DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Building Programming Languages, Construction by Construction

VIKTOR PALMKVIST

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Building Programming Languages, Construction by Construction

VIKTOR PALMKVIST

Master in Computer Science
Date: July 5, 2018
Supervisor: David Broman
Examiner: Mads Dam
Swedish title: Att bygga programmeringsspråk, konstruktion för konstruktion
School of Electrical Engineering and Computer Science

Abstract

The task of implementing a programming language is a task that entails a great deal of work. Yet much of this work is similar for different programming languages: most languages require, e.g., parsing, name resolution, type-checking, and optimization. When implementing domain-specific languages (DSLs) the reimplementation of these largely similar tasks seems especially redundant.

A number of approaches exist to alleviate this issue, including embedded DSLs, macro-rewriting systems, and more general systems intended for language implementation. However, these tend to have at least one of the following limitations:

- They present a leaky abstraction, e.g., error messages do not refer to the DSL but rather some other programming language, namely the one used to implement the DSL.
- They limit the flexibility of the DSL, either to the constructs present in another language, or merely to the syntax of some other language.
- They see an entire language as the unit of composition. Complete languages are extended with other complete language extensions.

Instead, this thesis introduces the concept of a *syntax construction*, which represents a smaller unit of composition. A syntax construction defines a single language feature, e.g., an if-statement, an anonymous function, or addition. Each syntax construction specifies its own syntax, binding semantics, and runtime semantics, independent of the rest of the language. The runtime semantics are defined using a translation into another target language, similarly to macros. These translations can then be checked to ensure that they preserve binding semantics and introduce no binding errors. This checking ensures that binding errors can be presented in terms of code the programmer wrote, rather than generated code in some underlying language.

During evaluation several limitations are encountered. Removing or minimizing these limitations appears possible, but is left for future work.

Sammanfattning

Att implementera ett programmeringsspråk är ett mycket arbetstungt åtagande. Detta trots att mycket av det som behöver göras inte skiljer sig särskilt mycket mellan olika språk, de flesta behöver exempelvis parsning, namnupplösning, typcheckning och optimering. För ett domänspecifikt programmeringsspråk (DSL) är denna upprepning ännu mer tydlig.

Det finns ett antal olika metoder för att hantera detta, exempelvis embeddade DSLer, macro-system, och mer generella system för programspråksimplementation. Dessa tenderar dock att ha en eller flera av följande begränsningar:

- De abstraktioner som introduceras ”läcker”, felmeddelanden kan exempelvis referera till abstraktioner i ett annat programmeringsspråk, nämligen det som användes för att implementera DSLet.
- DSLet som implementeras blir begränsat, antingen till vad som finns i implementationsspråket, eller till implementationsspråkets syntax.
- Ett DSL ses som den minsta hela beståndsdelen i systemet. Om delar av språket ska återanvändas eller inkluderas i ett annat måste hela språket följa med.

Denna avhandling introducerar istället *syntaxkonstruktioner* som minsta beståndsdel. En syntaxkonstruktion representerar en enskild del av ett språk, exempelvis en if-sats, en anonym funktion, eller addition. Varje syntaxkonstruktion definierar sin egen syntax, bindningssemantik och exekveringssemantik, utan referenser till språket som helhet. Exekveringssemantiken liknar en macro, den uttrycks som en översättning till ett implementationsspråk. Tack vare att bindningssemantiken är specifierad kan vi sedan kontrollera översättningen så att den inte kan introducera bindningsfel. Detta medför att felmeddelanden kan referera enbart till kod som programmeraren faktiskt skrev, istället för genererad kod i implementationsspråket.

Evalueringen påvisar flera begränsningar med systemet. Begränsningarna tycks lösbara, men detta arbete lämnas till framtiden.

Contents

Chapter 1

Introduction

A domain-specific programming language (DSL) is a programming language created to solve problems in some particular domain. There are numerous methods for creating such languages. A few examples include:

Embedded DSLs. An embedded DSL is a language designed as a library in a host programming language. They use the already existing capabilities of the host language, e.g., name binding, type systems, and other libraries. Some examples include Paradise [1], RSpec[1], and Rake[2].

However, embedding a DSL limits it to the capabilities of the host language. Additionally, any type errors produced on incorrect usage may refer to concepts in the host language, rather than the DSL.

External DSLs. On the other hand, an external DSL is implemented from the ground up. It is not part of a host language. Examples include CSS, SQL, and DOT[3].

While these DSLs are not limited by some host language, they also have no host language to take advantage of. Any capabilities that might have been provided by a host language must instead be reimplemented.

[1] http://rspec.info/
[2] https://ruby.github.io/rake/
[3] http://graphviz.org/

Macros. Macros, commonly featured in Lisp dialects, are syntax transformations that run at compile-time. From a practical stand point they allow a programmer to extend a language with new constructs that appear as though they are built in to the language, but are in fact implemented by some translation to the underlying language. They can be used to implement an embedded DSL.

This gives greater freedom than most other forms of embedded DSLs, but error messages are still likely to refer to the host language, and the syntax is still constrained to that of the host language; a Lisp macro still looks like Lisp.

Language-definition languages. This category is somewhat looser defined than the others, but refers to systems whose (primary or secondary) purpose is the construction of programming languages. Some examples include SoundX [11], Silver [17], and Racket [15].

The trade-offs vary by system, but they all tend to consider a language as the base building block when we want to reuse or extend a language: extend a full language with a full language extension.

This thesis falls under the language-definition language category, but takes a different approach. We introduce *syntax constructions*, which center around singular language constructs. A syntax construction could for example implement an if-statement, an anonymous function, or a pattern used in pattern matching. Each syntax construction describes its own syntax, its binding semantics, and its runtime semantics, independent of other parts of the language. The runtime semantics are specified in terms of a translation to another programming language, and this translation is checked to ensure that it always preserves the binding semantics.

An alternative angle is to consider syntax constructions as macros with some extra features: greater syntactical flexibility and the ability to reason about name bindings without macro expansion. This means that binding errors do *not* refer to the host language.

The self-contained nature of a syntax construction enables creating new programming languages by cherry-picking individual features

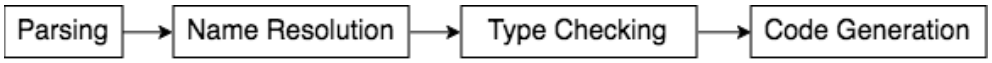

Figure 1.1: A common general structure for compilers and interpreters.

from previous languages, i.e., it enables reuse of a single language feature as opposed to an entire language. The unit of composition is a feature, not a language.

The remainder of this introduction will provide additional context and motivation for the thesis and the associated approach (Sections 1.1 through 1.3), a more precise problem statement (Section 1.4), and a list of contributions (Section 1.5).

1.1 Programming Language Implementation

The implementation of a programming language—be it an interpreter or a compiler—tends to be divided into phases, each dealing with their own aspect of the language. Figure 1.1 exemplifies this structure. For example, a parser details the syntax of the language, while name resolution handles names, scoping, and namespaces. A type checker details part of the semantics, what language constructs can be combined, then code generation or a simple interpreter codify the runtime semantics of the individual constructs. This means that the implementation of any single language construct is spread throughout the language implementation, necessitating it be considered as a whole. Adding a new construction requires changes in many places.

This also means that reuse of a language implementation is difficult; each phase makes assumptions on what has been done before it and what will be done later. Trying to use or replace a single phase gives little flexibility to make something different than what is already there.

Attempting to use a single language construct from a language is likewise difficult. The various aspects of its definition are spread throughout the different phases, quite possibly intertwined with code handling other constructs that happen to be similar in that particular language.

Despite this, there are some quite successful cases of language implementation reuse:

JVM. A large number of programming languages exists that can compile to Java bytecode, enabling them to reuse the implementation originally intended for Java. Examples include Scala, Clojure, and Kotlin. Other languages have the JVM as an additional backend, for example Ruby (JRuby) and Python (Jython).

JavaScript. Many languages can compile to JavaScript, thus reusing the entire language while adding something new on top. Examples include TypeScript (essentially JavaScript with static types), CoffeeScript (alternative syntax for JavaScript), and Elm (statically typed language with JavaScript as a compile target).

OCaml. ReasonML replaces the parser and surrounding tooling of OCaml, providing a different syntax to what is otherwise the same language.

LLVM. The LLVM project (previously an acronym for ”low level virtual machine,” now a name all by itself) defines an intermediate representation that can be used both as a target for a language and as a source for translation into machine code. This allows changing either frontend or backend while reusing the other, as well as a large number of optimizations defined on the intermediate representation. Languages using LLVM either by default or as part of an alternative backend include C, C++, Rust, and Haskell. Examples of instruction sets to which LLVM intermediate representation can be translated include x86, x86-64, ARM, and PowerPC.

.NET Framework. The .NET framework provides a common framework for interoperability between multiple programming languages. This is done by offering a single intermediate language called Common Intermediate Language (more commonly CIL), similar to Java bytecode.

Somewhat simplified, the key common characteristic of these examples is that the reused part tends to be from somewhere in the middle until the end of the original implementation; the replaced part has the first few phases, then all remaining phases are from the

original implementation. No one only replaces a few phases in the middle.

While this does allow reuse of large chunks of an implementation, much still has to be reimplemented. The different implementations of the first phases generally do much of the same work, all of them parse their language, then do name resolution, then maybe some type checking, and so on. Many details will differ, but the basic kind of work tends to stay the same.

We would ideally like to have more flexible forms of reuse, enabling reuse of singular language features, and not requiring reimplementation of name resolution and many other common tasks beyond specifying the particular details of the current language.

1.2 Macros

From a practical viewpoint, a macro is essentially syntax that appears as though it were built in to the programming language, but is in fact defined by a user through a translation to some other syntax. The following example is in Racket [6], a Lisp dialect:

```
; define-syntax-rule introduces a new macro,
; in this case with the name 'and'.
; This is a shortcutting 'and', i.e., 'b' will
; only be evaluated if 'a' is true
(define-syntax-rule (and a b)
  (if a b #f))

; Assuming the existence of a function trace
; that prints its argument and then returns
; it, we can demonstrate 'and' like so:
(and (trace #t) (trace #f))
; => #f
; This prints #t, but not #f, i.e., (trace #f)
; was not evaluated
```

This would not be possible to implement as a function in Racket since functions require their arguments to be fully evaluated before they

are called, i.e., the second argument would be evaluated before `and`, regardless of the value of the first argument.

This particular example shows a simple example where the arguments of the macro are simply inserted into a form of template, but macros in general can be far more complex and powerful. Macros can utilize recursion as well as the full power of Lisp, i.e., a macro in general is a function from syntax to syntax that runs at compile-time.

The expressive power provided here is great; any arbitrary computation can be done at compile time. Examples include type checking (typed racket [15]), destructuring and pattern matching (Clojure, standard library[4] and external library[5] respectively), and coroutines (Clojure[6]). Racket (which was used in the example above) takes this one step further and describes itself as an "ecosystem for developing and deploying new languages", all of which is powered by its macro system.

There are some drawbacks, however, and we will spend the next few sections examining some of them.

1.2.1 Problems of Abstraction

The abstractions provided by macros can easily become leaky abstractions without special care. By leaky abstraction we mean an abstraction that might in some case expose its implementation to a user, or one that cannot be reasoned about correctly without knowing its implementation.

Name Capture

A naive macro expander using textual replacement can introduce accidental name capture. To understand what accidental name capture is, consider the following example (using Racket syntax) intended to calculate $a + a + b$:

[4] https://github.com/clojure/clojure/blob/master/src/clj/clojure/core.clj
[5] https://github.com/clojure/core.match
[6] https://github.com/clojure/core.async

```
(define-syntax-rule (double-add a b)
; let introduces a new binding, in this case
; 'x' is bound to the value of evaluating 'a'
  (let ([x a])
    (+ x x b)))
```

The extra `let`-binding is there to ensure we only compute `a` once. Macros are expanded at compile-time and receives its arguments as expressions, i.e., they are not values, but rather computations that produce values. The more obvious implementation, `(+ a a b)`, would compute `a` twice.

The following expression will exhibit accidental name capture when macro expansion uses pure textual replacement:

```
(let ([x 2])
  (double-add 1 x))
; expands to:
(let ([x 2])
  (let ([x 1])
    (+ x x x)))
; which evaluates to
(+ 1 1 1) = 3
; while our intended semantics would evaluate to:
(+ 1 1 2) = 4
```

Notable here is that the above would not produce an error of any kind, the result would simply be silently wrong.

It is worth noting here that Racket solves this problem, as do most other macro systems, through rewriting or more complicated forms of name resolution (e.g., [7, 5]). A notable exception is the C preprocessor that makes no such attempt.

Errors after Expansion

In some cases, the fully expanded code is incorrect, either because the macro was poorly implemented, or because it was used improperly. For example:

```
(define-syntax-rule (new-let x e body)
  (let ([x e])
    body))

(new-let 2 x
  (+ 1 x))
```

Here we define a new binding construction, but misuse it; the identifier and the value are switched. The macro expansion will succeed and produce the code below, but at that point an error message will be produced about improper use of **let**, even though **let** never occurs in the original code.

```
(let ([2 x])
  (+ 1 x))
```

The error exposes the implementation of the macro. The generated code may be introduced through several layers of macro expansion and can be quite complicated, making the connection between the error message and the actual error even less clear.

In contrast to accidental name capture, this error *does* appear in Racket, as described above.

In a statically typed language this problem has an additional form: type errors after macro expansion. A macro might expand to code that is incorrect in the sense that it has type errors. Considering this case is important, as one of the strengths of a statically typed language is the dialog between the programmer and the typechecker: the programmer writes code that is wrong, and the typechecker provides guidance on what to fix in the form of error messages. If these error messages refer to the expansion and not the original code, then the guidance is very hard to follow.

This thesis does not address types, but rather leaves them for future work.

1.2.2 Problems of Syntax

One advantage of macros is their ability to essentially introduce syntactic sugar to a language at a user level. Despite this, most commonly used macro systems place some fairly strict limits on the newly introduced syntax. For example, a C preprocessor macro must be an identifier, possibly followed by an argument list, and a Lisp macro must be a list of valid forms, started by a symbol.

Such restrictions simplify parsing the language, since it ensures that the grammar is fixed and cannot be changed during parsing.

1.3 Desired Ambiguity

When designing a grammar to describe the syntax of a programming language, ambiguity tends to be an undesirable property. There are multiple reasons for this. For example an unambiguous grammar may be faster to parse (e.g., the Earley parsing algorithm [3] has a better worst-case complexity for unambiguous grammars), or one might consider the possibility of encountering code that, while syntactically correct, does not have a well defined meaning as a bad thing.

Instead, this thesis argues that there is a place for ambiguity in programming language grammars, if we presuppose that code is read more often than it is written. Consider the expression `1 & 3 == 1`, where `&` is bitwise ”and” and `==` is equality. The result of the expression depends on the relative precedence of the two operators. Thus a reader can know the precise semantics of all operators and values involved, and still not know what the code does.

This is of course a tradeoff. The same argument could be made for an expression such as `1 + 2 * 3`, yet requiring explicit grouping for all expressions seems excessive. The difference stems from the frequency each operator is used, e.g., most code uses basic arithmetic and logical operators, but bitwise operators are relatively infrequent. The former are also taught early in mathematics, including precedence, thus most programmers would know the exact meaning of `1 + 2 * 3` while `1 & 3 == 1` would be less obvious.

Worse, the meaning of the latter expression varies by language, even if the languages have the same meaning for the operators. The evaluation of that same expression in C and Python differs:

C	Python
`1 & 3 == 1`	`1 & 3 == 1`
`1 & (3 == 1)`	`(1 & 3) == 1`
`1 & 0`	`1 == 1`
`0`	`True`

Thus a programmer who knows the precedence in one language may read the code, understand the semantics of all operators involved, and yet misunderstand the semantics of the expression.

This thesis thus suggests the following: if most programmers would agree on the interpretation of some code it should be parsed unambiguously as such. If the correct interpretation is non-obvious then the programmer writing the code should be asked to clarify their intent, most likely by using parentheses for explicit grouping. The latter can be achieved with an ambiguous grammar and good error messages (Section 5.6.1 shows a good first step towards such error messages).

The key then is to provide tools with which a language designer has ample control of where ambiguity is surfaced to an end user. This control should deal with both ambiguity and unambiguity, neither should appear accidentally when the intent was the opposite.

1.4 Problem Statement

Previous methods for constructing programming languages either allow the creation of small reusable language constructs, or syntactical flexibility, but not both. Specifically, the research problem of this thesis is to

> design a programming language construction approach that enables the creation and composition of individual language features that are self contained in terms of syntax and semantics, while minimizing restrictions on syntactical flexibility.

In particular, it should be possible to extend a language by cherry-picking one or a few features from another language, without having to reimplement them.

Features here refers to things such as an if-statement, an anonymous function, or a pattern used in pattern matching.

This stands in contrast with most other approaches that either compose entire languages (i.e., you get the entire language or nothing at all), or spread the semantics of individual features across several phases of compilation.

The goal above states very little in regards to the form of a solution, nor does it provide much assistance in evaluating the usefulness of any particular solution. To alleviate this issue, the following additional design goals will guide the design and evaluation of the approach.

Tower of languages: It should be possible to define new languages in terms of other languages, very much akin to Lisp tradition, as a form of reuse.

Syntactical freedom: Language features should be able to specify new syntax, ideally with no constraints.

Abstraction preservation: No usage of a language feature should expose its implementation to an end user.

Good error messages: Improper use or implementation of language features should present the user with understandable error messages.

Composition through cherry-picking: It should be possible to reuse individual features of languages, i.e., the unit of composition must be smaller than a full language.

Reasoning without context: Expanding upon the previous point, a language feature should be as self contained as possible, to permit reasoning about it without full awareness of its context.

These goals are referred to throughout the thesis to motivate choices or highlight strengths and flaws with the chosen approach.

1.4.1 Delimitations

There are a few aspects of programming languages that this thesis does not address, but rather leaves for future work:

Type safety: Type safety is not considered. Incorrect usage of features that would give rise to type errors is not detected and will most likely present errors in terms of the implementation of the features being (mis)used. This affects abstraction preservation and good error messages.

Lexing: This thesis assumes that parsing is done on a token stream, i.e., the syntactical freedom of a feature is limited by a predefined tokenization. This affects syntactical freedom. In practice, it mostly means that the syntax of the following things cannot be customized:

- Valid identifiers
- Literals, e.g., integers and strings
- Comments

Indentation: This thesis assumes syntax to be whitespace independent, which precludes defining languages such as Python where indentation is important. This affects syntactical freedom.

Namespaces and Importing: Namespaces and other similar methods of partitioning definitions and later importing them are not considered, only lexical scoping is used. This is sufficient for C and some similar languages, but not for Java and other languages with some form of modules. It does not explicitly affect any of the design goals, but it clearly limits the languages that can be designed.

1.5 Contributions

This thesis contributes the following:

- A method of specifying the syntax, binding semantics, and expansion of a language construction (macro) in a singular

location. This differs from most other solutions, wherein the unit of composition is a language, not an individual feature. The design considerations from a user perspective can be found in Chapter 3, while formalization is in Chapter 4.

- An approach for checking that a given expansion specification can never introduce binding errors (Section 4.7), and a justification that expansion always terminates when given finite input (Section 4.6).
- Two implementations of two non-trivial language subsets:
  - OCaml (Section 5.2.1), to evaluate the method's suitability in modeling a typical functional language, focusing on pattern matching.
  - Lua (Section 5.2.2), to evaluate the method's suitability in modeling a typical imperative language.

1.6 Research Method

The results of this thesis are evaluated through an implementation of the ideas introduced in Chapter 3 and later presented with more detail in Chapter 4. An overview of the implementation can be found in Section 5.1.

This implementation is then used to implement two non-trivial programming language subsets to evaluate the expressiveness of the method. These languages are discussed in Sections 5.2.1 (OCaml) and 5.2.2 (Lua). The languages were chosen to test the expressiveness of syntax constructions, with a particular focus for each:

- For OCaml the focus was to express a relatively standard functional programming language, with pattern matching as the most important feature to implement correctly.
- For Lua the focus was to express an imperative language, in particular, one with imperative control flow, e.g., if-statements, while-loops, and mutable variables.

The two language implementations are tested by ensuring that several small programs produce the same output regardless of

whether they were executed by the syntax construction implementation or the original implementation (Section 5.4).

Additionally, we discuss several other programming languages in terms of features they support that syntax constructions cannot express (Section 5.2.3).

To evaluate composability, we import patterns and pattern matching from OCaml into Lua (Sections 5.3.1 and 5.3.2, respectively).

Section 5.6 evaluates the error messages produced by the implementation in terms of information content and specificity.

Finally, the performance of the implementation is examined in Section 5.5.

1.7 Ethics, the Environment, and Societal Impact

Given the nature of this thesis, it has no direct ethical or environmental impact, but some indirect impact. The aim is to produce a convenient way of constructing new programming languages, i.e., the product of this thesis is a tool to construct tools (programming languages) to construct tools (programs). These final tools, programs, can of course be ethically questionable, or perfectly ethical, but considering this at such a removed level is too wide a topic to be practically applicable in this thesis.

On a more direct level, however, the purpose is essentially to make programming language construction easier, i.e., making the work of a certain subset of people easier, which seems like a morally positive thing to do.

Environmentally speaking, the most clear impact would be the power consumption required to run a program. This thesis makes no claim or effort on execution efficiency (though it is part of plans for future work), thus there is a likelihood that the programs produced would be inefficient, and thus have a high power consumption. Additionally, by making programs and programming languages easier to construct we increase the likelihood that programming will

be used for some problem where it would previously have been implausible. The impact of this is unclear and depends on a comparison with the impact of whatever would otherwise have been used, or nothing if the problem would not have been solved at all.

Chapter 2

Related Work

The related work is separated roughly based on the degree to which they solve the two problems introduced in sections 1.2.1 and 1.2.2: solutions for abstraction problems, solutions for syntax problems, and solutions that address both problems.

2.1 Abstraction Solutions

The solutions in this section address some part of the abstraction problem, i.e., they ensure that some aspect of the underlying implementation is invisible and properly abstracted away. However, they either do no work on concrete syntax, or have very limited syntactical freedom.

Accidental name capture is largely a solved problem in the Lisp world, most commonly through renaming (e.g., [7]). The latest iteration of Racket's [6] macro expander instead achieves binding hygiene by using sets of scopes [5].

Rather simplified, each identifier is equipped with a set of scopes based on, for example, its lexical context or the macro it was introduced by. Name resolution then compares the sets attached to binding identifiers and referencing identifiers to determine how they relate. The authors report the implementation as simpler to follow than the previous version using renaming, and while ambiguous

references are possible, they do not appear in practice. Neither of these solutions deal with errors after expansion.

λ_m [9] takes a different route and is an extension of the lambda calculus with macros and macro type signatures. These signatures describe the structure of macro arguments and results, including binding structure, and form an inspirational basis to the approach taken by this thesis in regard to hygiene and name binding.

There is a rather notable difference however: bindings in λ_m macros are always nested, in the sense that bindings are never available outside of the macro that introduced them. In contrast, consider these declarations in Haskell:

```
even n = case n of
  0 -> True
  1 -> False
  n -> odd (n - 1)
odd n = case n of
  0 -> False
  1 -> True
  n -> even (n - 1)
```

The functions `even` and `odd` are available both before and after their declarations, not only in some nested expression.

In addition, λ_m introduces definitions of α-equivalence and hygiene that do not depend on macro expansion and places focus on allowing reasoning about unexpanded programs, both of which are also present in this thesis.

Romeo [13] continue the path of typed macros but extends it to allow procedural macros, as opposed to the pattern matching and replacing of λ_m. However, like λ_m, Romeo focuses on nested bindings.

2.2 Syntax Solutions

The solutions in this section deal with defining syntax, generally in extensible ways. However, they deal only with syntax, not semantics.

The syntax definition formalism (SDF) [8] is a system with a combined notation for lexical and context-free grammar, along with a parser. Productions are listed individually, allowing non-terminals to be spread across files in a very similar fashion to the syntax constructions of this thesis. SDF also uses associativity annotations for operators instead of necessitating a manual rewriting of the grammar.

A specification in SDF is used to generate a lexical grammar and a context-free grammar, both of which are then used for the actual parsing. The productions in the specification also double to describe an abstract syntax tree, which is the end result. This means that SDF retains the separation of syntax and semantics, a single feature will be spread over several phases, of which SDF provides one. In case of an ambiguous parse SDF produces a parse forest, while this thesis attempts to describe the ambiguities in a useful way to the user.

SDF has also been extended [4] to handle layout-sensitive grammars through extra annotations, thus retaining the declarative nature of a grammar.

Silkensen and Siek [12] consider the problem of combining already constructed grammars in a scalable way. The main observation they use is that most domain specific languages deal with values of different types. With this in mind the different grammars can use these types, for example using non-terminals such as `Matrix` instead of `Expression`. Additionally, if identifiers can be specified to have a specific type (e.g. `Matrix`) they can be used as islands in island parsing. By using these two things the authors present a parsing algorithm that needs to examine far fewer possible parsings, even in the presence of many DSLs. The precise scalability claim can be found in the paper. The efficiency is based on the presence of type annotations on variable declarations, which are thus required.

2.3 Combined Solutions

This section contains solution that address both abstraction preservation and syntax. However, they deal with complete languages as the unit of composition.

SoundX [11] is a system for specifying extensible languages. It uses SDF (see section 2.2) to specify syntax and adds type rules and type judgements. A language extension is specified as a set of rewritings (macros) and type rules for the rewritings. Macros in SoundX are checked to guarantee that they preserve abstraction on a type level, i.e., they introduce no type errors during rewriting. Similarly, code can be type checked without performing rewriting.

The system works through a rewriting of derivation trees, not pure syntax, thus macros have access to not only types explicit in a program but also derived types.

Copper [16] and Silver [17] together form a system for extensible languages based on attribute grammars. Copper defines a grammar and a lexical scanner that work in tandem, where the scanner only returns tokens that the grammar defines as valid next tokens given the current state of the parse. The combination manages to parse more languages, since the scanner can work unambiguously in more cases. Silver allows modular language extensions on some host language, all of them defined using attribute grammars, along with some guarantees on their behaviour under composition [10].

Chapter 3

Design of Syntax Constructions

This section details the design and motivations behind syntax constructions, the main product of this thesis. Syntax constructions are macros, similar to those in various Lisp dialects, but with additional guarantees and capabilities. This chapter introduces the various parts of a syntax construction, all the while using the design goals (previously stated in Section 1.4) to motivate choices made.

The algorithms required to implement the semantics of syntax constructions are described to some extent here, as relevant for the design, but are explained more thoroughly in Chapter 4. This chapter is mostly written for a language designer using syntax constructions, while Chapter 4 contains more precise semantics.

3.1 Syntax Constructions and Syntax Types

Syntax constructions center around two concepts, the constructions themselves and their types. These types, which from here on out will always be referred to as syntax types, should not be confused with the types associated with a value. For example, the literal `4` has type `int`, while the syntax construction representing an integer literal has syntax type `Expression`.

A syntax construction is essentially a macro with some additional features and guarantees, while a syntax type is something along the

lines of **`Expression`**, **`Statement`** or **`Pattern`**. These syntax types should facilitate statements like: ”a sum is an expression and consists of an expression, a plus, and another expression”.

A singular syntax construction forms the unit of composition; they can be included or excluded in a language on an individual basis. A language is thus a set of syntax constructions.

3.2 Syntax

This section describes how the concrete syntax of a language defined with syntax constructions is defined. This section also uses the definition of a simple arithmetic language as a running example. The language contains addition, multiplication, grouping through parentheses, integer literals, and vectors. The language definition can be found in Listing 1. Note that Listing 1 only contains what is relevant for the concrete syntax. The omitted parts are written `<...>`.

From a syntactical point of view, a syntax construction is defined by the following things:

- Its name (e.g. `addition`, on line 2 in Listing 1).
- Its syntax type (e.g. **`Expression`**, on line 2 in Listing 1).
- A syntax description (e.g. `a:`**`Expression`** `"+"` `b:`**`Expression`**, on line 3 in Listing 1).
- Some (optional) extra data (e.g., precedence via `#prec` (line 5) and associativity via `#assoc` (line 6)).

The intuition to use here is that each syntax construction is a single production in a context-free grammar (CFG). For example, `addition` represents the following production:

⟨*Expression*⟩ ::= ⟨*Expression*⟩ ’+’ ⟨*Expression*⟩

All user defined syntax types essentially represent non-terminals. Additionally, there are five built in syntax types that represent terminals:

1. **`Integer`** matches an integer token, e.g., `42`.

```
1  syntax type Expression = <...>
2  syntax addition:Expression =
3    a:Expression "+" b:Expression
4  {
5    #prec 11
6    #assoc left
7    <...>
8  }
9  syntax multiplication:Expression =
10   a:Expression "*" b:Expression
11 {
12   #prec 12
13   #assoc left
14   <...>
15 }
16 syntax parens:Expression = "(" e:Expression ")"
17 { <...> }
18 syntax integerLiteral:Expression = i:Integer
19 { <...> }
20 syntax vector:Expression =
21   "[" e:Expression ("," es:Expression)* "]"
22 { <...> }
```

Listing 1: A simple language for basic arithmetic.

2. **`Float`** matches a decimal number token, e.g., `3.14`.
3. **`String`** matches a string token, e.g., `"foobar"`.
4. **`Identifier`** matches an identifier token, e.g., `name`.
5. **`Symbol`** matches punctuation and operators, e.g., "`(`", "`+`", and "`!=`".

As mentioned in the delimitations of this thesis (Section 1.4.1), parsing is done on an already tokenized file, i.e., a token stream. This token stream contains five kinds of tokens, where each of the terminals above match exactly of of those kinds. The lexical syntax of each of these is chosen to align as well as possible with the syntax of common programming languages, which works well enough in most cases.

Note also that the lexer produces no keywords. What would normally have been a keyword is instead lexed as an identifier, which a syntax construction can match against by using a quoted literal in the syntax description. Additionally, any identifier that appears as a quoted literal will *not* be matched by `Identifier`. For example,

```
syntax let:Expression =
  "let" x:Identifier "=" e:Expression
  "in" body:Expression
{ <...> }
```

has two quoted identifier literals: `"let"` and `"in"`. A language that contains this syntax construction will not be able to use `let` or `in` as identifiers.

The built in syntax type `Identifier` then matches any identifier that never appears in a quoted literal, effectively emulating reserved keywords.

Additionally, a syntax description can use extended backus-naur form (EBNF) operators: ”*” for zero or more repetitions, ”+” for one or more, and ”?” for zero or one. As an example, the syntax description of `vector` (line 21 in Listing 1) uses ”*” to enable writing a vector with any number of components. Having these EBNF operators allows us to express more language features as a single syntax construction. For example, `vector` would need multiple syntax constructions, and one more syntax type.

Using this intuition, and for the moment ignoring precedence and associativity, the language in Listing 1 represents the following grammar:

$$
\begin{aligned}
\langle \textit{Expression} \rangle \quad ::= \; & \langle \textit{Expression} \rangle \text{ '+' } \langle \textit{Expression} \rangle \\
| \; & \langle \textit{Expression} \rangle \text{ '*' } \langle \textit{Expression} \rangle \\
| \; & \text{'('} \langle \textit{Expression} \rangle \text{ ')'} \\
| \; & \langle \textit{Integer} \rangle \\
| \; & \text{'['} \langle \textit{Expression} \rangle \text{ (','} \langle \textit{Expression} \rangle)^* \text{ ']'}
\end{aligned}
$$

This grammar is quite ambiguous, but the syntax constructions above contain additional information we can use to construct an unambiguous grammar instead: precedence and associativity. Including precedence and associativity in a grammar is a rather well

known transformation that adds an extra non-terminal for every precedence level, like so:

⟨*Expression*⟩ ::= ⟨*Expression*⟩ '+' ⟨*Expression1*⟩
| ⟨*Expression1*⟩

⟨*Expression1*⟩ ::= ⟨*Expression1*⟩ '*' ⟨*Expression2*⟩
| ⟨*Expression2*⟩

⟨*Expression2*⟩ ::= '(' ⟨*Expression*⟩ ')'
| ⟨*Integer*⟩
| '[' ⟨*Expression*⟩ (',' ⟨*Expression*⟩)* ']'

This method of specifying precedence and associativity is also commonly available in parser generators, such as Yacc.

The precedence and associativity transformation above is slightly generalized from the common one: it applies to any production, not only to operators. Precedence affects recursive uses of the same syntax type (i.e., using **`Expression`** in an **`Expression`**), regardless of position in the syntax description, by only allowing recursing to the same level or the next. Associativity further affects this by allowing recursion to the same level only in the leftmost recursion (with left associativity) or the rightmost recursion (with right associativity).

This generalization was chosen to align with the normal definition of operator precedence, but to also be usable as a tool for disambiguation in other syntactic forms.

The subject of precedence bears further elaboration. It is required in some form or other, otherwise a very large class of languages would be inexpressible, e.g., $a + b * c$ would be ambiguous. However, precedence only has meaning when comparing multiple operators to each other, which is in direct conflict with the design goal of only requiring a user to consider each syntax construction in isolation. The current solution describes precedence as a number, which can then be compared with the precedence of any other syntax constructions without either referring to the other. We thus have no direct, explicit dependence between syntax constructions. However, it does present an implicit dependence on all other syntax constructions of the same syntax type. Other options might include explicitly stating one syntax construction as preferred over another, similar to SDF [8].

A reader with previous experience of context-free grammars may

have some qualms at this point; if the syntactical part of syntax constructions is just a context-free grammar with some rewriting help, won't ambiguous grammars be an issue? Context-free grammars are not unambiguous in the general case, and determining whether a given grammar is ambiguous is not always obvious. Syntax constructions provide no static guarantees on ambiguity, but provides good error messages when ambiguous code is encountered. See sections 4.3.3 and 5.6.1 for implementation and evaluation of these errors, respectively, and sections 5.2.1 (specifically p. 96) and 5.2.2 (specifically p. 103), amongst others, for consequences and evaluation.

As a final note, the capabilities described in this section are present in most parser generators in one form of another.

3.3 Bindings

To support the goal of abstraction preservation and good error messages, syntax constructions include name binding semantics. This brings us past the capabilities of parser generators.

To motivate the design of the binding semantics, this section will examine three examples.

First, a `let`-binding in OCaml introduces new names to a subtree (`a` is bound in `print_string a` in this case):

```
let a = "value" in
print_string a
```

We can codify these semantics with the following syntax construction:

```
syntax let:Expression =
  "let" x:Identifier "=" e:Expression
  "in" body:Expression
{
  #bind x in body
  <...>
}
```

The syntax type **Identifier** is a built in type that participates in name binding. `#bind x in body` specifies that `x` is part of a definition, otherwise it would be interpreted as a reference that must be bound in the current environment. In the former case we say that `x` is in a binding position, while in the latter case `x` is in a referencing position.

Second, in Java, variables cannot be used before their declaration, nor after the scope they were introduced in ends:

```
String first = "first";
{  // A block opens a new scope
  String second = "second";

  // Error: cannot find third
  System.out.println(first + second + third);

  String third = "third";
  System.out.println(first + second + third);
}  // Closes the new scope

// Error: cannot find second or third
System.out.println(first + second + third);
```

These semantics can be described using only capabilities introduced so far, but with some drawbacks. The idea is to treat each statement as if it contained all the statements after it in a subtree. This allows a declaration to bind a name in all following statements:

```
syntax declaration:Statement =
  t:Type x:Identifier "=" e:Expression
  ";" next:Statement
{
  #bind x in next
  <...>
}

syntax block:Statement =
  "{" s:Statement "}"
  ";" next:Statement
```

```
{
  <...>
}

syntax call:Statement =
  e:Expression
  "(" (a:Expression ("," as:Expression)*)? ")"
  ";" next:Statement
{
  <...>
}

syntax empty:Statement = {
  <...>
}
```

However, we have to introduce some boilerplate, each **Statement** must contain ";" next:**Statement**, and we need an additional empty **Statement** to terminate a list of statements.

This specification also breaks somewhat with intuition, conceptually our second example contains a list of statements, one of which contains sub-statements, but using the specification above we only ever refer to singular **Statement**s.

While bindings in subtrees are capable of expressing the desired semantics, the description is unintuitive and verbose.

However, our third example cannot be expressed in this way. A function in JavaScript can be used both before and after its declaration, permitting mutual recursion:

```
function foo(n) {
  console.log("foo", n);
  if (n > 0) bar(n-1);
}
function bar(n) {
  console.log("bar", n);
  if (n > 0) foo(n-1);
}
```

To express this with only nested binding we would require both function declarations to contain the other as a subtree.

To solve this, we introduce a separate kind of binding: a syntax construction is allowed to specify that an identifier is bound before itself, after itself, or both. This allows bindings that are not limited to subtrees. Additionally, we introduce a concept of scope, which bounds the area cover by this form of binding. To demonstrate both of these we have the following syntax construction, which codifies a function declaration in JavaScript:

```
1  syntax funcDecl:Statement =
2    "function" f:Identifier "("
3    (a:Identifier ("," as:Identifier)*)?
4    ")" "{" body:Statement+ "}"
5  {
6    #bind f before
7    #bind f after
8    #bind f, a, as in body
9    #scope (body)
10   <...>
11 }
```

Lines 6 and 7 state that the identifier `f` is bound before and after the function declaration, respectively. Line 8 makes the function and its parameters available in the body. Finally, line 9 introduces a scope around `body`, which prevents declarations in the body from being accessible from the outside.

This solution once again disconnects statements, allowing each to be defined in isolation, even when they need to affect adjacent statements with a binding.

The interaction between EBNF operators and scopes bears further elaboration. A name used in a `#scope` declaration refers to all repetitions of the name. For example, a **`match`**-expression in OCaml has the following form:

```
match list with
  | [] -> "empty"
  | [a] -> "one element: " ^ a
```

```
  | _ -> a (* error, a unbound *)
```

The bindings introduced in each alternative is only available in that alternative, the example thus has a binding error on the last line.

We could represent a `match`-expression with the following syntax construction:

```
1 syntax match:Expression =
2   "match" e:Expression "with"
3   arm:("|" pattern:Pattern "->" body:Expression)+
4 {
5   #scope (pattern body)
6   <...>
7 }
```

However, this definition is incorrect. The `#scope` declaration on line 5 introduces a single scope that covers *all* repetitions of `pattern` and `body`. Instead, we want the following declaration:

```
#scope arm:(pattern body)
```

This introduces a new scope for every repetition of `arm`, covering the `pattern` and `body` that occur within. Note that this works even in the case of nested EBNF operators. Given this syntax description:

```
outer:(inner:(a:T b:T)+ c:T)*
```

Then the following, more complicated, examples should be interpreted as follows:

1. `#scope inner:(a b)`

   Introduce a new scope for each repetition of `inner`, covering the `a` and `b` within. This means that any `a` or `b` can use declarations from any repetition of `c`, but not from any repetition of `a` or `b` in another repetition of `inner`.

2. `#scope outer:(a b c)`

   Introduce a new scope for each repetition of `outer`, covering the `a`s, `b`s, and `c` that occur within. This means that each `a`, `b`,

and `c` can use declarations from any other `a`, `b`, or `c`, as long as they occur within the same repetition of `outer`.

3. `#scope (c inner:(b))`

   Introduce a new scope covering all repetitions of `c`, then introduce a nested sub-scope for each repetition of `inner` that covers the `b` within. This means that:

   - All repetitions of `a` can use declarations from all other repetitions of `a`, but none from any `b` or `c`.
   - Each repetition of `b` can use declarations from any repetition of `a` or `c`, but none from any other `b`.
   - Each repetition of `c` can use declarations from any repetition of `a` or any other `c`, but none from `b`

The last declaration also demonstrates that scopes need not be contiguous: `a` can occur in-between different repetitions of `c`, yet it will not be a part of the single scope that covers all repetitions of `c`.

3.4 Expansion

Syntax constructions, being essentially macros, need a way to specify how they expand into whatever underlying language they are implemented on top of. Additionally, the goal of abstraction preservation requires that the implementation of a syntax construction is never exposed to an end user. As such, the expansion must never fail or produce a malformed program.

To achieve this the expansion is specified in a fairly limited way, which makes checking the implementation simpler.

Furthermore, the system requires a base language, some point at which point expansion is finished. Syntax constructions themselves pose no requirements on this language and instead allow a construction to be marked as ”builtin”, meaning part of the base language.

As expansion proceeds and syntax constructions are replaced by syntax constructions closer to the base language, a similar

transformation will take place on syntax types, a program will go from using the types of its language, to the types of the host language, etc., all the way down to the base language. To ensure that the final expanded program is syntactically correct, each syntax type must designate either its underlying type or that it is a syntax type of the base language, i.e., **builtin**. For example:

```
syntax type BaseExpression = builtin
syntax function:BaseExpression =
  "fun" x:Identifier "." e:Expression
{
  #bind x in e
  builtin
}
syntax functionApplication:BaseExpression =
  f:BaseExpression a:BaseExpression
{ builtin }
```

In the simplest case, an expansion is simply a syntactical expression in the target language, with elements of the original syntax construction spliced in, similar to quote and unquote capabilities in similar systems. For example:

```
1  syntax type Expression = BaseExpression
2  syntax let:Expression =
3    "let" x:Identifier "=" e:Expression
4    "in" body:Expression
5  {
6    #bind x in body
7    #scope (body)
8    #scope (e)
9    BaseExpression` (fun `id(x). `t(body)) `t(e)
10 }
```

Line 1 introduces **Expression** as a new syntax type, and that **Expression**s should expand to **BaseExpression**s. Line 9 is the expansion template of `let`, which will now be explained in greater detail.

BaseExpression` specifies that what follows should be parsed as a

BaseExpression. This is necessary as a consequence of two things:

1. We want the expansion to be flexible. This means that we might not expand to a **BaseExpression** right away, instead we expand to something that will itself eventually expand to a **BaseExpression**.

2. The exact same syntax can have different meanings depending on where it appears in a source file. For example, in OCaml:

```
1  match list with
2    | [1] -> [1]
```

   Line 2 here contains the syntax "`[1]`" twice; first as a pattern and then as an expression.

A core underlying language tends to be designed to be simple, it might for example only have expressions, i.e., everything is a **BaseExpression**. This means that both occurrences of "`[1]`" will expand to something with syntax type **BaseExpression**, but with very different semantics.

Returning to the expansion template of `let`, we must thus specify the syntax type we wish to parse.

Next is a function (`fun `id(x). `t(body)`), which is immediately applied to an argument (`` `t(e) ``). Here, `` `id `` and `` `t `` are similar to "unquote" in Lisp-like macro systems; they denote holes in the template into which we splice other expressions. In this case, we splice in `x`, `e`, and `body`, which were introduced in the syntax description on lines 3-4.

The splicing syntax has six forms: one for each kind of token (i.e., `` `int ``, `` `float ``, `` `str ``, `` `id ``, and `` `sym ``), and one for any other syntax type (`` `t ``). These are there to ensure that the parser knows how to interpret the splice, which is necessary since it is parsing a language that may produce very different interpretations based on what is spliced in.

It is worth noting here that in most cases this tag is not strictly necessary; we have already seen the syntax description, thus we already know that `x` is an **Identifier**. However, the parser in the current implementation is context-free, and this form of memory of

things we have previously seen requires a context-sensitive parser. The requirement of tagging a splice is thus a limitation of the current implementation, and not necessarily a fundamental limitation of the method.

If the original syntax construction contains EBNF operators, some elements may occur multiple times, forming lists of syntactical elements. These may not be used directly in the expansion, where only singular syntactical elements are allowed. To produce a singular syntactical element from a list one may instead use a fold:

```
syntax switch:Expression =
  "switch" e:Expression "{"
  cases:("case" test:Expression
         ":" result:Expression)*
  "default" ":" default:Expression "}"
{
  Expression`
    let x = `t(e) in
    `t(foldr cases rest
         (Expression` if `t(test) == x
                        then `t(result)
                        else `t(rest))
         default)
}
```

Here a switch is translated to a series of `if`-expressions, checking for equality with each case in turn. The **`foldr`** expression takes four arguments: what list to fold over, the name of the result so far (the accumulator), the expansion template that will be used to construct the next result, and the initial value.

A reader with previous experience of folds may be somewhat surprised by the syntax used here: nothing looks terribly much like a function and no name is given to the current item in the list being folded over. Using psuedocode and named arguments the above fold is essentially equivalent to:

```
foldr(list = cases,
      initial = default,
      f = ((test, result), rest) =>
```

```
        (Expression` if `t(test) == x
                      then `t(result)
                      else `t(rest)))
```

The function is run once per repetition of `cases` and can use all named syntax constructions that are not in a nested inner repetition or in a repetition that does not contain `cases`. For example, in `example` below the following things are true:

- The first two folds are equivalent, since `second` and `a` repeat exactly as often.
- All bodies can use `a` as a singular value.
- Only `body3` and `body4` can use `b` as a singular value.
- The third fold will use `body3` once per occurrence of `b` across all repetitions of `second`, while the nested fold in the fourth will only use `body4` once per `b` in the *current* repetition of `second`.

```
syntax example:T =
  first:"first"+ second:(a:T b:T*)+
{
  foldr second acc
    <body1>
    <initial>

  foldr a acc
    <body2>
    <initial>

  foldr b acc
    <body3>
    <initial>

  foldr second acc1
    (foldr b acc2
      <body4>
      <initial>)
    <initial>
}
```

Having all named syntax elements implicitly available turned out to be more convenient than requiring some form of pattern matching or explicitly mentioning the ones you wanted to use, and presents no ambiguity for the reader as long as you already know the syntax.

The example uses a right fold, additionally left fold and versions without an initial value are available, the latter only being usable if the list contains at least one value (i.e., it was constructed using the ”+” operator, not ”*” or ”?”).

3.4.1 Expansion Checking and Correctness

One of the design goals of syntax constructions is that they must provide good error messages when some aspect of the system is misused. For the purposes of this section a good error message presents a user with the information required to fix the error, and presents that information in terms of code that user is working with. Notably, then a poorly implemented syntax construction should result in an error message for the syntax construction implementer, not for an unsuspecting end-user using the syntax construction. Similarly, an end-user misusing a syntax construction should get an error message referring to their code, not to its expansion.

To ensure that this split is maintained, we require that expansion never introduces an error, i.e., error-free code is expanded into error-free code. Following the delimitations in Section 1.4.1, ”error-free code” here means code that has no binding errors, i.e., no unbound references and no redefinitions in the same scope.

This places a fairly stringent requirement on the syntax construction implementer: the expansion of a syntax construction must never introduce an error, no matter what.

Presenting an error to an end-user misusing a syntax construction is then straight-forward: perform name resolution on the unexpanded abstract syntax tree and report any error encountered. The fact that we can do this at all stems from the presence of binding semantics on syntax constructions, this cannot be done in most Lisp dialects since a Lisp macro does not have this information. If there are no errors, then the code is correct and can be fully expanded without introducing

errors.

Ensuring that an expansion will never introduce an error is more difficult. The expansion must be correct regardless of the structure of the surrounding syntax tree. For example, the following definition of `let` contains a somewhat subtle bug:

```
syntax let:Expression =
  "let" x:Identifier "=" e:Expression
  "in" body:Expression
{
  #bind x in body
  #scope (body)
  Expression` (fun `id(x). `t(body)) `t(e)
}
```

If `e` contains a declaration that is available after `e` (i.e., a binding introduced by `#bind after`), and `body` uses it, then this expansion will introduce an error. The expansion has `e` appearing *after* `body`, which breaks the reference and introduces an error.

The language designer may at this point consider this somewhat nonsensical, perhaps no expressions in their language expose names in that way, but since syntax constructions are made for composition and reuse we cannot assume that to always be true. Thus checking assumes an open universe of syntax constructions, i.e., all syntax types conceptually include syntax constructions that both export and use names in all possible ways, and an expansion must not introduce an error, even in their presence.

In this particular case the solution is to place `e` in its own scope, which prevents `body` from referencing any names from `e`:

```
syntax let:Expression =
  "let" x:Identifier "="
  e:Expression "in"
  body:Expression
{
  #bind x in body
  #scope (body)
  #scope (e)
```

```
  Expression` (fun `id(x). `t(body)) `t(e)
}
```

Additionally, this aligns better with the intended semantics; it specifies explicitly that nothing bound in `e` can be used elsewhere.

The process by which the expansion template of a syntax construction is checked for correctness is described in the next chapter, in Section 4.7.

Chapter 4

Formalization

This chapter takes the informal description of Chapter 3 and develops a set of more formal definitions and algorithms. Figure 4.1 shows an overview of the different parts and how they relate to each other in terms of inputs and outputs.

For the purposes of this section the system as a whole takes two inputs:

A list of tokens: The source code we wish to parse and process, represented as a list of tokens.

A set of syntax constructions: All the syntax constructions that may be used throughout the system. A subset of these specify the programming language in which the source code is written. The rest are part of the core language, or any intermediary language.

The final output is a fully expanded *instance* of some syntax construction. For all intents and purposes, an instance is an abstract syntax tree (AST) of the source code we are processing, and the two terms will mostly be used interchangeably. ”Fully expanded” means that it only contains constructs from the core language, all higher-level language constructs have been expanded and replaced with lower-level constructs.

Section 4.2 defines the various components that make up a single syntax construction, which all other sections will then use.

We will now briefly describe the different parts of Figure 4.1 and

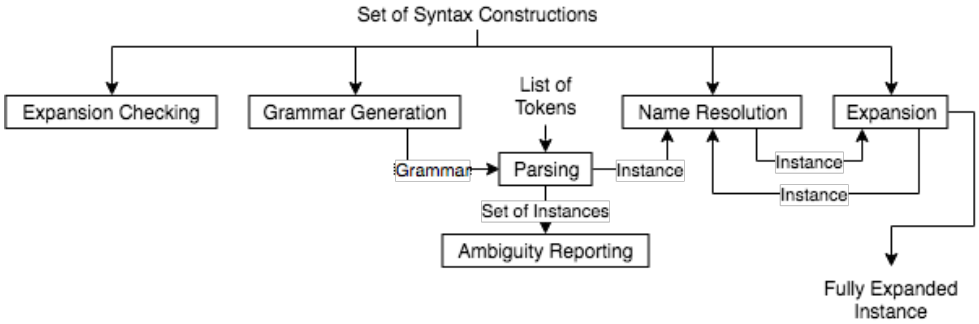

Figure 4.1: The different algorithms that formalize syntax constructions, and how they relate to each other.

describe their relation to each other. Expansion checking can be run entirely separately from the rest of the system, but requires previous knowledge from the other parts, and will thus be explained last.

Grammar generation (Section 4.3.1) takes a set of syntax constructions, more precisely the set that defines our language, and produces a context-free grammar (CFG).

Parsing (Section 4.3.2) takes the CFG from grammar generation and the list of tokens describing the source code and produces zero or more instances (ASTs). Zero instances means parsing failed, one means it succeeded, more than one means the source code was ambiguous.

Ambiguity reporting (Section 4.3.3) takes a set of instances and produces an error message of the form ”these parts of the source code are ambiguous”, thus being more informative than the trivial ”the source code is ambiguous”.

Name resolution (Section 4.5) takes the complete set of syntax constructions and an instance, and checks the instance for binding errors. The binding semantics for an instance is specified in the syntax construction of which it is an instance. Name resolution also performs (binding preserving) renaming to ensure that each symbol occurs at most once in a binding position.

Expansion (Section 4.6) takes an instance where each symbol appears in binding position at most once and expands a single instance somewhere in the AST. Note that the resulting instance might have a

symbol in a binding position more than once, and there may be more instances that could be expanded. The general case will thus interleave expansion and name resolution until no more instances can be expanded, at which point the final, fully expanded instance has been produced.

Expansion checking (Section 4.7) examines a single syntax construction to ensure that it cannot introduce a binding error through expansion. This means that if an AST has no binding errors, then the expansion of some instance cannot introduce a new binding error, i.e., all errors can be reported in terms of the original source code.

As mentioned previously, expansion checking can be run entirely independently of the rest of the system, none of the other steps require it. In particular, expansion checking requires no source code and no other syntax constructions than the one currently being checked. However, all statements regarding the abstraction preserving nature of syntax constructions requires that each syntax construction has been checked and found correct. In practice, expansion checking should thus be performed before all other steps.

4.1 Running Example

The remainder of this chapter will feature a running example: a programming language (defined in Listing 2) and a small source code example in that language (Listing 3). The reader may wish to bookmark these listings, as this chapter is fairly long and they will be referred to throughout. This section will explain the example in greater detail.

The language consists of six syntax constructions: `top`, `var`, `lit`, `sum`, `def`, and `block`, and one syntax type: **`Expression`**. The language is implemented in terms of a Lisp-like core language (which can be found in Appendix A, also defined using syntax constructions).

Line 1 of Listing 2 defines the syntax type **`Expression`**, and states that its representation is the same as that of the syntax type **`Core`**, which is provided by the core language. We do this, instead of just using **`Core`** directly, for parsing reasons: when we attempt to parse

```
syntax type Expression = Core
syntax top:File = e:Expression
{ File` `t(e) }
syntax var:Expression = id:Identifier
{ Core` `id(id) }
syntax lit:Expression = i:Integer
{ Core` `int(i) }
syntax sum:Expression =
  a:Expression "+" b:Expression
{
  #prec 2
  #assoc left
  Core` (sum `t(a) `t(b))
}
syntax def:Expression =
  "def" id:Identifier "=" e:Expression
{
  #bind id after
  #prec 1
  Core` (bind_after `id(id) `t(e))
}
syntax block:Expression =
  "{" (es:Expression ";")* "}"
{
  #scope (es)
  foldl es prev
    (Core` (seq `t(prev) `t(es)))
    (Core` 0)
}
```

Listing 2: The language that will be used as a running example.

```
{
  def a =
    {
      def a = 4;
      5;
    };
  a + 2 + 3;
}
```

Listing 3: Source code for the running example. Should evaluate to 10.

an **`Expression`** we will only try to parse syntax constructions explicitly defined to have syntax type **`Expression`**. This means that our new language is syntactically separated from the core language.

The remainder of Listing 2 defines the six syntax constructions. As an example, consider `sum`, defined on lines 8-14. Broadly speaking, a syntax construction is defined in three parts: a *header* (line 8), a *syntax description* (line 9), and a *body* (lines 10-14):

Header: The header declares the name of the syntax construction (`sum`) and its syntax type (**`Expression`**).

Syntax description: The syntax description describes the concrete syntax of the syntax construction, i.e., how it should be parsed when we parse source code in our language.

Body: The body is split in two parts:

- Some optional extra information, e.g., precedence (line 11), binding information (in `def`, on line 18), or scoping information (in `block`, on line 25).
- An expansion template (line 13), specifying how this syntax construction is to be expanded.

The beginning of line 13, **`Core`**`, specifies that what follows should be parsed as a syntax construction of syntax type **`Core`**. Had it instead said **`Expression`**` then the remainder of the body would have been parsed as a syntax construction of syntax type **`Expression`**. Both would be valid when considering the syntax type of the expansion, as it only needs to be representationally equal to **`Expression`**. In general, parsing uses nominal syntax types while expansion uses representational syntax types. These serve as parallels to nominal and structural type systems, where the former treats differently named types as different types, while the latter treats all types with the same structure as the same type.

The remainder of line 13 is an expression in the core language, a call to the function `sum` with two arguments: `` `t(a) `` and `` `t(b) ``. These arguments represent the **`Expression`**s `a` and `b` from the syntax description. The "`t`" in `` `t(a) `` means that we are using a syntax construction `a`, while `` `int(i) `` (e.g., in `lit` on line 7) instead means that we are using an integer from the syntax description. This extra

syntax for using elements from the syntax description in the expansion template is due to a limitation in the parser, and is discussed in more detail in the previous chapter, in Section 3.4.

The syntax description of `block`, on line 23, demonstrates an additional capability: repetitions via "`*`". Syntax constructions additionally support "`+`", and "`?`", as commonly seen in regular expressions and EBNF. The "`*`" after the parentheses on line 23 denote that a `block` can contain zero or more **`Expression`**s, each followed by a semicolon.

The expansion template, on lines 25-28, must then reduce the unknown number of **`Expression`**s to a exactly one. This is done through a left fold, folding over `es` and naming the accumulator `prev`.

Line 27 contains an expansion template, and is essentially the function we are folding with. Here `prev` will refer to the accumulator, and `es` will refer to the *current* element of the list we are folding over. `seq` is sequential composition, defined in the core language, i.e., it evaluates its first argument and then returns the second.

Line 28 contains the initial value of the accumulator: an expression that will evaluate to `0`. This means that an empty `block` evaluates to zero, but otherwise has no effect, since sequential composition discards the previous value.

Three of the syntax constructions in Listing 2 interact with name binding: `var`, `def`, and `block`.

`var` introduces a reference to a binding. This is signified by the **`Identifier`** in the syntax description that does not appear in a `#bind` declaration. We say that the identifier is in a *referencing position*, since it should reference a binding elsewhere. This is similar to a free identifier, as commonly used in the literature, but makes no statement on whether the identifier appears bound or free.

`def` introduces a binding *after* itself, as signified by the `#bind id after` declaration on line 18. We say that this identifier is in a *binding position*, since it is used to introduce a new binding. In particular, the binding is not available in `e`, this syntax construction does not allow recursive definitions. Had there been an additional declaration of the form `#bind id in e`, then recursive definitions

would have been allowed.

`block` introduces a *scope*, preventing bindings inside the `block` from being available outside. This is declared through `#scope (es)` on line 23, which means "introduce a new scope that covers all repetitions of `es`."

And finally, **`File`** is a syntax type introduced by the core language to represent an entire file. When parsing source code the parser will try to parse a syntax construction with this syntax type to cover the entire file. In grammar terms, **`File`** is the start symbol.

4.2 Defining Syntax Constructions

This section will introduce the various components that make up a single syntax construction, finally stating the complete definition of a syntax construction at the end of the section.

Some parts of this formalization will use a line above an expression to mean a sequence of expressions. For example, $\overline{i}$ denotes a sequence of is, i_n denotes the nth element in the sequence, and $\overline{i_n + 1}$ denotes a sequence of $i_n + 1$ for each n (i.e., a mapping over the sequence). For example, given the sequence $\overline{i} = [1, 2, 3]$, we have $\overline{i_n + 1} = [2, 3, 4]$. Additionally, given two sequences $\overline{a} = [1, 2]$ and $\overline{b} = [3, 4]$, $\overline{(\overline{a}, b_n)} = [([1, 2], 3), ([1, 2], 4)]$, i.e., $\overline{a}$ stays a sequence since it has no index and retains the line. An empty sequence will be written ϵ.

Additionally, as is common in functional programming languages, we will use _ to denote a wildcard, something whose value is unimportant or is allowed to have any value. For example, $f(\_) = 4$ is a function that ignores its argument and always produces 4.

Syntax Type

A syntax type t is given by a name, drawn from the set of all possible syntax type names. We will write them like the syntax types in Listing 2, e.g., **`Expression`**, **`Core`**, and **`File`**.

Line 1 in Listing 2 introduces the syntax type **`Expression`**, and additionally defines it to have the same representation as **`Core`**. We

write:

$$\texttt{Expression} \approx \texttt{Core}$$

We further define $\approx$ to be reflexive, symmetric, and transitive.

For our example language all syntax constructions have the syntax type **`Expression`**, except `top` which has syntax type **`File`**.

The core language defines the syntax type **`Core`** as follows:

```
syntax type Core = builtin
```

This declaration only introduces the new syntax type, it does not declare it to have the same representation as any other syntax type.

Additionally, syntax type declarations must refer to previously declared syntax types. This means that each partition formed by $\approx$ has exactly one syntax type declared as **`builtin`**.

Syntax Description

A *syntax description* describes the concrete syntax of the syntax construction it belongs to. It is defined by the following grammar:

$$
\begin{array}{rcl}
d & ::= & \textbf{identifier} \\
& | & \textbf{string} \\
& | & \textbf{integer} \\
& | & \textbf{float} \\
& | & \langle \textit{token} \rangle \\
& | & t \\
& | & \textbf{seq}\ \overline{d} \\
& | & \textbf{star}\ d \\
& | & \textbf{plus}\ d \\
& | & \textbf{question}\ d
\end{array}
$$

Note that although the syntax of a syntax description is an abstract syntax, it describes a concrete syntax, and will thus contain tokens. For the remainder of this chapter, tokens will be written quoted and with a different font. For example, `"foo"` is an identifier token, while `"4"` is an integer token.

The first four alternatives of the grammar match any token of the correct type, e.g., **integer** will match any integer token. ⟨*token*⟩ represents any literal used in the syntax description, for example `"+"` on line 9 in Listing 2. These will match a token containing precisely that text, a single plus symbol in the example. **seq** represents a parenthesized sequence (e.g., on line 23 in Listing 2), while **star**, **plus**, and **question** represent "`*`", "`+`", and "`?`", respectively.

For example, $d_{\texttt{block}}$), the syntax description of `block` (line 23 in Listing 2), is given by:

seq `"{"` (**star** (**seq** **`Expression`** `";"`)) `"}"`

Since the syntax used in Section 3.2 allows multiple direct children (i.e., without using an extra set of parentheses), but the syntax above does not, we implicitly insert **seq** around the description to get an equivalent description.

Another notable difference between the syntax descriptions in Listing 2 and d is that of names. The names appearing in the syntax descriptions in Listing 2 do not appear in the syntax description in the formalization, they are instead reintroduced in a different form later in this section.

From here we define the concept of a leaf of a syntax construction, those descendants that appear atomic: the first six alternatives in the grammar above are leaves.

Syntax Construction Instance

We will now define an *instance* of a syntax construction. When the parser determines that some source code should be parsed as a syntax construction c, we create an instance of it. These are technically not part of the definition of a syntax description, but later concepts require knowledge of the structure of an instance.

An instance i is defined by the following grammar:

⟨*i*⟩ ::= c **:>** i
| ⟨*token*⟩

| **seq** $\overline{\langle inner \rangle}$
| **rep** $\overline{\langle inner \rangle}$

The alternatives of i each represent, in order:

- An instance, parsed from a syntax type t. In particular, an instance of a syntax construction c always has the form c **:>** i, meaning that i is an instance of c.
- A single token, parsed from one of the first five alternatives in s.
- A sequence, parsed from a **seq**.
- A repetition, parsed from any of the three last alternatives in s (i.e., the EBNF operators).

For example, the `block` on lines 3-6 of Listing 3 has the following instance:

```
block :>
  seq "{"
      (rep (seq (def :> ...) ";")
           (seq (lit :> ...) ";"))
      "}"
```

where `def` **:>** ... is the instance of the `def` on line 3, and `lit` **:>** ... is the instance of the `lit` on line 4.

As a parallel to the leaves of a syntax construction, we define the leaves of an instance to be the ⟨*token*⟩s and ⟨*instance*⟩s that are reachable without passing through some other ⟨*instance*⟩. The leaves of the `block` instance above are the curly braces, the semi-colons, the `def` **:>** ..., and the `lit` **:>** ....

More concretely, we compute the set of leaves of an instance using *leaves*:

$$
\begin{aligned}
\mathit{leaves}(c \textbf{ :> } i) &= \mathit{children}(i) \\
\mathit{children}(\langle \textit{token} \rangle) &= \{\langle \textit{token} \rangle\} \\
\mathit{children}(c \textbf{ :> } i) &= \{c \textbf{ :> } i\} \\
\mathit{children}(\textbf{seq } \bar{i}) &= \bigcup_n \mathit{children}(i_n) \\
\mathit{children}(\textbf{rep } \bar{i}) &= \bigcup_n \mathit{children}(i_n)
\end{aligned}
$$

These will be useful in later sections (e.g. name resolution) that deal with leaves, the instance to which they belong, and the syntax construction the instance was instantiated from.

Paths

We will now reintroduce the names used in the syntax descriptions of Listing 2. Each name will be replaced by a path, a series of steps traversing d or i, selecting the element(s) to which the name referred. The specific formulation used below allow the same path to be used both in a syntax description and an instance, and additionally provide a simple way to refer to specific repetitions (from EBNF operators), a concept that is important for name resolution and expansion.

A path is given by the following grammar:

$$
\begin{array}{lll}
p & ::= & n : p \\
 & | & * : p \\
 & | & []
\end{array}
$$

Where n is a positive number. We will additionally write $[a, b, c]$ as shorthand for $a : b : c : []$. Informally, a step n traverses through the nth child, while a step $*$ traverses through *all* children. We will generally use p to range over paths, and P to range over sets of paths.

Each path belongs to some syntax construction c, and should only be used for traversal in the syntax description of c, or an instance of c. This is equivalent to saying that a name referring to an element of the syntax description of c is only meaningful for c.

For traversal, we define a function $follow$. This function allows us access to the element(s) the names referred to, and once we introduce more operations for manipulating paths, access to specific subsets of what the names referred to. To traverse d:

$$
\begin{array}{rcl}
follow([], i) & = & i \\
follow(n : p, \mathbf{seq}\ \bar{i}) & = & follow(p, i_n) \\
follow(\_ : p, \mathbf{star}\ i) & = & follow(p, i) \\
follow(\_ : p, \mathbf{plus}\ i) & = & follow(p, i) \\
follow(\_ : p, \mathbf{question}\ i) & = & follow(p, i)
\end{array}
$$

Note that we only allow a $*$ to step through one of the EBNF operators, $follow(* : p, \textbf{seq}\ \bar{i})$ is not defined. To traverse i:

$$\begin{aligned} follow([], i) &= \{i\} \\ follow(n : p, \textbf{seq}\ \bar{i}) &= follow(p, i_n) \\ follow(n : p, \textbf{rep}\ \bar{i}) &= follow(p, i_n) \\ follow(* : p, \textbf{rep}\ \bar{i}) &= \bigcup_n follow(p, i_n) \end{aligned}$$

As a consequence on the limitation on $*$, a traversal in d produces a single element, while traversal in i produces a (possibly empty) set of elements.

Translating a name to a path is now a matter of finding the corresponding node d' in the syntax description d, and producing a path p such that $follow(p, d) = d'$, and every traversal through one of **star**, **plus**, or **question** is done using a $*$. For example, the syntax description of `block` (on line 23 in Listing 2) has the following form:

seq `"{"` (**star** (**seq** **`Expression`** `";"`)) `"}"`

The name `es` (also on line 23) refers to the **`Expression`**. We find that **`Expression`** appears in the second element of the outer **seq**, inside the **star** expression, in the first element of the inner **seq**. The path $[2, *, 1]$ describes this traversal, i.e., $follow(2, *, 1], d) =$ **`Expression`**.

The `block` instance on lines 3-6 in Listing 3 has the following form:

```
block :> i = block :>
                seq "{"
                   (rep (seq (def :> ...) ";")
                        (seq (lit :> ...) ";"))
                   "}"
```

We thus find that $follow([2, *, 1], i) = \{($`def :> ...`$), ($`lit :> ...`$)\}$. If we wished to only access the first repetition of `es`, we could in stead use the path $[2, 1, 1]$: $follow([2, 1, 1], i) = \{($`def :> ...`$)\}$

As a final point, $follow$ will occasionally be used "backwards", i.e., we have an instance i and one of its leaves i', and we wish to find a path p such that $follow(p, i) = \{i'\}$.

Scope

A *scope* introduces a limit on where a binding introduced by `#bind before` or `#bind after`: no binding introduced inside a scope is available outside of it. It is given by the following grammar:

$$
\begin{array}{lll}
s & ::= & \textbf{scope}\ p\ P\ s \\
 & | & \textbf{near}
\end{array}
$$

where p is a path and P is a set of paths. The first alternative represents a scope introduced via a `#scope` declaration (e.g., line 25 in Listing 2). p describes how often the scope repeats (i.e. `#scope repeat:(...)`), while P contains paths to the leaves included in the scope (i.e. `#scope (leaf1 leaf2)`). The final component is the parent scope.

The second alternative is the special scope **near**, which is the ancestor scope of all other scopes. It represents the scope to which the syntax construction instance belongs, i.e., the nearest surrounding scope.

Note that, similar to paths, each scope belongs to a single syntax construction. They will not be discussed without a clear connection to a syntax construction or instance.

We also introduce the concept of $cover$: the cover of a scope is a set of paths to the leaves it contains:

$$
\begin{array}{rcl}
cover(\textbf{scope}\ \_\ P\_) & = & P \\
cover(\textbf{near}) & = & \{\text{The leaves not covered by another scope}\}
\end{array}
$$

We require that all leaves are covered by some scope, defaulting to **near** if no other scope applies. Additionally, no two scopes may cover the same leaf. To create nesting scopes we instead create parent-child connections between two scopes.

For example, `block` in Listing 2 has two scopes: **near** and $(\textbf{scope}\ []\ \{[2, *, 1]\}\ \textbf{near})$. Their covers are $\{[1], [2, *, 2], [3]\}$ and $\{[2, *, 1]\}$, respectively. The latter scope corresponds to the `#scope (es)` declaration on line 25, thus the cover is a single path to `es`. **near** then covers everything else, in this case the curly braces and the semicolon.

Expansion Template

An *expansion template* describes how the expansion of a particular syntax construction should be created. It is essentially an instance with holes, which are filled using elements from the instance being expanded or folds over elements from the instance being expanded. An expansion template is given by the following grammar:

$$
\begin{array}{lcl}
e & ::= & p \\
& | & \textbf{acc}\ n \\
& | & \textbf{foldl}\ p\ e\ e \\
& | & c\ \textbf{:>}\ i_e \\
\\
i_e & ::= & \langle \textit{token} \rangle \\
& | & e \\
& | & \textbf{seq}\ \overline{i_e} \\
& | & \textbf{rep}\ \overline{i_e}
\end{array}
$$

The **acc** n alternative refers to the accumulator of a surrounding fold, the innermost if $n = 0$, second inner-most if $n = 1$, etc. This is essentially a de bruijn index, except instead of a lambda binding an argument, we have a fold binding an accumulator. The actual implementation additionally has a right fold, as well as *fold1* variations of both, where the first element of the list is used as the initial value of the accumulator while folding over the tail of the list, but these are omitted here for brevity.

The `block` syntax construction in Listing 2 has the following expansion template (lines 26-28):

foldl $[2, *, 1]$
  `coreseq` **:>** (**seq** `"("` `"seq"` (**acc** 0) $[2, *, 1]$ `")"`)
  `corelit` **:>** (**seq** `"0"`)

`coreseq` and `corelit` are defined in the core language mentioned in the description of the running example, in Section 4.1. This definition can be found in Appendix A.

An additional point here is that of recursion: many macro systems allow a macro to expand to code containing the macro itself, but syntax constructions do not (neither directly nor indirectly). Disallowing recursion loses some expressive power, however, we

contend that EBNF operators and folds mostly make up for this loss in practice, and we gain some nice termination guarantees instead. This will be discussed further in Section 4.6.

Precedence and Associativity

The precedence of a syntax construction is a number or $+\infty$. The former represents a syntax construction that has a defined precedence (e.g., `sum`, via `#prec 2` on line 11 in Listing 2), while the latter represents a syntax construction without a defined precedence (e.g., `var`, defined on lines 4-5 in Listing 2).

Associativity is represented by one of three values:

a ::= **left** | **right** | **none**

left and **right** represent left and right associativity, while **none** represents undefined associativity.

Syntax Construction

Putting it all together, a syntax construction c is given by a tuple

$$c = (t_c, d_c, S_c, B_n, B_b, B_a, p_c, a_c, e_c)$$

with the following components:

t_c	The syntax type of c
d_c	The syntax description of c
S	The scopes introduced by c, a set of scopes.
B_n	The nested bindings of c, a set of tuples (p, P) where the first element is a path to an identifier and the second is a set of paths to the leaves where it is in scope.
B_b and B_a	The bindings c exports before and after itself, respectively. Sets of paths to identifiers.
p_c	The precedence of c, $+\infty$ unless otherwise specified.
a_c	The associativity of c.
e_c	The expansion template of c.

A 9-tuple is somewhat unwieldy, thus later sections will generally refer to "the B_n of c" to refer to the nested bindings of c, etc., instead

of listing the entire tuple and picking out the relevant component. The components that appear with c in their subscript are exceptions to this, e.g., t_c refers to the syntax type of c.

As an example, the syntax construction `block`, defined on lines 22-29 of Listing 2 has the following form:

$$(t_{\texttt{block}}, d_{\texttt{block}}, S_{\texttt{block}}, B_n, B_b, B_a, p_{\texttt{block}}, a_{\texttt{block}}, e_{\texttt{block}})$$

where

$$
\begin{aligned}
t_{\texttt{block}} &= \textbf{Expression} \\
d_{\texttt{block}} &= \textbf{seq}\ \texttt{"\{"}\ (\textbf{star}\ (\textbf{seq}\ \textbf{Expression}\ \texttt{";"}))\ \texttt{"\}"} \\
S_{\texttt{block}} &= \{\textbf{near}, \textbf{scope}\ []\ \{[2, *, 1]\}\ \textbf{near}\} \\
B_n &= \emptyset \\
B_b &= \emptyset \\
B_a &= \emptyset \\
p_{\texttt{block}} &= +\infty \\
a_{\texttt{block}} &= \textbf{none} \\
e_{\texttt{block}} &= \textbf{foldl}\ [2, *, 1] \\
&\qquad \texttt{coreseq}\ \textbf{:>}\ (\textbf{seq}\ \texttt{"("}\ \texttt{"seq"}\ (\textbf{acc}\ 0)\ [2, *, 1]\ \texttt{")"}) \\
&\qquad \texttt{corelit}\ \textbf{:>}\ (\textbf{seq}\ \texttt{"0"})
\end{aligned}
$$

From this we also define our concept of a language: a language defined using syntax constructions is a set of syntax constructions.

4.3 Source Code Parsing

The diagram in Figure 4.1 contains three components that are relevant to parsing code written in a language defined using syntax constructions: Grammar Generation, Parsing, and Ambiguity Reporting. This section will describe them, along with their connections to each other.

4.3.1 Grammar Generation

The algorithm for translating a set of syntax constructions to a context-free grammar proceeds as described below, using the

following notation:

N_p^t	A non-terminal generated by the constructions c such that $t = t_c \wedge p = p_c$.
$next(N_p^t)$	The non-terminal with the next higher precedence.
N_-^t	Shorthand for the non-terminal generated by the lowest precedence group in t.
N_+^t	Highest precedence group in t. Note that $N_+^t \neq N_-^t$ is not true in general, some syntax types only contain syntax constructions of a single precedence level.

1. Group syntax constructions by their syntax type, then by precedence.
2. For each such group, generate a non-terminal N_p^t:
   - For each syntax construction c, generate one production according to its syntax description:
     - **identifier**, **integer**, **float**, and **string** each match one token of the corresponding kind.
       * **identifier** should not match an identifier with a symbol that is used as a literal in any syntax construction.
     - A token literal matches a token with exactly the same contents.
     - References to a syntax type t' matches non-terminals according to the following, picking the first that applies:
       * If $t \neq t'$, match $N_-^{t'}$.
       * If c has the highest precedence in t, match N_-^t.
       * If c has undefined associativity, or is left-associative and t' is the leftmost occurrence, or is right-associative and t' is the rightmost occurrence, match $N_{p_c}^t$.
       * Otherwise match $next(N_{p_c}^t)$.

  - A **seq** matches each of its children in turn.
  - EBNF operators (**star**, **plus**, and **question**) match repetitions as expected.

3. For each non-terminal N_i^t except N_+^t add a production matching the non-terminal $next(N_i^t)$.

For our example language, grouping by syntax type and then precedence gives the following grouping:

- **`File`**

  $+\infty$ `top`

- **`Expression`**

  1 `def`

  2 `sum`

  $+\infty$ `var`, `lit`, and `block`

Generating production rules according to point 2 gives the following grammar:

⟨*FileInf*⟩ ::= ⟨*Expression1*⟩

⟨*Expression1*⟩ ::= 'def' ⟨*Identifier*⟩ '=' ⟨*Expression1*⟩

⟨*Expression2*⟩ ::= ⟨*Expression2*⟩ '+' ⟨*ExpressionInf*⟩

⟨*ExpressionInf*⟩ ::= ⟨*Identifier*⟩
| ⟨*Integer*⟩
| '{' (⟨*Expression1*⟩ ';')* '}'

where ⟨*Identifier*⟩ will not match an identifier that is exactly `"def"`.

Note how the three **`Expression`**s refer to each other: ⟨*Expression1*⟩ only references itself, ⟨*Expression2*⟩ references itself and ⟨*ExpressionInf*⟩, and ⟨*ExpressionInf*⟩ only references ⟨*Expression1*⟩. These references are determined by the precedence and associativity of the syntax construction that generated each production.

This grammar is not particularly useful however, since neither ⟨*Expression2*⟩ nor ⟨*ExpressionInf*⟩ are reachable from the start symbol ⟨*FileInf*⟩. Step 3 solves this by introducing an extra production to each of ⟨*Expression1*⟩ and ⟨*Expression2*⟩, which produces our final grammar:

⟨*FileInf*⟩ ::= ⟨*Expression1*⟩

⟨*Expression1*⟩ ::= 'def' ⟨*Identifier*⟩ '=' ⟨*Expression1*⟩
| ⟨*Expression2*⟩

⟨*Expression2*⟩ ::= ⟨*Expression2*⟩ '+' ⟨*ExpressionInf*⟩
| ⟨*ExpressionInf*⟩

⟨*ExpressionInf*⟩ ::= ⟨*Identifier*⟩
| ⟨*Integer*⟩
| '{' (⟨*Expression1*⟩ ';')* '}'

4.3.2 Parsing

We here assume the presence of a parsing algorithm that can handle an arbitrary context-free grammar, for example the Earley [3] parsing algorithm. In particular, if the source code cannot be parsed unambiguously then we must be able to construct all possible parse trees.

With this in mind, the inputs of this step are the following:

- A context-free grammar produced by the previous section.
- A token stream representing the source code to be parsed.

The output then has three possible forms:

- A single parse tree, from which we produce an instance. This is the successful case. We proceed to the next step, Name Resolution, in Section 4.5.
- No parse trees, we could not parse the source. This is the simple failure case, report an error to the user.
- Two or more parse trees, from which we produce one instance each. This is the ambiguous case, the more complex failure case, in which case we move on to the next section, Ambiguity Reporting.

4.3.3 Ambiguity Reporting

If the parse step produced multiple instances we have encountered source code that was ambiguous, i.e., it could be parsed in multiple ways. We could trivially report to the user that the code is ambiguous and that they must clarify their intent, but merely stating that an entire file is ambiguous provides very little assistance in locating the problem.

Instead we wish to produce an error that refers only to the actually ambiguous parts of the source file. Furthermore, we wish to present the different parse trees in an understandable way.

Starting with the second point, assume that no operators have defined precedence or associativity and consider the following expression:

$$a + b * c$$

It can be parsed in two ways:

$$(a + b) * c$$
$$a + (b * c)$$

The former is a product of a sum and a variable, the latter is a sum of a variable and a product. Further describing the inner structure of the sub-expressions (sum of two variables and product of two variables respectively), gives little to no extra useful information for understanding the two parsings. Consider this more complicated example:

$$a + b + c + d$$

The parsings are as follows:

$$((a + b) + c) + d$$
$$(a + (b + c)) + d$$
$$(a + b) + (c + d)$$

$$a + ((b + c) + d)$$
$$a + (b + (c + d))$$

All of these are sums, the first two of a sum and a variable, the next of two sums, the last two of a variable and a sum. This form of description, i.e., a single top-level node and a shallow description of its children, will be referred to as a two-level representation.

As an example, consider the following code:

```
1 + 2 + 3
```

Parsing this code using the running example language (Listing 2), but without associativity for `sum`, we obtain the following two instances:

```
top :> (seq
  sum :> (seq
    (sum :>
      seq (lit :> seq "1")
          "+"
          (lit :> seq "2"))
    "+"
    (lit :> seq "3")))
```

```
top :> (seq
  sum :> (seq
    "+"
    (lit :> seq "1")
    (sum :>
      seq (lit :> seq "2")
          "+"
          (lit :> seq "3"))))
```

Replacing these with two-level representations of the actual ambiguity gives us the following:

- A `sum` of a `sum` and a `lit`.
- A `sum` of a `lit` and a `sum`.

There is an additional subtlety in what we wish to consider as a two-level representation, stemming from the presence of internal structure in the form of **seq** and **rep**. In the example above we only consider full instances, not the internal structure, but we could just as well have chosen the following two-level representations:

- A **seq** of a `sum` and a `lit`.
- A **seq** of a `lit` and a `sum`.

This choice is discussed in more detail in the evaluation, in Section 5.6.1. The remainder of this section will assume the second alternative (i.e., use **seq** and **rep**) and leave the first alternative as a potential engineering decision.

The task of producing an error message then becomes selecting subtrees from the syntax trees such that:

- The unselected parts are identical across the different syntax trees, including covered source area (i.e., we report all ambiguities).
- The selected subtrees from the same position across the syntax trees cover the same source area (i.e., each report has a clear connection to the source).
- At least two selected subtrees in the same position of their respective syntax tree have different two-level representations (i.e., all reports actually show different parsings).
- No selected subtree is contained in another (i.e., we only report one ambiguity per source file area).
- No selected subtree can be replaced by one of its children (also replacing the corresponding subtrees from the other syntax trees) while still satisfying the points above (i.e., the selected subtrees are minimal).

To accomplish this, we use a form of shallow equality where two nodes are equal if they cover the same source area and have the same kind (e.g., they are instances of the same syntax construction). With this in mind we traverse all the syntax trees in parallel, top to bottom, until we encounter a node that is shallowly different between at least two syntax trees. If any sibling of the node also is shallowly different, select the parent, otherwise select the node and keep traversing down the siblings.

Finally, present to the user a set of two-level representations for each selected subtree across the syntax trees.

As an example, consider our running example, but once again with associativity removed for `sum`. Since Listing 3 contains `a + 2 + 3` we know that parsing it will produce multiple instances, in this case two. In the interest of brevity we will only show two levels of the

instances at once, the node currently being compared and its children, but not its grandchildren. We will denote shallow equality by $=_s$. Stepping through the ambiguity reporting we start by examining the top-level node of the respective trees:

$$\texttt{top} \textbf{ :> } (\textbf{seq} \ldots) \quad =_s \quad \texttt{top} \textbf{ :> } (\textbf{seq} \ldots)$$

$$\textbf{seq } (\texttt{block} \textbf{ :> } \ldots) \quad =_s \quad \textbf{seq } (\texttt{block} \textbf{ :> } \ldots)$$

$$\texttt{block} \textbf{ :> } (\textbf{seq} \ldots) \quad =_s \quad \texttt{block} \textbf{ :> } (\textbf{seq} \ldots)$$

$$\textbf{seq } \texttt{"\{"} \ (\textbf{rep} \ldots) \ \texttt{"\}"} \quad =_s \quad \textbf{seq } \texttt{"\{"} \ (\textbf{rep} \ldots) \ \texttt{"\}"}$$

$$\begin{array}{rcl} \texttt{"\{"} & =_s & \texttt{"\{"} \\ \textbf{rep } (\textbf{seq} \ldots) \ (\textbf{seq} \ldots) & =_s & \textbf{rep } (\textbf{seq} \ldots) \ (\textbf{seq} \ldots) \\ \texttt{"\}"} & =_s & \texttt{"\}"} \end{array}$$

$$\begin{array}{rcl} \textbf{seq } (\texttt{def} \textbf{ :> } \ldots) \ \texttt{";"} & =_s & \textbf{seq } (\texttt{def} \textbf{ :> } \ldots) \ \texttt{";"} \\ \textbf{seq } (\texttt{sum} \textbf{ :> } \ldots) \ \texttt{";"} & =_s & \textbf{seq } (\texttt{sum} \textbf{ :> } \ldots) \ \texttt{";"} \end{array}$$

Under normal circumstances we would now have to descend into both children (i.e., both (**seq** `def` ...) and (**seq** `sum` ...)), but in the interest of brevity we will instead note that the first child is fully equal across the two trees (there is no ambiguity there). Descending through it will thus never come across any instances that are not shallowly equal. We will thus only concern ourselves with descent into the second child:

$$\begin{array}{rcl} \texttt{sum} \textbf{ :> } (\textbf{seq} \ldots) & =_s & \texttt{sum} \textbf{ :> } (\textbf{seq} \ldots) \\ \texttt{";"} & =_s & \texttt{";"} \end{array}$$

$$\textbf{seq } (\texttt{sum} \textbf{ :> } \ldots) \ \texttt{"+"} \ (\texttt{lit} \textbf{ :> } \ldots) \quad =_s \quad \textbf{seq } (\texttt{var} \textbf{ :> } \ldots) \ \texttt{"+"} \ (\texttt{sum} \textbf{ :> } \ldots)$$

This is the first point where we see any inequality, but the two instances are still shallowly equal, since both have the form $(\mathbf{seq}\ \ldots)$ and both cover the same area of the source code.

$$\begin{array}{rcl} \texttt{sum}\ \textbf{:>}\ (\mathbf{seq}\ \ldots) & \neq_s & \texttt{var}\ \textbf{:>}\ (\mathbf{seq}\ \ldots) \\ \texttt{"+"} & \neq_s & \texttt{"+"} \\ \texttt{lit}\ \textbf{:>}\ (\mathbf{seq}\ \ldots) & \neq_s & \texttt{sum}\ \textbf{:>}\ (\mathbf{seq}\ \ldots) \end{array}$$

Note here than none of the children are shallowly equal. In particular, the `"+"` tokens cover different areas of the source code (they are the second and first "+" in the source code respectively). We thus select the parent and present that as the one ambiguity in the source code.

4.4 Further Properties of Paths

This section will introduce further path-related concepts that will be used in the later sections.

As a reminder, a path allows a syntax construction to refer to its children through a traversal. Each step of the traversal is either an index n that traverses down the child at index n, or a $*$ traversing through *all* children. We write the empty path $[]$, prepending n to a path p as $n : p$, and use $[a, b, c]$ as shorthand for $a : b : c : []$.

We now define a relation $p_1 \subseteq p_2$ between two paths, where

$$\begin{array}{ll} p \subseteq p & (i.e., \subseteq \text{ is reflexive}) \\ n : p \subseteq n : p' & \text{if } p \subseteq p' \\ n : p \subseteq * : p' & \text{if } p \subseteq p' \end{array}$$

We will now justify the notation. Given the following:

- Two paths p_1 and p_2 belonging to a syntax construction c.
- d_c, the syntax description of c.
- $p_1 \subseteq p_2$.

We can now state two things about the result of following p_1 or p_2:

- $follow(p_1, d_c) = follow(p_2, d_c)$.

- For every instance $(c$ **:>** $i)$ of c, we have:
  $follow(p_1, i) \subseteq follow(p_2, i)$.

We see this by noting that the structure of an instance mirrors that of the syntax description, and that a $*$ step is only allowed through an EBNF operator (i.e., **star**, **plus**, or **question** in the syntax description, and **rep** in an instance). We say that p_1 is a more specific version of p_2.

For example, $[2, 1, 1] \subseteq [2, *, 1]$. The former selects the first sub-expression in a `block` instance (Listing 2), while the latter selects *all* sub-expressions.

This relation is important for scopes in name resolution, where we define a leaf, reachable by path p, to be in a scope $(\textbf{scope}\ \_\ P\ \_)$ if $\exists p' \in P.\ p \subseteq p'$.

We also wish to be able to specialize a path p using some other path p', i.e., $specialize(p, p') \subseteq p$. Note that the fourth case below allow p and p' to diverge (i.e., we do not require $p' \subseteq p$), in which case the specialization is only applied to their shared prefix.

$$
\begin{aligned}
specialize(p, []) &= p \\
specialize(* : p, n : p') &= n : specialize(p, p') \\
specialize(* : p, * : p') &= * : specialize(p, p') \\
specialize(n : p, n' : p') &= \begin{cases} p & n \neq n' \\ n : specialize(p, p') & n = n' \end{cases}
\end{aligned}
$$

For example, $specialize([2, *, 2, *], [2, 1, 1, 1]) = [2, 1, 2, *]$.

This will mostly be used during expansion, where we wish to find the elements that are part of the current iteration of **foldl**.

4.5 Name Resolution

Name resolution serves two purposes:

- Discover binding errors.
- Ensure that each symbol appears in binding position only once.

Somewhat more formally, name resolution is a structure and binding preserving transformation that replaces each symbol in an abstract syntax tree with newly generated ones so as to ensure that each symbol appears in a binding position at most once across the entire tree. Additionally, name resolution detects the presence of binding errors. The root of the tree we are to process will be referred to as $root$, and will by construction have the shape c **:>** i (i.e., it is an instance of some syntax construction c).

We assume that each node is unique and distinct from all other nodes in the tree, and in particular that it can be used as the key in a mapping. This could, for example, be accomplished by attaching a unique number to each node across the tree. Furthermore, we attach an ordering between two nodes if neither is a descendant of the other. A node a is before another node b if it appears earlier in the tree, in which case we will write $a < b$. This is used to support `before` and `after` bindings, where the `after` bindings of a might be in scope in b only if $a < b$ ("might" stems from additional scoping rules, which are explained later in this section).

This section is also strict about the distinction between an identifier and a symbol: an identifier is a node in the syntax tree, it has a location in the source file and contains a symbol, which is essentially just a string. As an example,

```
{
  def a = 1;
  def a = 2;
}
```

has two identifiers with the same symbol: "a". The identifiers are different, since they cover different areas of the source file, and they are in different places in the AST. We will write $symbol(i)$ to denote the symbol of the identifier i. A symbol will be written as its text in quotations, i.e., "a" is a symbol, not to be confused with `"a"`, which is an identifier (strictly speaking, it is an identifier token, thus written the same as other tokens).

First, we wish to find all the identifiers that are in a binding position:

$$
\begin{array}{rcl}
\mathit{binders}(c \texttt{ :> } i) & = & i_b \cup i_a \cup i_n \cup i_c \\
\text{where} & & \\
i_b & = & \{\mathit{follow}(p, i) \mid p \in B_b \text{ of } c\} \\
i_a & = & \{\mathit{follow}(p, i) \mid p \in B_a \text{ of } c\} \\
i_n & = & \{\mathit{follow}(p, i) \mid (p, \_) \in B_n \text{ of } c\} \\
i_c & = & \bigcup_{(c' \texttt{ :> } i') \in \mathit{leaves}(c \texttt{ :> } i)} \mathit{binders}(c' \texttt{ :> } i')
\end{array}
$$

For example, using the instance for the innermost `block` (writing it as `block` **:>** i) on lines 3-6 in Listing 3:

$$
\begin{array}{rcl}
\mathit{binders}(\texttt{block :> } i) & = & i_b \cup i_a \cup i_i \cup i_c = \{\texttt{"a"}\} \\
\text{where} & & \\
i_b & = & \{\mathit{follow}(p, i) \mid p \in \emptyset\} = \emptyset \\
i_a & = & \{\mathit{follow}(p, i) \mid p \in \emptyset\} = \emptyset \\
i_i & = & \{\mathit{follow}(p, i) \mid (p, \_) \in \emptyset\} = \emptyset \\
i_c & = & \mathit{binders}(\texttt{def :> } \ldots) = \{\texttt{"a"}\}
\end{array}
$$

Second, we need a mapping Γ from $\mathit{binders}(\mathit{root})$ to new unique symbols. We require the mapping to be a bijection, but otherwise the particular details are unimportant.

Third, we wish to determine the scope of each node in the tree. Similar to syntax constructions and their instances we introduce scope instances, described by the following grammar:

$$
\begin{array}{rcl}
s' & ::= & \textbf{scope-inst } P \; s' \\
& | & \textbf{root-scope}
\end{array}
$$

The first case represents most instances and consists of two components: a set of paths to the leaves it covers, and its parent scope instance. The second case is the root scope.

For example, the outermost `block` instance in Listing 3 (lines 1-7) introduces one scope instance:

$$\textbf{scope-inst } \{[2, *, 1]\} \;\; \textbf{root-scope}$$

We will further write $s'_c \to s'_p$ to mean that s'_p is the parent scope of s'_c, and $s' \to^* s''$ as the reflexive, transitive closure of this relation. Since each node will be assigned to a single scope we will write $i \in s'$ to

mean that i belongs to scope instance s', which then selects a unique scope instance per i.

Additionally, we will define the scope instance of $root$ to be **root-scope**, i.e.:

$$root \in \textbf{root-scope}$$

Given a scope s and a path p to a leaf, both belonging to an instance c **:>** i, the scope instance s' of s is given by $instantiate(c \textbf{ :> } i, p, s)$:

$$\begin{aligned}
instantiate(c \textbf{ :> } i, p, \textbf{near}) &= s' \text{ where } c \textbf{ :> } i \in s' \\
instantiate(c \textbf{ :> } i, p, (rep, P, s)) &= (P', parent) \\
instantiate(c \textbf{ :> } i, p, \textbf{scope } r\ P\ s) &= \textbf{scope-inst } P'\ s' \\
&\text{where} \\
r' &= specialize(r, p) \\
P' &= \{specialize(p', r') \mid p' \in P\} \\
s' &= instantiate(c \textbf{ :> } i, p, s)
\end{aligned}$$

We are now ready to determine the scope instance any non-root leaf belongs to. Given a leaf i, that is a leaf of c **:>** i', find the scope s in c that covers i, then instantiate it:

$$\frac{\begin{array}{cc} i \in leaves(c \textbf{ :> } i') & follow(p, i') = \{i\} \quad p \subseteq p' \\ p' \in cover(s) & s \in scopes \text{ of } c \end{array}}{i \in instantiate(c \textbf{ :> } i', p, s)}$$

We can now start considering actual bindings. For each node we will compute a set of available bound symbols. Each binding is represented by a 3-tuple consisting of a symbol, a scope instance, and a generated symbol. The first two components denote that the symbol definition is in the given scope instance, while the last component is the generated replacement symbol from Γ above.

As an example, $\{(\text{"foo"}, s'_1, \text{"bar"}), (\text{"foo"}, s'_2, \text{"baz"})\}$ has two bindings of the symbol "foo", one bound in s'_1, the other in s'_2, replaced by "bar" and "baz" respectively.

The *lookup* function below defines shadowing behavior: a binding of a symbol in a child scope shadows the same symbol in a parent scope.

The first case also assumes that we will never produce a binding set with more than one element that share the first two components, which would represent a redefinition of a symbol in the same scope.

$$lookup(sym, s', \gamma) = \begin{cases} sym' & (sym, s', sym') \in \gamma \\ lookup(sym, s'', \gamma) & (sym, s', \_) \notin \gamma \wedge s' \rightarrow s'' \\ \bot & (sym, s', \_) \notin \gamma \wedge s' = \textbf{root-scope} \end{cases}$$

At this point we have our definitions for binding errors: an unbound symbol produces $\bot$ during lookup, while a redefinition produces a binding set where two or more elements share the first two components.

The latter error has some subtleties. We could defer the error until *lookup* would find multiple choices, i.e., make it an error to have an ambiguous reference instead of making it an error to redefine a symbol, but the latter tends to be a programming error even if it does not lead to any ambiguity.

The exported `before` and `after` bindings of an instance $(c \textbf{ :> } i) \in s'$, written as $E_b(c \textbf{ :> } i)$ and $E_a(c \textbf{ :> } i)$ respectively, are given by:

$$\begin{aligned} E_b(c \textbf{ :> } i) &= immediate \cup transitive \\ \text{where} & \\ immediate &= \bigcup_{p \in B_b} \{(symbol(follow(p, i)), s', \Gamma(follow(p, i)))\} \\ transitive &= \{i'' \mid i'' \in E_b(c' \textbf{ :> } i') \\ &\quad , (c' \textbf{ :> } i') \in leaves(c \textbf{ :> } i) \\ &\quad , (c' \textbf{ :> } i') \in s'\} \end{aligned}$$

$$\begin{aligned} E_a(c \textbf{ :> } i) &= immediate \cup transitive \\ \text{where} & \\ immediate &= \bigcup_{p \in B_a} \{(symbol(follow(p, i)), s', \Gamma(follow(p, i)))\} \\ transitive &= \{i'' \mid i'' \in E_a(c' \textbf{ :> } i') \\ &\quad , (c' \textbf{ :> } i') \in leaves(c \textbf{ :> } i) \\ &\quad , (c' \textbf{ :> } i') \in s'\} \end{aligned}$$

immediate are the bindings that are exposed by the instance itself (i.e., via `#bind`), while *transitive* are the bindings exported by its leaves.

Bindings can also be defined in a nested manner, via B_n. The bindings visible in a node $i \in leaves(c \textbf{ :> } i')$ where $follow(p, i') = \{i\}$, is given by $E_n(i)$:

$$\begin{aligned} E_n(root) &= \emptyset \\ E_n(i) &= E_n(c \textbf{ :> } i'\}) \cup \\ &\quad \{symbol(follow(p, i')) \mid (p, P) \in B_n \wedge \exists p' \in P.\ p \subseteq p'\} \end{aligned}$$

Note the difference between $E_a(i)$, $E_b(i)$, and $E_n(i)$: the former two give the set of bindings *exported* by i, while the latter gives the set of bindings *usable* in i.

With all this setup, name resolution is a matter of replacing the symbol in each identifier that was parsed from **identifier** (as opposed to a token literal) with a newly generated symbol. Given an identifier $id \in leaves(c \textbf{ :> } i)$ with scope $id \in s'$, $follow(p, i) = \{id\}$, and $syntax$ being the syntax description of c, we will only perform the below substitution if $follow(p, syntax) = \textbf{identifier}$.

$$\begin{aligned} symbol(id) &\leftarrow \begin{cases} \Gamma(id) & id \in binders(root) \\ lookup(symbol(id), s', \gamma) & id \notin binders(root) \end{cases} \\ \text{where} & \\ \gamma &= b \cup a \cup E_n(id) \\ b &= \{x \mid x \in E_b(c' \textbf{ :> } i'), (c' \textbf{ :> } i') \in s'', s' \rightarrow^* s'', (c' \textbf{ :> } i') > i\} \\ a &= \{x \mid x \in E_a(c' \textbf{ :> } i'), (c' \textbf{ :> } i') \in s'', s' \rightarrow^* s'', (c' \textbf{ :> } i') < i\} \end{aligned}$$

4.6 Expansion

Expanding the instance of a syntax construction is relatively straightforward, but the formalization turns out to be surprisingly involved. The base idea is that an expansion is a template into which we splice subtrees from the instance we are expanding, similar to pattern based macros in Lisp tradition. As an additional precondition we assume that the symbols present in the initial instance only occur in binding position at most once.

The process is made somewhat more complicated by the presence of EBNF operators that generate lists, which must be reduced to a single subtree through folds. To perform such a fold we require specific paths to each of the elements to be folded over, to be handed to *specialize* later:

$$\begin{aligned} specs([], i) &= \{[]\} \\ specs(n : p, \textbf{seq } \overline{i}) &= \{n : p' \mid p' \in specs(p, i_n)\} \\ specs(n : p, \textbf{rep } \overline{i}) &= \{n : p' \mid p' \in specs(p, i_n)\} \\ specs(* : p, \textbf{rep } \overline{i}) &= \bigcup_n \{n : p' \mid p' \in specs(p, i_n)\} \end{aligned}$$

Note that while this function uses set-notation, the order of the paths will be important while folding. The order is easily recovered however using a simple lexicographic sorting of the paths. All later uses of *specs* will thus be considered to return an ordered sequence of paths, rather than a set.

As an example, consider the outermost `block` instance (covering all lines) obtained from the code in Listing 3. It has the following form:

```
block :>
  seq "{"
      (rep (seq (def :> ...) ";")
           (seq (sum :> ...) ";"))
      "}"
```

As such $specs([2, *, 1], \texttt{block :> } \ldots) = [[2, 1, 1][2, 2, 1]]$.

Expansion (via *expand* below) then traverses the template (the fourth argument, e_c), while keeping track of the current item in a fold (p), the related accumulators ($\overline{acc}$), and the instance being expanded (i). Most cases are fairly straightforward recursion:

$$\begin{aligned} expand(i, p, \overline{acc}, p') &= i' \text{ where } follow(specialize(p', p), i) = \{i'\} \\ expand(i, p, \overline{acc}, \textbf{acc } n) &= acc_n \\ expand(i, p, \overline{acc}, c \textbf{ :> } i') &= c \textbf{ :> } expand(i, p, \overline{acc}, i') \\ expand(i, p, \overline{acc}, \langle token \rangle) &= \langle token \rangle \\ expand(i, p, \overline{acc}, \textbf{seq } \overline{i'}) &= \textbf{seq } \overline{expand(i, p, \overline{acc}, i'_n)} \\ expand(i, p, \overline{acc}, \textbf{rep } \overline{i'}) &= \textbf{rep } \overline{expand(i, p, \overline{acc}, i'_n)} \end{aligned}$$

The one remaining case, **foldl**, is somewhat more involved:

$$expand(i, p, \overline{acc}, \textbf{foldl}\ \ p'\ f\ i') \quad = \quad foldl(f', f_0, P)$$

where

$$\begin{array}{lcl} f' & = & \lambda p'\ acc'.\ expand(i, p', acc'\ \overline{acc}, f) \\ f_0 & = & expand(i, p, \overline{acc}, i') \\ P & = & specs(specialize(p', p), inst) \end{array}$$

and *foldl* is a regular left fold over a list.

As an example, `block` has the following expansion template:

foldl $[2, *, 1]$
  `coreseq` **:>** (**seq** `"("` `"seq"` (**acc** 0) $[2, *, 1]$ `")"`)
  `corelit` **:>** (**seq** `"0"`)

Expanding the instance given above proceeds as follows, using i to refer to that instance:

$$\begin{array}{l} expand(i, [], \epsilon, \textbf{foldl}\ \ [2, *, 1]\ (\texttt{coreseq}\ \textbf{:>}\ \ldots)\ (\texttt{corelit}\ \textbf{:>}\ \ldots)) \\ \quad = foldl(f', f_0, P) \end{array}$$

where

$$\begin{array}{lcl} f' & = & \lambda p'\ acc'.\ expand(i, p', [acc'], \texttt{corelit}\ \textbf{:>}\ \ldots) \\ f_0 & = & expand(i, [], \epsilon, \texttt{corelit}\ \textbf{:>}\ (\textbf{seq}\ \texttt{"0"}) \\ & = & \texttt{corelit}\ \textbf{:>}\ expand(i, [], \epsilon, \textbf{seq}\ \texttt{"0"}) \\ & = & \texttt{corelit}\ \textbf{:>}\ (\textbf{seq}\ expand(i, [], \epsilon, \texttt{"0"})) \\ & = & \texttt{corelit}\ \textbf{:>}\ (\textbf{seq}\ \texttt{"0"}) \\ P & = & specs(specialize([2, *, 1], []), i) \\ & = & specs([2, *, 1], i) \\ & = & [[2, 1, 1], [2, 2, 1]] \end{array}$$

foldl will then call f' twice, once with $p' = [2, 1, 1]$ and $acc' = f_0$, and once with $p' = [2, 2, 1]$ and $acc' = f_1$, where f_1 is the result of the previous call. It is here useful to note that a subtree that does not contain either **acc** n or a path always expands to itself, hence we will omit those expansions below.

The first f' call proceeds as follows:

$$
\begin{aligned}
f'\ [2,1,1]\ f_0 &= \mathit{expand}(i, [2,1,1], f_0, \texttt{coreseq}\ \textbf{:>}\ (\textbf{seq}\ \ldots)) \\
&= \texttt{coreseq}\ \textbf{:>} \\
&\quad \mathit{expand}(i, [2,1,1], f_0, \textbf{seq} \\
&\quad\quad \texttt{"("} \\
&\quad\quad \texttt{"seq"} \\
&\quad\quad \textbf{acc}\ 0 \\
&\quad\quad [2,*,1] \\
&\quad\quad \texttt{")"}) \\
&= \texttt{coreseq}\ \textbf{:>} \\
&\quad \textbf{seq} \\
&\quad\quad \texttt{"("} \\
&\quad\quad \texttt{"seq"} \\
&\quad\quad \mathit{expand}(i, [2,1,1], f_0, \textbf{acc}\ 0) \\
&\quad\quad \mathit{expand}(i, [2,1,1], f_0, [2,*,1]) \\
&\quad\quad \texttt{")"} \\
&= \texttt{coreseq}\ \textbf{:>} \\
&\quad \textbf{seq} \\
&\quad\quad \texttt{"("} \\
&\quad\quad \texttt{"seq"} \\
&\quad\quad f_0 \\
&\quad\quad (\texttt{def}\ \textbf{:>}\ \ldots)\ \text{since} \\
&\quad\quad\quad \mathit{follow}(\mathit{specialize}([2,*,1],[2,1,1]), i) \\
&\quad\quad\quad\quad = \mathit{follow}([2,1,1], i) \\
&\quad\quad\quad\quad = (\texttt{def}\ \textbf{:>}\ \ldots) \\
&\quad\quad \texttt{")"} \\
&= f_1
\end{aligned}
$$

The second call, with the identical steps omitted, then follows:

$$
\begin{aligned}
f'\ [2,2,1]\ f_1 \;=\; & \texttt{coreseq} \textbf{ :>} \\
& \quad \textbf{seq} \\
& \quad\quad \texttt{"(" } \\
& \quad\quad \texttt{"seq"} \\
& \quad\quad \mathit{expand}(i, [2,1,1], f_1, \textbf{acc}\ \ 0) \\
& \quad\quad \mathit{expand}(i, [2,1,1], f_1, [2,*,1]) \\
& \quad\quad \texttt{")"} \\
= \; & \texttt{coreseq} \textbf{ :>} \\
& \quad \textbf{seq} \\
& \quad\quad \texttt{"(" } \\
& \quad\quad \texttt{"seq"} \\
& \quad\quad f_1 \\
& \quad\quad (\texttt{sum} \textbf{ :>} \ldots) \text{ since} \\
& \quad\quad\quad \mathit{follow}(\mathit{specialize}([2,*,1],[2,2,1]),i) \\
& \quad\quad\quad\quad = \mathit{follow}([2,2,1],i) \\
& \quad\quad\quad\quad = (\texttt{sum} \textbf{ :>} \ldots) \\
& \quad\quad \texttt{")"}
\end{aligned}
$$

At this point we can consider termination guarantees: the expansion of a single syntax construction instance must terminate. We see this easily from the definition of expand and the observation that the expansion template is finite. Furthermore, since syntax constructions may not recursively expand into instances of themselves (neither directly nor indirectly), the complete expansion of a syntax tree must also terminate. We can see this by noting that when we expand an instance of a syntax construction c then none of the instances introduced by the expansion (i.e., not the ones copied over from the previous instance) are instances of c, nor can they expand into instances of c. To expand those instances we thus never need c, i.e., the number of syntax constructions required for expansion is reduced by one. Since the number of syntax construction is finite we will eventually reach a point where we cannot expand any further since no remaining syntax construction has an expansion, i.e., the syntax tree is fully expanded.

4.7 Expansion Checking

Expansion checking takes as input a single syntax construction c and checks that expanding an instance of c cannot introduce a binding error. As mentioned in previous section, we will assume that any instance we are to expand has no symbol that appears in binding position twice. The core idea behind the approach used in this section is to generate a set of instances of c, then expand each of them and check that no new binding errors are introduced. There are two points that complicate this approach:

1. The expansion must introduce no errors regardless of the initial instance's context and children. In particular, a checked expansion should be correct even when used together with syntax constructions that are not yet defined, i.e., we cannot assume a closed universe of syntax constructions.
2. The EBNF operators **star** and **plus** have no bound on the number of repetitions. Thus even if we handle the open universe of syntax constructions, we still need to handle an infinite number of instances with different internal structure.

We will address the first point initially, and return to the second later in the section.

As a running example we will check the `block` syntax construction on lines 22-29 in Listing 2. The components that are relevant for this section are the syntax type, the syntax description, the scopes, the bindings, and the expansion template:

$$
\begin{aligned}
t_{\texttt{block}} &= \textbf{Expression} \\
d_{\texttt{block}} &= \textbf{seq}\ \texttt{"\{"}\ (\textbf{star}\ (\textbf{seq}\ \textbf{Expression}\ \texttt{";"}))\ \texttt{"\}"} \\
S_{\texttt{block}} &= \{\textbf{near}, \textbf{scope}\ []\ \{[2, *, 1]\}\ \textbf{near}\} \\
B_n &= \emptyset \\
B_b &= \emptyset \\
B_a &= \emptyset \\
e_{\texttt{block}} &= \textbf{foldl}\ [2, *, 1] \\
&\qquad \texttt{coreseq}\ \textbf{:>}\ (\textbf{seq}\ \texttt{"("}\ \texttt{"seq"}\ (\textbf{acc}\ 0)\ [2, *, 1]\ \texttt{")"}) \\
&\qquad \texttt{corelit}\ \textbf{:>}\ (\textbf{seq}\ \texttt{"0"})
\end{aligned}
$$

Out of these, only the syntax type and syntax description are directly relevant for expansion checking. Since the approach centers around generating instances, expanding them, and checking the expansion for errors, we will need to perform expansion (which requires the expansion template) and name resolution (which requires scopes and bindings).

As an initial observation, the leaves on an instance place restrictions on what an expansion can do without introducing errors. For example, consider this code:

```
{
  def a = 5;
  a;
}
```

It parses as the following instance of `block`:

```
block :>
  seq "{"
      (rep (seq (def :> ...) ";")
           (seq (var :> ...) ";"))
      "}"
```

The first **`Expression`** leaf (i.e., `def` **:>** ...) introduces a binding for the symbol "a" `after` itself, which the second **`Expression`** leaf (i.e., `var` **:>** ...) references. An expansion of this instance is thus not allowed to reorder the two **`Expression`**s, nor remove the first, since that would leave an unbound reference. Note that the second can be removed without introducing an error, since it only removes the referencing identifier.

To ensure that the expansion respects its children we introduce *placeholder leaves*, leaves that are maximally demanding. The idea is that replacing a placeholder leaf with a real leaf should never increase the restrictions placed on the expansion. A placeholder that represents an instance references every symbol available to it, and binds one symbol `before` itself, and one `after` itself. A placeholder identifier in referencing position imports every symbol available to it. An identifier in a binding position does not need to be replaced by a

placeholder, as the only thing it can do is contain a single symbol, which is distinct from all other symbols in binding position.

We will write placeholder instances as f, and placeholder identifiers as j.

Replacing the real leaves in the above instance with placeholder leaves produces an instance with the same internal structure, but more demanding children:

```
block :>
  seq "{"
      (rep (seq f1 ";")
           (seq f2 ";"))
      "}"
```

Each placeholder instance is assigned the syntax type the syntax description requires, i.e., in the example, both f_1 and f_2 have syntax type **`Expression`**.

We now expand name resolution to handle placeholders, but discard the renaming aspect, as we are only interested in binding errors.

By definition, for a placeholder instance $f \in s'$ we have:

$$\begin{aligned} E_b(f) &= \{(b_f, s', b_f)\} \\ E_a(f) &= \{(a_f, s', a_f)\} \end{aligned}$$

where b_f is the symbol exported `before`, and a_f is the symbol exported `after`. These symbols should be distinct from each other, and from every other symbol in the initial instance.

To compute the symbols a particular placeholder instance or identifier conceptually references, we need the set of available bindings in a particular node. The set of bindings γ_i available in node i is given by:

$$\begin{aligned} \gamma_i &= \{sym \mid (sym, \_, \_) \in b \cup a \cup E_n(i)\} \\ \text{where} & \\ b &= \{x \mid x \in E_b(c' \textbf{ :> } i'), (c' \textbf{ :> } i') \in s'', s' \rightarrow^* s'', (c' \textbf{ :> } i') > i\} \\ a &= \{x \mid x \in E_a(c' \textbf{ :> } i'), (c' \textbf{ :> } i') \in s'', s' \rightarrow^* s'', (c' \textbf{ :> } i') < i\} \end{aligned}$$

Compare with the definition of γ at the end of Section 4.5.

Additionally, to ensure that no previously exported symbols are hidden by the expansion, we require the symbols exported `before` and `after` an instance c **:>** i, given by:

$$\begin{aligned} E'_b(c \text{ :> } i) &= \{sym \mid (sym, \_, \_) \in E_b(c \text{ :> } i)\} \\ E'_a(c \text{ :> } i) &= \{sym \mid (sym, \_, \_) \in E_a(c \text{ :> } i)\} \end{aligned}$$

For the example instance with placeholders, we find that there are no binding errors, and the following:

$$\begin{aligned} E'_b(\texttt{block} \text{ :> } \ldots) &= \emptyset \\ E'_a(\texttt{block} \text{ :> } \ldots) &= \emptyset \\ \gamma_{f_1} &= \{b_{f_2}\} \\ \gamma_{f_2} &= \{a_{f_1}\} \end{aligned}$$

This means that the instance exports no symbols, neither `before` nor `after`, and f_1 and f_2 can reference b_{f_2} and a_{f_1}, respectively. Expanding the instance produces the following instance:

```
coreseq :> (seq
  "("
  "seq"
  coreseq :> (seq
    "("
    "seq"
    corelit :> (seq "0")
    f1
    ")")
  f2
  ")")
```

In this instance we find that there are no binding errors, and the following:

$$\begin{aligned} E'_b(\texttt{coreseq} \text{ :> } \ldots) &= \{b_{f_1}, b_{f_2}\} \\ E'_a(\texttt{coreseq} \text{ :> } \ldots) &= \{a_{f_1}, a_{f_2}\} \\ \gamma_{f_1} &= \{b_{f_2}\} \\ \gamma_{f_2} &= \{a_{f_1}\} \end{aligned}$$

This means three things:

- The expansion exports *more* symbols than the original instance. This cannot introduce an error, since the original instance (and its context) does not have the same symbol in binding position more than once. The newly exported symbols can thus not be previously bound anywhere in the surrounding context, as they were either bound in the initial instance, but not exported, or introduced by the expansion, which introduces new, unique symbols.
- The placeholders have access to exactly the same set of symbols after the expansion as before it, i.e., no reference has been broken.
- No new binding errors were introduced (since there were no binding errors at all).

The second point requires more work in general than in this particular case: the expansion may duplicate a placeholder. To ensure that no errors can be introduced we must examine each placeholder in the expansion. For each placeholder f' (or j') in the expansion, copied from a placeholder f (or j) in the initial instance, we require that $\gamma_f \subseteq \gamma_{f'}$ (or $\gamma_j \subseteq \gamma_{j'}$), i.e., the placeholder in the expansion has access to all the symbols the initial placeholder had access to.

Finally, we check that all instances in the expansion have the correct syntax type. We require that each instance has a syntax type that is representationally equal (i.e., $\approx$, defined in Section 4.2) to the expected syntax type.

For the expansion, the expected syntax type is the syntax type of the initial instance. In the example, the initial instance has syntax type **`Expression`**, while the expansion has syntax type **`Core`**. Since **`Expression`** $\approx$ **`Core`** (by the declaration on line 1 in Listing 2) we find no error here.

For all other instances of form c' **:>** i' or f' in the expansion, the expected syntax type is given by the syntax description of its parent. For example, `coreseq` (defined in Appendix 5.1.1) has the following syntax description:

seq `"("` `"seq"` **Core Core** `")"`

The two occurrences of **Core** correspond to (`coreseq` **:>** ...) and f_2 in the example expansion. The former has syntax type **Core**, while the latter has syntax type **Expression**. Both are representationally equal to **Core**, thus neither is erroneous.

To summarize, an instance i (that contains placeholders) that expands to an instance i' is expanded without introducing errors if:

- The expansion introduces no new binding errors.
- $E'_b(i') \subseteq E'_b(i)$ and $E'_a(i') \subseteq E'_a(i)$ (i.e., the expansion exports at least the symbols exported by the initial instance).
- For every placeholder f' (or j') in i', copied from a placeholder f (or j) in i, we require that $\gamma_f \subseteq \gamma_{f'}$ (or $\gamma_j \subseteq \gamma_{j'}$) (i.e., the copy has access to at least the same symbols).
- The syntax type of every instance is representationally equal with the expected syntax type, which is given either by the initial instance or the syntax description of the parent instance.

We now move on to the matter of generating an instance, and in so doing return to the second point on what complicates this approach: a syntax construction has a potentially infinite number of instances with different internal structure.

When we generate instances, we wish to generate unique identifiers when they are in binding position, and unique *placeholder* identifiers when they are in referencing position. The values given to the leaves that do not participate in name resolution do not matter, but they must still be there to preserve the validity of paths, thus we give them some arbitrary dummy value. A syntax type t generates a placeholder instance of syntax type t. **seq** is a simple recursive case, and the EBNF operators (**star**, **plus**, and **question**) are similar recursive cases, but with different numbers of repetitions. With this in mind, and somewhat informally, the set of all instances that can be generated from a syntax description, given that all child instances and referencing identifiers are placeholders, is given by gen:

$$
\begin{array}{rcl}
gen(\textbf{identifier}) & = & \begin{cases} \{\textit{new-identifier}\} & \text{binding position} \\ \{\textit{new-placeholder-identifier}\} & \text{referencing position} \end{cases} \\
gen(\textbf{string}) & = & \{\texttt{"dummy"}\} \\
gen(\textbf{integer}) & = & \{\texttt{0}\} \\
gen(\textbf{float}) & = & \{\texttt{0.0}\} \\
gen(\langle \textit{token} \rangle) & = & \{\langle \textit{token} \rangle\} \\
gen(t) & = & \{\textit{new-placeholder-instance}\} \text{ (with syntax type } t\text{)} \\
gen(\textbf{seq } \overline{d}) & = & \{\textbf{seq } d'_1\ d'_2 \ldots \mid d'_1 \in gen(d_1), d'_2 \in gen(d_2), \ldots\} \\
gen(\textbf{star } d) & = & \bigcup_{n=0}^{\infty} \{\textbf{rep } d'_1\ d'_2\ \ldots\ d'_n \mid d'_1, d'_2, \ldots, d'_n \in gen(d)\} \\
gen(\textbf{plus } d) & = & \bigcup_{n=1}^{\infty} \{\textbf{rep } d'_1\ d'_2\ \ldots\ d'_n \mid d'_1, d'_2, \ldots, d'_n \in gen(d)\} \\
gen(\textbf{question } d) & = & \bigcup_{n=0}^{1} \{\textbf{rep } d'_1\ d'_2\ \ldots\ d'_n \mid d'_1, d'_2, \ldots, d'_n \in gen(d)\}
\end{array}
$$

where *new-** give new identifiers and placeholders, unique across the entire generated instance. For example, $gen(d_{\texttt{block}})$ gives the following instances:

```
block :>
  seq "{"(rep)"}"

block :>
  seq "{"
      (rep (seq f1 ";"))
      "}"

block :>
  seq "{"
      (rep (seq f1 ";")
           (seq f2 ";"))
      "}"

block :>
  seq "{"
      (rep (seq f1 ";")
           (seq f2 ";")
           (seq f3 ";"))
      "}"
```

... etc. ...

If we could expand and check each of these instances we could be sure that the expansion can never introduce a binding error, given that the placeholder leaves are indeed more demanding than any real leaf. However, the **star** and **plus** cases introduce an infinite number of instances, which are clearly too many to check. The hypothesis of this thesis is that those cases can be given an upper bound of 2 and 3 repetitions respectively, without loss of safety.

The justification centers around the capabilities of the **`foldl`** construct in the expansion template, by which the repetitions are reduced to a single element. As a reminder, it has the following structure:

$$\textbf{foldl}\ p\ e_1\ e_2$$

p is the path to the elements we will fold over, e_1 is the expansion template that will be used for each element, and e_2 is the initial value of the accumulator. Somewhat informally, e_1 cannot inspect the current value of the accumulator or the current element, thus the only thing it can do is "wrap" them in some surrounding instance. With this in mind we examine three cases of evaluating a **foldl** expansion template:

1. We fold over zero elements. We produce exactly e_2.
2. We fold over one element. We produce e_2 and that element, "wrapped" in a single e_1.
3. We fold over two or more elements. We produce e_2 and those elements, "wrapped" in two or more e_1.

The hypothesis of this thesis is that errors can arise in three corresponding ways:

1. There is an error in e_2. Folding over zero elements will expose it.
2. There is an error in e_1, or the interaction between e_2 and e_1. Folding over a single element will expose it.

3. There is an error in the interaction between two "wrappings" of e_1. Folding over two elements will expose it.

As an additional note, it may appear as though folding over zero elements is unnecessary. However, e_1 may ignore the accumulator, thus to be fully general we must test this case as well.

We thus have a hypothesis by which we only need to generate 0, 1, or 2 repetitions of d in $(\mathbf{star}\ d)$. For $(\mathbf{plus}\ d)$ the implementation additionally has a *fold1* version, where the first element is used as the initial value of the accumulator. Here we require 1, 2, and 3 repetitions to see 0, 1, and 2 "wrappings", respectively.

With this, we alter the **star** and **plus** cases of gen:

$$\begin{aligned} gen(\mathbf{star}\ d) &= \bigcup_{n=0}^{2} \{\mathbf{rep}\ d'_1\ d'_2\ \ldots\ d'_n \mid d'_1, d'_2, \ldots, d'_n \in gen(d)\} \\ gen(\mathbf{plus}\ d) &= \bigcup_{n=1}^{3} \{\mathbf{rep}\ d'_1\ d'_2\ \ldots\ d'_n \mid d'_1, d'_2, \ldots, d'_n \in gen(d)\} \end{aligned}$$

Checking a single syntax construction c with syntax description d_c is then a matter of checking that every instance $i \in gen(d_c)$ fulfills all properties listed on page 77.

Chapter 5

Evaluation

The evaluation of this thesis is largely empirical and based on case studies facilitated by a proof of concept implementation of the descriptions in the previous section, along with a small interpreter for a core language. An overview of this implementation is given in Section 5.1.

The actual evaluation consists of case studies of expressibility (Section 5.2), reuse of language feature from one language in another (Section 5.3), a brief examination of the correctness of the implementations in the case studies (Section 5.4), a few performance metrics of the implementation (Section 5.5), and finally an examination of the errors presented to a user when something goes wrong (Section 5.6).

For evaluation, two programming languages have been implemented: a subset of Lua[1] and a subset of OCaml[2]. The former evaluates the expressibility of a relatively standard untyped, imperative language, while the latter evaluates the expressibility of a relatively standard functional language, with an extra focus on pattern matching. Note that the implementation of the OCaml subset does not have static type checking, following the delimitations in Section 1.4.1.

For each of the languages, a few interesting features are discussed, including their implementation, along with several limitations of the

[1] https://www.lua.org/
[2] https://ocaml.org/

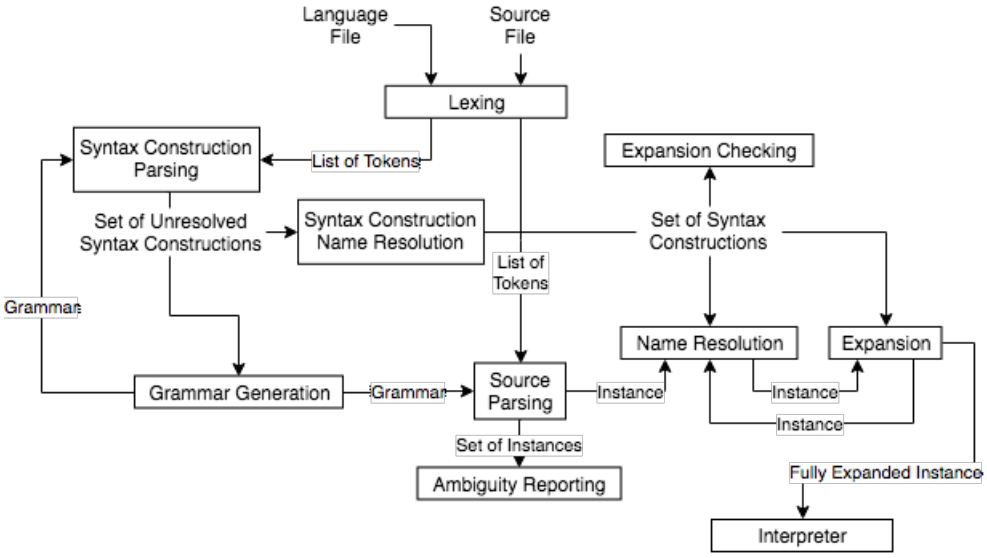

Figure 5.1: The components that make up the implementation, and their connections

system.

5.1 Implementation

This section contains a high level overview of the proof of concept implementation of the ideas in this thesis.

The implementation is written in Haskell and is available online[3]. It is structured similarly to compiler with multiple phases. The phases, represented as boxes in Figure 5.1, are as follows:

Lexing

The lexer is intended to be the simplest possible lexer that still produces results suitable for the grammars later generated by the syntax constructions. It lexes everything as identifiers, symbols, or string, integer or float literals. Most adjacent punctuation and special characters are merged into single symbols, with a few exceptions to

[3]https://github.com/elegios/master-thesis

create behavior similar to other programming languages. For example, parentheses will never merge, thus `(a++)` will lex as `symbol('('),identifier('a'),symbol('++')` and `symbol(')')`.

Syntax construction parser

The syntax construction parser is a simple context-free grammar, parsed using an Earley [3] parser. It is somewhat complicated by the need to parse arbitrary source code in the expansion template. This is solved by merging the syntax construction grammar with the generated grammar from grammar generation. Due to a limitation in the parser implementation used, the grammar cannot be modified during parsing, thus the expansion description of a syntax construction cannot use syntax constructions defined in the same file, only ones defined in an earlier file.

The syntax constructions are unresolved, meaning that no names are connected to their definitions. These unresolved syntax constructions are usable for grammar generation, but the remaining phases require additional processing.

Syntax construction name resolver

The syntax construction name resolver will prepare the parsed syntax constructions for use later in the implementation. All names used are resolved to locations in the syntax description and an expansion function is produced for syntax constructions that are not marked as builtin. Note that this phase does some checking of the syntax construction, but only what is necessary to produce the binding information and expansion function. The checking here is not sufficient to ensure that no binding errors appear during expansion.

Expansion checking

The expansion checker checks that an expansion can never introduce an error during expansion. Similar to type checking and various

other forms of static analysis, the expansion checker is not a necessary part, but its absence does of course remove any guarantees on errors.

Grammar generation

The grammar generator generates a context-free grammar from a set of syntax constructions and which will later be used to parse source code using an Earley [3] parser.

Ambiguity reporting

If the grammar is ambiguous, and the source code being parsed exposes this ambiguity, one parse tree will be produced for each possible parse, and all of them will be compared to determine the ambiguous sections of the source code.

Name resolution

The name resolver takes a syntax tree generated by the parser and checks for name binding errors. If none are found all symbols are replaced with newly generated symbols in such a way that each symbol appears in a binding position exactly once, while preserving the referencing relation between identifiers.

Expansion

The source code expander repeatedly finds a single syntax construction to expand, expands it, then runs the name resolver over the result, until all remaining syntax constructions are marked as builtin. This is the simplest implementation that will always be correct, but it almost always does far more work than necessary. See Section 5.5 for more information.

Core language interpreter

The core language interpreter evaluates a syntax tree composed only of syntax constructions that are part of the core language described in Section 5.1.1, but is largely unrelated to syntax constructions as such. It is only used for validation in Section 5.4.

5.1.1 Core Language

Syntax constructions does not specify a core language, but instead treats any construction marked as ”builtin” as a core language construct, expanding all constructions encountered until only builtin constructions remain. However, to evaluate the expressibility of semantics for syntax constructions such a language must exist. This section describes the core language used throughout the evaluation section.

Both language subsets are implemented using the same underlying language, i.e., the same set of syntax constructions marked as builtin (for more information, see Section 3.4). The core language is a largely functional language using curried functions and eager execution, with a few additions:

- Sequential composition.
- Non-nested bindings, i.e `defAfter` and `defAround`.
- Builtin values using special syntax, such as `#unit`, `#true` and `#plus`. More complex features introduced via such values include:

  Conditional execution via `#if`.

  Fixpoint operator via `#fix`.

  Mutable references via `#ref`, `#deref`, and `#assign`.

  Lists via `#cons`, `#head`, `#tail`, and `#nil`.

  Continuations via `#callcc`.
- Expression functions. Syntactically similar to a function (`efun x. body` vs. `fun x. body`) these work around a

limitation in syntax constructions. They are described in more detail in the next section, but in brief: they are functions that take an expression, returns an expression, and are evaluated at compile-time.

All syntax constructions in the core language have syntax type `BExpression`.

Expression Functions

Several syntax constructions in the evaluated languages turn out to require some information about their context, some examples include:

- A `return` statement must exit the surrounding function.
- A pattern in a pattern match needs a value to compare to.
- A `break` statement must exit the surrounding loop.

The only way to connect multiple syntax constructions in their current design is through the name binding system: one construction defines a binding, another uses it. This requires user interaction however, the identifiers must be in the source code written by the user. None of the examples above follow such a pattern, they are all expected to obtain their information through their lexical context.

In some cases this can be solved through a function, for example, a pattern might expand to a function that expects the value to compare to as its argument. Problems arise when the syntax construction both requires information from its context, and additionally needs to expose bindings via `#bind before` or `#bind after`. Functions introduce a scope around their body, preventing any such bindings from being visible.

To solve this issue we extend our core language with an additional kind of function, one that takes an expression as an argument and evaluates at compile-time. Essentially an anonymous macro. An `efun` will not prevent accidental name capture, but since they are evaluated after the expansion provided by the syntax construction implementation, name resolution has already happened, i.e., all identifiers that do not relate to each other have different symbols.

For example,

```
(efun a. #add a a) (#add 3 4)
// evaluates to
#add (#add 3 4) (#add 3 4)
// (at compile-time)
```

We define the syntax construction as follows:

```
syntax exprfun:BExpression =
  "efun" x:Identifier "." e:BExpression
{
  #bind x in e
  builtin
}
```

Now we can let a pattern take the value to compare through an `efun`, and provide the bindings we need.

The problem is that the above definition is insufficient; the binding semantics are not properly described.

To demonstrate why, consider the following code:

```
(efun a. defAfter x = a) x
```

`defAfter` creates a new name binding for the remainder of the scope, but does not allow a recursive definition; we could not replace `a` with `x` in the example since it is not bound. However, the second occurrence of `x` in the example *is* bound, since it occurs after the definition. Evaluating the `efun` produces the following:

```
defAfter x = x
```

The former produces no binding errors, but the latter does. Using a binding defined inside an `efun` in an argument to the same `efun` might create problems, but we have no way of expressing this restriction using syntax constructions. An expansion may thus appear to be correct to the expansion checker, since it preserves all specified binding semantics, and yet introduce a binding error when the `efun` is evaluated.

The second problem is one of inconvenience, many syntax

constructions may now need to take arguments and pass them on to their children, even though they themselves have no use of them. For example, to get information from a loop to a `break` statement it may need to be passed through many intermediate statements.

This additionally has an impact on composing multiple languages. No new syntax constructions can be added in-between the syntax construction that gives the context and the one that uses it without threading the information, but the odds of a syntax construction from a different language being written to thread the same information seem minuscule, thus importing a syntax construction from a different language is unlikely to work if either language uses that kind of contextual information.

Expression functions are thus not the ideal solution, but a better option is left for future work.

5.2 Case Studies in Expressiveness

This section evaluates the expressiveness of syntax constructions through a number of cases. The first two implement subsets of two pre-existing programming languages: OCaml and Lua. The former focuses on implementing pattern matching and a fairly standard functional programming language, while the latter focuses on implementing control flow commonly present in imperative programming languages. The final section examines several smaller examples not present in either language that demonstrate limitations of syntax constructions.

5.2.1 A Functional Language - OCaml Subset

The functional language implemented here is a subset of OCaml, with a focus on supporting pattern matching. This section will not examine the entirety of the language definition, most syntax constructions used are fairly simple and rote, focus will instead be placed on the implementation of patterns and pattern matching, as well as additional, difficult to express particulars of the language. The full language description can be found in Appendix B.

Patterns and Pattern Matching

Pattern matching is a feature found in most functional languages, allowing the programmer to simultaneously check the structure of data and extract internal data from it. Patterns describe the expected shape of the data, as well as what internal data should be extracted and bound to names for later use. They can be arbitrarily nested, allowing a simple way of expressing a potentially quite tedious checking process. Listing 4 shows an example of a match expression in OCaml, wherein the first match arm with a matching pattern is selected for execution.

```
let result = match [[4]] with
  | [[a], b] ->
    print_string "a:int, b:int list, both bound"
  | _ :: _ :: _ ->
    print_string "nothing bound, length >= 2"
  | [[4]] ->
    print_string "nothing bound"
  | _ ->
    print_string "nothing else matched"
```

Listing 4: Example match expression in OCaml

The above suggests a fairly clear formulation of the syntax and bindings of the match expression using syntax constructions.

```
syntax match:Expression =
  "match" e:Expression "with"
  arm:("|" p:Pattern "->" body:Expression)+
{
  #scope arm:(p body)
  #scope (e)
  <...>
}
```

Patterns expose their bound names using `#bind x after` and each match arm introduces a new scope containing its pattern and expression. The patterns themselves are also simple to express in terms of syntax and bindings:

```
syntax wildcardPattern:Pattern = "_" {
  <...>
}

syntax bindPattern:Pattern = id:Identifier {
  #bind id after
  <...>
}

syntax integerLiteralPattern:Pattern = i:Integer {
  <...>
}

syntax consPattern:Pattern =
  head:Pattern "::" tail:Pattern
{
  #assoc right
  #prec 10
  <...>
}
```

Implementing each of these is where problems arise; each pattern must do two things:

- Check a value provided by the enclosing pattern or match expression. The result of this check must be usable to change control flow.
- Possibly expose a bound name, where the bound value is derived from the checked value, but only needs to be valid if the check succeeded.

The former implies a function, each pattern might be implemented as a function receiving the value to check and producing a boolean value signifying the result of the check. The latter precludes the use of a function, since a function introduces a new scope, thus preventing any bound name from being available outside its body.

Other pattern match systems implemented as macros (e.g., [14]) tend to have a more complicated match macro, which constructs the final

checking code and binding code by examining the actual contents and / or results of the patterns. Syntax constructions on the other hand have no such functionality, child nodes may only be treated atomically, i.e., moved, removed, or duplicated. As such the composing mechanism must be present in the pattern implementation, but it cannot be a function, since it would hide name bindings.

For example, the following definition of `bindPattern` has an incorrect expansion template; the binding introduced is hidden inside the function:

```
syntax bindPattern:Pattern = id:Identifier {
  #bind id after
  BExpression` fun value.
    defAfter `id(id) = value
}
```

As discussed in Section 5.1.1 we will instead use `efun`s, essentially anonymous macros.

A pattern then becomes an `efun` taking a function to call if the match fails and a value to match against. Implementation becomes as follows:

```
syntax wildcardPattern:Pattern = "_" {
  BExpression` efun fail. efun value.
    #unit
}

syntax bindPattern:Pattern = id:Identifier {
  #bind id after
  BExpression` efun fail. efun value.
    defAfter `id(id) = value
}

syntax integerLiteralPattern:Pattern = i:Integer {
  BExpression` efun fail. efun value.
    #if (#equal value `int(i))
      (fun _. #unit)
      (fun _. fail #unit())
```

```
}

syntax consPattern:Pattern =
  head:Pattern "::" tail:Pattern
{
  #assoc right
  #prec 10
  BExpression` efun fail. efun value.
    (#if (#equal value #nil)
       (fun _. fail #unit)
       (fun _. #unit);
     defAfter headValue = #head value;
     `t(head) fail headValue;
     defAfter tailValue = #tail value;
     `t(tail) fail tailValue)
}
```

The pattern implementing matching against a list literal (i.e., `[a, b]` as opposed to `a :: b :: []`) is a straightforward extension of the cons pattern.

Continuing, the implementation of **match**, while not trivial, is a relatively straightforward composition of its match arms. For each arm, give the pattern the value to check, and a function to call to skip to checking the next arm in case the pattern fails, then evaluate the body and return the result. The implementation uses `callcc` to make an abortable function: `#callcc (fun abort. body)` evaluates to one of two things: the argument provided to `abort`, or the result of evaluating `body` if `abort` is never called.

```
syntax match:Expression =
  "match" e:Expression "with"
  arm:("|" p:Pattern "->" body:Expression)+
{
  #scope arm:(p body)
  #scope (e)
  #prec 1
  BExpression` #callcc (fun end. (fun val.
    `t(foldr arm prev
         (BExpression`
```

```
            #callcc (fun next.
              (`t(p) next val;
               end `t(body)));
            `t(prev))
        (BExpression` #crash)))
    `t(e))
}
```

Note that the implementation is somewhat naive, for example no work is shared between the arms, even if there is overlap in the patterns, and there are no warnings for overlapping patterns or incomplete coverage.

The syntax descriptions and binding specifications on the other hand are clear an concise and may be considered a success.

Patterns in Function Declarations

Patterns are not limited only to the **match** expression in OCaml, they can additionally be used in function definitions. The most common way of writing such a function uses **let** with some syntax sugar to allow writing all the arguments next to the function name, instead of as part of nested **fun** expressions. The two definitions below are equivalent, and we would like our OCaml subset to support both:

```
let add a b = a + b
let add = fun a -> fun b -> a + b
```

As an informal description of this feature, the identifier being bound in a **let** expression can be followed by any number of parameters (patterns), in which case the value being bound should be a curried function with those parameters. The syntax construction `topLet` below expresses this.

```
syntax topLet:Top =
  "let" x:Identifier args:Pattern*
  "=" e:Expression (";" ";")?
{
  #bind x after
  #scope (args (e))
```

```
  BExpression` defAfter `id(x) = `t(
    foldr args next
      (BExpression` fun x.
        (`t(args) (fun _. #crash) x;
         `t(next)))
    e)
}
```

Using `topLet` to expand the previous definition of `add` we obtain a syntax tree essentially equivalent to the following:

```
defAfter add =
  (fun x. <a> (fun _. #crash) x;
    (fun x. <b> (fun _. #crash) x;
      #plus a b))
```

where `<a>` and `<b>` are placeholders for the expansions of the patterns used for the two arguments. In this particular case, both will be `efun`s ignoring their first argument and using `defAfter` to expose their second argument. This works, and will work for all sensible patterns we might define for OCaml, but there is a problem. Consider the following two patterns (their expansion is unimportant for the current discussion and thus elided):

```
syntax equalPattern:Pattern =
  "=" "(" e:Expression ")"
{
  #scope (e)
  <...>
}

syntax aroundPattern:Pattern =
  id:Identifier
{
  #bind id before
  #bind id after
  <...>
}
```

`equalPattern` evaluates an expression and compares for equality,

while `aroundPattern` is the same as `bindPattern` except it binds before as well as after. `aroundPattern` is the real problem, but we need `equalPattern` to expose it.

As an example of how these patterns might be used,

```
let foo a =(a) = true
```

defines `foo` to be a function that takes two arguments. The arguments are compared, and if they are equal, the function returns `true`, otherwise it produces an error (failed pattern match).

Swapping the two patterns produces the code below, which should ostensibly produce the same result:

```
let foo =(a) a = true
```

However, expanding the **`let`** definition produces code equivalent to the following (using `<a>` for the expansion of `=(a)` and `<b>` for the expansion of `a`):

```
defAfter add =
  (fun x. <a> (fun _. #crash) x;
    (fun x. <b> (fun _. #crash) x;
      #plus a b))
```

The inner function introduces a binding for `a` in its body, but the outer function uses this binding. Yet this is an error, since bindings introduced in the body of a function are not available outside of it.

Expansion checking detects this and rejects the expansion specification, preventing an error from being introduced after expansion has started. The problem is that all patterns in OCaml that introduce bindings only do so for code appearing after themselves, thus this expansion is fine for all cases we are interested in. However, anyone could add a new pattern that works differently we must handle such cases as well.

There are two plausible ways around this, neither of which is supported by syntax constructions at present.

- Make a more flexible scope declaration. If `topLet` had a scope declaration along the lines of

`(args1 (args2 (args3(... (e)))))`, i.e., a new scope is introduced after each argument, disallowing previous patterns from referencing names bound in later patterns, properly reflecting the semantics of the expansion.

- Limit syntax constructions of syntax type **`Pattern`** to only allow binding after, capturing our intuition of how patterns in OCaml are supposed to work in the actual patterns rather than in syntax constructions that use them.

Lists and Ambiguous Syntax

A list literal in OCaml is a list of semi-colon separated expressions, enclosed in square brackets. The syntax description of a corresponding syntax construction is straight-forward and can be seen below:

```
syntax listLiteral:Expression =
  "[" (e:Expression (";" es:Expression)*)? "]"
{
  <...>
}
```

However, OCaml additionally supports sequential composition of expressions in the form of an operator, semi-colon. This syntax construction is also straight forward:

```
syntax sequentialComposition:Expression =
  e1:Expression ";" e2:Expression
{
  #assoc right
  #prec 2
  <...>
}
```

The resulting composed syntax is ambiguous however: a list literal of length greater than one will never be parsed unambiguously, there is nothing to distinguish the item separators from the sequential composition operator. The OCaml parser special cases this, parsing

`[1;2;3]` as a list of three elements and `[(1;2;3)]` as a list of one element.

Syntax constructions can implement this, but not in a convenient fashion. A first, almost convenient, approach would be to create a new syntax type, for example named **`Statement`**, moving sequential composition to it and creating a new syntax construction consisting of a single **`Expression`**. This allows us to state when we want an expression including sequential composition (**`Statement`**) or excluding sequential composition (**`Expression`**). Most previous uses of **`Expression`** would now instead use **`Statement`**, except list literals.

The first issue with this solution is partially that it is a workaround that requires changes in many places that very much requires each syntax construction to be considered in its context (thus breaking a design goal). The second issue is that it does not work: the original **`Expression`** contains syntax constructions with both lower and higher precedence than sequential composition. The rewrite above essentially gives sequential composition lower precedence than all constructions in **`Expression`**, thus altering how the language is parsed.

A proper solution might instead require three syntax types: **`ExpressionWith`**, **`ExpressionWithout`**, and **`ExpressionBase`**. The last contains the syntax constructions that were in **`Expression`** with higher precedence than sequential composition, while the first two contain duplicates of the remaining syntax constructions, except for sequential composition, which is only present in **`ExpressionWith`**.

This solution seems inconvenient enough to not really consider a solution, thus we will instead count it as a limitation of the system as currently designed.

5.2.2 An Imperative Language - Lua Subset

The imperative language implemented to test the expressiveness of syntax constructions is a subset of Lua. The purpose is to evaluate expressibility of common control flow constructs present in

imperative languages, as such tables and global variables, arguably the more unique aspects of the language, are not implemented. The control flow constructs here implemented are loops, `break` and `return`.

Loops and Break

A loop in an imperative language will evaluate its body for side effects, possibly multiple times, for as long as some condition holds. In addition certain statements exist to alter the normal control flow of a loop; in the case of Lua this is limited to the **`break`** statement, but other languages may contain statements such as **`continue`**, **`redo`**, or **`next`**.

This section details the implementation of a **`while`** loop and the **`break`** statement.

Starting with the former, **`while`**, the implementation is relatively straightforward. The condition check and body is expanded into a function that recursively calls itself to perform another condition check and iteration. The recursion is performed using the fixpoint operator `#fix`.

```
syntax while:Statement =
  "while" cond:Expression "do"
  body:Block "end"
{
  BExpression` #fix (fun repeat. fun _.
    #if (#deref `t(cond))
      (fun _.
        (`t(body);
         repeat #unit))
      (fun _. #unit))
}
```

Note that the body has syntax type **`Block`** as opposed to the more intuitive **`Statement`**+. This is due to a peculiarity of the Lua syntax, a block is a list of statements optionally terminated by a break or return statement, which is also the only place where a break or return statement is legal. Since the block structure appears in multiple

constructs beside **while** loops, e.g., **function**s and **if** statements, this formulation reduces code duplication.

We define a **Block** to expand to a **BExpression**, and define a single syntax construction `blockContent`:

```
syntax type Block = BExpression

syntax blockContent:Block =
  (stmnt:Statement ";"?)* ret:Terminator?
{
  #scope (stmnts ret)
  foldr stmnt next
    (BExpression` `t(stmnt); `t(next))
  foldr ret _
    (BExpression` `t(ret))
  (BExpression` #unit)
}
```

Continuing with the implementation of the **break** statement we run into some issues. **break** should transfer control flow past the end of the enclosing loop, regardless of the form of said loop, e.g., it should not matter if it is a **while** or **for** loop. This can be done by making all the loops construct an abort function using `#callcc`, but then this function has to be provided to the **break** statement somehow.

One solution is to have **break** expand to a function taking the abort function as an argument. However, **break** may appear inside some nested statement of the loop body, thus each statement of the body must also be supplied this function, in case it contains a **break**. The code below shows this implementation, threading the abort function through all the statements:

```
syntax while:Statement =
  "while" cond:Expression "do"
  body:Block "end"
{
  BExpression` fun _. #callcc (fun break.
    #fix (fun repeat. fun _.
      #if (#deref `t(cond))
        (fun _.
```

```
          (`t(body) break;
           repeat #unit))
        (fun _. #unit)))
}

syntax type Block = Statement

syntax blockContent:Block =
  (stmnt:Statement ";"?)* ret:Terminator?
{
  #scope (stmnts ret)
  BExpression` fun break.
  `t(foldr stmnt next
       (BExpression` `t(stmnt) break; `t(next))
     foldr ret _
       (BExpression` `t(ret) break)
     (BExpression` #unit))
}

syntax type Terminator = Statement

syntax break:Terminator = "break" {
  BExpression` fun break. break #unit
}
```

This implementation causes a new problem however: **local**, which introduces a new local binding, must also accept the same argument since it is a statement, but a function introduces a scope, preventing the binding from being exposed:

```
syntax local:Statement =
  "local" x:Identifier ("=" e:Expression)?
{
  #bind x after
  BExpression` fun _.
    defAfter `id(x) = #ref
      (#deref `t(foldr e _ (e)
                   (BExpression` #unit)))
}
```

```
function fibonacci(n)
  local prev = 0
  local curr = 1
  local temp
  for i = 1, n-1 do
    temp = b
    b = a + b
    a = temp
  end
  return curr
end
```

Listing 5: An example in Lua demonstrating a simple function

`break` must be enclosed in a function to receive the abort function, but `local` must not be enclosed in a function, otherwise it cannot expose its binding. A syntax construction has a very limited set of actions it can perform in regards to its child nodes during expansion (moving, removing and duplicating) and thus cannot distinguish between a statement that needs the abort function (e.g., `break` and `if`) and a statement that cannot be wrapped in a function (e.g., `local` and `function`) lest its binding is hidden.

The actual implementation instead uses functions that do not introduce a new scope, the expression functions introduced and discussed in Section 5.1.1. Replacing the outermost enclosing `fun` in each of the syntax constructions above with `efun` resolves the error and produces a correct implementation.

Functions and Return

A return statement in an imperative language returns control flow to the caller of a function, optionally providing a return value. Reaching the end of a function without encountering a return statement also returns control flow and is equivalent to a return statement without a return value. Listing 5 demonstrates a simple function with a single return.

In the case of nested functions a return statement should only return

from the inner-most function. The syntax of a return statement is trivial to describe and can be seen below.

```
syntax return:Statement =
  "return" value:Expression?
{ <...> }
```

The implementation is rather less obvious. The return statement must transfer control flow to a point external to itself, namely to the caller of the enclosing function. However, the situation is rather similar to the **break** statement (see Section 5.2.2), so we employ a similar solution: a function declaration introduces an abort function via `#callcc` and threads it to all statements via expression functions:

```
syntax function:Statement =
  "function" f:Identifier
  "(" (a:Identifier ("," as:Identifier)*)? ")"
  body:Block "end"
{
  #bind f before
  #bind f after
  #bind f, a, as in body
  BExpression` efun _. efun _.
    (defAround `id(f) = #fix (fun recur.
      (defAfter `id(f) = recur;
       `t(foldr a next
            (BExpression` fun `id(a). `t(next))
          foldr as next
            (BExpression` fun `id(as). `t(next))
          (BExpression`
            fun _. #callcc (fun return.
              `t(body) return
                       (fun _. #crash)))))))
}

syntax return:Return = "return" e:Expression? {
  BExpression` efun return. efun break. `t(
    foldr e _
      (BExpression` return (#deref `t(e)))
    (BExpression` return #unit))
```

```
}
```

Note that this change adds yet another wrapping `efun` around all statements, whether it should affect them directly or not.

Function Calls and Precedence

A function call in Lua appears syntactically as an expression evaluating to a function followed by an argument list within parenthesis, while the core language only has single argument functions and uses juxtaposition as application. The translation between the two is straightforward and is as follows:

```
syntax funcCall:Expression =
  f:Expression
  "(" (e:Expression ("," es:Expression)*)? ")"
{
  BExpression` `t(
    foldl es prev
      (BExpression` (#deref `t(prev)) `t(es))
    foldl e prev
      (BExpression` (#deref `t(prev)) `t(e))
    f)
    #unit
}
```

The final `#unit` being applied is to allow calling a function that takes no arguments. Compare with the definition of **`function`** in the previous section.

This definition produces an ambiguous grammar, e.g., `1 + f()` has two allowable parses: `(1 + f)()` and `1 + (f())`. The solution takes the form of precedence, give a function call higher precedence than arithmetic. This solves the ambiguity, but presents a new problem: `f(1 + 2)` is no longer parsed correctly.

The issue stems from the generalization of precedence that is used in syntax constructions (see Section 3.2 or Section 4.3.1). Whenever a syntax construction recursively uses its own syntax type (e.g., `funcCall`, which is an **`Expression`**, uses **`Expression`** in the

syntax description) those recursive uses may only contain syntax constructions of the same, higher, or undefined precedence. Since addition has lower precedence it is disallowed both in the function being called and in the argument list. The latter is undesired.

This can be solved by an extra layer of indirection. If the argument list is broken out into a separate syntax construction with a different syntax type it's uses of **Expression** will be entirely unconstrained.

The correct solution then becomes as follows:

```
syntax funcCall:Expression =
  f:Expression "(" args:ArgList ")"
{
  #prec 15
  BExpression` `t(args) `t(f) #unit
}

syntax argList:ArgList =
  (e:Expression ("," es:Expression)*)?
{
  BExpression` fun f. `t(
    foldl es prev
      (BExpression` (#deref `t(prev)) `t(es))
    foldl e prev
      (BExpression` (#deref `t(prev)) `t(e))
    (BExpression` f))
}
```

This seems clearly undesirable, assigning precedence affects more things than intended and rectifying the problem requires introducing a new syntax type that we most likely would not have wanted otherwise.

5.2.3 Other Limitations

Variables in Different Domains

Many programming languages have bindings that are split in different domains, the most common example probably being a split

```haskell
data T = V
err1 = T -- Not in scope: data constructor 'T'
ok = V :: T

err2 = case V of
  U -> "U" -- Not in scope: data constructor 'U'
  u -> "u"
```

Listing 6: Identifiers in Haskell are interpreted differently depending on their syntactical position, as well as the characters in their symbols.

between values and types. For example, Listing 6 shows the result of attempting to use an identifier representing a type in a value position: an unbound error. This despite the same identifier being used on the very next line, thus clearly being bound. The definitions end up in different domains where they neither conflict nor interact.

Additionally the patterns on the last two lines contain unbound identifiers, but the semantics differ based on the casing of the characters in their symbols. The first of the two matches against a data constructor (the domain was selected by the contents of the symbol), which must be bound, thus an error is produced, while the second binds `u` to the value of whatever is being matched.

Syntax constructions have only a single domain of bindings, and does not allow differentiating identifiers based on the form of their symbols, and can thus not express the above at all. Adding such a capability seems possible however, but is left for future work.

Prolog Pattern Matching

Section 5.2.1 evaluates the formulation of a pattern matching system using syntax constructions, used in the OCaml subset. However, other programming languages have pattern matching with more features. For example, Prolog allows matching an unbound variable twice in a pattern, with the semantic meaning of requiring the values at both positions to be equal. Listing 7 shows a simple example.

An implementation of the syntax and binding semantics of this in syntax constructions might look as follows:

```
abs(A, B) :- A < 0, B is -A.
abs(A, A) :- A >= 0.
```

Listing 7: Prolog rule stating the conditions for the second value being the absolute value of the first.

```
syntax variableBinding:Pattern = id:Identifier {
  #bind id after
  <...>
}
syntax equalityPattern:Pattern = id:Identifier {
  <...>
}
```

The first syntax construction binds the result of the pattern match, similarly to the pattern matching system in Section 5.2.1. The second has no `#bind id after` statement and thus requires `id` to be bound by the environment. If we disallow shadowing these syntax constructions are mutually exclusive; if the symbol is already bound we cannot bind it again without a redefinition error and we must choose `equalityPattern`, if it is unbound we cannot find that it refers to anything and we must choose `variableBinding`.

If we allow shadowing of bindings from an outer scope both syntax constructions will be valid in some cases, namely when the symbol is bound in an outer scope. In this case we could shadow it and choose `variableBinding`, or refer to the pre-existing binding and choose `equalityPattern`.

Syntax constructions allow shadowing of definitions from an outer scope, thus the above would not be guaranteed to be unambiguous. Furthermore, due to the way parsing is implemented the above would double the number of parse trees for every occurrence of an identifier in a pattern, which in a normal Prolog program would result in a very large number of parse trees. In the interest of performance the current implementation therefore requires all disambiguation to be done syntactically, thus this form of pattern matching could not be expressed at all.

5.3 Language Composition

This section evaluates the capabilities of syntax constructions in terms of language composition. Section 5.3.1 uses the patterns defined in the OCaml subset to implement destructuring in the Lua subset, while Section 5.3.2 attempts to import the **match** expression into Lua.

5.3.1 Patterns in Lua

We start off by adding destructuring to a **local** binding. The **local** syntax construction is originally implemented as follows:

```
syntax local:Statement =
  "local" x:Identifier ("=" e:Expression)?
{
  #bind x after
  #scope (e)
  BExpression` efun _. efun _.
    defAfter `id(x) = #ref
      (#deref `t(foldr e _ (e)
                   (BExpression` #unit)))
}
```

Importing all the **Pattern**s from OCaml, replacing **Identifier** with **Pattern**, and making the corresponding change in the expansion yields the following implementation:

```
syntax local:Statement =
  "local" x:Pattern ("=" e:Expression)?
{
  #scope (e)
  BExpression` efun _. efun _.
    `t(x) (fun _. #crash)
      (#ref (#deref `t(foldr e _ (e)
                         (BExpression` #unit))))
}
```

The semantics of the resulting syntax constructions are essentially as

expected, patterns can be used to destructure the value and introduce new bindings. However, a `local` binding can be mutated, which is here implemented by introducing a reference to a value, as opposed to purely the value itself. As such, any pattern that assumes a non-reference value will fail. This stems from a mismatch between the underlying core language and Lua; the core language is explicit about what is a reference and what is not, while Lua will frequently produce references and implicitly dereferences them when required. OCaml, which these patterns were written for, is more in line with the core language and performs no implicit dereference.

We can alleviate this mismatch somewhat by allowing the `local` construct to attempt the pattern both with and without dereferencing. This essentially adds auto-dereferencing to the pattern match, but only for non-nested cases. A pattern appearing somewhere within another pattern will not auto-dereference, so to handle this case we introduce a new pattern, `derefPattern`, to explicitly match against a reference.

```
syntax local:Statement =
  "local" x:Pattern ("=" e:Expression)?
{
  #scope (e)
  BExpression` efun _. efun _.
    (fun expr. #callcc (fun done.
      (#callcc (fun fail. `t(x) fail expr);
       `t(x) (fun _. #crash) (#deref expr))))
    (#ref (#deref `t(foldr e _ (e)
                     (BExpression` #unit))))
}
syntax derefPattern:Pattern =
  "ref" p:Pattern
{
  BExpression` efun fail. efun value.
    `t(p) fail (#deref value)
}
```

The remaining binding constructions in Lua (e.g., functions and for loops) can be changed in similar ways to support destructuring. Patterns in function arguments exhibit the same issues here as they

do in OCaml, see the end of Section 5.2.1, but otherwise work as expected. We have thus reused patterns from OCaml without needing to change them, only requiring changes to the parts of Lua that should use the patterns.

5.3.2 Match in Lua

Patterns can do more than simple destructuring, OCaml additionally has a `match` construct that enables conditional execution based on which pattern matches a given value. Reusing this construct in Lua turns out to be problematic however, for a few reasons:

1. OCaml consists largely of expressions while Lua makes a distinction between statements and expressions. As such, `match` is an expression. In OCaml it makes sense to have a control flow construct as an expression, but in Lua most computation requires statements, thus we likely want `match` to be a statement.
2. We may consider that in a real world scenario the two languages would likely have been developed entirely independent of each other, i.e., the syntax type `Expression` of OCaml is distinct from the syntax type `Expression` of Lua.
3. We only wish to bring in `match`, not the entirety of OCaml; it should be possible to write Lua in each of the branches, not OCaml.
4. The syntax of OCaml differs significantly from Lua. What is natural in one may very well not be in the other.

As such we have a few options for how to integrate `match`. Broadly speaking, these options can be split in two groups: explicitly exposing the `match` from OCaml, and creating a new syntax construction that expands into a `match`, without exposing it.

Reusing `match`

The former approach immediately runs into problems because of the second point above. To write `match` as a statement we require a

syntax construction of syntax type **Statement** consisting of a sole OCaml **Expression**, a form of bridging syntax construction. Additionally, to write a Lua statement inside a **match** branch we require a bridge in the opposite direction. The combination of these two syntax constructions produce an ambiguous grammar however; they can nest any number of times when parsing a statement.

In general, this problem will appear any time we wish to include a syntax construction that is recursive, i.e., that uses its own syntax type in the syntax description.

Expanding to match

The alternative reuse option is to create a new syntax construction that expands into a **match**, i.e., to follow the tower of languages approach:

```
syntax luaMatch:Statement =
  "match" e:Expression "with"
  arms:("case" p:Pattern ":" b:Block)+
  "end"
{
  #scope (e)
  #scope arms:(p b)
  BExpression` efun return. efun break.
    (defAfter e = #deref `t(e);
     `t(foldr arms next
          (Expression` match e with
            | `t(p) -> `t(b) return break
            | _ -> `t(next))
          (BExpression` #crash)))
}
```

This syntax construction expands into a nested series of **match** expressions from OCaml and can be used like so:

```
match 4 with
  case 1: print(1)
  case 2: print(2)
```

```
  case n: print("n")
end
```

5.4 Correctness

Syntax constructions provide no semantics or specification of a core language, thus the choice of core language and a means to run programs in it are largely irrelevant to this thesis. However, we wish to know that the languages defined using syntax constructions have the correct semantics. As such, the syntax construction implementation contains a simple interpreter for the eager, largely functional core language briefly described in Section 5.1.1.

A small set of programs, written in OCaml and Lua respectively, then form a sort of correctness test suite: if a program executes without any errors in the original implementation, and the syntax construction implementation can parse it, then the results should be the same no matter which implementation runs it. The former requirement stems largely from the absence of type checking in syntax constructions, and the latter ensures that the program is part of the defined language subset.

The testing is complicated by syntax constructions not supporting importing. Program correctness is checked by comparing printed output, but printing functions tend to be defined externally, via some form of library, standard or otherwise. To cover this difference we supply an extra prelude for each of the implementations. These define the expected printing functions.

As an example, the prelude for the syntax construction implementation of OCaml is as follows:

```
let p_int x = #debug x
let print_endline x = #debug x
```

The original OCaml implementation instead uses this prelude:

```
let p_int i = print_int i; print_newline ()
```

When a test program is run by either implementation we prepend the corresponding prelude. The programs themselves (without the prelude) are identical.

The test programs were chosen to ensure that between them, each syntax construction is used at least once. Additionally, the test programs use direct recursion, mutual recursion, and (in Lua) loop iteration. The OCaml subset has a total of 32 syntax constructions, while the Lua subset has 29 syntax constructions. The programs can be found under '`implementation/languages/ocaml`' and '`implementation/languages/lua`' in the implementation online[4], and are as follows:

- '`fib.ml`' and '`fib.lua`'. These programs implement functions for finding the nth fibonacci number, one with the quadratic recursive definition, and a linear version. They test most binding constructs, some control flow, and basic arithmetic.
- '`fizzbuzz.ml`' and '`fizzbuzz.lua`'. These programs implement fizzbuzz, i.e., for each number between 1 and 100, print:
  - "fizzbuzz", if the number is divisible by both 3 and 5.
  - "fizz", if the number is divisible by 3 but not 5.
  - "buzz", if the number is divisible by 5 but not 3.
  - the number itself, if its divisible by neither 3 nor 5.

  These programs test more control flow and comparisons.
- '`misc.ml`'. This program tests the various remaining syntax constructions in the OCaml subset, for example, boolean literals, anonymous functions and cons patterns.
- '`misc.lua`'. This program tests the various remaining syntax constructions in the Lua subset, for example, grouping by parentheses, **`break`**, and multiplication.

The test programs implement functions for calculating Fibonacci numbers with the quadratic recursive definition as well as the linear version, and two versions of FizzBuzz. The programs use all syntax

[4] https://github.com/elegios/master-thesis

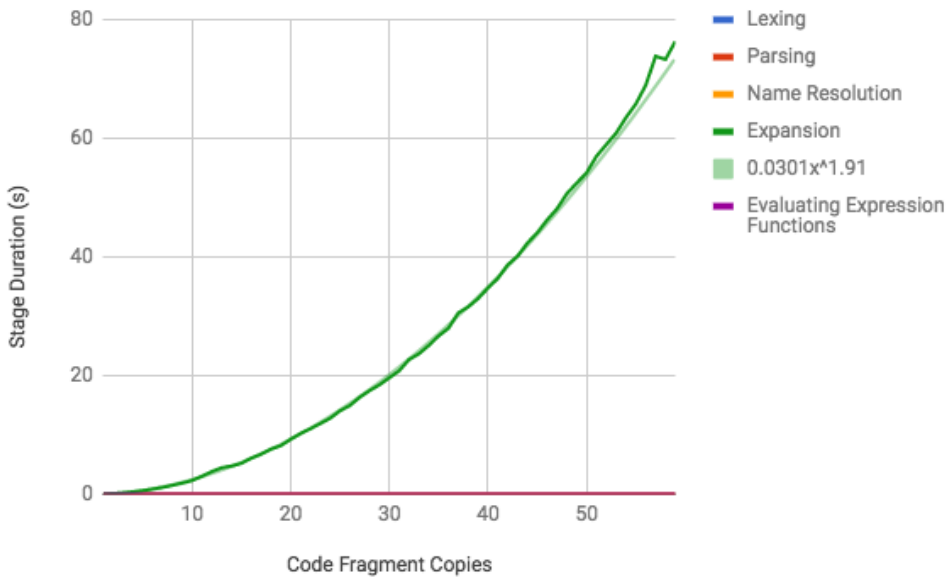

Figure 5.2: Phase runtime, the light green line is a best fit of the expansion phase

constructions at least once between them with the exception of list related constructions in OCaml.

5.5 Performance

This section evaluates the performance of the syntax construction implementation used throughout this thesis.

In practice all the sub-components of the implementation have been fast enough to not incur any noticeable delay for any of the programs tested, with one very notable exception: the expansion phase is slow. Figure 5.2 shows the result of running the tool on programs of varying length. These programs are constructed by duplicating and concatenating the code in Listing 8 an increasing number of times.

We believe the slowness of expansion to not be an inherent property of the method, but rather a consequence of the simplest possible implementation. Expansion of a syntax construction assumes that each symbol exists at most once in binding position (see Section 3.4.1

```
do
  function identity(x)
    return x
  end
end
```

Listing 8: Simple Lua program that can be concatenated with itself without causing name errors.

for more information), and name resolution uses generated unique symbols and renaming to ensure this to be true. However, an expansion may duplicate one of its sub-nodes, which might produce a tree that has the same symbol in binding position multiple times.

Thus we reach the conclusion that we may need to perform name resolution after an expansion, to replace the duplicated binding symbols with distinct symbols.

The naive implementation of expansion thus expands a single syntax construction, then runs name resolution on the entire abstract syntax tree to ensure the invariant holds, then repeats, until no more syntax constructions can be expanded. This means that name resolution will be run roughly once per node in the syntax tree. Name resolution is at least linear in the size of the tree, since it examines every node. Thus the naive implementation is at least quadratic in the size of the syntax tree, which aligns pretty well with Figure 5.2.

However, most syntax constructions do not duplicate its sub-nodes. They are generally used at most once, in which case the extra name resolution pass is unnecessary. In fact, none of the syntax constructions used to implement the OCaml and Lua subsets use a sub-node more than once, thus the entire syntax tree could be expanded in a single bottom-up traversal. The extra work of a new name resolution pass between each individual expansion is thus quite significant.

As for the remaining parts of the implementation, Figure 5.3 shows the runtime of each of the stages except expansion, for easier readability. Parsing requires an algorithm that can parse any context-free grammar and produce all syntax trees in case of ambiguities. The current implementation uses an off-the-shelf

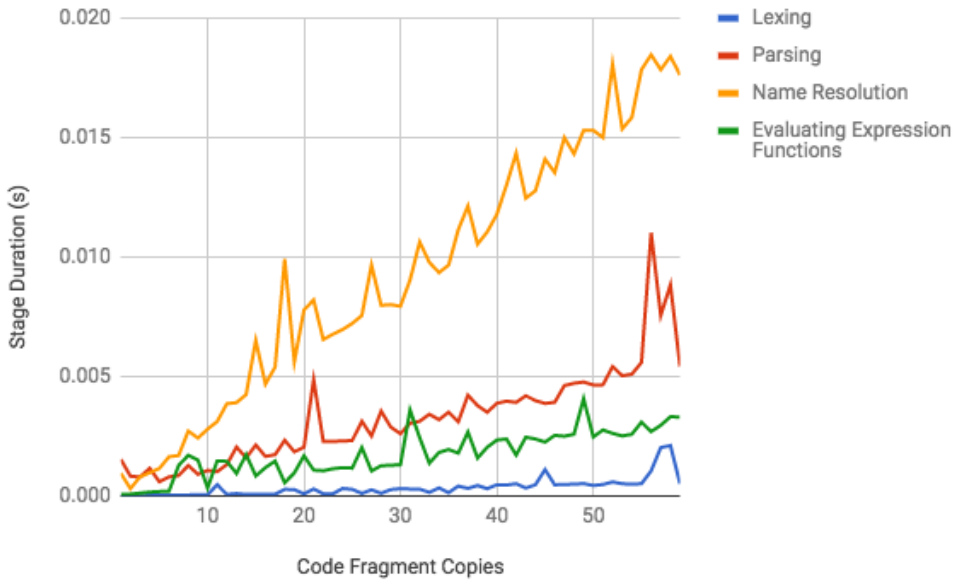

Figure 5.3: Stage runtime, excluding expansion

implementation[5] of the Earley [3] parsing algorithm. The algorithms used for grammar generation, expansion checking, expansion of a single instance (as opposed to full expansion of all instances), and name resolution have not been examined to determine time complexity, but have in practice had negligible impact on the run time of the implementation.

To summarize, the run time of the current implementation is dominated, by several orders of magnitude, by the expansion phase, which in the common case can be greatly optimized.

5.6 Error Reporting

Since one of the design goals is good error messages, this section examines the error messages produced by the implementation, and to some extent what the messages could be, given some more engineering and user experience effort.

Note that the errors examined are only those related to syntax

[5]https://hackage.haskell.org/package/Earley

```
syntax binaryAnd:Expression =
  a:Expression "&" b:Expression
{ <...> }
syntax binaryOr:Expression =
  a:Expression "|" b:Expression
{ <...> }
syntax variable:Expression =
  id:Identifier
{ <...> }
syntax parens:Expression =
  "(" e:Expression ")"
{ <...> }
```

```
 a  &  b  |  c
(a  &  b) |  c
 a  & (b  |  c)
```

Listing 9: Example of two operators without defined precedence, and the two interpretations

constructions, i.e., ambiguous source code, incorrect expansions, and name binding errors. Errors that are not covered include type errors and runtime errors, since this thesis makes no contribution to either kind of error.

5.6.1 Ambiguous Source Code

Syntax constructions provide no guarantee that a composed language has an unambiguous grammar. As such a user may write code that cannot be parsed unambiguously. At that point the user must be presented with a helpful error that can assist with solving the problem, without deep knowledge of the internals of the particular language, nor of grammars and parsing in general.

The error message should quickly direct the user to the actually ambiguous part of the code, and ideally also present what to change to select one of the possible interpretations.

Listing 9 shows an example of two binary operators without defined precedence, requiring the user to explicitly group the operators. In

this case the error produced will mark the correct part of the source code as ambiguous and present two different interpretations:

- A `binaryOr` of a `binaryAnd` and a `variable`.
- A `binaryAnd` of a `variable` and a `binaryOr`.

It is worth noting that the current implementation shows the above information, plus the source code location of each mentioned syntax construction instance (both start and end), but the location information is here omitted for brevity and readability.

Ideally the system could detect that `parens` have no effect beyond grouping, and that it could be used to force one interpretation or the other, thus using it instead of the textual representation above, but that functionality is not present at the moment. As such the implementation will correctly identify the ambiguous section of the code, and provide a clear enough representation of the different ways to interpret the ambiguity, but no suggested changes are presented.

Listing 10 shows a definition of a **`let`** expression used for a while during construction of the OCaml subset in Section 5.2.1. Since **`let`** has no defined precedence it can both be a direct part of a sum (first interpretation in the example) and have a sum in its body (second interpretation in the example), without breaking precedence. The error presented marks the correct area of code, and presents two different interpretations:

- A `sum` of a `let` and a `variable`.
- A `let` of a `bindingPattern`, an `integerLiteral` and a `sum`.

The latter interpretation is somewhat less obvious than the former, since we rarely think of a **`let`** expression in terms of its nested syntactical elements, but the meaning is clear enough after comparing the interpretation with the offending source code.

It is worth noting here that parentheses could be used for disambiguation, similar to above, but the syntax of OCaml dictates that the code in Listing 10 should be unambiguously parsed as the second alternative. The solution in this case is to give **`let`** a low precedence.

```
syntax sum:Expression =
  a:Expression "+" b:Expression
{
  #prec  11
  #assoc left
  <...>
}
syntax let:Expression =
  "let" x:Identifier args:Pattern*
  "=" e:Expression "in" body:Expression
{ <...> }
syntax bindingPattern:Pattern =
  id:Identifier
{ <...> }
syntax integerLiteral:Expression =
  i:Integer
{ <...> }
```

```
 let x = 1 in  2  +  x
(let x = 1 in  2) +  x
 let x = 1 in (2  +  x)
```

Listing 10: Example of a definition for a let expression leading to an ambiguous syntax, and an example with two interpretations

Listing 11 shows the list literal as originally implemented in the OCaml subset (see Section 5.2.1 for more details). The ambiguity occurs since the item separator in a list is the same as the sequential composition operator. The interpretations presented by the error message are as follows:

- A sequence of a `sequentialComposition` and a `*` repetition covering no source code.
- A sequence of a `sequentialComposition` and a `*` repetition covering `; c`.
- A sequence of a `variable` and a `*` repetition covering `; b ; c`.

These interpretations expose rather more implementation details than

```
syntax sequentialComposition:Expression =
  a:Expression ";" b:Expression
{
  #assoc right
  #prec 2
  <...>
}
syntax listLiteral:Expression =
  "[" (e:Expression (";" es:Expression)*)? "]"
{ <...> }
syntax variable:Expression =
  v:Identifier
{ <...> }
```

```
[ a ; b ; c ]
[(a ; b ; c)]
[(a ; b); c ]
[ a ; b ; c ]
```

Listing 11: Example of an ambiguous list literal. The alternative interpretations are presented as OCaml would parse them.

the previous ones. A `sequence` refers to a parenthesized list of syntactical elements in a syntax description, i.e., `(e: Expression (";" es:Expression)*)` above. A `*` repetition refers to a syntactical element repeated using an EBNF operator, i.e., `(";" es:Expression)*` above.

The `sequence` is reported instead of `listLiteral` since, implementation-wise, it is the closest node in the syntax tree that covers the entire ambiguity. The `listLiteral` also covers the square brackets, but those are parsed unambiguously. The repetition is reported as such instead of some number of `variables` simply because it was the easiest information to obtain given the current implementation.

A language implementer might appreciate the above representations of the interpretations, while an end user would most likely rather want the closest enclosing syntax construction and a list of its child syntax constructions, regardless of internal structure of the syntax

construction and it possibly covering some unambiguous parts of the code.

Since ambiguity reporting identifies the minimal subtrees that are ambiguous, either representation is simple to obtain. The user could thus potentially choose their own preference, depending on whether they are an end user or a language implementer.

Static and Dynamic Ambiguity Detection

The errors presented above bear something of a resemblance to runtime errors in a dynamic programming language, i.e., a programming language without compile time type checking: an ambiguous language only produces errors at parse-time and an incorrect program only produces errors at run-time.

Correctness in a dynamically typed language tends to depend more on testing, manual or automated, than static guarantees, since the latter are largely absent. Testing for ambiguity is rather straightforward, merely provide example code and state whether it is ambiguous or not, then try parsing it.

However, ambiguity is generally a property of a set of syntax constructions, not of any syntax construction in itself. Passing tests for one set of syntax constructions (i.e. a language) give little to no information for another set, even if the two intersect (i.e., they share syntax constructions). Thus a complete set of tests need to be created for each language.

This is analogous to integration tests in a dynamically typed language; a new system (i.e., a new composition of sub-systems) require new integration tests. However, regular programming languages additionally tend to have unit tests, which tests some unit of the program (e.g., a function) independently of the rest of the program. These tests give useful results about the tested units even when they are composed into a bigger system. Testing syntax construction languages for ambiguity has no corresponding concept, thus requiring the testing process to essentially start from scratch for every new language.

Static ambiguity detection on the other hand, i.e., compile-time

checking, would alleviate this effort greatly, since it would be automatic and ideally complete for each language without extra work for the language designer. This thesis makes no attempt at determining the ambiguity of a language, instead leaving such an approach for future work.

5.6.2 Expansion Specification Errors

The tool implemented for this thesis performs the checking necessary to ensure that no syntax construction has an expansion that may produce a malformed syntax tree. The errors presented when such an expansion is encountered are informative, but no effort has been put into making the presentation user friendly. Keeping that in mind, the errors presented are one of the following:

Missing export: The syntax construction specifies `#bind x before` or `#bind x after`, but the expansion does not expose `x`. This error is also produced if a syntax construction appears in the top-level scope in the syntax construction before expansion, but is in an inner scope or not present after expansion, since it could also export a symbol.

Dependency error: The checker uses a concept of a dependency, which is either an **`Identifier`** or a syntax construction along with a direction (before or after). These define the sources from which a reference may find its binding. Since any of these sources may be used the dependency set of a syntax construction after expansion must be a superset of its dependency set before expansion.

Self dependency: Following the definition of a dependency above, if a syntax construction depends on itself (i.e., on a copy of itself) after expansion, and it binds a symbol, then this symbol is defined (at least) twice. Technically this is only an error if at least two of the definitions are in the same scope, shadowing is allowed as long as it is in a child scope, but the implementation is currently slightly overly cautious and rejects any self dependency.

Binding error: Since an expansion can use any syntax in its definition

it may also introduce new bindings and identifiers. These are checked the same as normal name resolution and reported in a similar fashion.

Note that the expansion checking is performed on a single syntax construction, entirely independent of other syntax constructions or source code to parse.

The errors are presented in terms of hypothetical syntax trees with the syntax construction being checked as the root. References to elements in this tree are by position in the tree, since not all things are named in the syntax description and even things that are named might be duplicated because of an EBNF operator.

As a non-trivial example, consider this implementation of an **if**-statement (from the Lua implementation in Section 5.2.2). The expansion uses the core language builtin `#if`, which requires the two branches to be single argument functions that are passed `#unit` if they are chosen for execution.

```
1  syntax if:Statement =
2    "if" cond:Expression
3    "then" then:Block
4    elseif:("elseif" econd:Expression
5            "then" ethen:Block)*
6    ("else" else:Block)? "end"
7  {
8    #scope (then)
9    #scope elseif:(ethen)
10   #scope (else)
11
12   BExpression`
13     efun return. efun break. #if (#deref `t(cond))
14       (fun _. `t(then) return break)
15       (fun _. `t(
16         foldr elseif next
17           (BExpression` #if (#deref `t(econd))
18             (fun _. `t(ethen) return break)
19             (fun _. `t(next)))
20         foldr else _
21           (BExpression` `t(else) return break)
```

```
22          (BExpression` #unit)))
23  }
```

Expansion checking produces several errors, here slightly abbreviated and clarified, and further explained and summarized later:

1. **syntax**#[1] needs before exports from **syntax**#[4,0,1].
2. **syntax**#[1] needs before exports from **syntax**#[4,1,1].
3. **syntax**#[3] needs before exports from **syntax**#[4,0,1].
4. **syntax**#[3] needs before exports from **syntax**#[4,1,1].
5. **syntax**#[4,0,1] needs before exports from **syntax**#[4,1,1].
6. **syntax**#[4,0,3] needs before exports from **syntax**#[4,1,1].
7. **syntax**#[4,0,1] before exports should be exported.
8. **syntax**#[4,0,1] after exports should be exported.
9. **syntax**#[4,1,1] before exports should be exported.
10. **syntax**#[4,1,1] after exports should be exported.

The **syntax**#[<...>] expressions refer to elements in the syntax tree using the path concept introduced in Chapter 4. Note, however, that the paths in the implementation are zero-indexed, while the formalization refers to them as one-indexed. For example, **syntax**#[4,0,1] refers to the first repetition of econd (on line 4). We can see this by noting that the element with index 4 in the syntax description is elseif:(<...>)*, whose child with index 0 is elseif:(<...>) (note the absence of *), whose child with index 1 is econd:**Expression**. Similarly, **syntax**#[4,1,1] also refers to econd, but in the second repetition of elseif.

Using this information we can summarize the errors to the following list:

1. cond needs before exports from econd. (1, 2)
2. then needs before exports from econd. (3, 4)

3. `econd` needs `before` exports from later repetitions of `econd`. (5)
4. `ethen` needs `before` exports from later repetitions of `econd`. (6)
5. Both `before` and `after` exports from `econd` need to be re-exported. (7-10)

The errors arise because the expansion puts repetitions of `econd` inside functions, later repetitions nested further in, which restricts their exports, hiding them from most preceding elements as well as the surrounding syntax tree.

To solve this we wrap the conditions in their own scope and restrict their exports before expansion; we most likely do not intend for variables bound in the condition to be visible outside of the if-statement.

The errors presented here are perhaps not particularly nice to read or understand, but they contain the information needed to construct better messages, it ”just” requires some more engineering and user experience effort.

5.6.3 Binding Errors

The errors presented by the implementation in the presence of binding errors add nothing new beyond what is common in most compilers. The following errors are reported:

Undefined symbol: If an identifier is encountered in a non-binding position and no binding with the same symbol is in scope this error is reported, along with the source code location of the unresolved identifier.

Symbol already defined: Bindings may shadow bindings from outer scopes, but if a symbol is defined twice in the same scope, and the bindings cover an overlapping area of code, then this error is reported. The source code location of both definitions are included.

The overlapping requirement is important: a binding that can

be used both before and after its definition is defined by using both `#bind x before` and `#bind x after`. This binds the same symbol twice in the same scope, but since the bound areas do not overlap, no error is reported.

Since all errors are independent in the sense that none of them is the cause of any other, they can all be collected and presented to the user at once.

No binding errors can be encountered during expansion of syntax construction, nor when all syntax constructions are expanded, thus all binding errors are reported in terms of the original source code written by the end user.

Chapter 6

Limitations, Future Work, and Conclusion

This chapter summarizes the limitations discovered during evaluation, suggests potential improvements for future work, and concludes by comparing the results with the design goals introduced in Section 1.4.

6.1 Limitations and Future Work

The limitations discussed in Chapter 5, and the future work they each suggest, can be grouped into the following categories:

Contextual Information: Certain programming language features require information specified by their context without a programmer using an explicit reference to that context, for example, a `break` statement must abort the innermost surrounding loop without any reference to it. Other examples of this can be found in sections 5.2.2 (`break` and `return`) and 5.2.1 (pattern matching). Syntax constructions have no concept of this, it is instead implemented using either regular functions in the core language, or expression functions, introduced in Section 5.1.1. However, as elaborated later in the same section (starting on page 86), these cannot be correctly described using syntax constructions, thus the expansion checker cannot

guarantee an expansion's correctness. Additionally, they unnecessarily couple the implementation of different syntax constructions to each other.

A better solution for future work may be to create a form of lexical bindings, where a syntax construction may set the value of such a binding for all its descendants, and a syntax construction that uses such a binding may not be used where it is unset (e.g., `break` may only be used inside a loop, which would set such a binding).

Ambiguity Control: Syntax constructions allow for ambiguity in the syntax of a programming language where desired, but provides poor control over it. The language designer gets no help in ensuring its presence or absence but must instead test for it themselves. Additionally, when ambiguity is present but undesired the tools to remove it are frequently not very convenient, page 103 for example contains an unintentional ambiguity with a slightly inconvenient solution, while page 96 contains an unintentional ambiguity with a significantly inconvenient solution.

Future work must here come in two different, but related flavors:

- Control of where ambiguity appears. This requires static analysis of the grammar, which in the general case is is undecidable for a context-free grammar [2]. The question then becomes if a heuristic is sufficient, or if some more restricted form of grammar can support the desired workflow of defining the syntax of each syntax construction separately.
- Tools to remove or add ambiguity. Precedence as a number easily turns unwieldy, and turns out to be insufficiently flexible (e.g., see p. 96). Disallowing specific syntax constructions from appearing in certain positions would solve this particular problem (the expressions in a list literal cannot be sequential composition), and may prove to be a more flexible tool.

Separated Symbol Domains: All bound symbols are treated the

same by syntax constructions, but many languages partition them into groups, a common example being the separation of types from values: names bound as types are fully separated from names bound as values.

Disambiguation by binding or reference: Some programming languages have features that behave differently depending on whether an identifier is previously bound or not. Section 5.2.3 discusses this in more detail, but the most natural way to express this with syntax constructions is to define two syntax constructions, one for when the identifier is bound and one for when it is not, but otherwise syntactically identical, which the system as it stands cannot handle unambiguously.

Namespaces: Though explicitly ignored in this thesis (see delimitations in Section 1.4.1), namespaces and the ability to import definitions from them are important parts of many programming languages and should play a part in future work.

Performance: The current implementation has an inefficient expansion phase. However, as discussed in Section 5.5, we believe that a very large portion of the work performed is unnecessary; only the syntax constructions that duplicate a subtree might require an extra name resolution pass after expansion. Future work could thus likely greatly improve performance in the common case by tracking potential duplication, and omitting the name resolution pass when possible.

6.2 Conclusion

This thesis has presented a first attempt at a method for defining programming languages through individual features, allowing reuse and composition of much smaller parts than entire languages.

Evaluating based on the design goals stated after the initial problem statement (see Section 1.4) we find the following:

Tower of languages: A syntax construction defines its implementation in terms of some other language, using the

syntax of that language. However, the inability to treat subexpressions in a non-atomic way somewhat limit their expressive power compared to the macros commonly found in Lisp dialects.

Syntactical freedom: A language can express essentially any context-free grammar, thus the syntactical freedom is far greater than for example Lisp or C. However, the fixed tokenization precludes customization of for example comments or identifiers, and the rewrites required to avoid undesired ambiguity are sometimes significant.

Abstraction preservation: The expansion checker has not allowed any abstraction breaking expansions in practice, and the justification presented in the formalization appears sound.

Good error messages: The error messages presented are informative and specific, though more engineering work is required to make them truly palatable.

Composition through cherry-picking: Though technically doable, certain common situations make the resulting language ambiguous, requiring creating a new syntax construction simply to wrap the imported one. This boilerplate seems undesirable, but a better solution is left for future work.

Reasoning without context: The `#precedence` declaration present in some syntax constructions present an implicit dependency on all other syntax constructions with a declared precedence. Additionally, the rewrites mentioned earlier require changing or creating several syntax constructions in tandem, i.e., they must be considered in their context. A majority of the syntax constructions defined could however be considered without context.

Syntax constructions thus do not fully meet the goals initially presented, but they do provide a reasonable first step.

Bibliography

[1] Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. "Paradise: A Two-stage DSL Embedded in Haskell". In: *SIGPLAN Not.* 43.9 (Sept. 2008), pp. 225–228. ISSN: 0362-1340.

[2] David G. Cantor. "On The Ambiguity Problem of Backus Systems". In: *J. ACM* 9.4 (Oct. 1962), pp. 477–479. ISSN: 0004-5411.

[3] Jay Earley. "An Efficient Context-free Parsing Algorithm". In: *Commun. ACM* 13.2 (Feb. 1970), pp. 94–102. ISSN: 0001-0782.

[4] Sebastian Erdweg et al. "Layout-sensitive generalized parsing". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7745 LNCS (2013), pp. 244–263. ISSN: 03029743.

[5] Matthew Flatt. "Binding As Sets of Scopes". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: ACM, 2016, pp. 705–717. ISBN: 978-1-4503-3549-2.

[6] Matthew Flatt and PLT. *Reference: Racket*. Tech. rep. PLT-TR-2010-1. PLT Design Inc., 2010.

[7] MATTHEW FLATT et al. "Macros that Work Together: Compile-time bindings, partial expansion, and definition contexts". In: *Journal of Functional Programming* 22.2 (2012), pp. 181–216.

[8] J. Heering et al. "The syntax definition formalism SDF—reference manual—". In: *ACM SIGPLAN Notices* 24.11 (Nov. 1989), pp. 43–75. ISSN: 03621340.

[9] David Herman and Mitchell Wand. "A Theory of Typed Hygienic Macros". In: *Proceedings of the 17th European Symposium on Programming* 4960 (2010), p. 48. ISSN: 03029743.

[10] Ted Kaminski. “Reliably composable language extensions”. In: (2017).

[11] Florian Lorenzen and Sebastian Erdweg. “Sound type-dependent syntactic language extension”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*. Vol. 51. 1. New York, New York, USA: ACM Press, 2016, pp. 204–216. ISBN: 9781450335492.

[12] Erik Silkensen and Jeremy Siek. *Well-Typed Islands Parse Faster*. Ed. by Hans-Wolfgang Loidl and Ricardo Peña. Springer, Berlin, Heidelberg, 2013, pp. 69–84. ISBN: 978-3-642-40447-4.

[13] Paul Stansifer and Mitchell Wand. “Romeo”. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming - ICFP ’14*. Vol. 49. 9. New York, New York, USA: ACM Press, 2014, pp. 53–65. ISBN: 9781450328739.

[14] S. Tobin-Hochstadt. “Extensible Pattern Matching in an Extensible Language”. In: *ArXiv e-prints* (June 2011). arXiv: 1106.2578 [cs.PL].

[15] Sam Tobin-Hochstadt et al. “Languages As Libraries”. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA: ACM, 2011, pp. 132–141. ISBN: 978-1-4503-0663-8.

[16] Eric R. Van Wyk and August C. Schwerdfeger. “Context-aware scanning for parsing extensible languages”. In: *Proceedings of the 6th international conference on Generative programming and component engineering - GPCE ’07*. New York, New York, USA: ACM Press, 2007, p. 63. ISBN: 9781595938558.

[17] Eric Van Wyk et al. “Silver: An extensible attribute grammar system”. In: *Science of Computer Programming* 75.1-2 (Jan. 2010), pp. 39–54. ISSN: 01676423.

Appendix A

Language Definition: Formalization Core

```
syntax type File = builtin
syntax type Core = builtin
syntax coretop:File = e:Core
{ builtin }
syntax coreafter:Core =
  "(" "bind_after" id:Identifier v:Core ")"
{
  #bind id after
  builtin
}
syntax coresum:Core =
  "(" "sum" a:Core b:Core ")"
{ builtin }
syntax coreseq:Core =
  "(" "seq" a:Core b:Core ")"
{ builtin }
syntax corevar:Core = id:Identifier
{ builtin }
syntax corelit:Core = i:Integer
{ builtin }
```

Appendix B

Language Definition: OCaml Subset

```
syntax type Top = BExpression

syntax top:File = ts:Top+ {
  File` #unit; `t(
    foldl1 ts prev (BExpression` `t(prev); `t(ts)))
}

// ERROR: if args[1] has #bind before and args[0]
// uses it then args[0] will work before expansion,
// but not after. Same for letExpr
syntax topLet:Top =
  "let" x:Identifier args:Pattern*
  "=" e:Expression (";" ";")?
{
  #bind x after
  #scope (args (e))
  BExpression` defAfter `id(x) = `t(
    foldr args next
      (BExpression` fun x.
        (`t(args) (fun _. #crash) x;
         `t(next)))
    e)
}
```

```
syntax topLetRec:Top =
  "let" "rec" x:Identifier args:Pattern*
  "=" e:Expression
  more:("and" x2:Identifier args2:Pattern*
        "=" e2:Expression)* (";" ";")?
{
  #bind x, x2 after
  #bind x, x2 in e, e2
  #scope (args (e))
  #scope more:(args2 (e2))
  BExpression`
  defAround `id(x) = #fix (fun `id(x).
    `t(foldr args next
         (BExpression` fun x.
           (`t(args) (fun _. #crash) x;
            `t(next)))
       e));
  `t(foldl more prevdef
       (BExpression` `t(prevdef);
        defAround `id(x2) = #fix (fun `id(x2).
          `t(foldr args2 next
               (BExpression` fun x.
                 (`t(args2) (fun _. #crash) x;
                  `t(next)))
            e)))
     (BExpression` #unit))
}

syntax type Expression = BExpression

syntax debugPrint:Expression = "#" "debug" {
  BExpression` #print
}

syntax funApp:Expression =
  f:Expression arg:Expression
{
  #assoc left
```

```
  #prec 13
  BExpression` `t(f) `t(arg)
}

syntax letExpr:Expression =
  "let" x:Identifier args:Pattern*
  "=" e:Expression "in" body:Expression
{
  #prec 2
  #bind x in body
  #scope (body)
  #scope (args (e))
  BExpression` defAfter `id(x) = `t(
    foldr args next
      (BExpression` fun x.
        (`t(args) (fun _. #crash) x;
         `t(next)))
    e);
  `t(body)
}

syntax letRecExpr:Expression =
  "let" "rec" x:Identifier args:Pattern*
  "=" e:Expression "in" body:Expression
{
  #prec 2
  #bind x in e, body
  #scope (body)
  #scope (args (e))
  BExpression` defAfter `id(x) = #fix (fun `id(x).
    `t(foldr args next
         (BExpression` fun x.
           (`t(args) (fun _. #crash) x;
            `t(next)))
       e));
    `t(body)
}

syntax match:Expression =
```

```
  "match" e:Expression "with"
  arm:("|" p:Pattern "->" body:Expression)+
{
  #scope arm:(p body)
  #scope (e)
  #prec 2
  BExpression` #callcc (fun end. (fun val.
    `t(foldr arm prev
         (BExpression`
           #callcc (fun next.
             (`t(p) next val;
              end `t(body)));
           `t(prev))
       (BExpression` #crash)))
    `t(e))
}

syntax plus:Expression =
  a:Expression "+" b:Expression
{
  #prec 11
  #assoc left
  BExpression` #plus `t(a) `t(b)
}

syntax minus:Expression =
  a:Expression "-" b:Expression
{
  #prec 11
  #assoc left
  BExpression` #minus `t(a) `t(b)
}

syntax times:Expression =
  a:Expression "*" b:Expression
{
  #assoc left
  #prec 12
  BExpression` #multiply `t(a) `t(b)
```

```
}

syntax equal:Expression =
  a:Expression "=" b:Expression
{
  #assoc left
  #prec 10
  BExpression` #equal `t(a) `t(b)
}

syntax leq:Expression =
  a:Expression "<=" b:Expression
{
  #assoc left
  #prec 10
  BExpression` #leq `t(a) `t(b)
}

syntax parens:Expression = "(" e:Expression ")" {
  e
}

syntax cons:Expression =
  h:Expression "::" t:Expression
{
  #assoc right
  #prec 10
  BExpression` #cons `t(h) `t(t)
}

syntax seqComp:Expression =
  a:Expression ";" b:Expression
{
  #assoc right
  #prec 2
  BExpression` `t(a); `t(b)
}

syntax var:Expression = v:Identifier
```

```
{ BExpression``id(v) }
syntax unitLit:Expression = "(" ")"
{ BExpression` #unit }
syntax trueLit:Expression = "true"
{ BExpression` #true }
syntax falseLit:Expression = "false"
{ BExpression` #false }
syntax IntLit:Expression = n:Integer
{ BExpression``int(n) }
syntax StringLit:Expression = s:String
{ BExpression``str(s) }
syntax listLit:Expression =
  "[" (e:Expression (";" es:Expression)*)? "]"
{
  foldr e next
    (BExpression` #cons `t(e) `t(next))
  foldr es next
    (BExpression` #cons `t(es) `t(next))
  (BExpression` #nil)
}
syntax funLit:Expression =
  "fun" args:Pattern+ "->"
  body:Expression
{
  #prec 1
  #scope (args (body))
  foldr args next
    (BExpression` fun x.
      (`t(args) (fun _. #crash) x;
       `t(next)))
  body
}

syntax type Pattern = BExpression

syntax intPat:Pattern = i:Integer {
  BExpression` efun fail. efun val.
    #if (#equal `int(i) val)
      (fun _. #unit)
```

```
      (fun _. fail #unit)
}
syntax stringPat:Pattern = s:String {
  BExpression` efun fail. efun val.
    #if (#equal `str(s) val)
      (fun _. #unit)
      (fun _. fail #unit)
}
syntax truePat:Pattern = "true" {
  BExpression` efun fail. efun val.
    #if (#equal #true val)
      (fun _. #unit)
      (fun _. fail #unit)
}
syntax falsePat:Pattern = "false" {
  BExpression` efun fail. efun val.
    #if (#equal #false val)
      (fun _. #unit)
      (fun _. fail #unit)
}
syntax bindPat:Pattern = id:Identifier {
  #bind id after
  BExpression` efun _. efun val.
    defAfter `id(id) = val
}
syntax wildcardPat:Pattern = "_" {
  BExpression` efun _. efun _. #unit
}
syntax listPat:Pattern =
  "[" (p:Pattern (";" ps:Pattern)*)? "]"
{
  BExpression` efun fail. efun val.
    `t(foldr p next
         (BExpression` efun prev.
           (#if (#equal prev #nil)
              (fun _. fail #unit)
              (fun _. #unit);
            `t(p) fail (#head prev);
            `t(next) (#tail prev)))
```

```
        foldr ps next
          (BExpression` efun prev.
            (#if (#equal prev #nil)
               (fun _. fail #unit)
               (fun _. #unit);
             `t(ps) fail (#head prev);
             `t(next) (#tail prev)))
        (BExpression` (efun prev.
          #if (#equal prev #nil)
            (fun _. #unit)
            (fun _. fail #unit))))
    val
}
syntax consPat:Pattern =
  h:Pattern "::" t:Pattern
{
  #assoc right
  #prec 10
  BExpression` efun fail. efun value.
    (#if (#equal value #nil)
       (fun _. fail #unit)
       (fun _. #unit);
     defAfter headValue = #head value;
     `t(h) fail headValue;
     defAfter tailValue = #tail value;
     `t(t) fail tailValue)
}
```

Appendix C

Language Definition: Lua Subset

```
syntax top:File = body:Block {
  #scope (body)
  File` #callcc (fun return.
    `t(body) return (fun _. #crash))
}

syntax type Expression = BExpression

syntax var:Expression = id:Identifier {
  BExpression` `id(id)
}

syntax intLit:Expression = i:Integer {
  BExpression` `int(i)
}

syntax strLit:Expression = s:String {
  BExpression` `str(s)
}

syntax funcCall:Expression =
  f:Expression "(" args:ArgList ")"
{
```

```
  #prec 15
  #scope (args)
  #scope (f)
  BExpression` `t(args) `t(f) #unit
}

syntax type ArgList = BExpression

syntax argList:ArgList =
  (e:Expression ("," es:Expression)*)?
{
  BExpression` efun f. `t(
    foldl es prev
      (BExpression`
        (#deref `t(prev)) (#deref `t(es)))
    foldl e prev
      (BExpression`
        (#deref `t(prev)) (#deref `t(e)))
    (BExpression` f))
}

syntax debugPrint:Expression = "#" "debug" {
  BExpression` fun x. fun _. #print x
}

syntax true:Expression = "true" {
  BExpression` #true
}

syntax false:Expression = "false" {
  BExpression` #false
}

syntax equal:Expression =
  a:Expression "==" b:Expression
{
  #prec 8
  BExpression`
    #equal (#deref `t(a)) (#deref `t(b))
```

```
}

syntax leq:Expression =
  a:Expression "<=" b:Expression
{
  #prec 8
  BExpression`
    #leq (#deref `t(a)) (#deref `t(b))
}

syntax and:Expression =
  a:Expression "and" b:Expression
{
  #prec 7
  BExpression`
    #and (#deref `t(a)) (#deref `t(b))
}

syntax plus:Expression =
  a:Expression "+" b:Expression
{
  #prec 11
  #assoc left
  BExpression`
    #plus (#deref `t(a)) (#deref `t(b))
}

syntax minus:Expression =
  a:Expression "-" b:Expression
{
  #prec 11
  #assoc left
  BExpression`
    #minus (#deref `t(a)) (#deref `t(b))
}

syntax times:Expression =
  a:Expression "*" b:Expression
{
```

```
  #assoc left
  #prec 12
  BExpression`
    #multiply (#deref `t(a)) (#deref `t(b))
}

syntax divide:Expression =
  a:Expression "/" b:Expression
{
  #assoc left
  #prec 12
  BExpression`
    #divide (#deref `t(a)) (#deref `t(b))
}

syntax parens:Expression = "(" e:Expression ")"
{ e }

syntax type Statement = BExpression

syntax funcCallStmnt:Statement =
  f:Expression "(" args:ArgList ")"
{
  #scope (f)
  #scope (args)
  BExpression`
    efun _. efun _. `t(args) `t(f) #unit
}

syntax assignment:Statement =
  ref:Expression "=" value:Expression
{
  BExpression` efun _. efun _.
    #assign `t(ref) (#deref `t(value))
}

syntax local:Statement =
  "local" x:Identifier ("=" e:Expression)?
{
```

```
  #bind x after
  #scope (e)
  BExpression` efun _. efun _.
    defAfter `id(x) = #ref
      (#deref `t(foldr e _ (e)
                  (BExpression` #unit)))
}

syntax block:Statement = "do" b:Block "end"
{ b }

syntax if:Statement =
  "if" cond:Expression
  "then" then:Block
  elseif:("elseif" econd:Expression
          "then" ethen:Block)*
  ("else" else:Block)? "end"
{
  #scope (cond (then))
  #scope elseif:(econd (ethen))
  #scope (else)

  BExpression`
    efun return. efun break. #if (#deref `t(cond))
      (fun _. `t(then) return break)
      (fun _. `t(
        foldr elseif next
          (BExpression` #if (#deref `t(econd))
            (fun _. `t(ethen) return break)
            (fun _. `t(next)))
        foldr else _
          (BExpression` `t(else) return break)
        (BExpression` #unit)))
}

syntax while:Statement =
  "while" cond:Expression "do"
  body:Block "end"
{
```

```
  #scope (cond (body))
  BExpression`
    efun return. efun _. #callcc (fun break.
      #fix (fun repeat. fun _.
        #if (#deref `t(cond))
          (fun _.
            (`t(body) return break;
             repeat #unit))
          (fun _. #unit)) #unit)
}

syntax repeat:Statement =
  "repeat" body:Block "until" cond:Expression
{
  #scope ((body) cond)
  BExpression`
    efun return. efun _. #callcc (fun break.
      #fix (fun repeat. fun _. (
        `t(body) return break;
        #if (#deref `t(cond))
          (fun _. #unit)
          (fun _. repeat #unit))) #unit)
}

syntax numFor:Statement =
  "for" i:Identifier "="
  s:Expression "," e:Expression
  ("," step:Expression)?
  "do" body:Block "end"
{
  #bind i in body
  #scope (s)
  #scope (e)
  #scope (step)
  #scope (body)
  BExpression` efun return. efun _.
    (defAfter step = #deref
      `t(foldr step _ (step) (BExpression` 1));
     defAfter end = #deref `t(e);
```

```
    #callcc (fun break. #fix (fun rep. fun x.
      (defAfter `id(i) = x;
       #if (#leq x end)
         (fun _. (`t(body) return break;
                  rep (#plus x step)))
         (fun _. #unit)))
      `t(s)))
}

syntax function:Statement =
  "function" f:Identifier
  "(" (a:Identifier ("," as:Identifier)*)? ")"
  body:Block "end"
{
  #bind f before
  #bind f after
  #bind f, a, as in body
  #scope (body)
  BExpression` efun _. efun _.
    (defAround `id(f) = #fix (fun recur.
      (defAfter `id(f) = recur;
       `t(foldr a next
            (BExpression` fun arg.
              (defAfter `id(a) = (#ref arg);
               `t(next)))
          foldr as next
            (BExpression` fun arg.
              (defAfter `id(as) = (#ref arg);
               `t(next)))
          (BExpression`
            fun _. #callcc (fun return.
              `t(body) return
                       (fun _. #crash)))))))
}

syntax type Block = BExpression

syntax blockContent:Block =
  (stmnts:Statement ";"?)* ret:Return?
```

```
{
  #scope (stmnts ret)
  BExpression` efun return. efun break. `t(
    foldr stmnts next
      (BExpression`
        `t(stmnts) return break; `t(next))
    foldr ret _
      (BExpression`
        `t(ret) return break)
    (BExpression` #unit))
}

syntax type Return = BExpression

syntax return:Return = "return" e:Expression? {
  BExpression` efun return. efun break. `t(
    foldr e _
      (BExpression` return (#deref `t(e)))
    (BExpression` return #unit))
}

syntax break:Return = "break" {
  BExpression`
    efun return. efun break. break #unit
}
```