happens as the result of calling an explicitly-declared function. A function declaration, in addition to arguments, argument types and a return type, contains a single statement. Functions may be mutually recursive; however, before a function is used, it must be declared, either as a regular function declaration, or as a function prototype. Function prototypes appear as function declarations, without enclosed statements.

Variable declarations associate a type to a variable identifier, and optionally a value.

```
(define-syntax-class (var-init-csexp-decl%
                      is-var-init-csexp-decl?:)
  (id^ name^ type^ init^))
```

Type declarations associate a type with a type identifier, similar to a `typedef` declaration in C.

```
(define-syntax-class (type-csexp-decl%
                      is-type-csexp-decl?:)
  (id^ name^ type^))
```

7.6.2 Types

A C/sexp program can contain syntactic types. At type-check time, these syntactic types are resolved to semantic types, which are actually used for type-checking and related compiler activities. Syntactic types can be one of the 19 basic atomic types presented in figure 7.1, a named type, an enumerated type, a tagged product type (known as a `struct`), a tagged sum type (known as a `union`), a pointer type (parameterized by the type of the object that is being referred to), or a function type.

```
(define-syntax-class (union-csexp-type%
                      is-union-csexp-type?:)
  (tag^ id^ names^ types^))
```

The semantic types that these syntactic types refer to are covered in detail in section 7.7.1.

7.6.3 Statements

A function in C/sexp consists of a single statement. A statement is the basic unit of computation in a C/sexp program: it can perform control effects, such as branching, looping, function return and unstructured control operations (with `goto`), and contain expressions which can have addition control effects such as function call, and mutate variables. The targets of `goto` operations, known as labels, are also considered statements. Statements in C/sexp exactly correspond to statements in C, and thus will not be covered in depth here.

```
(define-syntax-class (if-then-else-csexp-stm%
                      is-if-then-else-csexp-stm?:)
  (cond^ if-true^ if-false^))
```

A statement may also be a `block`, which consists of a sequence of statements to be computed in order.

```
(define-syntax-class (block-csexp-stm%
                      is-block-csexp-stm?:)
  (stms^))
```

7.6.4 Expressions

```
(define-syntax-class (binary-csexp-exp%
                      is-binary-csexp-exp?:)
  (op^ left^ right^))
;;;op is one of '(mul div add sub mod lshift
;;;               rshift and xor or land lor)
```

Expressions in C/sexp represent values, either constant values, values stored as named variables, or computed values. Computed values may have some control effects, such as function call or simple branching, and may have other side effects, such as variable mutation. They are contained within statements either as predicates for control operations, or as values for function return, or a statement may consist entirely of a single expression. Expressions may contain subexpressions. Once again, C/sexp's expressions correspond exactly to C's expressions, and will not be covered in depth.

7.6.5 A generic function during parse-time

There is one static-semantic function that is used during parse time: `perform-decls:`. A statement may cause the parser to enter a namespace that is visible for the remainder of the current block. Therefore, parsing a block must consist of parsing a statement, using `peform-decls:` on that parsed statement to enter the correct namespace, and parsing the remainder of the block in this new namespace. For most statements, `perform-decls:` will be the identity function. For variable declarations, invocations of `perform-decls:` enter namespaces that includes the declared variable.

7.7 Static semantics

This section covers the static semantics of C/sexp in the abstract. A more in-depth explanation can be found in appendix A.

7.7.1 Types

The primary goal of the semantic-analysis phase of compilation is to associate types with expressions, and to determine that all type conditions for the language are satisfied. For this, we have a collection of classes encoding a semantic domain (*i.e.*, types), separate from the syntactic domain of types we've already encountered.

Types in C/sexp are associated with expressions, variables, and the subtypes of derived types.

7.7.2 Type categories

Following C99, all C/sexp types are in exactly one of three type categories: object types, function types and incomplete types. For each C/sexp type, exactly one of the following attributes should return a true value:

- `is-object-type?:` An object type indicates a program data object of known size.
- `is-function-type?:` A function type indicates that a variable or expression is associated with a function. Functions in C/sexp can not be passed as first class data; however, pointers to functions can be.
- `is-incomplete-type?:` An incomplete type indicates an object of a size that is not yet known at the program point it is encountered. For example, a variable can be declared to be of type of `struct foo` before `struct foo` is itself declared. At the point the variable is declared, `struct foo` is an incomplete type.

7.7.3 Predicates and conversions on types

Types in C/sexp are compared and converted with generic functions, based on the definitions in C99. New types must either provide methods for these generic functions, or delegate to a type that does, such as a base C/sexp type.

- `(compatible-type?:` t_1 t_2`)` The C99 standard defines two types to be *compatible* under certain conditions; usually, if they are the same type. There are some exceptions to this: for example, the types `(array` t`)` (indicating an array of type t and dynamic size) and `(*` t`)` (indicating a pointer to a value of type t) are compatible, but not the same.
- `(composite-type:` t_1 t_2`)` Under C99, a *composite* type can be constructed from two compatible types.
- `(type-accepts?:` t_1 t_2`)` This predicate returns true if, for all contexts where t_1 is allowable, the type t_2 is also allowable. Sometimes, this will result in an implicit conversion taking place.
- `(type-equal?:` t_1 t_2`)` This predicate returns true if the two types are the same.
- `is-signed-integer-type?: is-unsigned-integer-type?: is-integer-type?: is-real-floating-type?: is-complex-type?: is-floating-type?: is-arithmetic-type?: is-real-type?: is-character-type?: is-scalar-type?:` These predicates classify the object types of C/sexp, according to the categories of C99.

7.7.4 Type classes

We define a set of Ziggurat classes to represent C/sexp's types. These classes are instantiated during type analysis, and thus are separate from the syntax objects created during parsing.

- `basic-type%` This type class is used for the 18 non-`void` single-identifier types of the language. (*e.g.*`int`, `long-long-int`, `complex-double`, ...)
- `void-type%` The `void` type is an incomplete type that can never be completed.
- `function-type%` A function type is used to indicate that an expression has a function value.
- `enum-type%` In C/sexp, the enumeration type delegates to an integer type. The enumeration type is used merely to keep track of an enumeration tag.
- `pointer-type%` Pointers are references to values of other types. Pointers are always object types, even if the type of the referenced object is a function or incomplete.
- Aggregate types `incomplete-struct-type% struct-type% incomplete-union-type% union-type%` Structures are records with named fields, and unions are sum types with named options. Both aggregate types are incomplete until the end of a declaration that specifies all of their fields.
- `qualified-type%` A qualified type is a strict delegation object that associates a type qualifier, either `const`, `volatile` or `restrict` with another type. Strict delegation, as defined in section 5.2, allows us to delegate most semantic functions to the type being qualified.

```
d ∈ Declaration ::= ...
                  | (syntax z_static z_parse x ...)
x ∈ Syntax      ::= (expression i z)
                  | (declaration i z)
                  | (statement i z)
                  | (type i z)
                  | (syntax i z)
```

Figure 7.3: C/sexp syntax for defining macros. Extends figure 7.1.

7.7.5 Generic functions for typing

There are a few generic functions for typing syntax, which correspond to the non-terminals in the grammar.

- `(typecheck-program:` *p*`)` This generic function typechecks a program. It signals an error if any type errors are found in the program.
- `(typecheck-stm:` *s r t*`)` This typechecks a statement or declaration. The argument r indicates the return type of the enclosing function. The table t associates identifiers with their type: for example, identifiers bound to variables are mapped in the table to the type of that variable, identifiers bound to struct tags are mapped to that struct type, and so forth.
- `(exp-type:` *e t*`)` This returns the type of the expression e. The table t is as in `typecheck-stm:`, above.
- `(type-syntax-type:` *ty t*`)` This returns the semantic type of the type syntax ty. The table t is as above.
- `(perform-type-decls:` *s t*`)` Certain syntax nodes can alter the type table, such as variable declarations. This generic function returns the modified table.

7.8 Macros

C/sexp includes a declaration form for introducing macros, introduced in figure 7.3.

This is syntactically larger than the `let-syntax` form introduced in the Arith language, but the differences are not complex.

C/sexp `syntax` declarations can only be found at the program top-level. It would be possible to have a `let-syntax` form, as in Arith, or, indeed, Scheme, but this is not done, for two reasons:

- Top-level declarations fit the essentially declarative design of C and C/sexp programs. Other significant declarations (i.e. functions and types) can only occur at top level.

- Many C/sexp macros will want to introduce function and type declarations into the program top level. If syntax declarations are found also at top-level, it gives the macro writer a natural place to include such things.

Syntax definitions have at least three subforms.

- The first subform z_{static} is a Ziggurat expression that is evaluated when the syntax form is parsed. This is usually the place to include class definition, method definitions and the like. The Ziggurat definitions present in this subform are only visible in the subsequent subforms; consequently, a `syntax` declaration has associated with it a "global" scope. Ziggurat definitions present in one syntax definition are not visible in other definitions, which prevents macros interfering with each other.
- The second subform z_{parse} is a Ziggurat expression that must evaluate to a parse function. This parse function is invoked to provide the abstract syntax to be used in place of the syntax definition.
- Any subsequent subforms introduce new keywords to C/sexp. The form `(expression` *i z*`)` introduces the parser *z* for the keyword *i* when used as an expression. Likewise `declaration` introduces new declarative forms, `statement` introduces new statement keywords, `type` introduces new type keywords, and `syntax` introduces new syntax keywords (such as `expression`, `declaration`, `statement`, `type` and `syntax`). The `syntax` keyword is a useful tool in case a language designer adds a new syntactic context other than expressions, declarations, statements and types. This form allows further developers to introduce macros for this new syntactic context.

Some examples will illuminate this mechanism.

7.8.1 Include

The first example of a C/sexp macro is a feature that the C pre-processor includes as a basic form: syntax for including one file verbatim in another. To do so, we introduce a new syntax class, `include-csexp-decl%`.

The `include` declaration syntax object contains a list of declarations. Parsing the `include` form parses the file into this list; making sure that all of this parsing occurs in the same phase ensures that the compilation phases are kept in order.

This `include` form is based on a similar preprocessor directive present in C. It is a poor substitute for a true module system. One could use Ziggurat to implement such a module system, but this is outside of the scope of this work. Such an addition is interesting to consider, however; C lacks such modern facilities.

7.8.2 Cond

The next example is implementing a multi-armed if in C/sexp. We can define `cond` two ways: as an expression, and as a statement. We choose to do both. To see the difference, let's implement a simple comparison routine both ways:

```
(syntax
  (begin
    ;; an include class delegates to its parsed file
    (define-syntax-class include-csexp-decl% (file^ decls^)
      (lambda ()
        (object decl-block-csexp-decl%
                      source^ env^ decls^))))

  ;; no in-place expansion
  (lambda (env form)
    (parse-csexp-decl: env (syntax (block))))

  ;; parsing an include involves opening and parsing the file
  (declaration include
    (lambda (env form)
      (syntax-case form ()
        ((include afilename)
         (let ((filename (->datum: (syntax filename))))
           (if (string? filename)
               (let* ((prt (open-input-file-object filename))
                      (input (let recur ((dat (get-datum prt)))
                               (if (eof-object? dat)
                                   '()
                                   (cons dat
                                         (recur
                                          (get-datum prt)))))))
                 (object include-csexp-decl%
                         form env (syntax afilename)
                         (let recur ((input input) (env env))
                           (if (pair? input)
                               (let ((decl (parse-csexp-decl:
                                            env (car input))))
                                 (cons decl
                                   (recur (cdr input)
                                                (perform-decls:
                                                 decl env))))
                               '()))))
               (parse-error "bad input file name"
                            (syntax afilename)))))))))
```

Figure 7.4: C/sexp code (with embedded Ziggurat terms) defining the C/sexp `(include` *filename*`)` macro for including source material from another file.

```
(fun compare (* char) ((x int) (y int))
  (cond ((< x y) (return "less"))
        ((> x y) (return "more"))
        (else (return "equal"))))

(fun compare (* char) ((x int) (y int))
  (return (cond ((< x y) "less")
                ((> x y) "more")
                (else "equal"))))
```

We will implement this macro in two ways: first as a parser macro with no lazy delegation, and next as a macro with delegation. The approach of a parser macro is to rewrite the code immediately while parsing `cond` code. If one were to implement such a macro with the Scheme macro system, this is the approach one would take. This is perfectly feasible in Ziggurat for such a simple macro, but gives none of the benefits of Ziggurat, as we will see when we implement it as a delegation macro.

As a parser macro

To implement `cond` as a parser macro, the statement immediately expands into a tree of `if` statements, and the expression into a tree of conditional expressions. The implementation in figure 7.5.

This is more of a traditional Scheme macro in that expansion occurs during parsing. The advantage of lazy delegation is that it allows us to define our own static semantics. In this case, we might want to define specialized error messages.

As a delegation macro

We can implement the `cond` expression as a delegation macro. This implementation can be seen in figure 7.6. We only provide the implementation of `cond` as an expression, as the statement version is not significantly different from the expression version.

The benefit of this extra layer of indirection is that it allows us to define our own, specialized type-checking routines. In this case, we will want to simply have more precise error messages. More complex macros will cause us to write more complex semantics.

The implementation of the typing method for `cond` is in figure 7.7. This method proceeds very simply by (1) checking that all of the conditions are of scalar type, and then (2) checking that all of the actions have compatible type. The error message generated if condition (2) is unsatisfied will refer to the entire expression, as opposed to the generated `?:` expression.

7.8.3 List macro

C/sexp has only very primitive data-structure definitions that describe how the data is laid out in memory, and provides only monomorphic types. C/sexp's type system is not advanced enough to allow the programmer to specify type constructors; if we want

```
(syntax

 (begin) ; no lazy delegation means no classes or methods.

 ;; no static stuff either.
 (lambda (env form) (parse-csexp-decl: env (syntax (block))))

 (expression cond
   (lambda (env form)
     (syntax-case form (else)
       ((cond)
        (parse-error "cond exp with no else" form))
       ((cond (else x))
        (parse-csexp-exp: env (syntax x)))
       ((cond (cnd val) . more)
        (parse-csexp-exp: env
                          (syntax
                           (?: cnd val (cond . more))))))))

 (statement cond
   (lambda (env form)
     (syntax-case form (else)
      ((cond (else action))
       (parse-csexp-stm: env (syntax action)))
      ((cond (cnd action))
       (parse-csexp-stm: env (syntax (if cnd action))))
      ((cond (cnd action) . more)
       (parse-csexp-stm: env
                         (syntax
                          (if cnd action (cond . more))))))))
```

Figure 7.5: The implementation of multi-armed conditional as a parser macro.

```
(syntax
 (begin
   (syntax-class cond-csexp-exp% (clause-list^)
     (lambda ()
       (parse-csexp-expr: env^
         (let loop ((clause-list clause-list^))
           (if (pair? (cdr clause-list))
               (with-syntax ((condition
                              (object pre-parsed-datum%
                                      (caar clause-list)))
                             (first
                              (object pre-parsed-datum%
                                      (cdar clause-list)))
                             (rest
                              (object pre-parsed-datum%
                                      (loop (cdr clause-list)))))
                 (?: condition first rest))
                (object pre-parsed-datum%
                        (cadr clause-list))))))))

 (lambda (env form) (parse-csexp-decl: env (syntax (block))))

 (expression cond
   (lambda (env form)
     (syntax-case form (else)
       ((cond . pattern)
        (object csexp-cond-exp% form env
                (let recur ((pattern (syntax pattern)))
                  (syntax-case pattern (else)
                    (() (parse-error "cond exp with no else"
                                     form))
                    ((cond (else exp))
                     (parse-csexp-exp: env (syntax exp)))
                    ((cond (cnd action) . more)
                     (cons (cons (parse-csexp-exp:
                                  env (syntax cnd))
                                 (parse-csexp-exp:
                                  env (syntax action)))
                           (recur (syntax more)))))))))))

 (statement cond ...))
```

Figure 7.6: The implementation of multi-armed conditional as a delegation macro.

```
(define-method cond-csexp-exp% exp-type:
  (lambda (exp table)
    (let cnd-test-loop ((clauses clauses^))
      (if (pair? clauses)
          (begin
            (unless (is-scalar-type?: (exp-type: (caar clauses) table))
              (type-error "non-scalar type in cond expression"
                          (caar clauses)))
            (cnd-test-loop (cdr clauses)))))
    (let action-test-loop ((clauses clauses^)
                           (unified-type #f))
      (if (pair? clauses)
          (let ((first-type (expr-type: (cdar clauses) table)))
            (if unified-type
                (if (compatible-type?: unified-type first-type)
                    (action-test-loop (cdr clauses)
                                      (composite-type: unified-type
                                                       first-type))
                    (begin
                      (type-error "incompatible type in cond expression"
                                  form^)
                      (action-test-loop (cdr clauses) unified-type)))
                (action-test-loop (cdr clauses) first-type)))
          unified-type))))
```

Figure 7.7: It is possible to define a specialized typing method for `cond` forms.

them, we will need to introduce them as macros. For example, we might want to add parametric list types, similar to ones provided by Haskell or SML.

Let's introduce a new type constructor, `(list` *t*`)`, that describes linked lists, whose elements have type *t*. There are two primitive data constructors for building such lists: `(make-list` e_1 e_2`)` and `(null)` for the empty list, where e_1 has type t, and e_2 has type `(list` *t*`)`. We can distinguish empty from non-empty lists with the predicate `(null?` *e*`)`, and deconstruct non-empty lists with `(head` *e*`)` and `(tail` *e*`)`.

Note that `make-list`, `null?`, *etc.* are not functions, since C/sexp functions are monomorphic. They are syntactic keywords, which will have their own type-checking methods, distinct from the ones built-in to C/sexp that handle function calls.

Consider the example of a simple C/sexp function to add up the elements of an integer list, along with some global variable `master-list` of integer-list type:

```
(struct list ((car int) (cdr (* (struct list)))))

(fun cons (* (struct list)) ((car int) (cdr (* (struct list))))
  (var to-return (* (struct list))
                 (malloc (sizeof (* (struct list)))))
  (:= (-> to-return car) car)
  (:= (-> to-return cdr) cdr)
  (return to-return))

(var master-list (* (struct list))) ; Master list of ints.

(fun sum-elements int ((nums (* (struct list))))
  (var sum int 0)
  (while (! (== nums (cast (* void) 0)))
    (=+ sum (-> nums car))
    (:= nums (-> nums cdr)))
  (return sum))
```

Figure 7.8: The expanded code for the `sum-elements` example.

```
(var master-list (list int)) ; Master list of ints.

(fun sum-elements int ((nums (list int)))
  (var sum int 0)
  (while (! (null? nums))
    (=+ sum (head nums))
    (:= nums (tail nums)))
  (return sum))
```

Figure 7.8 shows the code into which this expands. For our one list type, the macro definition expands into a C/sexp definition for a type `(struct list)` which is the C/sexp actualization of the integer-list type. Additionally, the macro expands into a function `cons` (unused in this example), that constructs an object of type `(* (struct list))`. Once the macro definition has expanded thus, individual macros can be expanded thus:

macro	expanded form
`(list int)`	`(* (struct list))`
`(null?` e`)`	`(==` e `(cast (* void) 0))`
`(head` e`)`	`(->` e `car)`
`(tail` e`)`	`(->` e `cdr)`

The `syntax` definition for this macro consists of several parts. The skeleton of this definition is in figure 7.9. Parsing this definition returns an object of class `list-csexp-decl%`, that delegates into a list of other definitions. Particularly, for each list type used in the subsequent program, we will define:

```
(syntax
 ;; Here we have one-time Ziggurat code that will be visible
 ;; to multiple uses of the macros.
 (begin
   (define list-type-tag-alist '())

   ;; Syntax object for the list macro definition
   ...

   ;; Syntax objects for list operations
   ...

   ;; Type operations on list operations
   ...
   )

 ;; This code produces the C/sexp AST node for this entire
 ;; syntax-definition term.
 (lambda (env)
   (object list-csexp-decl%))

 ;; Definition of the (head exp) expression.
 (expression head
   (lambda (env form)
     (syntax-case form ()
       ((head exp)
        (object head-csexp-exp% form env
                        (parse-csexp-exp: env (syntax exp)))))))

 ;; Other macro definitions follow: tail, null?, null,
 ;; list and make-list
 ...)
```

Figure 7.9: A skeleton of the list macro definition.

```
;; Syntax object for the list macro definition
(define-syntax-class list-csexp-decl% ()
  (lambda ()
    (object decl-block-csexp-decl% source^ env^
      (map (lambda (x)
             (parse-csexp-decl: env^
                (syntax
                  (block
                    ;; the structure that holds the list data.
                    (struct ,(cadr x)
                            ((car ,(type->syntax: (car x)))
                             (cdr (* (struct ,(cadr x))))))
                    ;; the function to construct the list.
                    (fun ,(caddr x) (* (struct ,(cadr x)))
                      ((car ,(type->syntax: (car x)))
                       (cdr ,(* (struct ,(cadr x)))))
                      (var to-return (* (struct ,(cadr x)))
                           (malloc (sizeof
                                    (* (struct ,(cadr x))))))
                      (:= (-> to-return car) car)
                      (:= (-> to-return cdr) cdr)
                      (return to-return))))))
           list-type-tag-alist))))
.
```

Figure 7.10: The definition of the syntax class for the list macro declaration.

- A `struct` type declaration, which is the rewritten form of the list type.
- A function declaration that will be invoked in the rewritten form of the `make-list` expression.

The definition of this syntax class can be found in figure 7.10. It relies on a variable `list-type-tag-alist`, defined in the static section of the macro, that contains a list with three items for each list type in the subsequent program:

- A type that is used in the program as the parameter to a list type. For example, for the type `(list int)`, this entry would be equivalent to `(object basic-type% 'int)`.
- A generated identifier to be used as the tag of the generated structure.
- A generated identifier to be used as the name of the list constructor function.

Next, we define classes for the new syntax in figure 7.11. The transformations mentioned earlier for `head`, `tail`, `null?`, `null` and `list` head straightforward syntax expansions in the delegation instantiation methods of these classes. A `make-list`

expression translates into a function call to the type-directed function in the declaration block, which has a name found in the Ziggurat variable `list-type-tag-alist`. This list-constructor function is very short, non-recursive and non-iterative, and thus is likely inlined.

The delegation method for `cons-csexp-exp%` relies on the variable `list-type-tag-alist` to contain the name of the function it delegates to. However, there are two problems with this:

- When the macro is defined, `list-type-tag-alist` is empty.

- Even if the list were populated, the delegate instantiation method would need to know the type of its arguments in order to look up the name.

Both of these problems are solved via phase ordering. The `cons-csexp-exp%` type defines a `type^` field, that, upon parsing, is initialized to `#f`. At type-analysis time, this field is set to the appropriate type. This means that the delegation method cannot be invoked before type expansion, but this is doable by writing methods for all of the generic functions that will be invoked before the type-analysis phase. Fortunately, for C/sexp expressions, there are none. Also during type-analysis time, the various list operations populate the variable `list-type-tag-alist` appropriately. This is done via the functions `get-list-type-tag` and `get-list-cons`, which return, respectively, generated type tags or constructor functions for the list types that they are passed.

```
(define (get-list-type-tag type)
  (let loop ((alist list-type-tag-alist))
    (if (pair? alist)
        (if (type-equal?: type (caar alist))
            (cadar alist)
            (loop (cdr alist)))
        (let ((new-tag (genid (z-syntax tag)))
              (new-cons (genid (z-syntax cons))))
          (set! list-type-tag-alist
                `((,type ,new-tag ,new-cons) .
                  ,list-type-tag-alist))
          new-tag))))
```

```
;; Syntax objects for list operations

(define-syntax-class head-csexp-exp% (list^)
  (lambda ()
    (parse-csexp-exp: env^
      (with-syntax ((list list^))
        (-> list car)))))

(define-syntax-class tail-csexp-exp% (list^)
  (lambda ()
    (parse-csexp-exp: env^
      (with-syntax ((list list^))
        (-> list cdr)))))

(define-syntax-class null?-csexp-exp% (list^)
  (lambda ()
    (parse-csexp-exp: env^
      (with-syntax ((list list^))
        (== list (cast (* void) 0))))))

(define-syntax-class cons-csexp-exp% (car^ cdr^ type^)
  (lambda ()
    (parse-csexp-exp: env^
      (with-syntax ((car car^)
                    (cdr cdr^)
                    (cons-fun (if (is-list-type?: type^)
                                  (get-list-cons
                                   (list-type-type: type^))
                                  (syntax-error))))
        (cons-fun car cdr)))))

(define-syntax-class null-csexp-exp% ()
  (lambda ()
    (parse-csexp-exp: env^ (syntax (cast (* void) 0)))))

(define-syntax-class list-csexp-type% (type^ tag^)
  (lambda ()
    (parse-csexp-type: env^ (syntax (* (struct ,tag^))))))
```

Figure 7.11: The syntax classes for the list macro definition.

```
(define (get-list-cons type)
  (let loop ((alist list-type-tag-alist))
    (if (pair? alist)
        (if (type-equal?: type (caar alist))
            (caddar alist)
            (loop (cdr alist)))
        (let ((new-tag (genid (z-syntax tag)))
              (new-cons (genid (z-syntax cons))))
          (set! list-type-tag-alist
                `((,type ,new-tag ,new-cons) .
                  ,list-type-tag-alist))
          new-cons))))
```

We are introducing new types into C/sexp, and thus, we must have new Ziggurat classes representing those types. These are separate from the C/sexp syntax `(list` e`)`, which is used to represent the list type in C/sexp programs. We have two Ziggurat classes, for list types and null types. List types are parameterized by another type, while the null type is polymorphic: it is the type of the null list, which unifies with any other list type.

The implementation of the null-list type is in figure 7.12. One possible implementation of this type is to have it delegate to the C/sexp void pointer type `(* void)`. However, a side effect of this would be that C/sexp expressions that expect a pointer would accept a null list, and vice-versa. This is undesirable, so instead, we do not have the null-list type delegate to any other type.

As a consequence of this, we need to define methods for all of the generic functions defined for C/sexp types. These methods establish the null-list type as an incomplete type that is compatible with any list type. The composite type of the null-list type and another list type is the other type; thus, `(null)` and `(list int)` unify to `(list int)`.

The implementation of the list-type class is in figure 7.13. One possible implementation of this type is to have it delegate to the equivalent structure type. However, this would result in the same problems as implementing the delegate of the null-list type as `(* void)`, so we define the list-type class without a delegate. Once again, we must define methods for all of the generic functions. These define a list type to be an object type, that is compatible with list-types with compatible arguments, or the null-list type.

The type of the `head` of an expression of type `(list` α`)` is α. The implementation of this is in the method for `exp-type:` at `head-csexp-exp%`, shown in figure 7.14. The typing most of the rest of the expressions follow directly and are not shown.

The type of a `make-list` expression is more complicated, and is found is figure 7.15. The type of a `make-list` expressions depends on the types of its arguments. If the type of the second argument is a null type, and the type of the first argument is α, then the type of the `make-list` is `(list` α`)`. Otherwise, the typing function checks the compatibility of the first and second arguments, and the type of the new list is the type of the tail. As a side-effect of calling `exp-type:` on a `make-list` expression, two things happen:

- The type of the list is stored in the field `type^`.

```
;; Type operations on list operations
(define-class (null-type% is-null-type?:) ())
(define-class (list-type% is-list-type?:)
              ((type^ list-type-type:)))

;; methods for null type.
(define-method null-type% incomplete-type?:
  (lambda (x) #t))

(define-method null-type% ->string:
  (lambda (x) "null"))

(define-method null-type% type-accepts?:
  (lambda (expected got)
    (is-null-type?: got)))

(define-method null-type% compatible-type?:
  (lambda (first second)
    (or (is-list-type?: second) (is-null-type?: second))))

(define-method null-type% composite-type:
  (lambda (first second) second))

(define-method null-type% type-equal?:
  (lambda (expected got)
    (is-null-type?: got)))

(define-method null-type% type->syntax:
  (lambda (type)
    (type->syntax: (object pointer-type%
                           (object basic-type% 'void)))))
```

Figure 7.12: The typing methods for the null-list type.

```
;; methods for the list type.
(define-method list-type% object-type?:
  (lambda (x) #t))

(define-method list-type% ->string:
  (lambda (x) (string-append (->string: type^) " list")))

(define-method list-type% type-accepts?:
  (lambda (expected got)
    (or (is-null-type?: got)
        (and (is-list-type?: got)
             (type-accepts?: type^
                             (list-type-type: got))))))

(define-method list-type% compatible-type?:
  (lambda (first second)
    (or (and (is-list-type?: second)
             (compatible-type?:
              type^ (list-type-type: second)))
        (is-null-type?: second))))

(define-method list-type% composite-type:
  (lambda (first second)
    (cond ((list-type?: second)
           (object list-type%
                   (composite-type: type^
                                    (list-type-type: second))))
          ((null-type?: second) first)
          (else second))))

(define-method list-type% type-equal?:
  (lambda (expected got)
    (and (is-list-type?: got)
         (type-equal?: type^ (list-type-type: got)))))

(define-method list-type% type->syntax:
  (lambda (type)
    (object list-csexp-type% (syntax list) #f
                  type (get-list-type-tag type^))))
```

Figure 7.13: Typing methods for the list type.

```
(define-method head-csexp-exp% exp-type:
  (lambda (exp table)
    (let ((list-type (exp-type: list^ table)))
      (if (is-list-type?: list-type)
          (list-type-type: list-type)
          (begin
            (type-error (string-append
                         "attempt to take head of "
                         (->string: list-type))
                        exp)
            reference-int-type)))))
```

Figure 7.14: A method to calculate the type of a `head` expression.

- An entry is entered in `list-type-tag-alist` for the type of the list, if it is not already present. This is done as a side-effect of `get-list-type-tag`.

This is to enable the delegation methods presented earlier. It is essential, then, that type syntax objects not delegate until after the typing phase of compilation. Fortunately, this is doable if no new generic functions are defined on C/sexp.

7.9 From language extensions to language embeddings

The last macro we introduced for C/sexp, the list macro, provides new methods for the static semantics of the base language. Lists have their own type system: a very limited polymorphism, much less powerful than the `NULL`-based polymorphism available in C/sexp, but much more apt to find syntax misuse. Now that we have established how to link static semantics, we have broken a major barrier to embedding languages: reconciling radically different notions of static semantics. Additionally, we can link static semantics much more complicated than the type system. In the next few chapters, we'll see the kind of more ambitious embedding enabled by this technique.

```
(define-method cons-csexp-exp% exp-type:
  (lambda (exp table)
    (let ((car-type (exp-type: car^ table))
          (cdr-type (exp-type: cdr^ table)))
      (cond
        ((is-null-type?: cdr-type)
         (let ((list-type (object list-type% car-type)))
           (get-list-type-tag car-type)
           (set! type^ list-type)
           list-type))
        ((is-list-type?: cdr-type)
         (if (type-accepts?: (list-type-type: cdr-type)
                             car-type)
             (begin
               (set! type^ cdr-type)
               cdr-type)
             (begin
               (type-error (string-append
                            "incompatible list type "
                            (->string: car-type)) exp)
               (set! type^ cdr-type)
               cdr-type)))
        (else
         (let ((exp-type (object list-type% car-type)))
           (set! type^ exp-type)
           (type-error
            "attempt to make a list with a non-list tail"
            exp)
           exp-type))))))
```

Figure 7.15: A method to calculate the type of a list constructor expression.

Chapter 8

Recap

While Arith is designed to be as simple as possible so as to demonstrate the basic architecture of Ziggurat, C/sexp is designed to be useful, and thus is more complex. However, the language is structured using the same elements. It is worthwhile at this point to go back and recap those elements, so as to demonstrate that the elements that allow language extension in Arith allow the same facility to C/sexp.

8.1 Language structure

C/sexp's design, although much more complex than Arith's, consists of the same three basic parts.

- The concrete syntax of C/sexp uses all of the facilities of Ziggurat parsing. It is s-expression-based, but is still quite complex: it has a number of non-terminals (declarations, statements and so forth), and a number of namespaces (for types, variables, expression keywords, statement keywords and so forth). In addition, there is syntax for defining new syntax. The `syntax` keyword in the top-level declaration namespace allows the programmer to add productions to any of these non-terminals, and keywords to any of the namespaces.

- The abstract syntax of C/sexp consists of a set of classes that, while larger than in Arith, is really not any more complex. In addition, as part of the type system, C/sexp has a number of classes without corresponding concrete syntax, consisting of `void-type%`, `function-type%` and so forth.

- The static semantics of C/sexp consists of a number of generic functions, ordered into four phases.

  - Parsing: during the parsing phase and thereafter, abstract syntax may have simple accessor and predicate generic functions called upon them. For example, the generic function to test whether an expression is a C/sexp variable expression, `is-var-csexp-decl?:`, may be called during this phase.

In addition, the static environment-building method, `perform-decls:`, may be called during this phase.

- Static analysis: during this phase, type analysis and semantic checking take place. Consequently, during this phase and thereafter, type-checking methods such as `typecheck-stm:` may be called during this phase.
- Code-generation: Although we do not focus on code generation in this dissertation, there is a generic function, `generate-c-code:` that simply translates the abstract syntax of a C/sexp program to a concrete C program. Since the abstract syntax of C/sexp is based on the concrete syntax of C, this function is trivial.
- Additional static analysis: In later chapters, we will perform control- and data-flow based debugging analysis. We have not yet specified this, but the generic functions that constitute this analysis will occur during this phase.

The more complex structure of C/sexp over Arith allows for more complex extension.

8.2 Language extension

If we recall the definitions of language extension presented in section 2.7, we have seen a couple of examples of extension via macro.

- The *cond* macro: This is an example of a simpler macro that does not add to the static semantics of the underlying language at all.
- The list macros, including *cons*, *head*, *tail*, etc.: These macros are a bit more complex. They are still specified via rewriting, but have associated static semantics. The typing generic functions present in C/sexp have methods at these new pieces of syntax. Because the language extension uses the same generic functions as the underlying language, the extended static semantics work seamlessly with other language extensions, including other instantiations of the same set of macros.

In coming chapters, we will see examples of language embedding and language modification. Language embedding is a special case of language extension, where a portion of the program written in a new language is entirely rewritten into the base language, in this case C/sexp.

- In the next chapter, we will see a language for specifying parsers, similar to Yacc or Bison. Since we embed the parser language within C/sexp, parsers can exist alongside other language extensions. This language has its own static semantics, but in chapter 10, we will see how it provides methods for base C/sexp static-semantic generic functions. The use of generic functions ensures that even if other macros in the same program contain semantic extensions, the composite language works as expected.

- In chapter 11, we will introduce a domain-specific language, called Topsl, for specifying web-based surveys, embedded within C/sexp. Similar to the parser language, this language has its own static semantics, but because of the use of generic functions, the composite language works as expected even in the presence of other language extensions.

Topsl also presents an example of language modification. Modification is merely taking a language and adding a new piece of syntax that does not rewrite into the underlying language, or a new static-semantic generic function.

- Topsl has a piece of C/sexp embedded within it. However, this version of C/sexp has a new generic function, used to determine the survey questions present in the embedded code.

These more complex examples of language extension in the coming chapters are, thus, merely specialized forms of the extensions we have already seen.

Chapter 9

LALR(1) parser languages

We have seen that C/sexp's Ziggurat-based macro system handles small language extensions, such as multi-armed if statements and polymorphic list types, quite well. However, if we truly wish to test this facility, we need to implement a radically different language within C/sexp. A good choice for such a language to implement via Ziggurat is a context-free parser generator language, similar to Yacc or Bison, for two reasons. Firstly, such jobs are commonly done via a domain-specific language, thus showing that Ziggurat is useful for typical metaprogramming tasks. Secondly, a parser has specialized static semantics which we can implement in Ziggurat, thus showing the advantage of this approach. Since context-free grammars are computationally much simpler than Turing machines, for example, it is possible to design a dependency-analysis algorithm that gives relatively precise results. Since the CFG language is emdedded within a general-purpose language, dependency analysis of programs that use both languages can be made more precise.

It is worth noting here that the parsing in this section has no connection to the parsing in chapter 5. In that chapter, we were defining parsers for Ziggurat-based languages with s-expression-based concrete syntax. This section defines languages for parsing arbitrary languages with any LALR(1) grammar. Chapter 5 presents a fundamental piece of Ziggurat, whereas the languages presented in this section are example uses of Ziggurat.

This parser generator uses two languages: a high-level context-free grammar language, called Wisent, and a lower-level push-down automaton language, called Aurochs.[1] The high-level Wisent delegates to the low-level Aurochs, but Aurochs is designed to be independent of any target language; thus, in order to encode semantic actions, both the high-level and low-level languages are parameterized by elements of another language. How this is done is covered in section 9.1.3.

These two languages form a language tower. In general with language towers, as we go from high-level to low-level, we go from languages with more static guarantees inherent in the language to ones with more computational power but fewer guarantees. Later in this chapter and the next, we will see how these guarantees on Wisent code

[1] The wisent (pronounce "veesahnt") is best described as a European bison, while the aurochs is an extinct species that is an ancestor of modern cattle.

allow for better analysis of the translated C/sexp code. We will see how semantic information can be derived at the high level of Wisent code, and be applied at the relatively low level of C/sexp code.

9.1 Wisent

Wisent is designed to be straightforward for anyone who has used a parser-generator tool, such as Yacc or Happy. For example, a parser for a simple expression language might be written

```
(parser pmain token get-token
  (tokens ; defines the terminals
    (left MULT DIV) ; left-assoc, equal precedence
    (left PLUS MINUS) ; left-assoc, lower prec than DIV
    (type int NUM) ; carries value of type int
    (eos EOS)) ; end-of-stream token
  ;; This defines a non-terminal of the language: in this
  ;; instance, all semantic actions will print the syntax,
  ;; and calculate the indicated value.
  (non-term EXP int
    (=> (NUM)
        (block (printf "NUM %d\n" ($ 1))
               (return ($ 1))))
    (=> (EXP MULT EXP)
        (block (printf "MULT %d %d\n" ($ 1) ($ 3))
               (return (* ($ 1) ($ 3)))))
    (=> (EXP DIV EXP)
        (block (printf "DIV %d %d\n" ($ 1) ($ 3))
               (return (/ ($ 1) ($ 3)))))
    (=> (EXP PLUS EXP)
        (block (printf "PLUS %d %d\n" ($ 1) ($ 3))
               (return (+ ($ 1) ($ 3)))))
    (=> (EXP MINUS EXP)
        (block (printf "MINUS %d %d\n" ($ 1) ($ 3))
               (return (- ($ 1) ($ 3))))))
  (non-term PROGRAM int ; program = single expression
    (=> (EXP) (return ($ 1))))
  (start PROGRAM)) ; start token is PROGRAM
```

The keyword `parser` is a C/sexp macro that will expand into the definition of a new C/sexp type `token`, a number of constructors `MULT`, `DIV` and so forth for that type, a function prototype `get-token` to be defined by the programmer that should produce objects of type `token`, and a function `pmain` that takes the output of `get-token` and parses it. We will get into some of the inner workings of this macro in section 9.2.2.

9.1.1 Concrete syntax

$r \in$ Parser ::= d ...
$d \in$ Declaration ::= (tokens e ...)
| (non-term i t p ...)
| (start i)
| (block d ...)
| (include c)
$e \in$ Token ::= i
| (type i t)
| (left e ...)
| (right e ...)
| (error i)
| (eos i)
$p \in$ Production ::= (=> (i ...) sa)
| (=> (i ...) i sa)
$c \in$ Constant
$t \in$ Type
$sa \in$ Action

The syntax of Wisent is parameterized by two embedded-language forms: $t \in$ Type and $sa \in$ Action. In the C/sexp implementation of the language, these are C/sexp syntactic types and statements, respectively.

9.1.2 Abstract syntax

The abstract syntax of Wisent consists, as usual, of a set of class declarations. In this section, these class declarations will be presented along with a description of the meaning of the class. To recall chapter 3 and section 5.4, the basic structure of a syntax-class declaration used in this section is

```
(define-syntax-class (class-name class-predicate)
  ((field-name field-accessor) ...)
  delegate-instantiation-method)
```

Using this, we can define the set of class declarations that make up the abstract syntax of Wisent.

Parser

```
(define-syntax-class (wisent-parser% is-wisent-parser?:)
  ((decls^ wisent-parser-decls:))
    (lambda ()
      (transform-parser decls^)))
```

A Wisent parser is specified by a sequence of declarations. The order of declarations determines the precedence of tokens, but has no effect on identifier scope. The scope of all Wisent identifiers is the entire grammar being defined; it is an error to declare two terminals or non-terminals with the same identifier.

A parser object has an equivalent Aurochs PDA, specified later in this chapter. The Ziggurat function `transform-parser` transforms the parser into its PDA form. The definition of `transform-parser` is as an ordinary non-generic function. This transformation is not unusual; it is merely the standard LALR CFG-to-PDA transformation [1], implemented as a macro, in the vein of Owens et al. [75] Again, what is novel in the Ziggurat approach is not the particular dynamic semantics of Wisent, as seen in the LALR translation code, but rather the fact that we can link static semantics across the language tower, as we shall see.

Token declarations

```
(define-syntax-class (wisent-decl-tokens%
                      is-wisent-decl-tokens?:)
  ((tokens^ wisent-decl-tokens-tokens:)))
```

A token declaration declares a number of non-terminals. The order of these non-terminals determines their precedence: if token A is declared before token B, token A has higher precedence than token B. Multiple token declarations can occur in the same parser; in this case, tokens in the first declaration have higher precedence than those in the second declaration.

Non-terminal declarations

```
(define-syntax-class (wisent-decl-non-term%
                      is-wisent-decl-non-term?:)
  ((name^ wisent-decl-non-term-name:)
   (type^ wisent-decl-non-term-type:)
   (productions^ wisent-decl-non-term-productions:)))
```

Non-terminals in the Wisent are declared with the `non-term` keyword. This declares a non-terminal of name `name^`, with type `type^`, and productions in the list `productions^`. It is up to the embedding language to determine how to parse types; how to specify this is shown below. Like terminals, non-terminals do not have a namespace, and are likewise unadorned identifiers.

Start-token declarations

```
(define-syntax-class (wisent-decl-start%
                      is-wisent-decl-start?:)
  ((name^ wisent-decl-start-name:)))
```

The `start` keyword declares a non-terminal to be the start non-terminal of the language. The static semantics of Wisent places restrictions on this non-terminal, as we shall see in the next section.

Declaration blocks

```
(define-syntax-class (wisent-decl-block%
                      is-wisent-decl-block?:)
  ((decls^ wisent-decl-block-decls:)))
```

The `block` keyword is used to put a sequence of declarations where a single declaration is usually found. Again, a declaration block does not interact with any namespace, since Wisent does not have namespaces.

Include declarations

```
(define-syntax-class (wisent-decl-include%
                      is-wisent-decl-include?:)
  ((filename^ wisent-decl-include-filename:)
   (decls^ wisent-decl-include-decls:))
  (lambda ()
    (cond
      ((not (pair?: decls^))
       (object wisent-decl-block% source^ env^ '()))
      ((not (pair?: (cdr: decls^)))
       (car: decls^))
      (else
       (object wisent-decl-block% source^ env^ decls^)))))
```

Similar to the `include` declaration in C/sexp, `include` includes the declarations found in another file. Parsing an `include` has the effect of loading the referenced file, and parsing the declarations into the field `decls^`. The `wisent-decl-include%` object delegates to these declarations.

Valued tokens

```
(define-syntax-class (wisent-token-valued%
                      is-wisent-token-valued?:)
  ((id^ wisent-token-valued-id:)
   (type^ wisent-token-valued-type:)))
```

A valued token is introduced by the `type` keyword. A valued token is used for tokens with an associated value, such as identifiers and numerical constants. A valued token has an associated type, which is inherited from the languages the parser is embedded in.

Unvalued tokens

```
(define-syntax-class (wisent-token-unvalued%
                      is-wisent-token-unvalued?:)
  ((id^ wisent-token-unvalued-id:)))
```

An unvalued token is introduced in Wisent by an unadorned identifier. An unvalued token has no associated semantic value, and is useful for keywords, whitespace and the like. In some languages Wisent can be embedded within, such as SML, the unvalued token could be implemented as a valued token with unit type. However, not all languages implement a unit type; for example, C/sexp does not,[2] and we still want to have the semantic guarantees that an unvalued token would give us. For example, we want it to be a compile-time error to attempt to take the value of an unvalued token.

Token precedence

```
(define-syntax-class (wisent-token-left%
                      is-wisent-token-left?:)
  ((tokens^ wisent-token-left-tokens:)))

(define-syntax-class (wisent-token-right%
                      is-wisent-token-right?:)
  ((tokens^ wisent-token-right-tokens:)))
```

The `left` and `right` keywords declare a number of tokens to be left- and right-associative, respectively. It is an error for one precedence declaration to be within another. This is checked during well-formedness-checking time; this design decision was made to keep the grammar relatively simple, though it would be possible to implement this restriction directly in the grammar.

Special tokens

```
(define-syntax-class (wisent-token-error%
                      is-wisent-token-error?:)
  ((id^ wisent-token-error-id:)))

(define-syntax-class (wisent-token-eos%
                      is-wisent-token-eos?:)
  ((id^ wisent-token-eos-id:)))
```

The `error` and `eos` tokens declare two special non-terminals, the error token and end-of-stream token respectively. These tokens are unvalued, and as with other tokens, may not be re-declared.

[2] C/sexp does have a void type. However, void is not a type in the same sense that int is a type; a programmer cannot declare a variable to be of void type, for example.

Productions

High-level Wisent only provides one sort of production, when not extended by macros.

```
(define-syntax-class (wisent-production%
                      is-wisent-production?:)
  ((rhs^ wisent-production-rhs:)
   (action^ wisent-production-action:)
   (precedence^ wisent-production-precedence:)))
```

A production is found inside a non-terminal declaration. It specifies the sequence of identifiers on the right-hand side of the non-terminal; a non-terminal can contain several productions, indicating that all of the productions have the same left-hand side. All productions contain a semantic action, parameterized by the embedding language, similar to the type syntax in a non-terminal. Productions also optionally have a `precedence^` field, which should be an identifier referring to a terminal, which indicates the precedence of the production. For example, the production `(=> (EXP DIV EXP) MULT ...)` would have the same precedence as a `DIV` terminal.

Parameterized syntax: type syntax and semantic actions

The above abstract syntax has fields for type syntax and semantic actions, which are left out of the language proper, and must be specified when the parser language is embedded within another. For example, in the C/sexp embedding, a Wisent type is merely a C/sexp type, and a semantic action is a C/sexp statement. We define a new piece of abstract syntax, `csexp-parser-action%`, that represents the semantic action of a non-terminal; this new syntax is a wrapper for a C/sexp statement.

```
(define-syntax-class (csexp-parser-action%
                      is-csexp-parser-action?:)
  ((stm^ csexp-parser-action-stm:)
   (identifier-mapping^
    csexp-parser-action-identifier-mapping:))
  (lambda () stm^))
```

A semantic action can have `($` n`)` expressions, which refer to the semantic values of terminals and non-terminals on the left-hand-side of the production. These new expressions are not C/sexp statements. By adding these, we are extending the language; thus, we must do all of the things one must do to extend a language, including writing new typing functions.

During LALR translation, these expressions get translated to C/sexp variables. Therefore, the semantic action object has a mapping from integers to identifiers, called `identifier-mapping`. These identifiers associated with integer n resolves to the C/sexp variable which holds semantic value n on the left-hand-side of the production.

As with the language towers we've seen before, to add new parse functions, we must enter a new syntactic environment. The generic parse functions for parsing types

and semantic actions are `parse-wisent-type:` and `parse-wisent-action:`, and they are defined for objects of class `wisent-compatibility-env%`.

```
(define-class wisent-compatibility-env%
  (action-parser^ type-parser^ super^)
  (lambda () super^))

(define-method wisent-compatibility-env%
               parse-wisent-action: action-parser^)

(define-method wisent-compatibility-env% parse-wisent-type:
  type-parser^)
```

This compatibility environment encapsulates the language parameterization. In the case of C/sexp, `type-parser:` will parse a C/sexp type, and `action-parser:` will parse a C/sexp statement. Other languages will have other parse functions returned by this method.

We define a new function, (`enter-wisent-env` *env parse-action parse-type*), that enters a syntactic environment for the Wisent language. When language implementors wish to embed Wisent in an underlying language, thy call this function with three arguments: (1) the enclosing environment, (2) a parse function for a semantic action, and (3) a parse function for an underlying type. For embedding Wisent parsers within C/sexp programs, for example, in the static part of the C/sexp macro definition this function is invoked:

```
(enter-wisent-env env parse-wisent-action parse-csexp-type)
```

The function *parse-wisent-action* requires a little explanation. Semantic actions are sections of code that calculate the semantic values of productions. For example, for the Arith parser above, the semantic action of an expression calculates the numerical value of that expression. When Wisent is embedded in C/sexp, a semantic action is a statement that returns its semantic value via a `return` statement. Most semantic actions will need to use the semantic values of other productions; for example, the semantic action for e_1 `+` e_2 needs to use the semantic values of e_1 and e_2. So, we will need to slightly extend the C/sexp language such that, when parsing a semantic action, we will add a new expression that will access the semantic values of subforms.

```
(define (parse-wisent-action env form)
  (let ((value-table (make-hash-table)))
    (let ((new-env (enter-csexp-exp-env env)))
      (add-keyword-parser!: new-env (syntax $)
                            (parse-parser-semantic-value
                             value-table))
        (object csexp-parser-action% form env
                (parse-csexp-stm: new-env form)
                value-table))))
```

The new expression keyword, $, introduces a new kind of C/sexp expression, a semantic value. This new expression is actually a macro, that expands into a C/sexp variable, such that ($ n) always expands to the same variable in a single semantic action.[3]

```
(define-syntax-class (csexp-parser-semantic-value%
                      is-csexp-parser-semantic-value?:)
  ((num^ csexp-parser-semantic-value-num:)
   (id^ csexp-parser-semantic-value-id:))
  (lambda () (object var-csexp-exp% source^ env^ id^)))

(define (parse-parser-semantic-value tbl)
  (lambda (env form)
    (syntax-case form ()
      (($ ndat)
       (let ((n (->datum: (syntax ndat))))
         (if (and (integer? n) (> n 0))
             (object csexp-parser-semantic-value% form env
                     n (value-table-get tbl n))
             (parse-error "expected a number"
                          (syntax ndat))))))))
```

By defining a new class for semantic actions, we have the ability to link C/sexp type-checking and Wisent well-formedness checking, as we shall see.

9.1.3 Static semantics

The static semantics of Wisent is concerned with well-formedness checking: making sure that common-sense structural properties apply to a parser. We define a number of generic functions

```
(define-generic (parser-well-formed?: parser)
  (lambda (parser) #t))


(define-generic (decl-well-formed?: decl
                                    error-defined?
                                    eos-defined?
                                    start-defined?
                                    table)
  (lambda (decl error-defined? eos-defined? table) #t))
```

[3]This particular style of specifying semantic values was chosen for its familiarity to anyone who has programmed in Yacc or Bison. Other language embeddings might use different means to specify semantic values; for example, a functional language may require that a semantic action be a function that takes the semantic values as arguments. Ziggurat is flexible enough to allow these alternate parameterizations; the design provided here is merely one point in a design space.

```
(define-generic (production-well-formed?:
                 prod non-term table)
  (lambda (prod non-term table) #t))


(define-generic (token-well-formed?: token
                                     error-defined?
                                     eos-defined?
                                     associativity-okay?
                                     table)
  (lambda (token error-defined? eos-defined?
           associativity-okay? table) #t))
```

The methods of these generic functions verify mundane properties such as

- That there is only one error token
- That there is at most one start token, and that it identifies a non-terminal
- That no terminal is defined twice
- That no two terminals or non-terminals use the same identifier

The flags `error-defined?` and so forth indicate whether certain declarations, such as the error token, have already been seen, since it is an error to declare them more than once.

The well-formedness function for parser actions

```
(define-generic (parser-action-well-formed?:
                 action non-term token-map)
  (lambda (action non-term tokens) #t))
```

has no methods in Wisent, since semantic actions are a facet of the language embedding. It is up to the language embedder to define this method. For example, in the C/sexp embedding, this method type-checks the enclosed statement.

```
(define-method csexp-parser-action%
               parser-action-well-formed?:
  (lambda (action non-term tokens)
    (let* ((table (get-current-type-table))
           (new-table
            (object semantic-value-table% tokens table))
           (return-type
            (type-syntax-type:
             (parser-decl-non-term-type: non-term)
             table)))
      (typecheck-stm: stm^ return-type new-table))))
```

The class `semantic-value-table%` defines strict delegation objects that delegate to a type environment. It contains information about the semantic values of the right-hand side of the production. Objects of class `semantic-value-table%` define methods for the generic function `semantic-value-get:`, which maps integers with their appropriate terminal or non-terminal, or the special symbols `no-action`, `too-high` or `too-low` in case of an improper mapping. Using this generic function, it is now possible to define a specialized `exp-type:` method for semantic actions, as shown in figure 9.1.

Since we have a well-formedness method that does C/sexp type-checking, and that type-checking may additionally cause some parser well-formedness checking, we have linked the static semantics of the two languages. However, since the Wisent embedding in C/sexp is a macro, it would be possible not to have a specialized `exp-type:` method for semantic-value expressions, and instead have the generic function handled by rewriting. The reason that this better done this way is that, as we shall see in section 9.2.4, the rewriting results in a lot of new code, and thus the error messages would be difficult to understand.

The rewriting of Wisent into C/sexp is accomplished in two steps. The first step is rewriting the parser into the Aurochs push-down language.

9.2 Aurochs

Aurochs is a language for specifying push-down automata. A programmer could write a parser as a push-down automaton in Aurochs, although this is not recommended. Aurochs is intended to be used as an intermediate language between the parser description and its underlying implementation, specifying the equivalent push-down automaton for a Wisent parser. Thus, the delegate of a Wisent parser is always an equivalent Aurochs parser. The delegate of that Aurochs parser will be a parser in the underlying language, in this case C/sexp.

Like Wisent, Aurochs is a parameterized language, with semantic actions in the underlying language. When a Wisent parser delegates to an Aurochs parser, the semantic actions of the CFG parser are used directly as the semantic actions of the PDA parser. Aurochs has some static checks that can be checked, although these properties are not as strict as can be checked at the CFG level. Since the semantic actions of the PDA are the same as those of the CFG, the checking of the PDA and the underlying language are linked in the same was as in Wisent.

9.2.1 The language

A parser in the PDA language describes the push-down automaton that carries out the parse. This language implementation provides no innovations on the LR parse algorithm, and thus this discussion of the language will be brief.

```
(define-method csexp-parser-semantic-value% exp-type:
  (lambda (exp table)
    (let ((tok (semantic-value-get: table (- num^ 1)
                                    (lambda (x) x))))
      (cond
        ((eq? tok 'no-action) (pass))
        ((or (eq? tok 'too-high) (eq? tok 'too-low))
         (well-formedness-error
          "semantic value out of range"
          exp)
         reference-int-type)
        ((is-parser-decl-non-term?: tok)
         (type-syntax-type:
          (parser-decl-non-term-type: tok) table))
        ((is-parser-token-valued?: tok)
         (type-syntax-type:
          (parser-token-valued-type: tok) table))
        ((is-parser-token-unvalued?: tok)
         (well-formedness-error
          "attempt to use semantic value of unvalued token"
          exp)
         reference-int-type)
        ((not tok)
         (well-formedness-error
          "semantic value out of range"
          exp)
         reference-int-type)
        (else
         (well-formedness-error
          "unknown semantic value"
          exp)
         reference-int-type)))))
```

Figure 9.1: We can define a specialized C/sexp type method for the semantic value expressions introduced by Wisent.

```
p ∈ Parser        ::= (parser t r s)
t ∈ Tokens        ::= (tokens i ...)
r ∈ Rules         ::= (rules (n (rule (non-terminal i)
                                      (slots n)
                                      (action sa))) ...)
s ∈ States        ::= (states (n (state pa ...))  ...)
pa ∈ PDAAction    ::= (comment f)
                    | (shift (i) n)
                    | (reduce (i) n)
                    | (accept (i))
                    | (goto i n)
n ∈ Integer
sa ∈ SemanticAction
```

A parser `(parser t r s)` contains three things: a list of tokens *t*, a numbered list of rules *r*, and a numbered list of states *s*. Tokens, as in the CFG language, are merely identifiers, but states and rules require more explanation.

A rule `(rule (non-terminal i) (slots n) (action pa))` performs a reduction. In performing the reduction, the LR parser takes *n* semantic values off the stack, binds them to the semantic value expressions in *sa*, and evaluates it for a new semantic value, which it then places back on the stack as the semantic value of *i*.

An LR parser state `(state sa ...)` is defined by a list of actions. These actions are the standard actions of an LR parser, each guarded by a lookahead token.[4]

A shift action `(shift (i) n)`, guarded by lookahead token *i* shifts a token onto the stack, and then goes to state *n*.

A reduce action `(reduce (i) n)`, guarded by token *i*, performs the reduction in rule *n*, and then branches to another state based on the state of the stack after the appropriate number of slots have been removed.

An accept action `(accept (i))` completes parsing if lookahead token *i* is seen.

A goto action `(goto i n)` indicates that once a reduction is performed, if the current state is on the top of the stack, and the semantic value *i* is on the stack, the parser should branch to state *n*.

9.2.2 Translation of Wisent into Aurochs

Objects of class `wisent-parser%` delegate to objects of class `aurochs-pda%`, through the function `transform-parser`. This algorithm for doing so implements the standard LALR parsing algorithm [1] due to DeRemer and Pennelo [21].

The calculator-language parser shown earlier delegates to the Aurochs PDA shown here.

[4]This language allows for only one lookahead token. However, this lookahead token is enclosed in parentheses for forward compatibility: if this language is modified at a later date to allow for multiple lookahead tokens, parsers written in this version of Aurochs will be compatible.

```
(parser
 (tokens MULT DIV PLUS MINUS NUM EOS)
 (rules
  (0 (rule (non-terminal start) (slots 2) (action ...)))
  (1 (rule (non-terminal EXP) (slots 1) (action ...)))
  (2 (rule (non-terminal EXP) (slots 3) (action ...)))
  (3 (rule (non-terminal EXP) (slots 3) (action ...)))
  (4 (rule (non-terminal EXP) (slots 3) (action ...)))
  (5 (rule (non-terminal EXP) (slots 3) (action ...)))
  (6 (rule (non-terminal PROGRAM) (slots 1))))

 (states
   (0 (state (comment (PROGRAM "=>" ("." EXP)))
             (comment (EXP "=>" ("." EXP MINUS EXP)))
             (comment (EXP "=>" ("." EXP PLUS EXP)))
             (comment (EXP "=>" ("." EXP DIV EXP)))
             (comment (EXP "=>" ("." EXP MULT EXP)))
             (comment (EXP "=>" ("." NUM)))
             (comment (start "=>" ("." PROGRAM EOS)))
             (shift (NUM) 3)
             (goto EXP 1)
             (goto PROGRAM 2)))

   (1 (state (comment (PROGRAM "=>" (EXP ".")))
             (comment (EXP "=>" (EXP "." MINUS EXP)))
             (comment (EXP "=>" (EXP "." PLUS EXP)))
             (comment (EXP "=>" (EXP "." DIV EXP)))
             (comment (EXP "=>" (EXP "." MULT EXP)))
             (shift (MULT) 5)
             (shift (DIV) 6)
             (shift (PLUS) 7)
             (shift (MINUS) 8)
             (reduce (EOS) 6)))

   (2 (state (comment (start "=>" (PROGRAM "." EOS)))
             (accept (EOS))))
   (3 (state (comment (EXP "=>" (NUM ".")))
             (reduce () 1)))
   (4 (state (comment (start "=>" (PROGRAM EOS ".")))
             (reduce () 0)))

   (5 (state (comment (EXP "=>" ("." EXP MINUS EXP)))
             (comment (EXP "=>" ("." EXP PLUS EXP)))
             (comment (EXP "=>" ("." EXP DIV EXP)))
             (comment (EXP "=>" (EXP MULT "." EXP)))
             (comment (EXP "=>" ("." EXP MULT EXP)))
             (comment (EXP "=>" ("." NUM)))
             (shift (NUM) 3)
             (goto EXP 12)))
```

```
(6 (state (comment (EXP "=>" ("." EXP MINUS EXP)))
          (comment (EXP "=>" ("." EXP PLUS EXP)))
          (comment (EXP "=>" (EXP DIV "." EXP)))
          (comment (EXP "=>" ("." EXP DIV EXP)))
          (comment (EXP "=>" ("." EXP MULT EXP)))
          (comment (EXP "=>" ("." NUM)))
          (shift (NUM) 3)
          (goto EXP 11)))

(7 (state (comment (EXP "=>" ("." EXP MINUS EXP)))
          (comment (EXP "=>" (EXP PLUS "." EXP)))
          (comment (EXP "=>" ("." EXP PLUS EXP)))
          (comment (EXP "=>" ("." EXP DIV EXP)))
          (comment (EXP "=>" ("." EXP MULT EXP)))
          (comment (EXP "=>" ("." NUM)))
          (shift (NUM) 3)
          (goto EXP 10)))

(8 (state (comment (EXP "=>" (EXP MINUS "." EXP)))
          (comment (EXP "=>" ("." EXP MINUS EXP)))
          (comment (EXP "=>" ("." EXP PLUS EXP)))
          (comment (EXP "=>" ("." EXP DIV EXP)))
          (comment (EXP "=>" ("." EXP MULT EXP)))
          (comment (EXP "=>" ("." NUM)))
          (shift (NUM) 3)
          (goto EXP 9)))

(9 (state (comment (EXP "=>" (EXP MINUS EXP ".")))
          (comment (EXP "=>" (EXP "." MINUS EXP)))
          (comment (EXP "=>" (EXP "." PLUS EXP)))
          (comment (EXP "=>" (EXP "." DIV EXP)))
          (comment (EXP "=>" (EXP "." MULT EXP)))
          (shift (MULT) 5)
          (shift (DIV) 6)
          (reduce (PLUS) 5)
          (reduce (MINUS) 5)
          (reduce (EOS) 5)))

(10 (state (comment (EXP "=>" (EXP "." MINUS EXP)))
           (comment (EXP "=>" (EXP PLUS EXP ".")))
           (comment (EXP "=>" (EXP "." PLUS EXP)))
           (comment (EXP "=>" (EXP "." DIV EXP)))
           (comment (EXP "=>" (EXP "." MULT EXP)))
           (shift (MULT) 5)
           (shift (DIV) 6)
           (reduce (PLUS) 4)
           (reduce (MINUS) 4)
           (reduce (EOS) 4)))
```

```
(11 (state (comment (EXP "=>" (EXP "." MINUS EXP)))
           (comment (EXP "=>" (EXP "." PLUS EXP)))
           (comment (EXP "=>" (EXP DIV EXP ".")))
           (comment (EXP "=>" (EXP "." DIV EXP)))
           (comment (EXP "=>" (EXP "." MULT EXP)))
           (reduce (MULT) 3)
           (reduce (DIV) 3)
           (reduce (PLUS) 3)
           (reduce (MINUS) 3)
           (reduce (EOS) 3)))

(12 (state (comment (EXP "=>" (EXP "." MINUS EXP)))
           (comment (EXP "=>" (EXP "." PLUS EXP)))
           (comment (EXP "=>" (EXP "." DIV EXP)))
           (comment (EXP "=>" (EXP MULT EXP ".")))
           (comment (EXP "=>" (EXP "." MULT EXP)))
           (reduce (MULT) 2)
           (reduce (DIV) 2)
           (reduce (PLUS) 2)
           (reduce (MINUS) 2)
           (reduce (EOS) 2)))))
```

This is, of course, a parse table presented in the form of a program with an associated static semantics. This semantics is what drives code generation of Aurochs into an underlying general-purpose language.

Perhaps it would be better to state this in the reverse direction: parse tables are really programs (typically reified as numeric data). Most parser tools gloss over this fact; it is a pleasant property of Ziggurat that the "parse table" is more accurately presented as a program term, and so can be operated upon by all the machinery that Ziggurat provides for program terms.

9.2.3 Static semantics of Aurochs

Aurochs syntax nodes define a `parser-well-formed?:` method. However, there are only a few static properties can be checked at this translation level. The properties that we can check are:

- That the actions of rules are well-typed.
- That each reduction references an existing rule.
- That if a rule action references semantic value n, the rule consumes at least n semantic values from the stack.
- That the PDA is deterministic. *E.g.*, no state has multiple actions guarded by identical lookaheads.

It's important to note that a statically well-formed Wisent program is a statically well-formed Aurochs program. Thus, it is valid for the `parser-well-formed?:` method of a Wisent parser to override that of the delegated Aurochs method.

It would be perfectly possible in this system to implement Pottier-style LR-parser types [80] for Aurochs. Such types are associated with stack operations, and provide a type-safety guarantee regarding the semantic-value stack. Since this stack is of heterogeneous type (*i.e.*, semantic values on the stack are of more than one type), it is possible for a poorly-constructed Aurochs program to misuse a popped value. Pottier's types provide a static guarantee that this will not happen, and, additionally, that the stack will not be empty when a value is popped from it. However, since Wisent has its own type system, this would give no additional assurances about a Wisent parser. What it would do would be to allow the parser generator to associate Wisent types with Aurochs types, which could then be checked, giving the programmer an added guarantee that the parser generator is correct.

9.2.4 C/sexp code generation

Up until now, we've seen examples of language *extension,* as defined in section 2.7. However, we do not want Wisent to be merely an extension of C/sexp. Wisent is not rooted to any implementation language; it is intended to be *embeddable* within a variety of languages. Therefore, Aurochs parsers do not delegate directly to any other syntax node. We need a wrapper to embed Aurochs (and, by delegation, Wisent) into an implementation language. Therefore, we define a class `csexp-parser%` that has, as its delegate-instantiation method, the code-generation algorithm.

```
(define-syntax-class (csexp-parser% is-csexp-parser?:)
  ((name^ csexp-parser-name:)
   (token-type^ csexp-parser-token-type:)
   (read^ csexp-parser-read:)
   (parser^ csexp-parser-parser:))
  (lambda () ...))
```

A parser generated from Aurochs in C/sexp is a top-level declaration, and thus the embedding C/sexp macro must declare it as such.[5]

[5]It is possible to implement the embedding in other ways, for example, as an expression. However, the structure of C/sexp encourages spreading complicated computations across several statements and several top-level functions. Therefore, a parser is embedded as a function at top level, so that invoking the parser takes the form of a function call.

```
(declaration parser
  (lambda (env form)
    (syntax-case form ()
      ((parser name type read . decls)
       (object csexp-parser% form env
               (parse-identifier: env (syntax name))
               (parse-identifier: env (syntax type))
               (parse-identifier: env (syntax read))
               (parse-parser env (syntax decls)))))))
```

The delegate-instantiation method for a `csexp-parser%` translates an Aurochs parser into the equivalent C/sexp code. Recall our running calculator-language example. We will step through the rewritten code line-by-line to understand how it works.

The first thing that the parser must do is establish the types that will be necessary for the encoding. Terminals and non-terminals together form an enumeration. The code in this chapter will be the translated C code rather than C/sexp, for ease in reading.

```
enum token_2 {start, MULT_1, DIV_1, PLUS_1, MINUS_1, NUM_1,
              EOS_1, EXP, PROGRAM};
```

Actual tokens can have semantic values, and thus the actual token type is a tagged union of the types of these semantic values.

```
typedef struct token_1 {enum token_2  kind;
                        union value_1 {int start;
                                       int NUM;
                                       int EXP;
                                       int PROGRAM;}
                        value;} token;
```

Next, for each terminal, we have a constructor, visible to the programmer outside of the rewritten code. He will need to be able to construct terminals to feed into the parser, for example, to build a lexer.

```
token NUM(int value) {
  token to__return_2;
  (to__return_2).kind=(NUM_1);
  ((to__return_2).value).NUM=(value);
  return to__return_2;
}

...
```

Next, for each rule, we generate a function. This is directly from the semantic actions found in the parser, since the semantic value Wisent expressions rewrite into C/sexp variables. Because this is generated code, two identifiers that use the same symbol are differentiated in the generated code by having different numerals

appended to their symbol; for example, if the symbol `semantic-value` is used to refer to two different variables, the two variables in the generated code are referred to as `semantic__value_0` and `semantic__value_1`.

```
int rule_3(int semantic__value_9) {
  printf("NUM %d\n", semantic__value_9);;
  return semantic__value_9;
}
```

Next, we have a function prototype for the lexer function. The programmer who defined the parser is also responsible for defining the lexer.

```
token get__token();
```

Next, we define a "state" function, that performs the actions for the current state, seen in figure 9.2.

Next, we define the parser "rule" function, that determines the correct semantic-action function, and performs stack manipulation before calling the goto function.

```
int rule(int rule_1, int* state__stack,
         token* term__stack, token next__token) {{
  switch (rule_1) {
    ...
    case 3: {{
              state__stack=((state__stack)-(3));
              term__stack=((term__stack)-(3));
              token to__push_3;
              (to__push_3).kind=(EXP);
              ((to__push_3).value).EXP=
                  ((rule_5)
                   (((*((term__stack)+(0))).value).EXP,
                    ((*((term__stack)+(2))).value).EXP));
              return (goto)
                     (to__push_3,*(state__stack),
                      state__stack,term__stack,next__token);
            }}
    ...
```

Next, we define the parser "goto" function.

```
int state_1(int state, int* state__stack,
            token* term__stack, token next__token) {
  switch (state) {
   ...
    case 10: {if (((next__token).kind)==(MULT_1))
               {
                 *((state__stack)++)=(state);
                 *((term__stack)++)=(next__token);
                 return (state_1)(5,state__stack,
                                  term__stack,
                                  (get__token)());
               }
              else {if (((next__token).kind)==(DIV_1))
               {
                 *((state__stack)++)=(state);
                 *((term__stack)++)=(next__token);
                 return (state_1)(6,state__stack,
                                  term__stack,
                                  (get__token)());
               }
              else {if (((next__token).kind)==(PLUS_1))
               {
                 return (rule)(4,state__stack,
                               term__stack,next__token);
               }
              else {if (((next__token).kind)==(MINUS_1))
               {
                 return (rule)(4,state__stack,
                               term__stack,next__token);
               }
              else {if (((next__token).kind)==(EOS_1))
               {
                 return (rule)(4,state__stack,term__stack,
                               next__token);
               }
              else {return 0;
             }}}}}}
...
```

Figure 9.2: The rewritten C state function for the running example.

```
int goto(token nonterm, int state, int* state__stack,
         token* term__stack, token next__token) {{
  switch (state) {
    ...
    case 8: {
             if (((nonterm).kind)==(EXP)) {{
               (state__stack)++;
               *((term__stack)++)=(nonterm);
               return (state_1)(9,state__stack,
                                term__stack,next__token);
               }}
             else {return 0;
               }}
    ...
```

Finally, we can define the parser "main" function, that starts the entire computation.

```
int pmain() {{
  return (state_1)(0,(int*)((malloc)((sizeof(int))*(100))),
                   (token*)
                   ((malloc)((sizeof(token))*(100))),
                   (get__token)());
  }}
```

This entire computation is recursive. However, it is tail-recursive, and many modern compilers such as GCC [91] optimize tail-calls so that they are no less efficient than an iterative construct. For compilers where this is not the case, it is possible to re-engineer the PDA-to-C/sexp translation to use iteration instead of recursion.

9.3 Parser macros

Wisent need not be the top of the language tower. It is possible to write macros that expand into Wisent parsers, or parts of Wisent parsers. These "parser macros," are macros in the parser language, and thus are different from the "parser macros," presented in section 7.8.2, which are macros whose machinery is limited to the parse phase of compilation.

Wisent expressions can be produced by macro expansion in one of two ways. The macro writer can simply write a C/sexp macro that expands into a parser. Additionally, we can define extensions to Wisent language defining syntax for macros within a parser, similar to macros in C/sexp. This syntax extends the grammar in section 9.1.1.

$d \in$ Declaration ::= ...
| (syntax z z x ...)
$x \in$ Syntax ::= ...
| (nonterminal i z)
| (production i z)

This syntax should be familiar; it is nearly exactly that for macros in C/sexp. A parser macro has three parts: a section of Ziggurat code to evaluate for effect, a Ziggurat function to produce a piece of syntax to stand in for the entire syntax declaration, and a sequence of new non-terminals and productions.

We will see examples of each kind of macro. First, we will see a macro inside of a parser, that adds a new form of production to the language, the comma-separated list. Second, we will see a C/sexp macro that expands into a full parser: we will see a lexer language that expands to the parser language.

9.3.1 Example: comma-separated list

Some productions are common enough to be abstracted, and so we allow parser macros to expand into parser productions. The example we will see in this section is the comma-separated list, used, for example, in specifying function arguments in ALGOL-like languages.

A comma-separated list is a new production.

$p \in$ Production ::= `(csl` i i`)` | ...

If i_1 and i_2 are terminals or non-terminals, the production `(csl` i_1 i_2`)` will match a list of i_1, separated by i_2. So, for example, `(csl EXP COMMA)` will match `EXP COMMA EXP COMMA EXP`, but not `EXP COMMA EXP COMMA`. The semantic action of the `csl` macro is to produce a list of the semantic values of the elements of the comma-separated list.

The `csl` macro needs to insert a number of new non-terminals into the parser. The parser

```
(parser pmain token get-token
  (include "csl.parser")
  (tokens SUM MUS (type int NUM) DOT (eos EOS))
  (non-term EXP int (=> (NUM) (return ($ 1))))
  (non-term EXPLIST (list int) (csl EXP DOT))
  ...)
```

will expand into

```
(parser pmain token get-token
  (non-term g100 (list int)
    (=> (EXP) (return (make-list ($1) (null))))
    (=> (EXP DOT g100) (return (make-list ($ 1) ($ 3)))))
  (tokens SUM MUS (type int NUM) DOT (eos EOS))
  (non-term EXP int
    (=> (NUM) (return ($ 1))))
  (non-term EXPLIST (list int)
    (=> (g100) (return ($ 1))))
  ...)
```

An outline of the CSL macro can be found in figure 9.3. The excluded machinery maintains the list `csls` of the forms of comma-separated list used in the parser. Items are added to this list by the function `get-csl-non-term`, and the list is used in the expansion of the `csl-decl%` macro. Similar to the structure of the list macro in section 7.8.3, elements are added to this list as the further code is parsed, and then this list is used in the expansion of the macro declaration.

There is something very important to note here: Even though a CSL is a Wisent macro, it generates code that makes use of the list machinery. Lists are C/sexp macros! However, since parsers expand into raw C/sexp code, and lists expand into raw C/sexp code independently, they both can be used together. This does mean, on the other hand, that a new embedding of Wisent into a base language would require re-implementation of the CSL macro.

9.3.2 Lexer generator

We can use language towers to turn our parser generator into a lexer generator. We can design a regexp-based lexer language, and then use Ziggurat to translate lexers into CFG parsers, where our tokens are individual characters.

Lexer language

The language to write a lexer should be intuitive for anyone who has used a lexer-generator tool, such as Lex [59] or Alex [22]. A lexer consists of a sequence of token declarations, where a token declaration is a regular expression to be matched and a semantic action (a C/sexp statement) to be executed in the event of a match. The semantic action is parameterized not only by the string that matched the pattern, but by an accumulated value, similar to the way `fold` functions work in functional languages [66]. We can write a lexer for the above expression language:

```
(lexer make-token-list (list tok) (return (null))
  ((: (| #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9)
      (* (| #\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9)))
    (token accum)
    (return (make-list (app NUM (app atoi token)) accum)))
  ("*" (accum) (return (make-list (app MULT) accum)))
  ("/" (accum) (return (make-list (app DIV) accum)))
  ("+" (accum) (return (make-list (app PLUS) accum)))
  ("-" (accum) (return (make-list (app MINUS) accum)))
  ((| " " "\r" "\n") (accum) (return accum)))
```

This defines a C/sexp function, `make-token-list`, that takes a string, and produces a list of tokens. The full concrete syntax is quite simple.

```
(syntax
 (begin
;;; csls is a list of all the csls defined.
;;; a csl is specified by the list:
;;; (item separator generated-non-term-name type-object)
   (define csls '())

    (define-syntax-class csl-decl% ()
     (lambda ()
       (syntax-object parser-decl-block% source^ env^
          (map
           (lambda (x)
             (parse-parser-decl: local-env
               (syntax
                 (non-term #,(object pre-parsed-datum%
                                     (caddr x))
                           #,(object pre-parsed-datum%
                                     (cadddr x))
                                 (=> (#,(car x))
                                     (return (make-list
                                              ($ 1) (null))))
                                 (=> (#,(car x) #,(cadr x)
                                                  #,(caddr x))
                                     (return (make-list
                                              ($ 1) ($ 3))))))))
           csls))))
   ...)

 (lambda (env)
   (syntax-object csl-decl% (z-syntax decl) env))

 (production csl
   (lambda (env form)
     (syntax-case form ()
       ((csl item separator)
        (let ((item (parse-identifier: env (syntax item)))
              (separator
               (parse-identifier: env (syntax separator))))
          (parse-parser-production: env
            (syntax (=> (#,(get-csl-non-term item separator))
                        (return ($ 1)))))))))))
 )
```

Figure 9.3: The basic outline of the CSL macro.

$l \in$ Lexer	::= `(lexer` *i t sa d* `...)`	
$d \in$ Declaration	::= `(`*re* `()` *sa*`)`	*; Optional identifiers are*
	| `(`*re* `(`i_{acc}`)` *sa*`)`	*; bound to accumulated*
	| `(`*re* `(`i_{acc} i_{match}`)` *sa*`)`	*; value & matched text.*
re $\in$ RegExp	::= *c*	*; Character constant*
	| *s*	*; String constant*
	| `(:` *re* `...)`	*; Sequence*
	| `(|` *re* `...)`	*; Choice*
	| `(*` *re* `...)`	*; 0 or more*
sa $\in$ Action		*; Semantic action*
$t \in$ Type		
$c \in$ Character		
$s \in$ String		

Unlike the parser language, the lexer language is fixed to its target: a parser embedded within C/sexp. Therefore, the language-embedding parts, semantic actions and type syntax, are tied to embedding language statements and types.

Token declarations take one of three forms.

- `(`*re* `()` *sa*`)` If a token matching the regular expression *re* is found, the semantic action *sa* is evaluated for a new accumulated value.
- `(`*re* `(`i_{acc}`)` *sa*`)` If a token matching *re* is found, the C/sexp variable i_{acc} is bound to the accumulated value, and *sa* is evaluated in the scope of i_{acc}.
- `(`*re* `(`i_{acc} i_{match}`)` *sa*`)` If a token matching *re* is found, the C/sexp variable *i*`match` is bound to the C/sexp string that matched the regular expression, the C/sexp variable *i*`acc` is bound to the accumulated value, and *sa* is evaluated in this new scope.

Regular expressions are constructed out of a handful of primitive combinators.

- `(:` *re* `...)` Forms a regular expression that is a concatenation a sequence of regular expressions.
- `(|` *re* `...)` Forms a regular expressions that is the alternation of a set of regular expressions.
- `(*` *re*`)` Forms the Kleene closure of a regular expression.
- Any character, `#\a` `#\b` ... Forms a regular expression that matches exactly one character.
- Any string is equivalent to the concatenation of its characters.

It's worth noting that regular expressions form their own independent language, and thus it would be possible to do language extension on regular expressions, though this is not yet implemented. For example, it would be possible to construct an "at least once" operator `+`, such that `(+` *re*`)` would expand to `(:` *re* `(*` *re*`))`.

9.4 Advantages of the Ziggurat approach

Parser-generating technology of the sort presented in this chapter is not novel. Indeed, there are many useable LALR(1) parser-generator languages available to today's programmer. In this chapter, we have seen some of the advantages that emerge from building a parser generator as a Ziggurat macro.

- The language is separable from its implementation, making it relatively easy to relocate Wisent and Aurochs to other implementation languages.
- Wisent has its own static semantics, for example, its own type rules. These type rules can be related to the type system of the implementation language by typing semantic actions.
- Macros can be build on top of Wisent, and may use facets of the implementation language, or remain agnostic towards them.
- These macros merge seamlessly with macros in the underlying language; Wisent semantic actions can use C/sexp macros.
- Macros that build on C/sexp's type system can also be used transparently with Wisent, even if semantic values take types created by the C/sexp macros.

The static semantics we have been observing in this chapter, that is, the type systems of C/sexp and Wisent, demonstrate these principles. However, the full power of this approach can perhaps be seen with a more computationally difficult semantics, such as flow analysis, which we will see in the next chapter.

Chapter 10

Slicing C/sexp: Linking analyses across language levels

Ziggurat's main focus is to allow the static semantics of different languages to be mixed when one is embedded within another. So far we have seen how C/sexp's type system can be extended for language extensions and embeddings. However, type analysis is only one form of static analysis; there are others which can benefit from the semantic linking that Ziggurat allows. In this section, we will see how program slicing can be made more precise through semantic linking.

Program slicing is a debugging method used by programmers for reducing a program to an interesting subset. Starting with a particular program behavior, slicing reduces that program to a minimal subset that produces that behavior [111]. For slicing in C/sexp, this will take the form of a Ziggurat function that takes a C/sexp statement, and produces a set of definitions and statements that influence the behavior of that starting statement. Deriving a minimal slice is in general impossible. The precision of a slice depends on the algorithm used to generate it; various algorithms are more suited to different programming models. Because of this, slicing provides an excellent opportunity for static semantic extension. Ziggurat will allow us to use different slicing algorithms for different languages. As an added benefit, when a single program incorporates several languages, Ziggurat will allow us to use these multiple algorithms *even while slicing a single program:* the correct algorithm will be used for the relevant piece of code.

We provide a basic slicing algorithm for C/sexp in the form of generic functions on C/sexp syntax nodes in such a way that abstract syntax that delegates to C/sexp can provide methods for these functions, and provide a more precise slice. This algorithm is based on a system of constraints, and running the algorithm involves solving the constraints. The inner workings of the constraint solver are not integral to the thesis; thus, the focus of this chapter will be on the object-based methodology used to build the constraints, and the means by which this algorithm can be extended for new languages.

10.1 Definitions

For purposes of slicing in C/sexp, we need to define a few terms. We must first define data and control dependency.

A syntax node A is control-dependent on a syntax node B if the computed behavior of B can determine whether control reaches A. The "computed behavior" of a statement is defined as its effect; for example, the computed behavior of a statement containing a single assignment is the value that gets stored in the left-hand side of the assignment, as well as the changing value of that left-hand side. Likewise, the computed behavior of an expression is its effect as well as its computed value.

A syntax node A is data-dependent on a syntax node B if the computed behavior of B can determine the computed value of A.

Collectively, control- and data-dependency are simply called dependency. We define dependency conservatively, such that if α is a potential dependency set for a given program that is larger than a dependency set β for the same program, and if β is correct, then α is correct too. It is undecidable to determine the smallest such dependency set for any program, and thus our main goal in Ziggurat will be to describe specialized dependency algorithms that are more precise for a subset of programs— in particular, programs that use multiple languages.

We will define the *slice* of a program at a syntax node as the set of syntax nodes that it is dependent on. For example, consider the C/sexp code:

```
(fun foo int ()
  (return 3))

(fun bar int ()
  (return 4))

(fun main int ()
  (var x int)
  (var y int)
  (:= x (foo))
  (:= y (bar))
  (return x))
```

If we sliced on the statement `(return x)`, we would get the slice:

```
(fun foo int ()
  (return 3))

(fun main int ()
  (var x int)
  (:= x (foo))
  (return x))
```

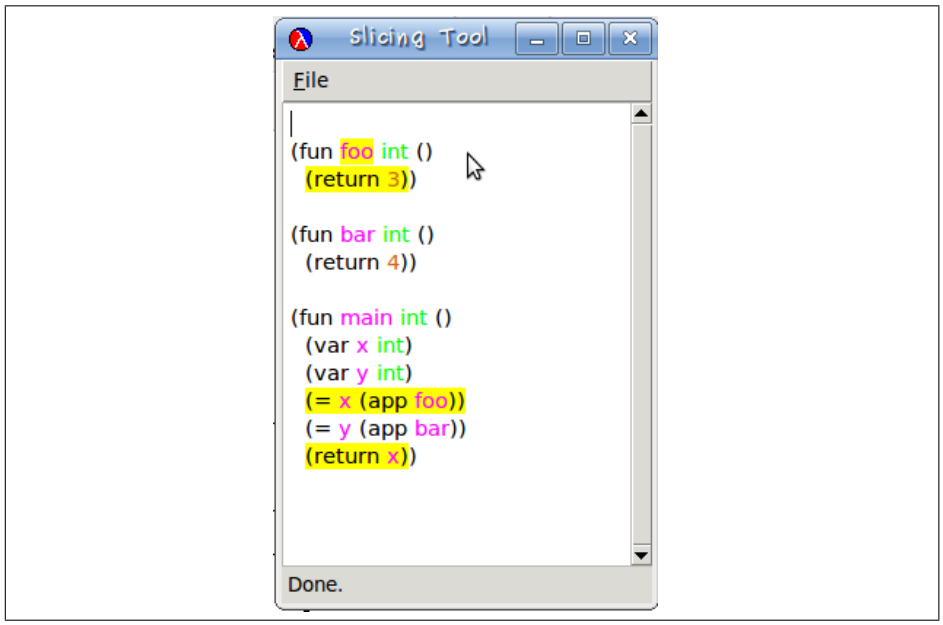

Figure 10.1: An example of using the slicing tool.

I have built a GUI tool to display a slice. The tool displays a piece of C/sexp code, and when the user clicks on a syntax node, the tool highlights all of the syntax that is in the slice. Performing this highlighting is straightforward: since syntax objects retain source-location information, the tool simply walks over the syntax objects in a slice, and highlights the appropriate source locations. This tool uses the MrEd editor toolkit to interact with the user. An example of generating the slice presented above in the GUI toolkit can be seen in figure 10.1.

For simplicity, we will make the conservative approximation that dependency is transitive: that is, if A is dependent on B and B is dependent on C, then A is dependent on C. Because of this assumption, we can set up the dataflow analysis of a C/sexp program as a group of simultaneous set equations. Since this algorithm will be implemented by generic functions, the algorithm should be structured as a walk over the parse tree. We write a set of generic functions to produce *goals* that, when solved, will produce a program slice. These goals are the primary part of the Tsuriai library.

10.2 Tsuriai

Tsuriai is a set-based constraint language and solver, intended to be used to solve dataflow problems, but not directly tied to any fixed notion of dataflow. Tsuriai constraints consist of mutually recursive set declarations, called *goals*. Ziggurat provides a library of macros and functions to define goals and run them. When a goal is run, Tsuriai finds the least fixed point of these declarations. For example, evaluating the Ziggurat expression

```
(all (<- x (U (set 1 2) y))
     (<- y (U (set 2 3) x)))
```

will result in a Tsuriai goal, assuming `x` and `y` are bound to Tsuriai variables. We can run this goal with the Ziggurat function `run`:

```
(run table
  (all (<- x (U (set 1 2) y))
       (<- y (U (set 2 3) x))))
```

This will result in the mapping $\mathtt{x} \mapsto \{1, 2, 3\}, \mathtt{y} \mapsto \{1, 2, 3\}$. The argument `table` to `run`, above, specifies the form of the return value of the `run` expression: specifically, that the resulting mapping should be returned as a hash table that maps Tsuriai variables to their bound sets, represented as lists.

Variables are created with the `var` form. Evaluating the Ziggurat expression `(var` *e*`)` creates a fresh variable. The expression *e* is merely a marker used for error reporting; evaluating the same `var` expression several times will result in distinct Tsuriai variables, even if the markers used in their definitions are identical.

There are four ways to build goals in Tsuriai.

- The most basic goal is `succeed`. The goal `succeed` automatically succeeds without binding any variables.
- A goal that binds a variable can be written `(<-` *v s*`)`. The expression *v* must evaluate to a Tsuriai variable. The expression *s* must evaluate to a *selector*: a selector builds a set from a mapping from Tsuriai variables to sets. Selectors will be covered in greater detail below.
- Goals can be combined with the expression `(all` *g g* `...)`.
- Sets can be abstracted over with the expression `(maps` *l s*`)`. The expression *l* should evaluate to a function of one argument that returns a goal. Running this *maps* goal selects a set by the selector *s*, applies the function *l* to each element in this set for a new goal, which is then run. For example, running the goal

  ```
  (all
   (<- x (set 1 2))
   (maps (lambda (y) (<- x (U x (- y)))) x))
  ```

  will result in the mapping $\mathtt{x} \mapsto \{-2, -1, 1, 2\}$.

Selectors take a current mapping of variables to sets, and form a new set.

- A selector that selects a constant set is written `(set` *i* `...)`. The expressions *i* are arbitrary values; Tsuriai allows any Ziggurat value as a set item.
- Tsuriai variables are also selectors. A Tsuriai variable selects its current value in the mapping.

- The selector `(U s s ...)` selects the union of the sets selected by the selectors *s* .... Other set operations are not allowed; in this way, we guarantee that a goal will reach a fixed point. The exception is `maps`, which we will see below.

A goal can be run with the expression `(run s g)`. After finding the least fixed point of the goal *g*, this expression evaluates to the set selected by the selector *s*. For use in `run` expressions, Tsuriai provides a special selector, `table`, that produces a hash table mapping variables to list representations of their mapped sets.

Tsuriai variables, selectors and goals are first-class objects in Ziggurat. By returning them from lazy delegation methods, we can build dataflow algorithms from generic functions.

10.3 Dataflow with Tsuriai goals

To slice on a statement, we will need to determine its dependency set, as defined in section 10.1. In order to find this, we will need to solve a number of set equations simultaneously. We will use Tsuriai to express and solve these equations, but first, we will need to define a number of Tsuriai variables.

- For each C/sexp statement and expression, we will need to know every other statement or expression that helps determine its behavior, that it is dependent on. For this, then, we will need a Tsuriai variable for each expression and statement. We can use attributes to make this association; since attributes are memoized, we guarantee that this attribute will return the same Tsuriai variable for each syntax node.

  ```
  (define-attribute dependence-var:
    (lambda (this)
      (var this)))
  ```

  Since we will frequently use this generic function, we will abbreviate it `d!`.

  In addition, we define a Tsuriai variable `heap-var` to indicate "dependent on the state of the store." This is a very conservative approach to alias analysis: any expression that loads any value from the store is potentially dependent on any expression that stores a value.

- For each variable, we will need to know every statement or expression that helps determine its value. We cannot use attributes for this, because different objects can refer to the same C/sexp variable: therefore, since in C/sexp variables are represented by identifiers, we define a table `variable-table` that maps identifiers to Tsuriai variables.

- For each expression that is used as a function, we will need to know every function it may refer to. (In most cases, this will obviously resolve to one function.) Similarly to the above case, we define a table `function-table` to represent this mapping. For expressions where it is not clear what function they refer to, we define a Tsuriai variable `unknown-function-var`.

- For each function, we will need a variable to represent the expressions and statements that the return value of that function may be dependent on. We define an attribute, `return-var:` that holds the Tsuriai variable that represents this.

Now that we have defined these variables, we can build a goal to solve for them by traversing the syntax tree in a single pass, via recursive calls of a generic function `csexp-dependence:`. [1] Most of this traversal is straightforward, so we will only see some illustrative examples.

The simplest goal definition is the dependency goal for constant expressions. A constant expression will naturally have no data dependence, since it will always give the same computed value. So, we only need concern ourselves with its control dependence.

```
(define-method const-csexp-exp% csexp-dependence:
  (lambda (this current-control)
    (<- (d! this)
        (U (d! this)
           (set this)
           (cobj-enclosing-control: current-control)))))
```

The argument `current-control` to the generic function `csexp-dependence:` is an object of class `control-object%` containing Tsuriai variables representing information passed down from lexically enclosing elements. This method uses the field `enclosing-control`, which is a Tsuriai variable that represents the set of statements and expressions that the current expression is control-dependent on. For example, in the C/sexp expression `(?: x 5 4)` the constant expression `5` is dependent on the variable `x`.

In addition to the control dependency, a constant expression is dependent on two other things:

- `(d! this)` This may seem to be redundant; however, it is necessary in case dependency information for this expression is also defined elsewhere.
- `(set this)` An expression is dependent on itself.

The dependency goal for variable expressions is only slightly more complicated.

```
(define-method var-csexp-exp% csexp-dependence:
  (lambda (this current-control)
    (<- (d! this)
        (U (d! this) (set this)
           (lookup-variable name^)
           (cobj-enclosing-control: current-control)))))
```

[1] It's worth noting that this generic function must be considered part of the static semantics of the language, and thus included in the specification of such in appendix A. Likewise, when we get to dataflow on parsers, We will need to have an analogous generic function for Wisent.

In addition to the dependency introduced above, variable expressions are also dependent on any statements or expressions that may determine the value of the variable they represent. The function `lookup-variable` looks up a Tsuriai variable in `variable-table`, or defines one if none is found. Variables are dependent mostly on assignment expressions.

Let's look at an assignment expression. In addition to the control dependency introduced above, we will want to also generate the goal that the left-hand side is dependent on the right-hand side.

```
1 (define-method assign-csexp-exp% csexp-dependence:
2   (lambda (this current-control)
3     (all (csexp-dependence: lhs^ current-control)
4          (csexp-dependence: rhs^ current-control)
5          (<- (d! this)
6              (U (d! this) (set this) (d! rhs^)
7                 (cobj-enclosing-control: current-control)))
8          (<- (lvalue-variable: lhs^)
9              (U (lvalue-variable: lhs^) (d! this))))))
```

To explain this goal, let's look at it line-by-line.

- Lines 3-4: the left-hand and right-hand sides of this assignment will generate new dependence goals. The goal generated by the assignment should represent those goals.
- Lines 5-6: In addition to the basic dependency of the constant expression, assignment expressions are dependent on their right-hand side.
- Lines 7-8: The Tsuriai variable the represents the left-hand side of this expression (either a Tsuriai variable representing a C/sexp variable or `heap-var`) is dependent on this expression.

The most complicated method for `csexp-dependence:` is for function application.

```
1  (define-method funapp-csexp-exp% csexp-dependence:
2    (lambda (this current-control)
3      (all
4       (csexp-dependence: fun^ current-control)
5       (<- (d! this)
6           (U (d! this) (set this) (d! fun^)
7              (cobj-enclosing-control: current-control)))
8       (let loop ((args args^))
9         (if (pair?: args)
10            (all (csexp-dependence: (car: args)
11                                    current-control)
12                 (loop (cdr: args)))
13                 succeed))
```

```
14      (maps (lambda (fn)
15              (all (<- (d! this)
16                       (U (d! this) (set fn)
17                          (return-var: fn)))
18               (let loop ((args (fun-args: fn))
19                          (actuals args^))
20                 (if (and (pair?: args) (pair?: actuals))
21                     (all (<- (lookup-variable (car: args))
22                              (U (lookup-variable (car: args))
23                                 (d! (car: actuals))))
24                      (loop (cdr: args) (cdr: actuals)))
25                     succeed))))
26            (likely-function: fun^)))))
```

This can be explained line by line.

- Lines 4-7: the first part of the dependency method is boilerplate: we must solve for the function expression, and provide basic dependency for the application.
- Lines 8-13: we must loop over the arguments, and solve for each actual argument.
- Lines 14-26: we loop over all of the possible functions for the function expression. This is either a single function (in the case of an identifier that indicates a function) or `unknown-function-var`, in the case of a calculated function pointer. For each possible function, we apply the enclosed function.

  For each possible function, the function application is dependent on the return value of that function. Additionally, the formal arguments of the function are dependent on the actual arguments of the function application.

By solving these mutually recursive definitions, we can discover the statement and expressions that a statement or expression is dependent on. This is the program slice from that statement or expression.

10.4 Slicing

Slicing is a three-step process:

- Generate a dependency goal for the entire program.
- Solve the goal for a table.
- To slice on a statement, look up the statement in the table. All of the statements and expressions in the resulting set are in the slice.

For example, in the program

```
(fun foo int ()
  (return 3))

(fun bar int ()
  (return 4))

(fun main int ()
  (var x int)
  (var y int)
  (:= x (foo))
  (:= y (bar))
  (return x))
```

if we were to slice on `(return x)`, we would generate a goal that would represent several facts:

- `(return x)` is dependent on the variable `x`.
- `x` is dependent on the assignment `(:= x (foo))`.
- `(:= x (foo))` is dependent on the function `foo` and its return value.
- The return value of the function `foo` is dependent on the statement `(return 3)`.

As a result, we obtain the slice:

```
(fun foo int ()
  (return 3))

(fun main int ()
  (var x int)
  (:= x (foo))
  (return x))
```

This is simple enough if the code does not contain macros. It turns out that it is no more difficult if the code does contain macros.

10.4.1 Slicing of macro application via rewriting

Since slicing was implemented via generic function, no additional work is required to slice macro applications. For example, delegation gives us a ready-made method to slice occurrences of the `cond` macro presented in section 7.8.2. When applying the generic function `csexp-dependence:` to the C/sexp code, as follows:

```
(include "printf.clib")
(include "cond.csexp")

(fun rabbit-count (* char) ((human-number int))
  (var x (* char))
  (:= x "too many")
  (cond ((== human-number 1) (:= x "one"))
        ((== human-number 2) (:= x "two"))
        (else (printf "you're counting too high!")))
  (return x))
```

the generic function is delegated to the rewritten form:

```
(include "printf.clib")
(include "cond.csexp")

(fun rabbit-count (* char) ((human-number int))
  (var x (* char))
  (:= x "too many")
  (if (== human-number 1)
      (:= x "one")
      (if (== human-number 2)
          (:= x "two")
          (printf "you're counting too high!")))
  (return x))
```

which, when slicing on `(return x)` gives us the slice:

```
(include "printf.clib")
(include "cond.csexp")

(fun rabbit-count (* char) ((human-number int))
  (var x (* char))
  (:= x "too many")
  (cond ((== human-number 1) (:= x "one"))
        ((== human-number 2) (:= x "two")))
  (return x))
```

correctly removing the irrelevant condition from the `cond` statement.[2] The slice presented here is the highlighted output of the slicing tool; since the source location of expanded code is derived from the source location of the original macro, the slicing tool is able to highlight the correct macro call.

[2]For this example, we are ignoring potential side-effects of the undefined function `printf`.

In many cases the slice given to us by delegation is too conservative, and we will want to make it more precise by providing a method for `csexp-dependence:`. An example of this is in the parser macro.

10.5 Slicing parsers is difficult

Wisent parsers can be sliced by delegating to the expanded code, as with any code that uses macros. However, this will tend to give a very conservative slice: semantic actions depend on one another, but which depend on which can be difficult to discover automatically. For example, in the program

```
(parser pmain token get-token
  (tokens EQ COLON (type int NUM) ID (eos EOS))

  (non-term NUM-EXP int
    (=> (NUM) (return ($ 1))))

  (non-term ID-EXP int
    (=> (ID) (return 0)))

  (non-term EXP int
    (=> (EQ NUM-EXP) (return ($ 2)))
    (=> (COLON ID-EXP) (return ($ 2)))))
```

Slicing on the semantic action of `(=> (COLON ID-EXP) (return ($ 2)))` produces a slice containing the entire program. However, this is unnecessarily conservative: if the production `(=> (COLON ID-EXP) (return ($ 2)))` is used, then clearly the non-terminal `NUM-EXP` will never be used, and much of the parser is irrelevant to the slice. This imprecision is due to the fact that when the parser is rewritten to C/sexp, all semantic actions are called from the single function `rule`, based on the contents of a parser stack located in the heap. Without lazy delegation, it's not clear how the slice could be made more precise.

The parser stack is located in the heap. Perhaps slicing could be made more precise by providing better alias analysis. However, alias analysis tells us which pointers may refer to the same location in memory, so it's not clear that a better alias analysis algorithm would provide a more precise slice. In addition, the precision of this method is highly dependent on the translation of Aurochs to C/sexp. If the translation method were to copy the parser table to a static value, and then interpret the resulting PDA via this table, as many parser-generators do, the situation is even worse. The dependency algorithm would need to interpret this resulting table, somehow. By decoupling the dependency algorithm from the translation, Ziggurat can manage even this case.

10.6 Better slicing of parsers via lazy delegation

Instead of attempting to come up with a slicing algorithm that works perfectly for all applications, we can use lazy delegation to specialize our slicing algorithm for different embedded languages. For example, it's not clear how to derive a general slicing algorithm that works well with C/sexp generated by the parser macro, but it is easy enough to build a slicing algorithm specifically for the parsing language Wisent. Likewise, connecting it to the general slicing algorithm is straightforward: we merely need to override the `csexp-dependence:` generic function.

The slicing algorithm for the parser is straightforward, and can be implemented as Ziggurat code in the static part of a C/sexp macro. We define a Tsuriai variable for each non-terminal in the parser. For each semantic action, we run `csexp-dependence:` for that C/sexp statement, with only two new features: all semantic values found in that action will be dependent on their relevant non-terminal variable, and the variable of the current non-terminal will be dependent on the return value of the semantic action. For example, in the parser declaration

```
(non-term EXP int
  (=> (COLON ID-EXP) (return ($ 2)))))
```

we would define two new Tsuriai variables: one for `EXP`, and one for `ID-EXP`. Running `csexp-dependence:` on the expression `($ 2)` would result in the expression `($ 2)` being dependent on the Tsuriai variable for `ID-EXP`, and running `csexp-dependence:` on the statement `(return ($ 2))` would result in the Tsuriai variable for `EXP` being dependent on the return statement.

The slices produced by the slicing algorithm are now much more precise. Running the above example now gives us the slice

```
(parser pmain token get-token
  (tokens EQ COLON (type int NUM) ID (eos EOS))

  (non-term ID-EXP int
    (=> (ID) (return 0)))

  (non-term EXP int
    (=> (COLON ID-EXP) (return ($ 2)))))
```

It's worth noting that this extended algorithm, in addition to providing a much more precise slice, runs a lot more quickly. The parser tool generates a lot of code: tens of lines of source can easily become thousands of lines of generated code. Thus, dataflow analysis that takes seconds with the extended algorithm can take minutes if dataflow is performed by delegation.

Since the slicing goals are not limited to any one language, providing a better slicing algorithm for an embedded language can create a better slice for the entire composite program. For example, consider the parser:

```
(fun display int ((arg int))
  (if (< arg 0) (= arg (- arg)))
  (return (* arg 100)))

(parser pmain token get-token
  (tokens (left PLUS DISPLAY) (type int NUM) (eos EOS))
  (non-term EXP int
    (=> (NUM) (return ($ 1)))
    (=> (EXP PLUS EXP) (return (+ ($ 1) ($ 3)))))
  (non-term PROGRAM int
    (=> (DISPLAY EXP) (return (display ($ 2))))
    (=> (EXP) (return ($ 1))))
  (start PROGRAM))

(var instring (* char))

(fun get-token token ()
     (var in (* char) (strtok instring " "))
     (printf "<%s>" in)
     (= instring (cast (* char) 0))
     (cond
      ((== in (cast (* char) 0)) (return (EOS)))
      ((== 0 (strcmp in "+")) (return (PLUS)))
      ((== 0 (strcmp in "display")) (return (DISPLAY)))
      (else (return (NUM (atoi in))))))


(fun main int ((argc int) (argv (* (* char))))
     (:= instring (readline "> "))
     (var result int)
     (:= result (pmain))

     (printf "result: %d" result))
```

If we slice on the statement `(return ($ 1))` by delegation, the result is shown in figure 10.2. This is clearly much too conservative.

Due to delegation, the slicing algorithm made the assumption that `($ 1)` could become bound to the return value of the function `display`. By simple inspection, this is impossible. The result of `display` can only be the semantic value of a non-terminal of type `PROGRAM`, and `($ 1)` can get the semantic value of a non-terminal of type `EXP`. As a result of this imprecision, the entire function `display` is included in the slice. Fortunately, this is exactly the precision that providing a specialized method gets us. We can see the better slice in figure 10.3.

Here, the function `display` is not included in the slice. The specialized dependence method has improved precision *even for parts of the program in a different language.* Providing a specialized method for one language improves precision wherever

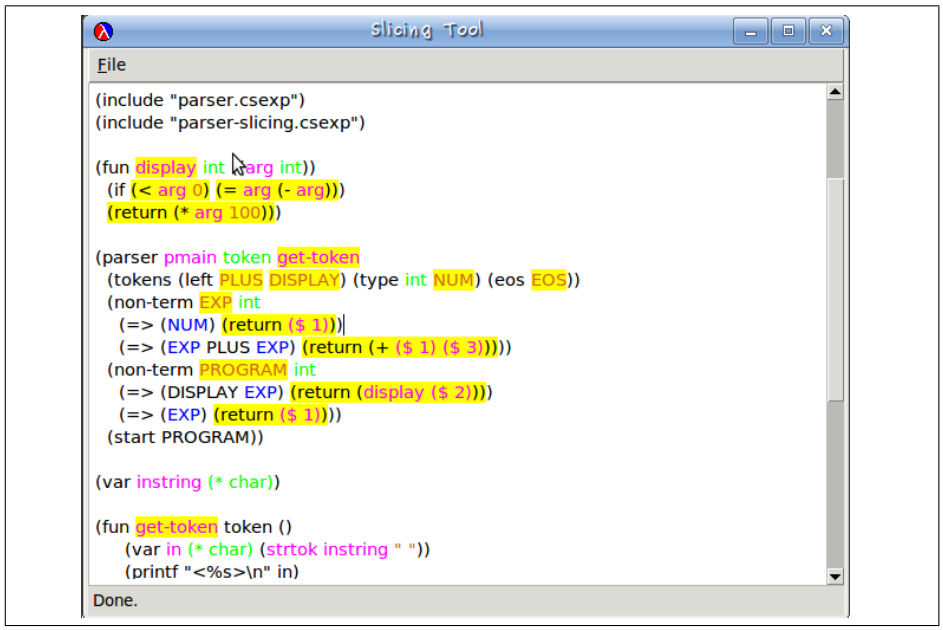

Figure 10.2: The result of slicing a parser via delegation.

that language is used, even in programs where that language is simply embedded, and not the primary language.

10.7 Dependence functions for parsers

The code to implement the algorithm is much longer than the previous methods, but uses the same basic structure, and can be explained in much the same way.

```
(define-method csexp-parser% csexp-dependence:
  (lambda (this current-control)
    (let ((token-table (make-identifier-table)))
      (let loop ((non-terms (parser-non-terminals: parser^)))
        (if (pair? non-terms)
            (begin
              (identifier-table-put!: token-table
                                      (car non-terms)
                                          (d! (car non-terms)))
              (loop (cdr non-terms)))))
      (let loop ((tokens (parser-tokens: parser^)))
        (if (pair? tokens)
            (begin
              (identifier-table-put!: token-table
                                      (car tokens)
```

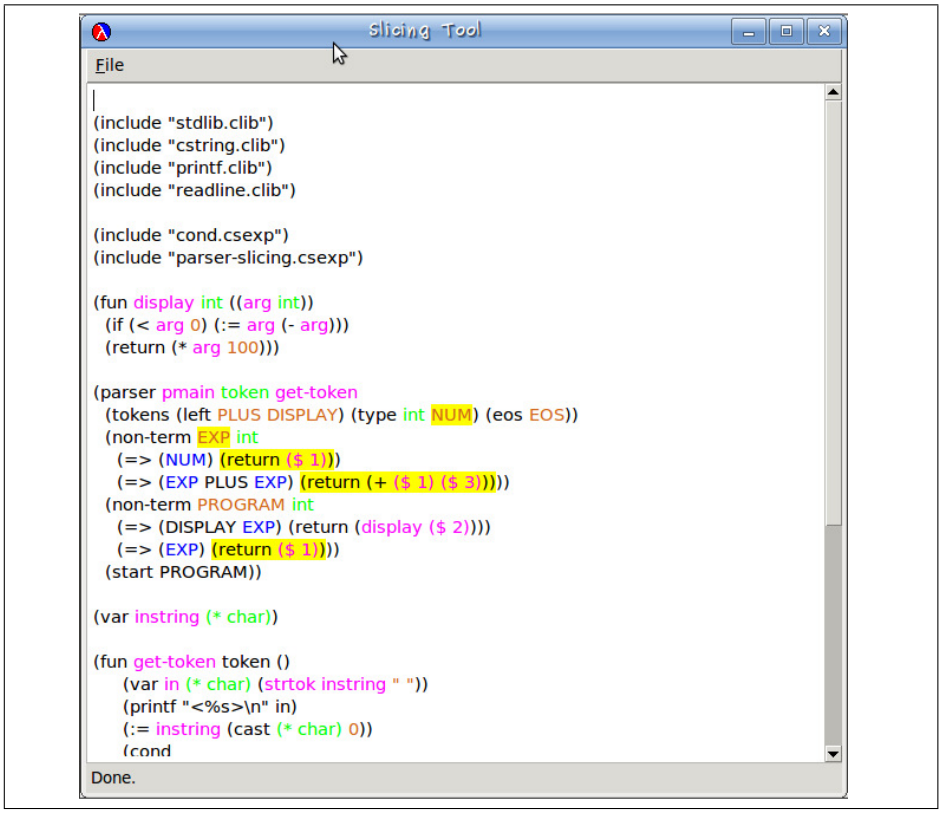

Figure 10.3: The result of slicing a parser via a specialized method.

```
                                        (d! (car tokens)))
          (loop (cdr tokens)))))
(let loop ((decls (parser-parser-decls: parser^)))
  (if (pair? decls)
      (all
       (parser-dependence: (car decls) token-table
                           current-control)
       (loop (cdr decls)))
      (let loop ((non-terms
                  (parser-non-terminals: parser^)))
        (if (pair? non-terms)
            (all
             (<- (d! (car non-terms))
                 (U (d! (car non-terms))
                    (set (car non-terms))))
             (loop (cdr non-terms)))
            (let loop ((tokens
                        (parser-tokens: parser^)))
```

```
34                    (if (pair? tokens)
35                        (all
36                         (<- (d! (car tokens))
37                             (U (d! (car tokens))
38                                (set (car tokens))))
39                         (maps (lookup-function read^)
40                               (lambda (fn)
41                                 (<- (d! (car tokens))
42                                     (U (d! (car tokens))
43                                        (return-var: fn)))))
44                         (loop (cdr tokens)))
45                        (<- (lookup-function name^)
46                            (U (lookup-function name^)
47                               (d! (parser-start:
48                                    parser^))))))))))))))
```

- Lines 3-17: we define a table, called `token-table`, that maps terminals and non-terminals to their corresponding Tsuriai variable.
- Lines 18-23: we go through the declarations of the parser, generating dependence information for each. The method to do so is below.
- Lines 24-31: each non-terminal is dependent on itself.
- Lines 32-44: each terminal is dependent not only on itself, but on the return value of the read function.
- Lines 45-48: finally, the return value of the parse function generated by the parser macro is dependent on the value of the start token of the grammar.

The dependency information of a parser declaration is calculated by the generic function `parser-dependence:`.

```
1  (define-method parser-decl-non-term% parser-dependence:
2    (lambda (this token-table current-control)
3      (let* ((new-control
4              (object control-object%
5                      (d! this) #f #f #f
6                      this
7                      (make-identifier-table))))
8        (let loop ((productions productions^))
9          (if (pair? productions)
10             (all
11              (parser-dependence: (car productions)
12                                  token-table new-control)
13              (loop (cdr productions)))
14             (let ((nt-var
15                    (identifier-table-get: token-table name^)))
16               (<- nt-var (U nt-var (return-var: this)))))))))
```

This method performs three actions of note.

- Lines 3-7: we first set up a new `control-object%`, specifying that this current function being analyzed is the "this" object. We will use this while analyzing semantic actions, which may include `return` statements and thus need to know where the value being returned will flow to.
- Lines 8-15: we loop over each production, and generate dependency information for each.
- Line 15: we assert that the Tsuriai variable for the non-terminal on the left-hand-side is dependent on the return values of the semantic actions.

Generating dependency information for a production is a matter of going through the right-hand-side of the production, and gathering the Tsuriai variables of the terminals or non-terminals into a list. We store this list in a new object of type `parser-control%`, that delegates to the current control object.

```
1  (define-method parser-production% parser-dependence:
2    (lambda (this token-table current-control)
3      (let ((rhs-control
4             (let loop ((rhs rhs^))
5               (if (pair? rhs)
6                   (cons (identifier-table-get: token-table
7                                                (car rhs)))
8                         (loop (cdr rhs)))
9                   '())))))
10       (csexp-dependence: action^
11                          (object parser-control% rhs-control
12                                  current-control)))))
```

Finally, we can generate dependency information for the semantic action. This involves taking the Tsuriai variables of the current production as generated above, associating them with their corresponding C/sexp variables, and generating dependency information for the C/sexp code.

```
1    (define-method csexp-parser-action% csexp-dependence:
2      (lambda (this current-control)
3        (let loop ((rhs (parser-control-rhs: current-control))
4                   (n 1))
5          (if (pair? rhs)
6              (all
7               (let ((val-var (lookup-variable
8                               (value-table-get value-table^ n))))
9                 (<- val-var (U val-var (car rhs))))
10              (loop (cdr rhs) (+ n 1)))
11              (csexp-dependence: stm^ current-control)))))
```

This gives us all of the machinery we need to provide the smaller slices we have seen in the previous section, and will see in the next.

10.8 A larger example

The example in the previous section demonstrated the basic principles behind slicing of parsers. However, to see the true extent to which conservative slicing provides too large of a slice, we must consider a somewhat larger example. The code presented in appendix B takes a language for a four-operation calculator, and produced three-operand pseudo-assembly code.

Consider slicing on the semantic action of the production right-hand-side `(=> (NUM) (return (generate-constant ($ 1)))`. The conservative slicer will not be able to trace through the parser, and give us a slice consisting of the entire program! We can do much better. After all, the semantic action for a production only consisting of a number does not depend on a function for generating a two-operand instruction.

The augmented slicer gives us a much tighter slice, seen in figure 10.4. As opposed to the previous slice which contained 150 lines, we now have merely 30. We have managed to reduce the slice to a fifth of its previous size. Calculating precise flow analysis can be very difficult, and in this example, we have done it easily. The advantage of Ziggurat is not that it provides better flow analysis, but rather that it allow the language designer to use what they already know about the language they are designing in order to provide better static semantics. The algorithm to analyze control flow of a parser is easy and straightforward, and Ziggurat allows us to implement it easily and use it to provide better control flow of any program containing a parser.

```
;;; generate a constant.
(fun generate-constant compile-state ((value int))
  (var to-return compile-state)

  (:= (fld to-return value) (itoa value))
  (:= (fld to-return listing) (null))
  (return to-return))


;;; parser parse-calc returns a compile-state.
(parser parse-calc token get-token
  (tokens (left MUL DIV ADD SUB) LPAREN RPAREN
          (type int NUM) (eos EOS))
  (non-term NUM-EXP compile-state
    (=> (NUM) (return (generate-constant ($ 1)))))
  (start PROGRAM))


;;; a simple lexer.
(var instring (* char))

(fun get-token token ()
  (var in (* char) (strtok instring " "))
  (:= instring (cast (* char) 0))
  (cond
   ((== in (cast (* char) 0)) (return (EOS)))
   ((== 0 (strcmp in "+")) (return (ADD)))
   ((== 0 (strcmp in "-")) (return (SUB)))
   ((== 0 (strcmp in "*")) (return (MUL)))
   ((== 0 (strcmp in "/")) (return (DIV)))
   ((== 0 (strcmp in "(")) (return (LPAREN)))
   ((== 0 (strcmp in ")")) (return (RPAREN)))
   (else (return (NUM (atoi in))))))
```

Figure 10.4: A minimal slice of the four-operation calculator compiler.

Chapter 11

Topsl

In recent years, domain-specific languages have found increased use in programming services for the web [8]. One such domain-specific language is Topsl [61], a language for writing web-based psychological surveys, and a language that is prime for implementation in Ziggurat. Topsl, in its published form, is implemented as an embedded language within Scheme, parameterized by Scheme in much the same way the Wisent language is parameterized by a semantic action language; thus, we will not be implementing Topsl in its published form, but instead, a variant of Topsl embedded within C/sexp.

A simple survey to query users on how much they enjoyed various restaurants might be written:

```
(survey
 (page
  (free times
"How many restaurants have you visited in the last month?"))
 (cpage (topsl-var times-raw times)
        (var times int (atoi (topsl-val times-raw)))
        (for (var i int 0) (< i times) (post+ i)
          (free restaurant-name "Name of restaurant")))
 (cpage (topsl-var names restaurant-name)
        (while (! (topsl-end? names))
          (topsl-bind single-restaurant-name
                      (topsl-val names))
          (likert taste "I liked the cuisine at "
                  single-restaurant-name)
          (:= names (topsl-next names)))))
```

This survey asks the user how many restaurants he or she has visited in the last month. Based on this answer, he or she is presented with a number of free-response spaces and asked to name the restaurants. For each named restaurant, the survey asks the user to rate the cuisine on a 7-point scale.

Topsl programs have a unique piece of static semantics: the codebook. Researchers must know all of the questions that might be asked during a survey, before that survey is run, so that statistical analysis packages will be able to analyze the results. In Topsl, the codebook of a particular survey is an XML document containing all of this information. The static semantics of Topsl, then, includes a generic function `topsl-codebook:`, that, given a survey, returns just such an XML document. The codebook for the above example would be:

```
<query type="free" key="times">
  <text>
How many restaurants did you visit in the last month?
  </text>
</query>
<query type="free" key="restaurant__name">
  <text>
Name of restaurant
  </text>
</query>
<query type="likert" key="taste">
  <text>
I liked the cuisine at
  </text>
  <field key="single__restaurant__name">
</query>
```

11.1 Structure of the Topsl language

The grammar of Topsl is presented in figure 11.1. A Topsl program consists of a single survey. Portions of the survey may be generated at survey-time, and to allow this, Topsl surveys may include C/sexp code. Running a survey binds Topsl variables to values. Each query has a variable name that it binds, and multiple queries can bind the same values. However, when the results of a survey are reported, the answers to all queries are represented, even if some bind the same variables.

A Topsl `survey` declaration contains a sequence of survey pages. In the resulting survey, these will be presented to the survey-taker in the order they appear in the survey source. The appearance of later pages may depend on the answers to questions in earlier pages.

A Topsl *page* contains a collection of queries which are presented to the user simultaneously. There are four kinds of page declarations: ordinary pages, sequences, code pages and C/sexp pages. Ordinary pages, declared with the `page` keyword consist of a name, followed by a sequence of queries. A sequence declaration `(seq` *p* `...)` allows the programmer to introduce a number of pages. A code page `(code` *s* `...)` includes a sequence of C/sexp declarations that may be used in later C/sexp pages. A code page does not actually generate a survey page. A C/sexp page `(cpage` *s* `...)` contains a

```
v ∈ Survey          ::= (survey p ...)
p ∈ Page            ::= (seq p ...)
                      | (page q ...)
                      | (cpage s ...)
                      | (code s ...)
q ∈ Query           ::= (likert i l ...)
                      | (free i l ...)
l ∈ Query-Element   ::= c
i                     |
c ∈ String
s ∈ Statement       ::= ...
                      | q
                      | (topsl-bind i e)
                      | (topsl-var i i)
                      | (topsl-val e)
                      | (topsl-end?  e)
                      | (topsl-next e)
```

Figure 11.1: The grammar for Topsl.

series of C/sexp statements that generate a sequence of queries that are presented to the survey-taker.

A *query* is a single question presented to the survey-taker. A query has a type (either Likert or free-response), a Topsl variable to bind, and a sequence of query elements. These query elements will be evaluated to strings, and then concatenated before being presented to the survey-taker.

There are two basic query types: Likert queries and free-response queries. A Likert query, developed by Rensis Likert [60], presents the user with a statement, and gives the user seven possible responses to judge their agreement with that statement, "strongly disagree," "disagree," "somewhat disagree," "neither agree nor disagree," "somewhat agree," "agree," or "strongly agree." The value of the bound variable is an integer between 0 and 6, with 0 indicating "strongly disagree" and 6 indicating "strongly agree."

The other type of query is a free-response query. A free-response query simply gives a user an entry blank for entering a freeform response. The variable in this sort of query is bound to a string representing the survey-taker's answer.

A query *element* is either a string, or a Topsl variable. Topsl variables are usually bound by previous queries, although they can be bound in embedded C/sexp code.

11.2 C/sexp embedding

Defining a `cpage` allows the survey-writer to insert C/sexp code to generate a page into the survey. The language of a `cpage` is C/sexp augmented with new statements to allow linkage with Topsl.

- `(likert` *i l* `...)` `(free` *i l* `...)` Evaluating a `likert` or `free` statement adds a likert-style or free-answer query into the page. The syntax of such a query in a `cpage` is identical to queries in a `page`. As a side effect of this syntactic similarity, in the absence of syntax declarations, ordinary pages are a subset of `cpages`. The Topsl page

```
(page (free name "What is your name?")
      (free quest "What is your quest?")
      (free color "What is your favorite color?"))
```

  is semantically equivalent to

```
(cpage (free name "What is your name?")
       (free quest "What is your quest?")
       (free color "What is your favorite color?"))
```

  Part of the purpose of a `cpage` is to allow pages to be generated based on the answers to previous survey question. This is done by further new C/sexp syntax that binds survey responses to C/sexp variables, and C/sexp values to Topsl variables.

- `(topsl-bind` *i e*`)` A `topsl-bind` statement binds the C/sexp value expressed by the expression *e* (which should evaluate to a string) to the Topsl variable *i* in the remainder of the `cpage`.

- `(topsl-var` i_1 i_2`)` A `topsl-var` declaration binds the responses indicated by the Topsl variable i_2 to the C/sexp variable i_1. There may be several survey questions that bind the Topsl variable i_2, and thus i_1 is bound to an iterator, similar to a Scheme list of strings. Further syntax is required to extract individual survey responses.

- `(topsl-end?` *e*`)` A `topsl-end?` expression evaluates to a boolean value indicating whether there are survey responses remaining in the response iterator that *e* evaluates to, similar to `null?` in Scheme.

- `(topsl-val` *e*`)` If there are survey responses remaining in the iterator that *e* evaluates to, `topsl-val` returns the (string) value of the first, similar to `car` in Scheme. It is an error to apply `topsl-val` to a completed iterator.

- `(topsl-next` *e*`)` If there are survey responses remaining in the iterator that *e* evaluates to, `topsl-next` evaluates to the iterator containing the remainder of the responses, similar to `cdr` in Scheme.

These new syntax classes are implemented as rewriting extensions on top of C/sexp. However, the language of a `cpage` is different from C/sexp: it adds a new piece of static semantics, the codebook.

11.3 Topsl codebook

A Topsl codebook is an XML summary of all of the queries that may be asked during a survey. Although queries can be generated dynamically, Topsl query labels and other query elements are static, and thus can be enumerated by a simple scan of the source code.

Codebooks are generated by means of an attribute, `topsl-codebook:`. When called on a survey fragment, this attribute returns a string containing the portion of the codebook corresponding to that fragment.

Invoking `topsl-codebook:` on the query `(free restaurant-cuisine "How would you describe the cuisine at " restaurant-name)` produces the codebook fragment

```
<query type="free" key="restaurant__cuisine">
  <text> How would you describe the cuisine at </text>
  <field key="restaurant__name">
</query>
```

Likewise, invoking the generic function `topsl-codebook:` on the query `(likert restaurant-rating "I enjoyed the food at " restaurant-name)` produces the codebook fragment

```
<query type="likert" key="restaurant__rating">
  <text> I enjoyed the food at </text>
  <field key="restaurant__name">
</query>
```

The effect of invoking `topsl-codebook:` on a `survey`, `seq` or `page` is straightforward. A `page` invokes `topsl-codebook:` on each enclosed query, and returns the concatenated result. Invoking `topsl-codebook:` on a `survey` or a `seq` is similar: it invokes `topsl-codebook:` on each enclosed page and returns the concatenated result.

Invoking `topsl-codebook:` on a `cpage` is somewhat more complicated. The embedded C/sexp code can have Topsl queries embedded within it, and those queries must appear in the codebook. In the absence of macros, a straightforward walk of the source code will reveal all of the embedded queries. However, to allow macros, we must extract the queries via generic function, and thus we change the static semantics of the embedded C/sexp: we add an attribute, `embedded-queries:`, that, when called on a C/sexp statement, will return a list of embedded queries. This attribute should be evaluated during static-analysis time.

11.4 Macros in Topsl

The Topsl language is implemented in Ziggurat, which makes it easily extensible. There are two primary ways programs in Topsl can make use of language extensions:

- Since C/sexp is embedded within Topsl, Topsl programs can make use of C/sexp extensions transparently.[1]
- Topsl allows macro definitions that expand into pages or queries.

11.4.1 C/sexp macros embedded in Topsl

Language extensions that do not modify the C/sexp language work in Topsl without change. For example, the survey

```
(survey
 (code (include "list.csexp")
       (var users (list (* char)) (null)))
 (page (free username "What is your name?"))
 (cpage (topsl-var name username)
        (for (var i (list (* char)) users)
             (! (null? i))
             (:= i (tail i))
          (topsl-bind compared-user (head i))
          (likert likeness "I like the user " compared-user)
          (:= users (make-list (topsl-val name) users)))))
```

contains an occurrence of the list macro of section 7.8.3, and behaves as it should. In addition, since the generic function `csexp-topsl-queries:` delegates to rewritten C/sexp, it gives the correct codebook

```
<query type="free" key="username">
  <text> What is your name? </text>
</query>
<query type="likert" key="likes">
  <text> I like the user </text>
  <field key="compared__user">
</query>
```

By simply re-using the C/sexp language infrastructure, we get language extension in the embedded language for free. However, the new language has new syntax categories, and this gives us new opportunities for extension.

11.4.2 Topsl macros

We modify the Topsl language to allow macros that expand to Topsl pages and queries. Just as in C/sexp, where we added a new `syntax` form, here we add new `define-page` and `define-syntax` forms.

[1] As noted earlier, the language embedded within Topsl is not exactly C/sexp, due to the added static semantics. However, the new generic function takes place after code generation, which means that C/sexp macros by-and-large work transparently.

$p \in$ Page ::= ...
| `(define-page` *i z*`)`
| `(define-query` *i z*`)`

The implementation of `define-page` and `define-query` present no innovations over the extension mechanisms we have already seen.

Topsl has very few primitive forms. The embedded C/sexp can be used for most of the control and data effects we need; however, the mechanisms to manipulate Topsl data are very limited. We can remedy this with language extension. For example, we might want a `loop` form to iterate over all of the bindings of a particular Topsl variable.

```
(define-page loop
  (lambda (env form)
    (syntax-case form ()
      ((loop v key code)
       (parse-topsl-page: env
         (syntax (cpage (for (topsl-var raw key)
                             (! (topsl-end? raw))
                             (:= raw (topsl-next raw))
                        (var v (* char) (topsl-val raw))
                        code)))))))))
```

The form `(loop` *v key c*`)` binds each string value bound to the Topsl variable *key* to the C/sexp variable *v* in the C/sexp code *c*. With this form defined, the survey at the beginning of this chapter can be rewritten

```
(survey
 (page
  (free times
"How many restaurants have you visited in the last month?"))
 (cpage (topsl-var times-raw times)
        (var times int (atoi (topsl-val times-raw)))
        (for (var i int 0) (< i times) (post+ i)
          (free restaurant-name "Name of restaurant")))
 (loop names restaurant-name
  (block
    (topsl-bind single-restaurant-name (topsl-val names))
    (likert taste "I liked the cuisine at "
            single-restaurant-name))))
```

Since the Topsl codebook is generated via generic function, the codebook generated by this survey containing macros is exactly the same as its macro-free counterpart.

Chapter 12

Related work

The subject of language extension has recently been of interest to a number of projects, due in part to the proliferation of domain-specific languages for Web services. Ziggurat's design has been influenced by several of these related projects.

12.1 Metaprogramming

Some recent work in domain-specific macros has been in "metaprogramming," the development of tools related to source-to-source program translation. A successful metaprogramming tool in wide use today is Stratego [102], a program-to-program source code translation toolkit based on term-rewriting systems.

Stratego allows one to design a domain-specific language, and specify its dynamic semantics by rewriting programs into an underlying language, such as Java. Thus, Stratego makes it easy to write source-to-source compilers, which is similar to the functionality presented by macro systems. An advantage of Stratego over Ziggurat and other Scheme-like macro systems is that Stratego allows for non s-expression-based grammars. However, Stratego does not provide tools for semantic extension. In order to implement a type system for a domain-specific language designed with Stratego, the source-to-source compiler would first check the types of the source program, translate the program text, and then allow the compiler of the underlying language to perform its own type checking. It is not possible to link the static semantics of the domain-specific language to that of the underlying language, as in Ziggurat. This complete separation of language semantics is good for translating languages that have dissimilar semantics, such as translating from a database-query language to a high-level language, but does not work as well for finer-grain language extension, such as the `cond` language extension presented earlier.

Another difference between Stratego and Ziggurat is that Stratego is a tool for full-program transformation, and does not allow languages to be embedded, as in macro systems. The MetaBorg tool [11], which is built on top of Stratego, does allow languages to be embedded. For example, one could implement a regular-expression language or a GUI-specification language inside of Java in MetaBorg—tasks one might ac-

complish with a macro system. However, language embedding is still coarser-grained than syntactic extensions such as `cond`, which are possible with macro systems. In MetaBorg, it is not possible for source-language terms to appear in the same syntactic context as embedded terms. Ziggurat aims to augment the syntactic extensions provided by macro systems with corresponding semantic extensions, whereas MetaBorg aims to allow languages with dissimilar static semantics to be embedded.

A system with a similar approach to Stratego's is Metafront [9], a tool for specifying languages with the goal of making them extensible. Metafront allows a language extension to specify how it adds to the grammar of a language, and then to provide rules to transform programs in the extended language to the base language. Metafront, like Stratego, is a full-program transformation toolkit, without semantic extensibility. In addition, the transformation rules are not Turing-complete, which guarantees that the compiler will terminate, but does not allow the power of full low-level macros.

An alternate approach to metaprogramming is multi-stage programming, implemented in MetaML [96]. MetaML takes many ideas from macro technology. In MetaML, syntax is manipulable data, similar to the way macros translate syntax. Programs thus can output programs, which, in turn, can output other programs, providing staged compilation similar to macros. Semantic analysis is possible through a form of reflection: variables defined in one compilation stage are available in later stages. However, as in Stratego, the focus is on full-program transformation, unlike the local syntactic extensions available with macro systems.

Nanavati [71] also provides a system for extensible analysis. However, Nanavati does not provide an object-oriented interface to his syntax. In order to implement significant syntactic extensions in his system, the programmer is forced essentially to simulate an object-oriented architecture programmatically, which makes for a fairly awkward, tortured coding style. Ziggurat captures this structure directly in the linguistic mechanisms of the meta-language.

12.2 Macro systems

The implementation of syntax as objects, first introduced by Dybvig, Hieb and Bruggeman [38], opens up numerous possibilities.

The analysis of macros themselves is an important related topic. Programs containing macros can complicate tasks that are simple in languages that do not contain them. For example, macros complicate separate compilation: if one module depends on another module, it may depend on macros found within that module, meaning that compilation of the first module will require including the source of the second. Flatt [31] has developed a method of mixing the specification of macros with the specification of modules, allowing modules to be partially imported at compile-time, as necessary for macros, and thus allowing for separate compilation. The analysis of the interaction of macros and modules is an important step in applying macro technology to a language with a module system, one that is not covered by this work.

These compilable macros are implemented in the macro system currently employed in MzScheme. MzScheme uses syntax objects; an earlier version of the macro system, called McMicMac [54], performed semantic analyses on syntax. Ziggurat draws many

ideas from McMicMac. However, while Ziggurat attempts to be a general-purpose language toolkit, McMicMac is designed exclusively for Scheme, only dealing with the limited static semantics available in Scheme.

Others have mixed hygienic macros with languages with static semantics, without providing an explicit method of accessing those semantics. The language Dylan [84] contains a hygienic macro facility, without such a mechanism. The Java Syntax Extender [3] adds a hygienic macro mechanism to Java, again with no means to access the static semantics of the language. Even without such a mechanism, these systems are adequate for many sorts of language extensions.

The Maya macro system [4], on the other hand, allows macros to access the types of subexpressions. It does this by performing macro expansion in stages, delaying expansion until the types of subexpressions are available. This is similar to the laziness of lazy delegation, but it does not allow macro writers to specify their own static semantics.

12.3 Attribute-grammar approaches

A popular method of specifying static semantics is via attribute grammars. Another approach to mixing language extension and static semantics is to augment semantics specified via attribute grammars with metaprogramming facilities.

The JastAdd extensible Java compiler [27, 26] uses attribute grammars to enable tree rewriting, similar to the Stratego language-extension tool. Like Stratego and other metaprogramming tools, JastAdd is based on rewriting one full language to another, making it ideal for incrementally building a compiler.

Silver [101] is a language for describing programming-language grammars and their associated static semantics with the goal of making these languages extensible. An application of this system is the Java Language Extender framework, a tool for writing extensions to the Java language, both fine-grained, such as new loop constructs, and coarse-grained, such as embedding domain-specific languages.

Silver is based on an attribute-grammar framework. A language is described as a number of syntax productions. Each production defines a number of attributes, compile-time values which are calculated from the attributes of its ancestors in the abstract syntax tree (“inherited” attributes) and its children (“synthesized” attributes). Each non-terminal defines a fixed set of attributes to be calculated for each production of that non-terminal.

In Silver AG, the language extender is only required to provide definitions for a subset of the attributes defined on a production. This is enabled by a feature called forwarding [100]. Productions can have a special attribute, called its forwards-to attribute, which is similar to the delegate in Ziggurat. To calculate the value of any attribute not defined on a production, Silver AG calculates the production’s forwards-to attribute, which should return a syntactic node, and then delegates to this node, similar to how lazy delegation deals with generic functions on objects with undefined methods.

The difference between Silver and Ziggurat involves how they deal with the interleaving of parsing and analysis. Both allow this, quite explicitly. In Silver, the forwards-to attribute is an attribute like any other, and thus can be calculated from the

attributes of the node's ancestors and descendants. Since attributes are calculated based on the values of other attributes, the order of evaluation of attributes is very important, and is determined via data dependence; the attribute specification is purely functional, and thus expansion can be delayed until it is required. Thus, if the forwards-to attribute depends on other attribute values, Silver will perform expansion after other semantic analyses.

Ziggurat allows this behavior by means of the laziness in lazy delegation. Ziggurat uses only simple data-flow to order the computations defined in the meta-language: it does not perform expansion until it is needed by a generic function being applied to a syntactic node lacking a method for that function. This puts the burden on the language designer to prevent expansion from taking place until the information necessary for it is available, but allows expansion in cases where it may be difficult to automatically determine the order in which things should be evaluated. We have found this linguistic on-demand mechanism fairly natural and intuitive to use.

Ziggurat's ability to delay expansion using lazy delegation is an important design element of the system. Sophisticated macros that mediate shifts between language levels can be written to exploit static-semantics computations performed at the higher language level to drive the rewrite into the lower language level. For example, the expander (*i.e.*, the delegation method) can perform type-dependent translation, or use the results of a flow analysis to select between translation strategies.

Consider implementing named, globally scoped constants in the Arith language. An example use of this feature might look like:

```
(with-constants
  (* (+ (constant x) (constant y))
     (define-constant (x 3)
       (define-constant (y 4)
         5))))
```

In the expanded code, we would like there to be an outermost let-binding for each constant being defined. The above code would look like:

```
(let (x 3)
  (let (y 4)
    (* (+ x y) 5)))
```

Although this example may seem a bit contrived, it is a simplification of something done frequently with macros: changing variable scope rules, something often made inaccessible by languages without metaprogramming facilities. Also, although it may not seem like it, this macro expansion requires a little bit of global analysis: in order to expand the constant-defining `with-constants` macro, the macro expander must first determine what constants are in the subsequent program.

Implementing this in Ziggurat is simple enough. The `with-constants` syntax object can define a field `constants^` intended to contain all of the constants in the subsequent program. Upon parsing a `constant` or `define-constant` form, the parse function adds the constant to the `constants^` field if it is not already in there.

This would not be difficult to implement in Silver AG, either. However, the design would be somewhat different. We would define a new attribute, `constants`, that would be a list of all of the constants in a program, and a list of the new identifiers they map to. The forwards-to attribute of the `with-constants` syntax node would then refer to the `constants` attribute of the subsequent program, and the forwards-to attribute of the `constant` syntax node would refer to the `constants` attribute of the `with-constants` node.

The advantage of the Ziggurat approach is in composability. When two attributes as in the Silver approach are combined, there is the possibility of circular dependency. Consider defining the same macro twice, with slightly similar name.

```
(with-constants
  (with-konstants
    (* (constant x) (konstant y)
       (define-constant (x 3)
         (define-konstant (y 4)
           5)))))
```

In order to observe this problem, the two macros need not be the same, but both must need to define a new attribute and refer to it in their forwards-to attribute. In this case, in order to calculate the `constants` attribute of the `konstant` node, the following attribute computations occur:

- Since the `konstant` node does not define the `constants` attribute, it must forward.
- Forwarding a `konstant` node requires calculating the `konstants` attributes of the `with-konstants` node.
- This requires calculating the `konstants` of the entire program, and specifically, the `constant` node.
- Since the `constant` node does not define a `konstants` attribute, it must forward.
- Forwarding a `constant` node requires calculating the `constants` attributes of the `with-constants` node.
- This requires calculating the `constants` of the entire program, which is where we started.

Forwarding is based on an earlier language-extension tool, called XL [62]. In XL, syntax nodes are parsed into records containing the attributes of that node. New static semantics are defined by extending these records, using a subtype-based extensible record facility built into the language. However, these records are not composable, limiting the macro designer to a linear tower of languages; macro packages cannot be combined in the same program. These extensible records have associated with them a subtyping relation on sets of attributes, similar to the attribute-grammar inheritance systems found in the LISA tool [67].

12.4 Semantic extension

An important related field of research is projects that have looked into semantic extension of languages, independent of macros. The JavaCOP project [2], for example, looks at extending Java's type system with type constraints. However, JavaCOP's syntactic mechanism only allows programmers to annotate their programs, instead of providing new type constructs. An interesting piece of future work for Ziggurat would be to see if lazy delegation could provide better syntax for JavaCOP's type annotations.

The J& project [72] establishes a language feature called "nested intersection." Not only is nested intersection an interesting language feature to implement via macros, but an application of their work is in providing fine-grain compiler extensibility through method inheritance. It is a possible future direction for Ziggurat research to explore combining nested intersection with lazy delegation to provide better, more terse descriptions of static semantics.

Chapter 13

Conclusion

In this dissertation, we have seen the basic structure of Ziggurat, and then demonstrated its utility via implementing and extending several languages.

- **Arith** is intended as a simple demonstration language, to show the basic function of Ziggurat. In Arith, we see the essential structure of a language, and how it is handled in Ziggurat:

  - Concrete syntax: the languages we design in Ziggurat are all s-expression-based. Parsing them is done by a slight modification of the methods used to parse Scheme.
  - Abstract syntax: the abstract syntax of Arith consists of a collection of classes, approximately one for each production in the language.
  - Static semantics: for Arith, we provide a small collection of generic functions that allow us to determine semantic facts about our program. As we extend our language, we have the option of writing specialized methods for these functions.

  Although this structure is simple, as we expand it to more complex languages, it will scale to more complex semantics, and the more complex tasks that those semantics are put to.

- **C/sexp** is a general-purpose language that provides the basis for language extensions and embeddings. Unlike Scheme, C/sexp presents challenges to a hygienic macro system handled by Ziggurat:

  - Multiple namespaces: hygiene mechanisms designed for Scheme's single namespace are not by themselves suited for C's syntactic landscape, where an identifier can have several roles simultaneously. The modifications presented in section 5.3 allow us to navigate this more complex landscape, without compromising the benefits of hygiene.

  - Static semantics: for several of the macros we write, we wish to modify C's static semantics in ways not easily provided by rewriting. For example, our `cond` macro needs to have specialized typing-error messages. Lazy delegation allows us to do this by writing custom methods for the generic functions for error checking. As another example, the list macro provides a slight modification to C/sexp's type system. By writing custom methods for the generic functions which constitute the type system, we can add these types to the system.
  - Type-directed translation: conversely, some macros need to have their rewrite methods dictated by the underlying semantics. For example, the list macro needs to produce data definitions and functions based on the type context in which it is used. In order to do this, we allow the delegate instantiation method to call semantic functions on other syntax nodes.

  In addition, we must ensure that macros work well together. For example, the `cond` macro relies on the type system to provide intelligent error messages. It must be the case that when the list macro adds to the type system, it does not break the error checking provided by the `cond` macro. This is achieved through the polymorphism of the object system: since the new types respond to the same generic functions as the base types, type-checking occurs correctly in either case.

- **Aurochs** and **Wisent** are domain-specific languages for a well-understood but specialized domain: parsers for LALR(1) languages. These present difficulties that are well-handled by Ziggurat:

  - A new namespace: Wisent introduces a new kind of name for terminals and non-terminals. This is completely different from any of the names in the underlying language, and should present no difficulty when the same identifier is used to represent, say, a C/sexp variable and a Wisent terminal. This is well-handled by the mechanism presented in section 5.3.
  - Language independence: both Aurochs and Wisent are not tied to any one underlying language. Their concrete syntax, abstract syntax and static semantics are not defined in terms of rewritten code. Defining a rewriting mechanism only occurs when the parser languages are embedded within another language, making it relatively easy to relocate them to another underlying language.
  - Semantic-value embedding: in addition to being embedded within another language, both Aurochs and Wisent have languages embedded within them, to provide semantic actions. The semantic-value language is the same as the underlying language. As a side-effect of this, macros defined in the underlying language work seamlessly in the semantic-value language, even if those macros define new static semantics.
  - Language-specific macros: We can define a macro system specifically for the parser languages. This means that not only can we create macros that expand into parsers, like in the case of the regular-expression language, but

we can write macros that expand into parts of parsers, like the macro for comma-separated lists.

- **Topsl** is a domain-specific language for a domain greatly afield of language design: designing custom web-based surveys for psychological experiments. Topsl demonstrates that the benefits of Ziggurat translate well into different domains, particularly:
  - Custom semantics: Topsl has a unique piece of static semantics, the codebook. A codebook is useful for statistical analysis, and must be determined uniquely for each survey. Ziggurat defines a method to build a codebook for an arbitrary survey, even in the presence of macros.
  - Language-specific macros: Ziggurat allows the programmer to write macros that expand into either complete surveys, or pieces of surveys.
  - Language embedding: in the same manner as Aurochs and Wisent, Topsl allows the programmer to embed code in the underlying language into the Topsl program. Because of the structure of Ziggurat, once again, macros in the underlying language work in the embedded pieces without modification, even if they define custom semantics.
- In addition, this dissertation explored semantics-based program analysis at several stages of the languages tower: **slicing**. Slicing is a dataflow-based analysis for debugging. Although slicing of C/sexp programs is computationally expensive and conservative, it is possible to make slicing more precise for the parser languages. Thus, when slicing a program that makes use of a parser, handling slicing via rewriting as would be done in most conventional macro systems is both slow and imprecise. However, in Ziggurat we can conquer both of these difficulties simply by writing specialized semantic methods for parsers.

The challenges tackled above are a direct result of the simple fact that languages have static semantics. Ziggurat addresses these challenges by adding static semantics to macros.

Chapter 14

Future work

The potential scope of Ziggurat is wide, and there are many possible avenues one could explore going forward. Our focus is currently on the following.

- **Certification.** Ziggurat permits syntax implementors to intercept and override analysis requests with their own methods. This is a key source of power for the system, but it is a two-edged sword: overloaded analyses, if they are buggy or malicious, can give wrong results, possibly leading to unsafe compilation. Since Ziggurat is intended as a general language-extension toolkit, this is unavoidable in the general case. A possible solution, though, is to require that certain analyses provide a certificate that their result obeys certain properties, *e.g.*, that they give a correct answer (of possibly varying precision), or that they do not violate safety rules. Certification in this scheme becomes just another analysis.

  In general, if a lower-level language includes checkably sound annotations expressing the support for a given assertion, then an analysis performed at a higher level in the tower can be expressed by expanding into annotated code in the lower-level language. Since these annotations will be checked by the static-semantics elements of the lower-level language implementation, there is no possibility of a buggy or malicious macro asserting a false claim undetected.

- **Further languages.** The Topsl language suggests a powerful avenue for further languages to explore in Ziggurat. Topsl presents an abstraction that is stateful, and translates it to a stateless service. This sort of translation happens naturally during macro expansion, which in turn suggests that Ziggurat may be very useful for implementing **"little languages" for web services.** There are a number of language-design paradigms that could be implemented—and combined—with Ziggurat.

  - **Stateless programming.** It is a central mantra that code written for the web be *stateless*: that is, it not maintain a "current state" for the client. However, saving and restoring information between each interaction is a tedious task for the programmer, and in fact, leads to a tortured programming style,

where steps that are sequential in the programmer's mind must be broken up, so that they can be executed in any order. It would be much easier to present a stateful abstraction to the programmer—such as an ordinary, high-level programming language like C/sexp—and have the language implementation translate code into a stateless style. This transformation can be done via CPS, defunctionalization, and lambda lifting [64] [19]. When implemented as a language embedding, this allows further language abstractions to be built on top of the stateless platform, such as the other abstractions in this list.

– **Tierless Programming.** Web applications consist of several *tiers.* The server-side must perform the work of the application, while the client-side has a user-interface program. In addition, the server may interact with a database, or may request information from remote sources: in fact, it is possible to build web applications entirely out of these service combinators [15]. In addition to coding the multiple tiers of the web application, the programmer must also code the middleware necessary to transfer information among them. This can be difficult, especially since different tiers may be in different languages; the user-interface software, for example, is typically implemented in Javascript. A solution to this in use today, is to allow the programmer to code a single "tierless" application, and then split it into multiple tiers through program transformation [18]. This program transformation could be implemented as a Ziggurat embedding, and could be combined with other language extensions.

– **Functional reactive programming.** Modern web applications expect a lot of their user interfaces: for example, a social networking site sign-up application might have to verify that a user name is available, a search form might have to offer suggestions based on partial input, and a mapping application might have to update location information as the user moves a marker around the map. A language paradigm based on keeping in sync multiple sources of information that change in real time is *functional reactive programming* [41]. Functional reactive programming has already been used for client-side web programming [68], implemented by translation into Javascript. It would be interesting to see how reactive programming would mix with other language paradigms present on this list, such as tierless programming: it is conceivable that a reactive program with a behavior defined on the server side, such as an active chat channel, would work when interacting with a client-side behavior consumer, such as a text box intended to display the chat.

– **Protocol specification.** Most web programmers do not need to specify a protocol; usually, "off-the-shelf" protocols will do nicely. However, there are times when applications require specialized protocols, for tasks such as authentication as in OpenID [82]. Sheehy and Pucella [85] have identified a problem mixing the way protocols are typically specified and a web-based infrastructure. Protocols are typically specified as bi-directional message-based conversations; this is known as a *transport model.* However, web

services communicate well by requesting resources that may be cached; this is known as a *resource model.* They propose a program transformation that inputs a protocol specified in a transport model, and outputs an equivalent protocol in a resource model. If this transformation were implemented as a macro, it would be possible to specify a protocol tersely within a web service that already uses a number of other language-based abstractions.

- **Further analyses.** The customizable code visualizer and slicing demonstrate how program analysis can aid in debugging. Other analyses could be used for better debugging, or for other tasks.

  - **Automatic documentation generator.** A compiler that produces a lot of functions (such as a parser generator) might produce a lot of bad documentation when run through an automatic documentation generator. However, a user of a parser generator would probably want their grammar to be inserted into the documentation directly.
  - **Abstract interpretation.** The slicer uses a very simple form of abstract interpretation, where expressions representing functions either have the value of a single function (in the case of identifiers that represent constant functions) or a universal "bottom" value (in the case of all other expressions). Earlier work with Ziggurat focused on an assembly language, and for control-flow analysis in that language, a slightly more complex version of abstract interpretation was used, and overridden when new control-flow primitives were introduced. It would be interesting to see if control-flow analysis algorithms such as 1CFA [86] or ΓCFA [69] requiring more complex forms of abstract interpretation would give more precise slices.
  - **Invariant detection.** Dynamic invariant detection tools [77] attempt to determine invariants by running a program. It is quite possible that languages up the tower have associated invariants; these can be used to refine the invariants detected dynamically by rejecting those invariants that are subsumed by static guarantees.

Appendix A

Semantic generic functions of C/sexp

In Ziggurat, the static semantics of a language consist of a fixed set of generic functions, organized into phases. The semantic generic functions of C/sexp are in four phases:

- Parsing
- Static analysis (type analysis)
- Additional analysis
- Code generation

Each generic method of the language belongs to a phase of compilation, and may not call generic functions of a later phase.

A.1 Parsing

During parse-time, there is only one generic function that may be called: `perform-decls:`. Calling this generic function on a statement, declaration, expression or type enters a namespace containing all of the names declared in that piece of syntax. For example, the declaration `(var x int)` declares the name `x` in the variable namespace. The expression `(sizeof (struct foo))` declares the name `foo` in the struct-tag namespace, and so forth.

```
;; perform-decls: syntax syntactic-environment ->
;;                syntactic-environment
;; enters a namespace with all names
;; declared in the argument syntax
(define-generic (perform-decls: form env)
  (lambda (form env) env))
```

A.2 Static analysis (type analysis)

C/sexp defines a set of generic functions that return the types of expressions, and accept or reject the program as a whole. During this phase, types are represented by a set of lazy-delegation classes, as discussed in section 7.7.1. These classes are distinct from the classes used to represent type syntax.

```
;; typecheck-program: csexp-program -> bool
;; validates that a C/sexp program is type-correct
(define-generic (typecheck-program: prog))
```

The following generic functions take a `type-table` argument. This variable is an identifier-table that maps variables, named types and struct tags to their declared types. Scope is not an issue for this mapping, since during parse-time each of these is resolved to a unique identifier.

```
;; typecheck-stm: (statement or declaration)
;;                type
;;                identifier-table -> bool
;; validates that a statement is type-correct
(define-generic (typecheck-stm: stm return-type type-table)
  (lambda (stm return-type table) #t))



;; exp-type: expression identifier-table -> type
;; determines the type of an expression
;; and checks that it is type-correct
(define-generic (exp-type: exp type-table)
  (lambda (exp table) reference-int-type))



;; type-syntax-type: type-syntax identifier-table -> type
;; determines the type of a piece of type syntax
(define-generic (type-syntax-type: type type-table)
  (lambda (exp table) reference-int-type))



;; perform-type-decls: (statement or declaration)
;;                     identifier-table ->
;;                     identifier-table
;; adds the variables, named types and
;; struct tags declared in the syntax
;; to the type table
(define-generic (perform-type-decls: stm type-table)
  (lambda (stm table) type-table))
```

Type objects in C/sexp have a number of methods defined for them, mostly to check the type properties defined in the C99 standard.

```
;; type-equal?: type type -> bool
;; tests two types for equality
(define-generic (type-equal?: first second)
  (lambda (first second) #f))
```

```
;; type->syntax: type -> type-syntax
;; turns a type back into syntax; not guaranteed unique
(define-generic (type->syntax: type))
```

According to section 6.2.5.1 of the C99 standard, there are three disjoint and complete type categories, object types, function types and incomplete types. Object types describe objects of known size, function types describe functions, and incomplete types describe objects of unknown size. C/sexp defines predicates for each.

```
;; object-type?: type -> bool
;; determines whether the type is an object type
(define-attribute object-type?: (lambda (this) #f))
```

```
;; function-type?: type -> bool
;; determines whether the type is a function type
(define-attribute function-type?: (lambda (this) #f))
```

```
;; incomplete-type?: type -> bool
;; determines whether the type is an incomplete type
(define-attribute incomplete-type?: (lambda (this) #f))
```

The C99 standard, in section 6.2.7, defines “compatible” types.

```
;; compatible-type?: type type -> bool
;; tests for compatible type as defined in
;; ISO WG14/N1124 section 6.2.7
(define-generic (compatible-type?: expected got)
  (lambda (expected got) #f))
```

```
;; composite-type: type type -> type
;; calculates the composite of two compatible types,
;; as defined in ISO WG14/N1124 section 6.2.7
(define-generic (composite-type: first second)
  (lambda (first second) first))



;; type-accepts? type type -> bool
;; determines whether an implicit conversion
;; can occur from expected to got,
;; as defined in ISO WG14/N1124 section 6.3
(define-generic (type-accepts?: expected got)
  (lambda (expected got) #f))
```

A.3 Further analysis

After type analysis has been done, control and data dependency can be determined. In C/sexp, this is done via a single generic function, that returns a Tsuriai goal to determine dependence, as discussed in section 10.3.

```
;; csexp-dependence: syntax control-object -> goal
;; returns a Tsuriai goal representing the
;; dependence information derived from this
;; piece of syntax
(define-generic (csexp-dependence: this current-control)
  (lambda (this current-control)
    (<- (d! this) (U (d! this) (set this)
                     (cobj-current-control:
                      current-control)))))
```

A.4 Code generation

The last phase in C/sexp compilation is code generation. This takes the form of a single generic function that returns a string containing the C99 equivalent of the argument syntax.

```
;; c-string: syntax -> string
;; returns a string containing the C99 equivalent
;; of the argument syntax
(define-attribute c-string:
  (lambda (x) "error"))
```

Appendix B

A simple compiler for a four-operation calculator

```
(include "stdlib.clib")
(include "stdio.clib")
(include "cstring.clib")
(include "math.clib")
(include "printf.clib")

(include "cond.csexp")
(include "list.csexp")
(include "parser-slicing.csexp")



;;; an instruction is a list of operands.
(type inst (list (* char)))

(type inst-list (list inst))



;;; a compile state is 1. a register to hold the current value
;;; and 2. a list of instructions.
(type compile-state
      (struct compile-state
              ((value (* char)) (listing inst-list))))
```

```
;;; concatenate two instruction lists.
(fun inst-list-append! inst-list ((left inst-list)
                                  (right inst-list))
  (var current-pointer (* inst-list) (& left))

  (while (! (null? (* current-pointer)))
    (:= current-pointer (& (tail (* current-pointer)))))
  (:= (* current-pointer) right)
  (return left))


;;; add an instruction to the end of a list.
(fun inst-list-add! inst-list ((add-me inst) (to-me inst-list))
  (var current-pointer (* inst-list) (& to-me))

  (while (! (null? (* current-pointer)))
    (:= current-pointer (& (tail (* current-pointer)))))
  (:= (* current-pointer) (make-list add-me (null)))
  (return to-me))


;;; print an instruction using printf.
(fun output-inst void ((inst inst))
  (var first int -1)

  (while (! (null? inst))
    (block
     (if first
         (block
          (printf "%s" (head inst))
          (:= first 0))
         (printf " %s" (head inst)))
     (:= inst (tail inst))))
  (printf " "))


;;; print a list of instructions.
(fun output-inst-list void ((inst-list inst-list))
  (while (! (null? inst-list))
     (block
      (output-inst (head inst-list))
      (:= inst-list (tail inst-list)))))


;;; concatenate two strings non-destructively.
(fun strdupcat (* char) ((left (* char)) (right (* char)))
  (var to-return (* char) (malloc (* (sizeof char)
                                     (+ (strlen left)
                                        (+ (strlen right) 1)))))
  (strcpy to-return left)
  (strcat to-return right))
```

```
;;; generate a register.
(var next-register unsigned-int 0)
(fun gensym (* char) ()
  (return (strdupcat "$" (itoa (post+ next-register)))))


;;; generate a constant.
(fun generate-constant compile-state ((value int))
  (var to-return compile-state)

  (:= (fld to-return value) (itoa value))
  (:= (fld to-return listing) (null))
  (return to-return))


;;; generate instructions.
(fun generate-2op compile-state ((op (* char))
                                 (left compile-state)
                                 (right compile-state))
  (var to-return compile-state)
  (var result-reg (* char) (gensym))

  (:= (fld to-return value) result-reg)
  (:= (fld to-return listing)
      (inst-list-append!
       (fld left listing)
       (inst-list-add!
        (make-list op
          (make-list result-reg
            (make-list (fld left value)
              (make-list (fld right value)
                         (null)))))
        (fld right listing))))
  (return to-return))


(fun generate-add compile-state ((left compile-state)
                                 (right compile-state))
     (return (generate-2op "ADD" left right)))

(fun generate-sub compile-state ((left compile-state)
                                 (right compile-state))
     (return (generate-2op "SUB" left right)))

(fun generate-mul compile-state ((left compile-state)
                                 (right compile-state))
     (return (generate-2op "MUL" left right)))

(fun generate-div compile-state ((left compile-state)
                                 (right compile-state))
     (return (generate-2op "DIV" left right)))
```

```
;;; parser parse-calc returns a compile-state.
(parser parse-calc token get-token
  (tokens (left MUL DIV ADD SUB) LPAREN RPAREN
          (type int NUM) (eos EOS))
  (non-term NUM-EXP compile-state
    (=> (NUM) (return (generate-constant ($ 1)))))
  (non-term EXP compile-state
    (=> (NUM-EXP) (return ($ 1)))
    (=> (EXP MUL EXP) (return (generate-mul ($ 1) ($ 3))))
    (=> (EXP DIV EXP) (return (generate-div ($ 1) ($ 3))))
    (=> (EXP ADD EXP) (return (generate-add ($ 1) ($ 3))))
    (=> (EXP SUB EXP) (return (generate-sub ($ 1) ($ 3))))
    (=> (LPAREN EXP RPAREN) (return ($ 2))))
  (non-term PROGRAM compile-state
    (=> (EXP) (return ($ 1))))
  (start PROGRAM))


;;; a simple lexer.
(var instring (* char))

(fun get-token token ()
  (var in (* char) (strtok instring " "))
  (:= instring (cast (* char) 0))
  (cond
   ((== in (cast (* char) 0)) (return (EOS)))
   ((== 0 (strcmp in "+")) (return (ADD)))
   ((== 0 (strcmp in "-")) (return (SUB)))
   ((== 0 (strcmp in "*")) (return (MUL)))
   ((== 0 (strcmp in "/")) (return (DIV)))
   ((== 0 (strcmp in "(")) (return (LPAREN)))
   ((== 0 (strcmp in ")")) (return (RPAREN)))
   (else (return (NUM (atoi in))))))


(fun main int ((argc int) (argv (* (* char))))
     (:= instring (malloc (* 100 (sizeof char))))
     (fgets instring 100 stdin)
     (var result compile-state (parse-calc))

     (output-inst-list (fld result listing)))
```

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Chris Andreae, Jame Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *OOPSLA '06: Proceedings of the 21st ACM SIGPLAN Conference on object-oriented programing, systems, languages, and applications*, pages 57–74. ACM Press, October 2006.

[3] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, volume 36, pages 31–42. ACM Press, November 2001.

[4] J. Baker and W. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *PLDI '02: Proceedings of the 2002 ACM SIGPLAN conference on programming language design and implementation*, volume 37, pages 270–281. ACM Press, May 2002.

[5] A. Bawden. Reification without evaluation. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 342–349, New York, NY, USA, 1988. ACM Press.

[6] A. Bawden. First-class macros have types. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 133–141. ACM Press, 2000.

[7] A. Bawden and J. Rees. Syntactic closures. In *LFP '88: Proceedings of the 1988 ACM Conference on LISP and functional programming*, pages 86–95, New York, NY, 1988. ACM.

[8] C. Brabrand. *Domain Specific Languages for Interactive Web Services*. PhD thesis, University of Aarhus, 2002.

[9] C. Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of the 2002 ACM SIGPLAN workshop on partial evaluation and semantics-based program manipulation*, volume 37, pages 31–40. ACM Press, March 2002.

[10] G. Bracha. Pluggable type systems. In *Workshop on revival of dynamic languages*, October 2004.

[11] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, pages 365–383. ACM Press, 2004.

[12] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, October 1964.

[13] W. E. Byrd and D. Friedman. From variadic functions to variadic relations: A miniKanren perspective. In *Scheme '06: Proceedings of the 2006 workshop on Scheme and functional programming*, September 2006.

[14] L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Technical Report 121, Digital Equipment Corporation SRC, 1994.

[15] Luca Cardelli and Rowan Davies. Service combinators for web computing. *IEEE Transactions on Software Engineering*, 25(3):309–316, 1999.

[16] W. Clinger. Hygienic macros through explicit renaming. *LISP Pointers*, 4(4), 1991.

[17] W. Clinger and J. Rees. Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 155–162, New York, NY, USA, 1991. ACM Press.

[18] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. *Formal Methods for Components and Objects*, 4709:266–296, 2007.

[19] Ezra Cooper and Philip Wadler. A located lambda calculus. `http://homepages.inf.ed.ac.uk/wadler/papers/located-lambda/located-lambda.pdf`.

[20] K. D. Cooper and K. Kennedy. Fast interprocedual alias analysis. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 49–59, New York, NY, USA, 1989. ACM Press.

[21] F. Deremer and T. Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Language Systems*, 4(4):615–649, October 1982.

[22] Chris Dornan, Isaac Jones, and Simon Marlow. Alex user's guide. `http://www.haskell.org/alex/doc/html/index.html`.

[23] J. J. Duby. Extensible languages: A potential user's point of view. In *Proceedings of the international symposium on extensible languages*, pages 137–140, New York, NY, USA, 1971. ACM.

[24] K. Dybvig, D. Friedman, and C. Haynes. Expansion-passing style: a general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–76, 1988.

[25] Kent Dybvig. Writing hygienic macros in scheme with syntax-case. Technical Report 356, University of Indiana Computer Science Department, 1992.

[26] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *OOPSLA '07: Proceedings of the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 884–885, New York, NY, USA, 2007. ACM.

[27] Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In *ECOOP '04: proceedings of the 2004 European conference on object-oriented programming*, volume 3086, pages 147–171. Springer, June 2004.

[28] M. Felleisen. *On the expressive power of programming languages*, volume 432, pages 134–151. Springer-Verlag, New York, N.Y., 1990.

[29] R. Findler, J. Clements, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: a progamming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[30] D. Fisher and O. Shivers. Static analysis for syntax objects. In *ICFP '06: Proceedings of the 11th ACM SIGPLAN international conference on Functional programming*, pages 111–121, New York, NY, USA, 2006. ACM Press.

[31] M. Flatt. Composable and compilable macros: you want it when? In *ICFP '02: Proceedings of the 7th ACM SIGPLAN international conference on functional programming*, 2002.

[32] D. Friedman. Object-oriented style. Invited talk at International LISP Conference, 2003.

[33] D. Friedman, W. Byrd, and O. Kiselyov. *The Reasoned Schemer*. The MIT Press, Cambridge, MA, 2005.

[34] C. Fruhwirth. Liskell: Haskell semantics with Lisp syntax. `http://clemens.endorphin.org/ILC07-Liskell-draft.pdf`.

[35] S. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In *ICFP '01: Proceedings of the 6th ACM SIGPLAN international conference on functional programming*, volume 36, pages 74–85. ACM Press, October 2001.

[36] K. Gray and M. Flatt. Compiling Java to PLT Scheme. In *Scheme '04: Proceedings of the 2004 Scheme workshop*, September 2004.

[37] K. E. Gray and M. Flatt. ProfessorJ: a gradual introduction to Java through language levels. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, pages 170–177, New York, NY, USA, 2003. ACM Press.

[38] R. Hieb, K. Dybvig, and C. Bruggeman. Syntactic abstraction in scheme. *LISP and Symbolic Computation*, 355:295–326, 1992.

[39] E. Hilsdale and D. Friedman. Writing macros in continuation-passing style. In *Scheme and functional programming '00*, 2000.

[40] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN conference on programming language design and implementation*, volume 23, pages 35–46, New York, NY, USA, July 1988. ACM Press.

[41] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. *Advanced Functional Programming*, 2638, 2003.

[42] G. Hutton and E. Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[43] ISO. Programming languages – C. Technical Report WG14 N1124, ISO/IEC, 1999.

[44] S. C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[45] T. Johnsson. Attribute grammars as a functional programming paradigm. In *FPCA '87: Proceedings of the 1987 conference on functional programming languages and computer architecture*, pages 154–173, 1987.

[46] S. Jones, T. Hoare, and A. Tolmach. Playing by the rules: rewriting as a practical optimisation technique. In *Proceedings of the 2001 Haskell workshop*, 2001.

[47] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[48] K. Kennedy and S. K. Warren. Automatic generation of efficient evaluators for attribute grammars. In *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on principles on programming languages*, pages 32–49, New York, NY, USA, 1976. ACM.

[49] O. Kiselyov. How to write seemingly unhygienic and referentially opaque macros with syntax-rules. In *Scheme '02: Proceedings of the 2002 Scheme Workshop*, 2002.

[50] O. Kiselyov. Macros that compose: systematic macro programming. In *GPSE '02: Proceedings of the 2002 conference on generative programming and system engineering*, 2002.

[51] O. Kiselyov, C. Shan, D. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on functional programming*, pages 192–203, New York, NY, USA, 2005. ACM Press.

[52] D. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[53] S. Krishnamurthi. *Linguistic Reuse*. PhD thesis, Rice University, 2001.

[54] S. Krishnamurthi, Y. Erlich, and M. Felleisen. Expressing structural properties as language constructs. In *ESOP '99: Proceedings of the 8th European symposium on programming languages and systems*, pages 258–272, London, UK, 1999. Springer-Verlag.

[55] S. Krishnamurthi and M. Felleisen. Toward a formal theory of extensible software. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on foundations of software engineering*, pages 88–98, New York, NY, USA, 1998. ACM Press.

[56] S. Krishnamurthi, M. Felleisen, and B. Duba. From macros to reusable generative programming. In *GCSE '99: Proceedings of the 1st international symposium on generative and component-based software engineering*, pages 105–120, London, UK, 2000. Springer-Verlag.

[57] D. Leijen and E. Meijer. Domain specific embedded compilers. In *DSL '99: Proceedings of the 2nd USENIX conference on domain-specific languages*, pages 109–122, 1999.

[58] D. Leijen and E. Meijer. Parsec: direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[59] M. E. Lesk and E. Schmidt. Lex—a lexical analyzer generator. *UNIX Vol. II: research system (10th ed.)*, pages 375–387, 1990.

[60] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140), 1932.

[61] M. Machenry and J. Matthews. Topsl: a domain-specific language for on-line surveys. In *Scheme '04: Proceedings of the 2004 Scheme workshop*, September 2004.

[62] W. Maddox. Semantically-sensitive macroprocessing. Technical Report CSD-89-545, University of California at Berkeley, Berkeley, CA, USA, 1989.

[63] J. Matthews and R. B. Findler. An operational semantics for Scheme. *Journal of Functional Programming*, 18(01):47–86, 2007.

[64] J. Matthews, R. B. Findler, P. Graunke, S. Krishnamurthi, and M. Felleisen. Automatically restructuring programs for the web. *Automated software engineering*, 11(4):337–364, October 2004.

[65] J. Mccarthy. sql-oo: An persistent layer for particularly constructed objects. `http://planet.plt-scheme.org/\#sql-oo.plt`.

[66] Erik Meijer and Graham Hutton. Bananas in space: extending fold and unfold to exponential types. In *FPCA '95: Proceedings of 7th international conference on functional programming languages and computer architecture*, pages 25–28, New York, 1995. ACM Press.

[67] M. Mernik, V. Zumer, M. Lenic, and E. Avdicausevic. Implementation of multiple attribute grammar inheritance in the tool LISA. *SIGPLAN Notices*, 34(6):68–75, June 1999.

[68] Leo Meyerovich, Michael Greenberg, Gregory Cooper, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax. `http://www.flapjax-lang.org/`.

[69] Matthew Might and Olin Shivers. Improving flow analyses via Γcfa: abstract garbage collection and counting. *SIGPLAN Notices*, 41(9):13–25, 2006.

[70] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Language Systems*, 21(3):527–568, 1999.

[71] R. Nanavati. Extensible syntax in the presence of static analysis. Master's thesis, Massachusetts Institute of Technology, September 2000.

[72] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software composition. In *OOPSLA '06: Proceedings of the 21st ACM SIGPLAN conference on object-oriented programing, systems, languages, and applications*, pages 21–36. ACM Press, October 2006.

[73] H. Oesterholt. ROOS—Rose object orientation for Scheme. `http://www.elemental-programming.org/roos.html`.

[74] R. Olinsky, C. Lindig, and N. Ramsey. Staged allocation: a compositional technique for specifying and implementing procedure calling conventions. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 409–421, New York, NY, USA, 2006. ACM Press.

[75] Scott Owens, Matthew Flatt, Olin Shivers, and Benjamin McMullan. Lexer and parser generators in Scheme. In *Scheme '04: Proceedings of the 2004 Scheme workshop*, pages 41–52, 2004.

[76] I. Pembeci, H. Nilsson, and G. Hager. System presentation - functional reactive robotics: an excercise in principled integration of domain-specific languages. In *PPDP '02: Proceedings of the 2002 conference on principles and practice of declarative programming*, October 2002.

[77] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *FSE '04: Proceedings of the 12th ACM SIGSOFT symposium on the foundations of software engineering*, pages 23–32, Newport Beach, CA, USA, November 2–4, 2004.

[78] S. Peyton-Jones, J. M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering (functional pearl). *SIGPLAN Notices*, 35(9):280–292, 2000.

[79] B. Pierce, editor. *Advanced Topics in Types and Programming Languages*. The MIT Press, Cambridge, MA, 2005.

[80] F. Pottier and Y. Regis-Gianas. Towards efficient, typed LR parsers. *Electronic Notes in Theoretical Computer Science*, 148(2):155–180, March 2006.

[81] F. Pottier and D. Remy. *The Essence of ML Type Inference*, pages 389–489. In Pierce [79], 2005.

[82] David Recordon and Drummond Reed. Openid 2.0: a platform for user-centric identity management. In *DIM '06: Proceedings of the 2nd ACM workshop on digital identity management*, pages 11–16, New York, NY, USA, 2006. ACM.

[83] S. Seefried, M. Chakravarty, and G. Keller. Optimising embedded DSLs using template Haskell. In *GPCE '04: Proceedings of the 2004 conference on generative programming and component engineering*, pages 186–205, 2004.

[84] Andrew Shalit, David Moon, and Orca Starbuck. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison Wesley Publishing Company, September 1996.

[85] Justin Sheehy and Riccardo Pucella. A resource-transfer model for cryptographic protocols.

[86] O. Shivers. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991.

[87] O. Shivers. Automatic management of operating-system resources. In *ICFP '97: Proceedings of the 2nd ACM SIGPLAN international conference on functional programming*, pages 274–279, New York, NY, USA, 1997. ACM Press.

[88] O. Shivers. The anatomy of a loop: a story of scope and control. In *ICFP '05: Proceedings of the 10th ACM SIGPLAN international conference on functional programming*, pages 2–14, Tallinn, Estonia, September 2005. ACM Press.

[89] O. Shivers and D. Fisher. Multi-return function call. *Journal of Functional Programming*, 16(4-5):547–582, 2006.

[90] R. Smith and D. Ungar. Programming as an experience: the inspiration for Self. In *ECOOP '95: Proceedings of the 9th European conference on object-oriented programming*, pages 303–330, London, UK, 1995. Springer-Verlag.

[91] Richard Stallman and the GCC Developer Community. Using the GNU Compiler Collection. `http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/`.

[92] T. A. Standish. Extensibility in programming language design. *SIGPLAN Notices*, 10(7):18–21, July 1975.

[93] G. Steele. Growing a language. In *OOPSLA '98 Addendum: Addendum to the proceedings of the 1998 conference on object-oriented programming, systems, languages, and applications*. ACM Press, 1998.

[94] C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1):11–49, April 2000.

[95] B. Stroustrup. An overview of C++. In *Proceedings of the 1986 SIGPLAN workshop on object-oriented programming*, pages 7–18, New York, NY, USA, 1986. ACM Press.

[96] W. Taha and T. Sheard. MetaML: multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

[97] D. Ungar and R. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings of the 1987 conference on object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.

[98] D. Vaillancourt. ACL2 in DrScheme. `http://www.ccs.neu.edu/home/dalev/acl2-drscheme/`.

[99] A. van Tonder. SRFI 72: Hygienic macros. `http://srfi.schemers.org/srfi-72/srfi-72.html`.

[100] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *CC '02: Proceedings of the 11th international conference on compiler construction*, pages 128–142, London, UK, 2002. Springer-Verlag.

[101] E. Van Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger. Attribute grammar-based language extensions for Java. In *ECOOP '07: Proceedings of the 2007 European conference on object-oriented programming*, pages 575–599. ACM, 2007.

[102] Eelco Visser. Meta-programming with concrete object syntax. In *GPCE '02: Proceedings of the 2002 ACM SIGPLAN/SIGSOFT conference on generative programming and component engineering*, pages 299–315. Springer-Verlag, 2002.

[103] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on principles of programming languages*, pages 307–313. ACM Press, 1987.

[104] P Wadler. Theorems for free! In *FRCA '89: Proceedings of the 4th international conference on functional programming languages and computer architecture*, pages 347–359, New York, 1989. ACM Press.

[105] P. Wadler. The expression problem. Mailing list, November 1998.

[106] D. W. Wall and M. L. Powell. The Mahler experience: using an intermediate language as the machine description. *SIGARCH Computer Archiecture News*, 15(5):100–104, 1987.

[107] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *ICFP '01: Proceedings of the 6th ACM SIGPLAN international conference on functional programming*, volume 36, pages 146–156. ACM Press, October 2001.

[108] M. Wand. Complete type inference for simple objects. In *Proceedings of the 1987 IEEE symposium on logic in computer science*, June 1987.

[109] K. Wansbrough. Macros and preprocessing in Haskell. `http://citeseer.ist.psu.edu/wansbrough99macros.html`, 1999.

[110] D. Weise and R. Crew. Programmable syntax macros. In *PLDI '93: Proceedings of the 1993 SIGPLAN conference on programming language design and implementation*, pages 156–165, 1993.

[111] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[112] N. Welsh, F. Solsona, and I. Glover. SchemeUnit and SchemeQL: Two little languages. In *Scheme '02: Proceedings of the 3rd workshop on Scheme and functional programming*, November 2002.

[113] M. Zenger and M. Odersky. Implementing extensible compilers. In *Proceedings of the workshop on multiparadigm programming with object-oriented languages*, Budapest, Hungary, June 2001.