An Interpreter for a Novice-Oriented Programming Language with Runtime Macros

by

Jeremy Daniel Kaplan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author . . . .
Department of Electrical Engineering and Computer Science
May 12, 2017

Certified by . . . .
Adam Hartz
Lecturer
Thesis Supervisor

Accepted by . . . .
Christopher Terman
Chairman, Masters of Engineering Thesis Committee

An Interpreter for a Novice-Oriented Programming Language with Runtime Macros

by

Jeremy Daniel Kaplan

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, we present the design and implementation of a new novice-oriented programming language with automatically hygienic runtime macros, as well as an interpreter framework for creating such languages. The language is intended to be used as a pedagogical tool for introducing basic programming concepts to introductory programming students. We designed it to have a simple notional machine and to be similar to other modern languages in order to ease a student's transition into other programming languages.

Thesis Supervisor: Adam Hartz
Title: Lecturer

Acknowledgements

I would like to thank every student and staff member of MIT's 6.01 that I have worked with in the years I was involved in it. I learned a lot about programming, teaching, and the way people think about programming. Those experiences and conversations helped motivate this project, and I would not have turned out nearly the same way otherwise.

In particular, I would like to thank Adam Hartz, who has been an amazing mentor throughout my career at MIT, always willing to discuss pedagogy, programming, and life in general.

I would also like to thank Kalin Malouf, Rodrigo Toste Gomes, Jeremy Wright, Kade Phillips, Geronimo Mirano, and Katy Kem for their willingess to listen to my ideas about teaching, programming, and teaching programming, in addition to being a wonderful support network.

Finally, I would like to thank my family for their unconditional love and support. I would not be anywhere near where I am now without them.

Contents

List of Figures

Chapter 1

Introduction

Our goal for this project was to design a programming language that can be used as a pedagogical tool for teaching introductory programming students. Our primary focus was in making a language with a simple notional machine [1] (for novice programmers) and powerful extensibility features (for experienced programmers). The language is aimed at late high school or early undergraduate students.

Some general-purpose languages are also used as introductory languages (Python, Java, and C++) [2]. These languages are powerful, feature-filled, and contain many shortcuts for common tasks, which are all benefits that come at the expense of complicated syntax or notional machines, so subsets of these languages are usually taught in their place. Students often search the web for help in completing assignments, but the world outside the teaching environment does not limit itself to the same subset of the language, so students may not be able to grasp the answers they find online, for example. Or, worse, the answer to their question may end up being a simple call to a standard library function, which trivializes the entire assignment!

Other languages languages explicitly designed for novices include Logo [3], PLT Scheme (now known as Racket) [4], and Scratch [5]. These languages are also powerful and feature-filled, but they bear little resemblance to other languages used in introductory programming courses. These languages may be successful in being easy for novices to learn, but students may struggle with the transition to a later programming course that uses different language. Within the same programming language family, some concepts can be translated with simple syntax transformations or notional machine analogies, but some of these teaching languages implement different paradigms entirely, so knowledge of one of these languages may not transfer well to another.

Our approach was to start with a very basic language (in features and syntax) by default and allow the user to extend the syntax at runtime with automatically hygienic macros, which cannot accidentally capture identifiers in the larger program. Macros are generally seen as an advanced programming concept, but part of our extensibility goal was to be able to introduce the concept of a runtime macro in a programmer's first language to enable them to make useful syntactic abstractions. We wanted students who learned this language to be able to easily move into other common languages, so the syntax of the base language is similar to Python or Ruby, and the extensibility features of the interpreter allow the language to borrow syntax and features from other languages.

In this thesis, we will start by describing the base language syntax and features (Chapter 2). We then describe the interpreter framework: a custom parser generator (Chapter 3), the language grammar itself (Chapter 4), and the evaluator (Chapter 5). We conclude with some commentary of the process and future work (Chapter 6).

Chapter 2

Base Language

In this chapter, we will describe the features of base language in our interpreter framework. The features were chosen to minimize complexity when presented to a novice programmer without excessively limiting the capabilities of the language. Many of the choices were made following the introductory language design principles set out by McIver and Conway in [6]. We wanted a minimal set of orthogonal features that worked the way a non-programmer would expect. We also wanted some features that would allow novices to move to other languages with few changes to their models of how these features worked.

In the first section, we will list the primitive features we chose to include in the base language. Next, we discuss the features we explicitly omitted. Finally, we describe the extensions to the language in the form of runtime macros and startup settings.

2.1 Primitives

The primitives in the base language include basic value types, the unordered collection type, functions, and control flow mechanisms. In this section, we describe our implementation and rationale for each type.

2.1.1 Data Types

The data types included in our base language are numbers, strings, booleans, the null value, functions, and unordered collections.

Numbers The base language implements numbers as arbitrary-precision rationals. We wanted to avoid teaching beginners about numeric representations (for example, two's-complement arithmetic or IEE754 floating-point numbers) because students' prior knowledge of basic arithmetic should transfer into programming. With rationals as the only numeric type, some arithmetic operations, such as square roots and logarithms, must either be approximated or represented symbolically. We considered providing access to a symbolic algebra system like SymPy [7], but we chose against it in order to avoid complicating the notional machine with "exact" and "inexact" versions of different calculations. Instead, we approximate any calculation involving irrationals as a rational number.

Strings The base language implements strings as immutable sequences of characters. We wanted to avoid problems with null-termination, mutability, and character encoding in our string implementation, so we implemented strings as their own data type.

Booleans The base language implements booleans as the singleton values `True` and `False`. We wanted to avoid the question of truthiness, so we implemented booleans as their own literal type instead of mapping them onto 0 and 1 or introducing the notion of truthy and falsy values.

Null The base language implements the null type as the singleton value `Nothing`. This is equivalent to Python's `None`, Ruby's `nil`, or JavaScript's `null`. There is no equivalent to JavaScript's `undefined` type.

Functions The base language provides a general mechanism for creating anonymous functions but not for creating named functions. Many languages conflate the concepts of defining a function and giving it a name, so we implemented only anonymous functions and allow them to be assigned to variables in the environment in the same way that all other values are. Functions must take a constant number of arguments, so they cannot have optional arguments or variadic arguments. Instead of using an explicit `return` keyword, functions return the value of the last statement that was executed in the body of the function.

Unordered Collections The base language implements collections as immutable key-value stores. We wanted the keys and values to be of any type, including unordered collections themselves. Using a mutable object as a key in a collection introduces the potential for key collisions if that object changes. We avoided the issues of unexpected key changes by making these collections (and all other primitives data types) immutable.

2.1.2 Control Flow

The base language includes two forms of control flow: conditional execution and indefinite loops. Both of these forms are expressions rather than statements so that they can be used as the final expression of a function body.

Conditional Execution We chose to implement conditional execution in its ternary form: a condition, a consequent, and an alternative. The consequent is executed in the case that the condition is satisfied; otherwise, the alternative is executed. A conditional expression evaluates to the value of the executed branch. We considered this form of conditional to be simpler than other forms, such as a pattern-matching construct (such as `match` in Scala) or a construct with multiple conditions (such as C-style `switch` or Lisp's `cond`).

Indefinite Loops The looping construct we chose to implement was a loop that repeatedly executes its body undil a condition is no longer satisfied usually referred to as a "while loop". We felt that this form most closely matched the indefinite loop constructs in other languages. In our loop structure, the condition is executed first, then the the body is executed if the condition is satisfied, and this process continues in a loop. The indefinite loop is an expression that evaluates to the value of the last execution of the body.

2.2 Omissions

We intentionally omitted some common programming language features from the base language. Notably missing from the list of features in the previous section are variable assignment and declaration, ordered collections, definite loops, and an object system. We chose not to implement these features because their syntax and semantics vary across languages. As part of our extensibility goals, we wanted to give instructors the freedom to include these features (and any others) at their discretion.

Assignment and Declaration The first problem with implementing assignment and declaration was the choice of syntax. For conformity with most programming languages, we would provide syntax that assigns the name `x` with the value `10` using `x = 10`. However, using the equals symbol for assignment is confusing to novices [8], so some languages use `x <- 10`, `x := 10`, or some other syntax. We encountered our second problem when deciding whether to include declaration as a separate syntax. In Python, for example, there is no separate declaration syntax, and a variable binding is created during assignment if a binding does not already exist. If a separate declaration syntax does exist, there is also the question of whether to make a distinction between constant and variable bindings. In JavaScript, as another example, declaring a variable with `let` or `const` makes the binding variable or constant, respectively. Rather than include any default syntax or semantics, we chose to implement the interpreter with enough flexibility to let an instructor decide what to do.

Ordered Collections Most languages use the same syntax for accessing the elements of ordered and unordered collections. This requires that novices associate the same syntax with multiple ideas (index of a list and key of a dictionary, for example), which can be confusing to a novice. Extending the language to include something that acts like an ordered collection is possible using macros, which gives the user the freedom to make choices such as whether index numbers should start at 0 or 1.

Definite Loops Almost all languages with looping constructs include both the definite and indefinite loop, although the forms of their definite loop differ. Some languages have a definite loop in which an iterator variable takes on values in some iterable (Python, Ruby), while others break the loop into a setup, condition, and update (C, Java, JavaScript). We saw that the indefinite loop could be used to implement any of these definite loops, so we designed the interpreter to allow a user to implement this through the extensibility mechanisms.

Objects Object-oriented programming can be a confusing way to introduce novices to programming concepts. Students learning object-oriented programming are forced to contend with method resolution and instance state in addition to understanding functions, variables, loops, and other constructs. Objects are useful abstractions for experienced programmers, but they complicate the notional machine, and the syntax and semantics of object systems differ across languages. For these reasons, we chose not to implement an object system in the base language. However, the extensibility features of the interpreter make it possible for a user to implement an object system of their own design.

2.3 Extensions

The power of this language and interpreter framework comes from the extensibility features. The user can modify the grammar of the language using runtime macros. In addition, the instructor can modify the interpreter through start-time settings files and through modifications to the interpreter itself.

2.3.1 Macros

Our goal for macros in this language was to allow users to extend the syntax of the language at runtime in order to add syntactic primitives they find useful or avoid repetition of common program structures in ways that functions were unable to. We wanted to make this process accessible to an introductory-level programmer, so we wanted to avoid requiring users to traverse ASTs or modify program text directly when implementing macros. We considered three existing macro systems: C preprocessor macros, Lisp runtime macros, and Rust compile-time macros.

The C preprocessor provides two types of macros: object-like and function-like. All macros are simultaneously expanded at compile time, and each macro is defined with a name (all uppercase by convention) that indicates to the compiler that macro expansion should occur. Object-like macros (such as `#define PI 3.14`) take no parameters and are generally used for constants. Function-like macros can take parameters (such as `#define ADD(x, y) x + y`), that are expanded in a call-by-name fashion. The text substitution causes problems with operator precedence (`a * ADD(b, c)` becomes `a * b + c`), which is usually solved by adding more parentheses in the expanded text. There are other problems with expanding complex statements in this way, although users of these macros have developed conventions for avoiding these errors[1].

Lisp macros have much more flexibility than C macros, and the homogeneous syntax of Lisp allows them to avoid the issue with operator precedence, as well as other issues faced by the C preprocessor. Lisp macros are expanded at runtime, and they are defined as an s-expression prefix and a transformation function from the source expression to a target expression, usually manipulating the source list directly. Because Lisp syntax is entirely made of lists, the function creates a list containing the symbols and values in the target expression. Different Lisp dialects provide different mechanisms for defining macros, but these are usually convenience functions around the same basic idea[2]. Common Lisp has `defmacro` and Scheme has

[1] `https://stackoverflow.com/questions/1067226/`

[2] `https://docs.racket-lang.org/guide/pattern-macros.html`

`define-syntax-rule`, for example.

Rust macros are similar to C macros in that they are expanded at compile-time, but they avoid some of their pitfalls (operator precedence and variable shadowing, for example)[3]. Rust macros are defined with a name, a syntax pattern, and a description of the replacement syntax. Metavariables captured in the syntax pattern are tagged with what kind of syntax they capture (expression, identifier, type, etc.). To use repetition, the macro is defined with special operators in the pattern (`$($x:expr),*`), and the same operator is used in the replacement code (`$(println!($($x),*))`). A macro defined this way would capture any number of comma-separated expressions, and use them as arguments (comma-separated) to the `println!` function.

In each of these languages, macro usage appears exactly like a function call (except for object-like C macros, which appear exactly like a variable). Without careful attention to macro name conventions, it can be difficult to know ahead of time whether that "function" will be a normal function or a macro.

Our design for the macro system has the following goals:

1. The syntax of macros should not be the same as function calls.
2. Macro expansion should perform substitutions on the syntax tree instead of on the text.
3. Users should not have to worry about name collisions between macros and the larger program.
4. Defining a macro should require minimal knowledge of syntax trees.
5. A user should be able to define a macro anywhere in the program.
6. The macro expansions should happen at runtime.

We used the pattern-matching idea from Rust in the form of a mini-language for defining captured syntax patterns, but we omitted the metavariable syntax-type annotations in favor of always capturing an entire expression or block. We allow the

[3]`https://doc.rust-lang.org/book/macros.html`

user to decide when to evaluate any captured code with a Lisp-style unquote operator. This mini-language is described in detail in Section 4.2.

2.3.2 Startup Settings

Some modifications to the language would be difficult or impractical to implement through macros. For example, the standard mathematical operators would be difficult to implement as runtime macros due to the necessity of operator precedence. We could have added more special syntax for defining unary and binary operators, but we found it easier to specify these language-level changes at interpreter startup. Part of our goal was to make these modifications easy to define and easy to share. For example, an instructor may want to provide some operators for one assignment and a completely different set for another (boolean operators versus set operators, for example).

Our interpreter design includes the ability to provide a settings file describing these language modifications at startup. This settings file contains the definition of any included operators and their precedence levels, some simple feature flags, and a mechanism for arbitrary changes to the grammar. The relevant fields in the settings file are described in Chapter 4 and Chapter 5.

2.4 Summary

Our goal for the base language was to implement a minimal set of teachable features, including common data structures and a function abstraction. Some examples of programs are included in Appendix B.

Chapter 3

Parser Generator

Our language and interpreter framework provide a high level of grammatical flexbility to the student using the language and to the instructor defining the interpreter. Our requirements for the parser generator were decided by our need to make programmatic changes to the parser at runtime and during interpreter startup. The runtime changes to the parser, in turn, required that our parser was able to handle parsing a program incrementally, waiting for new rules to be defined before parsing the parts of the program that used them. Our grammar also contains some rules that are not easily expressed using regular expressions, so the parser also needed to support these irregular rules.

Runtime changes to the parser are necessary to accomodate runtime macros. As we execute a program, macro definition statements can be used to define new syntax rules that will be added to the parser and be available for use immediately. Start time changes are necessary to accomodate custom operators and other startup settings.

We need more complex rules than regular expressions will allow because of how we chose to specify identifiers. Our identifiers are any number of non-whitespace characters that are not reserved words and do not contain any operator symbols. This rule, especially in the context of runtime-defined keywords, cannot be easily expressed as a regular expression.

We also need to be able to parse a prefix of the program text because we cannot depend on knowing all syntax structures before interpretation begins. One of the problems with interpreting a non-homogeneous language with runtime macros is that the rules for parsing later statements may not exist until partway through the interpretation of the program.

Based on these requirements, we evaluated parser generators based on their affordances for the kinds of grammar flexibility we needed:

1. The parser should support changes at runtime.

2. The parser should support changes at start time.

3. The parser should support parsing a prefix of the program text.

4. The parser should support rules beyond regular expressions.

3.1 Other Parser Generators

We considered four existing parser generators (PLY, PyPEG, Grako, and Arpeggio) but ultimately decided on a custom parser generator. Most parser generators work by reading in a specification of the grammar (usually in EBNF or PEG syntax) and generating code or parse tables that can be used as a parser. The rules that can be defined are usually restricted to EBNF, PEG, or regular expressions, although some generators allow for custom rules. These parsers are intended to be used to parse entire files containing programs (or other text) into one AST.

These parsers lack support for runtime changes because the generated tables are in an opaque binary format, and the generated code is difficult to extend. It is possible to work around this limitation, however, by adding new rules and re-generating the parser.

These parsers also lack support for incremental parses because they assume that there is a single rule that exists to match the entire program text. This results in parsers that produce an error if the entire file does not match the rules of the language. Without a partial parse, the only way to support macros is to require that all macros be defined before any other kind of statement executes, which turns our runtime macros into compile-time macros.

3.2 Custom Parser Generator

We designed our custom parser generator around the features described earlier: allowing changes at both start and runtime, allowing incremental parses, and allowing arbitrarily-complicated user-defined rules. Our approach was structured around the using parser combinators to define complex rules, but we retained the ability to define parser rules without using the combinators. We allow programmatic changes by defining the grammar using Python and exposing some of its internal data structures to the other parts of the interpreter. We support incremental parses by allowing the parser to emit a match without encountering an end-of-file (EOF). In the rest of this section, we describe the data structures and operation of our parser generator framework.

3.2.1 Data Structures

Our parser generator is centered around three kinds of objects: `Match` objects, `Parser` functions, and `Rule` functions. We provide a library of simple primitive rules, rule combinators, and a means of turning `Rule`s into `Parser`s.

Match Objects For our parser to support incremental parsing, we needed to know how much of the program text was matched and the location of that program fragment in the entire text. We also needed to support incremental AST creation during a parse, which we implemented as an extra field attached to a match.

This resulted in a `Match` object with the following attributes:

- `text`: the fragment of the source that was matched
- `position`: the starting position of this match in the source text
- `data`: unrestricted storage that we used for AST creation

Parser Functions A `Parser` is an infinite Python generator that takes two arguments: the source text and a start position in that text. The generator will initially emit `Match` objects representing the portions of the text that it recognizes starting at the specified position. To support backtracking in the parser, all possible `Match`es will be emitted in the order that they should be considered. When a `Parser` has exhausted the set of `Match` objects it can produce, it continues by emitting the infinite sequence of `None`s, at which point we consider the parse to have failed. Successsful parses are those that emit at least one `Match`.

Parser Rule Functions A `Rule` has exactly the same interface as a `Parser`, except that it need not be infinite. This allows us to define `Rule`s that only emit success cases, and we can turn them into proper `Parser`s by affixing the infinite sequence of `None`s to its generated `Match`es. This simplifies the grammar by allowing us to omit a boilerplate `while True:  yield None` at the end of every function in the grammar.

3.2.2 Rule Creation

Our parser generator was implemented as a library for specifying common `Rule` patterns in this framework. This library provides some functions for creating primitive rules (which deal with matching specific strings in the text) and some combinator functions (which deal with matching patterns or combinations of other rules). Each of these functions takes an optional callback function that is used to initialize the `data` field on each emitted `Match`.

Rule Primitives

We defined two primitive rules: `Raw` and `Regex`. These are used when matching simple strings or patterns in the text.

Raw A `Raw` rule emits at most one `Match` representing an exact string match at the specified position. The required argument when creating a `Raw` rule is the exact string to match. The callback function is called on the string that was matched. These rules are primarily used for keywords, operator symbols, and delimiters.

Regex A `Regex` rule emits at most one `Match` representing the results of using Python's `re.match` on a slice of the source string starting at the specified position. The required argument when creating a `Regex` rule is the regular expression to match. The callback function is called on the Python regular expression match object returned by the regular expression test. These rules are primarily used for numeric literals, string literals, and whitespace.

Rule Combinators

We defined rule combinators in order to easily specify common patterns in our grammar. These patterns include optional syntax, repeated syntax, and syntax alternatives, among others.

Optional The `Optional` combinator is used in places where certain syntax is allowed but not required, such as the alternative clause in a conditional. This combinator takes one rule as its required argument and emits either one or two `Match`es, so it always succeeds. If the provided rule parses sucessfully, the first object emitted is a `Match` representing that match. In this case, the callback function is passed the `data` from the original `Match`. Regardless of whether the rule parses sucessfully, the next `Match` emitted represents the match of the empty string, becasue the empty string is always a possible match for an `Optional` rule. In this case, the callback function is passed `None`.

Choice The `Choice` combinator is used when alternatives are allowed in syntax and exactly one of the alternatives should be used, such as having multiple kinds of numeric literals. This combinator takes a list of rules as its required argument, representing the possible options in the order that they should be tried. Each option is tried in order, and all `Match`es from the parse of one option are emitted before trying the next option. `Choice` succeeds when at least one of its options succeeds. The callback function is passed the `data` from the `Match` that is emitted.

Sequence The `Sequence` combinator is used when a there are rules should match in a specified order, such as the tokens required for an assignment statement. This combinator takes a list of rules as its required argument representing the necessary rules to match in order.

When parsing a `Sequence`, we iterate over each of the rules, starting the parse for each rule from the end of the match from the previous rule. If any rule in the sequence fails to parse, we backtrack by using the next match for the previous rule and retrying any later rules from there. The parse fails if backtracking exhausts all possibilities for the first rule, leaving no possible matches for the sequence. The parse succeeds when at least one compatible set of matches is found for every rule in the sequence. The callback function is passed a list containing the `data` from each `Match` in the sequence.

We found that we often needed whitespace in between the rules of our `Sequences`. Assuming a whitespace rule `ws` and some set of interesting rules `r1`, `r2`, and `r3`, this resulted in messy definitions like `Sequence([ws, r1, ws, r2, ws, r3])`, when we would rather define `Sequence([r1, r2, r3])` and have the whitespace match automatically. Therefore, `Sequence` takes one optional parameter called `skip_whitespace` which specifies the pattern of text to skip, defaulting to `Regex(\s*)`. If `skip_whitespace` is truthy, we try to match it before each rule in the sequence. Any matches from `skip_whitespace` are omitted when passing the list of data to the callback function. It is important to note that `skip_whitespace` must be able to match the empty string for that whitespace to be considered optional.

Star and Plus The `Star` and `Plus` combinators (named for the Kleene star and plus, respectively) are used when syntax elements can be repeated multiple times. `Star` handles the case where the syntax can be repeated zero or more times, while `Plus` requires at least one occurrence. Each of these combinators takes a single rule as its required argument representing the rule to repeatedly match. We assume that these combinators will be used to match as many repetitions as possible, so the matches that are emitted starting with the one with the greatest number of repetitions and ending with the empty match. The callback function is passed a list containing the `data` from each `Match`, in order, using the empty list for the empty match.

SeparatedStar and SeparatedPlus The `SeparatedStar` and `SeparatedPlus` combinators are used when a repeated syntax element is separated by another syntax element, such as function arguments separated by commas. `SeparatedStar` handles the case for zero or more repetitions, while `SeparatedPlus` requires at least one occurrence of the repeated element.

These combinators take two rules as their required arguments representing the main syntax to match and the separator to match. When parsing, we alternate matching the main rule and the separator rule until no more matches can be made. These matches are emitted in order from greatest to least repetitions, in the same way as `Star`. The callback function is passed a list containing the `data` from each `Match` emitted by the main rule, omitting the data for the separators by default. This combinator takes two optional arguments: `keep_separators` (in the case that the separators should be used in making the AST) and `skip_whitespace` (which operates exactly as in `Sequence`).

Usually, the separator rule matches something uninteresting, such as a comma or newline, and callback functions are written only for the main matches. However, knowledge about the separator itself may be required during a parse, such as knowing whether the separator was a comma or a semicolon within a MATLAB-style matrix literal, so setting `keep_separators` to `True` will retain the separator match data in the list passed to the callback function.

Like in `Sequence`, `skip_whitespace` is used to skip over whitespace between rules. In this case, `skip_whitespace` defaults to `None`, because whitespace is often part of the separator rule itself.

Single The `Single` combinator is included for the case where extra processing needs to be applied to the `data` of an existing rule's `Match`. Usually, that extra processing would simply be added to the callback function of the associated `Rule`, but we found no simple way to achieve that when generating macro rules (discussed in Section 5.4). The `Single` combinator takes one rule as its required argument and emits the same matches that that rule emits, but with the extra callback function applied to the `data` in that `Match`.

3.3 Summary

All the primitives and combinators in our parser generator library (except for `Single`) have equivalents in other parser generators, whether based on EBNF or PEG. However, standard EBNF or PEG parser generators are very fast. We trade off that parsing speed for more expressiveness in the kinds of grammars that can be defined. This allows us to define irregular rules, stateful parsers, and regular Python functions within our grammar.

Parser generators that output parse tables or generated code are notoriously difficult to debug. Because our a parser in our framework is just a Python program, a grammar can be debugged without the need of special tools or knowledge beyond standard Python techniques. The callback functions are a natural place to perform logging to determine parser progress or parser state.

Some usage examples are described in Appendix A. In Chapter 4, we use this parser generator to describe the syntax of our programming language.

Chapter 4

Grammar

In this chapter, we describe the implementation of a grammar for the base language using our parser generator framework. Our main goal for the language was to allow it to be introduced to the novice incrementally, with each new piece of syntax form being associated with a new programming concept. Our secondary goal was to provide the novice with tools and knowledge that would transfer to other programming languages, when possible. Toward this end, we developed the grammar with the following design philosophy:

1. Provide exactly one syntax for each data structure or control flow form

2. Avoid using the same symbols for different purposes

3. Use keywords instead of symbols to denote the start and end of blocks

4. Use conventions from existing languages where possible

In our framework, a grammar is represented by a `Grammar` object, a namespace containing syntax rules and the additional data structures they depend on. These rules and data structures are exposed to the rest of the interpreter so the interpreter can make changes to them in response to macros or interpreter settings. The rules are used to parse program text and create the AST we use to execute the program.

In Section 4.1, we describe the base language syntax rules themselves. In Section 4.3.2 we describe strategies for modifying and extending the the base grammar to accommodate macros and custom operators.

4.1 Rules

In this section, we discuss the rules defined in the base grammar. The majority of these rules are defined using the primitives and combinators provided by our custom parser generator (Chapter 3), although some rules are implemented by Python generators directly.

Rule definition order is important to a proper implementation of a grammar. Because a `Grammar` is created by executing a normal Python program, it is possible to encounter a `NameError` by using a Python variable before it has been defined. This means that co-recursive rules are defined in a way that avoids referencing nonexistent variables (similar to the `palindrome` rule in Figure A-2). In this section, we describe any rules defined in this way as if they were normal rules, for clarity. We also omit discussion of most callback functions, as they are only used to construct the AST for later use in the evaluator (Chapter 5).

We begin with a discussion of some special rules and then continue into the four main sections of the grammar: data structure rules, control flow rules, unary and binary operator rules, and the macro mini-language.

4.1.1 Special Rules

Our grammar includes some special rules that are often used to define other rules, even if they are not specific language constructs themselves. These include whitespace, expressions, statements, and blocks.

One of our goals was to let programmers decide their own indentation and whitespace conventions, so we minimized our use of required whitespace. It is possible to add significant whitespace to the grammar by storing the current indentation level as part of the parser state. This would further complicate the grammar, and goes against our principle of using keywords to delimit blocks, so we chose not to implement it in the base language.

However, we did enforce the requirement that there could be at most one statement per line. Multi-statement lines can be very useful, but these cases are rare at the introductory level. Changing this grammar to use a statement separator (such as a semicolon) is as simple as changing the separator for the `block` rule defined later in this section.

Whitespace For our grammar, we recognized four different kinds of whitespace: required multi-line whitespace (`ws_all`), optional multi-line whitespace (`ws_all_star`), newlines (`newline`), and optional single-line whitespace (`ws_one_line`).

Figure 4-1: Whitespace rules

```
ws_all = Regex(r'\s+')
ws_all_star = Regex(r'\s*')
newline = Regex(r'\n+')

def ws_one_line(source, position):
    for match in parse(ws_all_star, source, position):
        ws, *_ = re.split(r'\n', match.text)
        yield Match(ws, position, ws)
```

The definition of `ws_one_line` (optional one-line whitespace) is an example of a rule defined manually as a generator. The `ws_one_line` rule is used in cases where we need to skip over whitespace without crossing onto the next line (for example, in the `block` rule discussed below).

Expressions and Statements An expression is a combination of values, operators, and functions that evaluates to a value. Since statements also evaluate to values, the meaningful difference between an expression and a statement is that an expression can be used as the value in an assignment but a statement cannot. A macro definition is the only syntax form in the base language that is a statement but not an expression.

The available types of expressions and statements are extensible (see Section 4.3.2), so the grammar rules for `expr` and `statement` must be able to respond to these extensions. We accomplished this by defining two lists within the grammar to contain the different expression and statement rules. These lists are referred to as `expr_types` and `statement_types` which contain references to expression rules and statement rules, respectively. An `expr` or `statement` is simply a `Choice` with the associated list as the options. To add a new expression or statement type, we add the rule for that type to the appropriate list, so that any later parse can make use of that rule.

Blocks A block is comprised of any number of statements separated by newlines. In the `block` rule, we skip one-line whitespace between statements and newlines in order to make indentation (or trailing whitespace) insignificant. `newline` is used as a statement separator to require that at least one newline occurs between any two statements. This can be changed to any other rule (`Raw(';')`, for example), to use that as a statement separator instead.

Figure 4-2: The block rule

```
block = SeparatedStar(
    statement,
    newline,
    skip_whitespace=ws_one_line,
)
```

A block is used for the body of any multi-statement structure, such as a function or a conditional. In order to provide more freedom in programs, we chose to have conditionals and loops (discussed in Section 4.1.3) accept blocks for their conditions.

4.1.2 Data Rules

The rules described in this section support the primitive data types available in the base language. Each of these rules is an element in the `expr_types` list by default.

Numbers Our grammar supports numbers in either integer or decimal format, so as to maintain compatibility with other programming languages and ubiquitous math notation.

Figure 4-3: Number rules

```
integer = Regex(r'\d+')
decimal = Regex(r'\d+\.\d+')
number = Choice([decimal, integer])
```

Because decimal numbers become rational numbers exactly the same way that integers do, these rules could have been implemented as a single `number` rule with `Regex(r'\d+(\.\d+)?')`. We chose to use two separate rules to support a possible change to handling decimals as floating point numbers. An `integer` match is a prefix of a `decimal` match, so the order of options in `number` is important. The `decimal` rule comes before `integer` in the options list to force an attempt to match `decimal` before falling back to `integer` (in case a decimal point is not found).

Strings We had two goals for string literals: avoiding the need for escaped string delimiters and avoiding special syntax for multi-line strings. Our design was based on Perl's `qq` operator, although we do not allow string interpolation and we prefer the quoting operator to be a non-alphabetic character.

String literals begin with a dollar sign (which is otherwise unused by many programming languages) and a single non-whitespace character to use as a delimiter. We consider the string content to be every character in the file until the next occurrence of the delimiter (unless that delimiter is preceded by a backslash). In addition to escaping the string delimiter, we allow the usual special character escapes (tab characters, carriage returns, etc.) within the string. We accomplished this with the rule shown in Figure 4-4.

The named groups are used to determine the string delimiter and string content in the callback function for this rule. The callback function performs any unescapes necessary in the content portion of the string.

Figure 4-4: The string rule

```
string = Regex(
    r'\$(?P<delimiter>\S)'
    r'(?P<content>(\\((?P=delimiter)|[bfrntv\\])|[^\\])*?)'
    r'(\1)'
)
```

Literals The base language contains three more literals: the two booleans (`True` and `False`) and the null value (`Nothing`).

Figure 4-5: The literal rule

```
literal = Choice([Raw(l) for l in ['True', 'False', 'Nothing']])
```

Functions Function definition syntax was chosen to closely match function definition and usage from other programming languages. A function definition consists of a keyword (in this case, `function`), a sequence of function parameter names, a block of code for the body of the function, and the closing delimiter of the function body (in this case `end`). This syntax is described by the rule shown in Figure 4-6.

Figure 4-6: The function definition rule

```
function = Sequence([
    Raw('function'),
    Raw('('),
    SeparatedStar(identifier, Raw(','), skip_whitespace=ws_all_star),
    Raw(')'),
    block,
    Raw('end'),
])
```

Our function call syntax is also similar to other languages: a function succeeded by a parenthesized argument list. The rules defining this syntax are discussed as part of the general expression parser discussed in Section 4.1.4. An example of both function definition and function call syntax are described by the rule shown in Figure 4-7.

Figure 4-7: Function definition and call

```
show_and_square = function(x)
    print(x)
    x*x
end
show_and_square(2)
```

Dictionaries To maintain compatibility with other languages, we chose to implement our unordered collection with the same syntax as Python's dictionaries, which are similar in syntax to JavaScript objects and Go structs. This syntax is described by the rule shown in Figure 4-8.

Figure 4-8: The dictionary rule

```
dictionary = Sequence([
    Raw('{'),
    SeparatedStar(
        Sequence([expr, Raw(':'), expr]),
        Raw(','),
    ),
    Raw('}'),
])
```

Identifiers To allow maximum freedom in naming variables, any sequence of non-whitespace characters that is not already reserved is treated as an identifier. An identifier cannot be the same as a keyword, for example, nor can it contain any unary or binary operators. This rule could not easily be written using the combinators from our parser generator library, so we implemented it using a normal Python generator.

Our strategy for the identifer rule was to find the longest sequence of non-whitespace characters and find the longest prefix of that sequence that does not contain a reserved substring. We accomplish this by splitting the string on each operator and keeping the longest prefix that contains no operators. Finally, we verify that this prefix is not a keyword, a literal, or another reseved word.

Figure 4-9: The identifier rule

```
def identifier(source, position):
    invalid_names = keywords.union(literals).union(reserved_words)
    invalid_substrings = reserved_substrings.union(operators)

    for match in parse(Regex(r'\S+'), source, position):
        name = text
        for chars in invalid_substrings:
            pattern = re.escape(chars)
            name, *_ = re.split(pattern, name)
        if name and name not in invalid_names:
            yield Match(name, position, Node('identifier', name))
```

The `identifier` rule has no callback function because it is implemented as a parser rule directly. The `yield` line emits a `Match` with the data field already set as an instance of `Node` (the class we use for ASTs).

4.1.3 Control Flow Rules

The rules in this section describe the control flow syntax in the base language. Each of these structures is considered an expression, so these rules are elements in the `expr_types` list.

If Expressions Conditionals ("if expressions") are similar to conditionals in C-like languages ("if statements"), with the exception that the condition can be an arbitrary block instead of just a single expression. As usual, the "else" and associated block are optional. This syntax is implemented by the rule shown in Figure 4-10.

An example of an if expression is included in **??**.

While Expressions Indefinite loops ("while expressions") are similar to indefinite loops in C-like languages ("while loops"), with the exception that the condition can be an arbitrary block instead of just a single expression. This syntax is implemented by the rule shown in Figure 4-12.

An example of a while expression is included in Figure 4-13.

Figure 4-10: The if expression rule

```
if_expression = Sequence([
    Raw('if'), block,
    Raw('then'), block,
    Optional(Sequence([Raw('else'), block])),
    Raw('end'),
])
```

Figure 4-11: An if expression

```
if
    x = 1 + 2
    x > 3
then
    print($'Big number')
else
    print($'Small number')
end
```

Figure 4-12: The while expression rule

```
while_expression = Sequence([
    Raw('while'), block,
    Raw('do'), block,
    Raw('end'),
])
```

Figure 4-13: A while expression

```
x = 0
while
    x = x + 1
    x < 10
do
    print(x)
end
```

4.1.4 Expression Parsing

As part of our extensibility goal, the grammar supports arbitrary collections of unary and binary operators. By default, no operators are included, but we describe the process of adding operators in Section 4.3.2.

Because we cannot know the available operators (or their precedence levels) at start time, our expression parsing must be very general. Our expression grammar is based on the precedence climbing algorithm described in [9]. The algorithm handles unary prefix operators and infix binary operators at arbitrary precedence levels. However, the base language includes expression syntax (function call and dictionary access) that the algorithm does not support, so we added special cases to the expression parser to handle them. We could have chosen different syntax, but those forms of function call and collection access are ubiquitous. Another modified version of the algorithm (demonstrated at [10]) supports pseudo-infix expressions and could be used instead.

4.2 Macro Mini-Language

This section describes the mini-language included in the base language that is used for defining macros. Our goal for the macro mini-language was to allow new syntax to be defined at runtime in a user-friendly way. A macro definition consists of the following parts defined in the mini-language: the macro type, the macro name, the macro syntax, the macro body, and any additional metavariables required for the macro. We first describe the syntax of macros; we describe some examples in Section 4.2.1.

Macro Type The two types of macros in this language are expression macros and block macros, which capture expressions and blocks, respectively. Because almost everything is an expression, the distinction between these two is small: an expression macro is added to the `expr_types` list, but a block macro is added to the `statement_types` list. This means that syntax defined as a block macro cannot be captured by an expression macro.

Macro Name A macro name is required because we use the name to associate the syntax and body when making rules and expanding macros. This also allows macros to be re-defined later in the program. A macro name can be any valid identifier.

Macro Syntax The macro syntax portion of a macro definition describes the syntax rule for the macro that will be added to the grammar. We wanted these syntax rules to be accessible to programmers without experience writing parsers, so we chose to use simple pattern-matching rules that we can easily transform into parser rule functions.

In general, a macro syntax rule is a sequence of patterns, where a pattern can be a string (which is matched exactly, as a keyword), an identifier (which matches an expression or block, based on the type of macro), or a repetition (which matches more complex rules). A repetition consists of three parts: the pattern to repeat, the cardinality of the repetition, and the separator. The possible cardinalities are `*` (zero or more), `+` (one or more), or `?` (zero or one).

Figure 4-14: The macro mini-language rules

```
repetition = sequence([
    raw('('),
    syntax_pattern,
    choice([raw(c) for c in '*+?']),
    optional(string),
    raw(')'),
])

syntax_pattern = separated_star(choice([
    identifier,
    string,
    repetition,
]), ws_all)

macro_syntax = sequence([
    raw('syntax'),
    syntax_pattern,
])
```

Macro Body The macro body describes the block that will be evaluated upon macro expansion. Rather than ask the user to create a syntactically valid string (as in C-style macros) or tree structure (as in Lisp macros), this macro body is specified as a single block of program text. This allows the user to easily make function calls, define variables, or use conditionals and loops within the body of the macro. The body of the macro must be able to access the blocks or expressions captured by the syntax pattern, but we wanted to avoid the possible confusion caused by allowing direct modification of captured ASTs. Our solution to this was to introduce an unquote operator (named after Lisp's `,` operator), which can be used to access the syntax elements referred to by a metavariable. By default, the unquote operator is `@`.

Macro Metavariables We recognize a distinction between a variable (which is a name bound to a value in the program) and a metavariable (which is a name bound to a variable name in a macro). Metavariables are very much like Lisp's symbols, except that Lisp symbols are generally valid identifiers, wheras our generated metavariables are not. We cannot easily determine which identifiers used in the macro body are variables or metavariables, so we require the user to specify the set of additional metavariables. Any identifiers used in the syntax pattern will be interpreted as metavariables, so it is not necessary to specify those in this set.

Hygiene The main challenge in the macro system was providing a mechanism for automatic macro hygiene, which makes it easy to prevent a macro from unintentionally accessing identifiers in the main program. A macro is expanded using dynamic scope, so any identifiers it refers to may already exist in the scope in which it is expanded. This problem is not unique to this language. Lisp solves the hygiene problem by introducing the concept of a symbol and providing a function that can generate a new symbol that is has not yet been used as an identifier. We thought it would overly complicate the language to add symbols as a primitive data type, especially without the benefits of Lisp's code-as-data representation. Instead, we solved the problem by automatically generating symbols that cannot collide with variable names and requiring that all metavariables be explicitly declared.

4.2.1 Examples

In this section, we discuss some examples of macros.

Unless One of the simplest macros that can be defined is the `unless` macro (shown in Figure 4-15), which is the opposite of `if`: it executes the body only if the condition is not satisfied. Because it captures arbitrary blocks for both its condition and body, `unless` must be defined as a block macro. It requires no additional metavariables beyond the blocks that it captures, so we can omit that portion of the definition. The body of the macro transforms the `cond` and `body` blocks into the equivalent `if` expression by unquoting them (with the `@` operator) in the appropriate locations.

Figure 4-15: The `unless` macro

```
macro_block unless
  syntax $'unless' cond $'then' conseq $'end'
  transform
    if @cond then else @conseq end
end
```

List Figure 4-16 contains an example of a program that defines the `list` macro, with syntax exactly like that of a Python list. This macro assumes that `=` is used for assignment and that the `<` and `+` operators have been defined. The syntax line defines a rule that matches any number of comma-separated expressions, and stores those expressions in the metavariable called `values`. The body of the macro builds up a dictionary mapping indices to the values in `values`, starting from index 0. This requires two extra metavariables, `i` and `l`, to store the current index and partial list, respectively. At the end of the macro, we evaluate the complete list as the "return value" of the macro.

Figure 4-16: A `list` macro

```
macro_expr list (i, l)
  syntax $'[' (values * $',') $']'
  transform
    @l = {}
    @i = 0
    while @i < size(@values) do
        @l = insert(@l, @i, @values[@i])
        @i = @i + 1
    end
    @l
end

L = [$'Hello', 2]
print($'2 =', L[1])
```

4.3 Grammar Modification

There are a number of ways the grammar can be modified. Because the grammar is a regular Python object, we have the freedom to make modifications to it at runtime or at interpreter start time. At runtime, we use the evaluator to add macro rules to the parser. At startup, we use a settings file to add operator definitions and extra rules for other syntax, such as assignments.

4.3.1 Runtime Modifications

The only runtime modifications available are those performed by runtime macros. A macro definition causes the interpreter to create new parser rules according to the syntax pattern and add them to the front of the `expr_types` or `statement_types` list, depending on what type of macro is being defined. This causes the expression and statement rules to try parsing macros from newest to oldest before falling back to the expression and statement types defined by the base language.

4.3.2 Startup Modifications

There are three kinds of modifications to the grammar that can happen at startup: adding operators, changing the unquote operator symbol, and mutating the `Grammar` object after creation.

Operators Operators, as they relate to the grammar, are represented by a symbol, a precedence, and (in the case of binary operators) an associativity. We store these properties (along with an evaluation function for the evaluator) in the settings file. As a convention, operator precedence is defined as an integer, where an operator that binds more tightly has a higher precedence.

As shown in Figure 4-17, a binary operator definition consists of four parts: the precedence level, the operator symbol, the associativity, and the evaluation function. A unary operator is defined in the same way, except that an associativity is not needed. We assume that all unary operators are prefix operators.

Figure 4-17: Defining operators in the settings file

```
settings = {
    'binary_operations': [
        (2, '+', 'left', lambda l,r: l+r),    # binary addition
        (2, '-', 'left', lambda l,r: l-r),    # binary subtraction
        (3, '*', 'left', lambda l,r: l*r),    # binary multiplication
        (3, '/', 'left', lambda l,r: l/r),    # binary division
    ],
    'unary_operations': [
        (1, '~', lambda x: not x),            # unary not
    ],
}
```

Unquote Some users and instructors may prefer to use a symbol other than `@` as their unquote operator. A settings file defined as in Figure 4-18 will use a two caret symbols (^^) as unquote. This string will automatically be added to the set of reserved substrings.

Figure 4-18: Setting the unquote operator in the settings file

```
settings = {
    'unquote_operator': '^^',
}
```

Mutations The settings file can also define a list of hooks that will be run at the end of the grammar definition. An example of such a hook is included in the settings file in Appendix C.

This settings file uses a hook to define the missing syntax rules needed to support declaration and assignment. It first defines the rules inserts those rules at the beginning of the `statement_types` list. The special symbols used are then added to the `reserved_substrings` set to keep them from being parsed as identifiers.

The possible modifications are not limited to assignment rules. The grammar hooks are passed the entire grammar object as an argument, so these hooks can be used to remove rules, reorder precedence in expression or statement types, change the list of reserved words, add literals, or change anything else about the parser.

Chapter 5

Evaluator

In this chapter, we describe the structure of the evaluator in our interpreter framework. We discuss the built-in data types, the scoping environment structure, and the evaluation strategy.

Our primary goal for defining evaluation was to balance the principle of least surprise with existing programming language conventions. Novice programmers have little or no experience with concepts like scoping rules or function calls, so we tried to simplify the rules that govern the evaluation of programs, when possible. However, there are some cases in which existing terms or conventions are so strong that it would produce more confusion if we deviated from the norm.

We describe the built-in types and functions in Section 5.1, the environment model in Section 5.2, and the evaluation strategies in Section 5.3. Most of the built-in types and functions operate in standard or straightforward ways. The novel portion of the evaluator is the creation of macro syntax rules, which are described in Section 5.4.

5.1 Built-ins

In this section, we describe the data structures supporting the built-in data types and the functions used to operate on them. These functions are included in the base environment of the evaluator, so we refer to them as built-in functions.

5.1.1 Built-in Data Types

We defined each built-in data type as its own Python class. Each class inherits from an abstract base class called `Root`. In this section, we discuss the primitive and compound types separately.

Primitive Types

The simpler literal values in the base language are implemented as primitive types in the evaluator. In this section, we discuss the implementation details of these primitives.

Null and Booleans The base language implements three literals: a null type and the two booleans. When checking value equality, each of these literals is only equal to itself. We do not implement "truthy" or "falsy" checks for booleans, so `1` and `True` are not considered equal in any context.

Numbers We chose to implement numbers as arbitrary precision rational numbers, which has already been implemented by Python as the `fractions.Fraction` class. Our numeric type (`Rational`) subclasses `Fraction`. Novices struggle with the some of the rounding issues that arise when floating-point operations (the non-assocativity of addition, for example), so we chose not to implement floating-point numbers.

Strings Our strings (implemented by the `String` class) store their delimiter in order to display strings to the user the same way they were defined. The delimiter is only used for display. String equality and other operations are only concerned with the content of the string.

Compound Types

The remaining literals are implemented as compound types in the evaluator. In this section, we discuss the implementation details of these types.

Dictionaries A `Dictionary`, as presented to the user, is an immutable key-value store. The keys of a `Dictionary` can be arbitrary primitive or compound types (including the `Dictionary` type) due to the immutability of all language values. Dictionaries are considered equal to each other if the `frozenset`s of their key-value pairs are equal. This allows us to say that two dictionaries are equal if they print identically.

Functions We defined two classes for functions: `BuiltinFunc` and `NativeFunc`. `NativeFunc` is used to make user-defined functions, and `BuiltinFunc` is used to wrap Python functions so they satisfy the same calling interface as `NativeFunc` does.

`NativeFunc` stores its parameter names, the AST for its body, and a reference to its parent environment. At call time, it checks the number of arguments passed, sets up the execution environment, runs the function body block, and returns the value of that block. If too many arguments are passed, an error is raised. All functions are anonymous, so there is no name to store.

If `partial_application` is set in the settings file, passing fewer than the required number of arguments to a `NativeFunc` performs partial application, producing a new `NativeFunc` that takes that many fewer arguments than the original one.

5.1.2 Built-in Functions

The following is a list of all functions included in the base environment of the evaluator:

- `print(...)` is Python's `print` function.
- `length(string)` returns the length of the content of a `String`.
- `concatenate(string1, string2)` returns a `String` that is the concatenation of its two input `String`s, with `string1` as the prefix and `string2` as the suffix.
- `characterAt(string, index)` returns a single-character `String` containing the character at the specified index in `string`. This follows common convention

for indexing and starts at zero.

- `contains(dictionary, key)` exposes Python's `in` operator for `Dictionary` objects. It returns `Boolean(True)` when `key` exists as a key in `dictionary`.

- `insert(dictionary, key, value)` returns a new `Dictionary` containing the same elements as `dictionary` except with `key: value` added. If `key` already exists in `dictionary`, the old mapping is replaced with the new one.

- `remove(dictionary, key)` returns a new `Dictionary` containing the same elements as `dictionary` except with `key` (and its associated value) removed. If `key` does not exist in `dictionary`, no error is raised, and the returned value will be equal to `dictionary`.

- `keys(dictionary)` returns a `Dictionary` containing `index: key` pairs for `dictionary`. The indices start at zero, and the keys are in the same order that display to the user when `dictionary` is printed.

- `size(dictionary)` returns the number of key-value pairs in `dictionary`.

- `as_decimal(number)` returns a `String` whose content is the decimal representation of `number`. This is a way of converting the `Rationals` into their "usual" decimal notation, primarily intended as a convenient way to display the familiar represention of transcendental numbers or unfamiliar fractions.

5.2 Environments

The `Environment` class exists to keep track of stack frames and the parent relationships between them for variable lookup. A new environment is created each time a function is called, which gives the language function scope and lexical scope. `Environments` keep track of which bindings can be changed.

An `Environment` has the following instance attributes:

- `parent`: its parent `Environment` (or `None` if this is the base environment)
- `settings`: a reference to the global `settings` object, which stores the settings loaded at initialization time
- `variables`: a Python `dict` mapping names to their values
- `constants`: a Python `set` of names with constant bindings

Name Resolution We implemented the `Environment`'s name lookup method as the `__getitem__` magic method so we could use Python's bracket notation to access variables. Name resolution continues up the chain of parent `Environments` until the name is found in the `variables` attribute of an `Environment`. If resolution reaches the base environment and the name has not been found, a `KeyError` is raised.

Constant Declaration The `declare_constant(name, value)` method binds `name` to `value` in the environment and marks `name` as constant. If the binding already exists, this raises a `RedeclarationError`.

Variable Declaration The `declare_variable(name, value)` method binds `name` to `value` in the environment. If the binding already exists and the setting `redeclaration_is_an_error` has been set (or the existing binding is a constant), this raises a `RedeclarationError`.

Assignment The `assign(name, value)` method is used to set the value of an previously declared binding. If `name` is bound in this environment, we check that the binding is not constant and then set the value of that binding. If the binding is constant, we raise an `ConstantAssignmentError`.

If the name is not bound in this environment and the setting `declaration_required` has not been set, we declare a variable binding with that name and value in this environment. Otherwise, we assume that a declaration for this name has occurred in the parent environment chain and send the assignment to this environment's parent instead. If no `Environment` in the chain finds a valid binding to assign, we raise an `UndeclaredVariableError`.

5.3 Evaluator

The `Evaluator` class keeps track of the state of the interpreter (macros, operators, and the grammar) and provides an `evaluate` method that evaluates ASTs. It also defines the base environment of the interpreter.

The `evaluate(ast, environment)` method returns the result of evaluating `ast` in the context of `environment`. In this section, we describe the handling of each AST node type grouped by purpose.

5.3.1 Standard Nodes

In this section, we describe the evaluation strategies for the standard AST nodes common to most languages. These evaluate as expected by most experienced programmers.

Expressions and Statements The `expr` and `statement` node types are wrappers around a single expression or statement, respectively.

Numbers, Strings, and Literals A `number`, `True`, `False`, `Nothing`, or `string` node evaluates to an object of the associated data type (as described in Section 5.1).

Dictionaries A `dictionary` node contains child nodes representing key-value pairs. We evaluate each pair (key first, then value) and return a `Dictionary` containing these pairs.

An `access` node (representing a dictionary lookup expression such as `d[key]`) contains two child nodes: a dictionary and a key. We evaluate the dictionary node and the key node and then return the corresponding value for that key in the dictionary.

Operators A `binary_operation` or `unary_operation` node contains child nodes representing the operator symbol and the operands. We look up the evaluation function associated with the operator, evaluate the operand nodes, apply the evaluation function to the operand values, and return the result.

Blocks A `block` node contains a sequence of `statement` nodes. We evaluate each statement, in order, and return the value of the last statement evaluated. If the block is empty, we return `Nothing`.

Functions A `function` node represents a function definition and contains the parameter list and body of the function. We create a `NativeFunc` with those properties, setting the function's parent environment to be the current environment.

A `call` node (representing a function call such as `f(x, y)`) contains child nodes representing a function and an argument list. We evaluate the function node and each of the argument nodes, in order. We then call the function on the arguments and return the result.

Variables An `identifier` node signifies a variable lookup and contains a variable name. We return the result of looking up that name in the current environment.

An `assignment`, `declare_variable`, or `declare_constant` node contains an identifier and a value node, so we evaluate the value node and perform the appropriate action with the name and value. For the special case in which the identifier is actually an `unquote` node and therefore represents a metavariable (see below), we evaluate the unquote node to get a variable identifier before performing the action.

Control Flow An `if` node contains nodes for its condition, consequent, and alternative blocks. We evaluate the condition node to determine which branch to take, evaluate the correct branch, and return the results of evaluating that branch.

A `while` node contains the nodes for its condition and the body. We begin by evaluating the condition node. If the condition evaluates to `True`, we continue by evaluating the body. We then continue from the condition node, looping this process until the condition evaluates to False. The overall return value is the value returned by the last evaluation of the body.

5.3.2 Macro Nodes

In this section, we describe the evaluation strategies for the AST nodes involved in the macro system.

Macro Definitions The macro_block and macro_expr nodes are used to define new block macros or expression macros, respectively. Each of these nodes contains the name of the macro, any metavariables to be declared, the syntax description, and the macro body. When evaluating a macro node, we create the appropriate parser rule from the syntax description and add it to the grammar by appending it to the grammar's statement_types or expr_types list.

Macro Expansions A macro_expansion node denotes the expansion of a previously defined macro rule. We look up the macro name in the state of the evaluator, generate names for each of the metavariables, and update the current environment to contain the mapping of metavariable names to variable names. We then evaluate the body of the macro in the current environment and return the result.

Unquote An unquote node indicates the expansion of an expression containing metavariables. In this case, we evaluate the unquote node to obtain the corresponding identifier node and then evaluate that identifier node to obtain the associated value.

5.4 Creating Macro Rules

When evaluating a macro_block or macro_expr node, we create a new grammar rule describing the syntax that this macro should match. We use a function called make_syntax_rule to accomplish this. The callback functions for these rules are nontrivial, so we discuss them here.

5.4.1 Macro Syntax Rules

`make_syntax_rule` takes four arguments: the syntax rule specification, the node type to capture, a reference to the whole grammar, and a flag (called `multi`, for "multiple match") that indicates whether the rule is part of a repeated syntax specification. The data attribute of a `Match` emitted by this macro rule will contain a Python dictionary that maps metavariable names to the captured ASTs associated with each. In the case of repeated syntax, the value in the dictionary will be a list containing each AST that was captured, in order.

There are four types of syntax patterns that can be matched by the syntax specification of a macro (described in Section 4.2):

Sequence The syntax pattern for an entire macro is always a `Sequence` rule whose terms are the individual syntax patterns. These sub-patterns are created recursively, using `make_syntax_rule`.

Metavariable A metavariable rule is denoted by an identifier in the syntax specification. A metavariable matches one or more expressions or blocks, depending on whether this is part of a repeated rule and which type of macro is being defined. We can match an expression or block using the `expr` or `block` rule from the grammar, but we use the `Single` combinator to add a second callback for post-processing the captured AST.

Keyword A keyword rule is denoted by a string in the syntax specification. We create a new `Raw` rule matching the string exactly, and set its callback to always return an empty metavariable mapping (to simplify combining them). We also add this keyword to the set of reserved words in the grammar. This means that it is possible to make an existing variable name impossible to use by including it as a macro keyword.

Repeat A repeat rule is denoted by a repetition match in the syntax specification. We use the cardinality of the repetition to select among the `Optional`, SeparatedPlus, and SeparatedStar combinators and then make the appropriate rule using the repeated rule (made with make_syntax_rule) and separator rule (made with Raw).

5.4.2 Callback Functions

The callback functions for rules created by make_syntax_rule exist to collect metavariable mappings to captured ASTs. The metavariable and keyword rules define partial mappings, and the sequence and repeat rules combine them into complete mappings. The callback function for the entire macro rule converts this mapping into a combined AST node that can be passed to the evaluate method.

Metavariable The callback function for a metavariable rule takes the captured AST (an expr or block node) and returns the partial mapping of the metavariable name to this node. The multi flag indicates whether this metavariable is part of a repeated match. In the case that multi is not set, the mapping is simply {metavariable: AST}. In the case that multi is set, the mapping is {metavariable: [AST]}, since repeated matchings result in lists of ASTs.

Keyword The callback function for a keyword rule always returns the empty mapping, since keyword cannot create a metavariable bindings.

Repeat The callback function for a repeat rule depends on the cardinality of the repetition. For the zero-or-one cardinality (?, Optional), the callback is either passed None (indicating no match) or a partial metavariable mapping (from an earlier callback function). In the case of None, we return an empty mapping. In the case of a partial mapping, we return that mapping.

For the one-or-more (+, SeparatedPlus) and zero-or-more (*, SeparatedStar) cardinalities, the callback function is passed a list of partial mappings. We combine those into a single mapping by concatenating the lists for repeated metavariable bindings and overwriting the existing bindings for single metavariable bindings.

Sequence The callback function for a sequence rule is the same as for a repeat rule. We combine the partial bindings for each of the terms, handling repeated matches separately from single matches.

Macro The callback function for the entire macro rule converts the final metavariable mapping (from the sequence rule) into a single `macro_expansion` node. This node contains two children: the name of the macro, and the macro parameters in a form that will work with the unquote operator.

We expose the individual ASTs captured by a repeated match in the form of a special dictionary object (`LazyDictionary`). Using a normal `Dictionary` for this purpose, would cause every captured AST to be immediately executed at expansion time, which gives macros the same semantics as functions. We wanted the user to control the evaluation of captured ASTs within the body of the macro, so a `LazyDictionary` delays the evaluation of its values until they are accessed. It does not cache the results (so the `Lazy` prefix is a bit of a misnomer) becasue caching in this way would make it impossible to evaluate an AST for its side effects more than once.

5.4.3 Example

In this example, we describe the process of interpreting a program that defines the `list` macro. The source of this program is in Figure 4-16.

The syntax specification of this macro consists of a keyword, a repeated rule, and another keyword. When evaluating this definition, the strings `'['` and `']'` are added to the `reserved_substrings` set in the grammar. The syntax rule added to the grammar (at the front of the `expr_types` list) is equivalent to the rule shown in Figure 5-1. At expansion time, the body of the macro will run in an environment with the extra variable bindings shown in Figure 5-2.

Figure 5-1: An equivalent syntax rule for `list`

```
Sequence([Raw('['), SeparatedStar(expr, Raw(',')) Raw(']')])
```

Figure 5-2: Extra variable bindings for `list`

```
values: (identifier '$macrosym_0')
i: (identifier '$macrosym_1')
l: (identifier '$macrosym_2')
$macrosym_0: (LazyDictionary
               ((integer 0) (expr (string 'Hello' "'")))
               ((integer 0) (expr (integer 2)))
             )
```

Chapter 6

Discussion

We conclude this thesis with a discussion of the alternate choices for some of our design decisions and a list of possible future work.

6.1 Alternate Design Choices

Our original design for the language included three features not present in the final version:

1. the `goto` statement, which we eventualy found too difficult to implement properly in conjunction with other language features
2. feature blocks that could be combined in a way to produce a full interpreter with those features
3. a mini-language for defining operators within the language itself

6.1.1 Goto

Part of our goal in creating this language was to have novice programmers understand common programming constructs by implementing them within the language itself. As part of this goal, our original design did not include any looping constructs at all. We intended to have users implement their own `while` and `for` loops (or `do-while` and `for-each`) in order to see exactly how control flow operates in each of these loops. This required having a flow control construct more basic than a loop, and we decided that we would accomplish this by providing labels and the `goto` statement.

We were wary of the problems that `goto` can bring [11]. We wanted to avoid the possibility of unintentionally jumping out of one function into another, so `goto` was only allowed to jump to a label that was defined within the same scope. To achieve this, we had to be able to determine all the labels in the current scope before evaluating a function (in case a `goto` appeared later in the function). However, this approach did not work outside of functions because of the inability to parse more than one statement at a time (in the case that a statement was a macro), so we added the restriction that `goto` could only jump to a label that had already been "seen" in the course of evaluation.

This decision made it difficult to implement the evaluator. The state of evaluation had to include what statement was currently being executed and where in the program it was. Environments had to store the mappings from labels to program locations, and each statement was assigned a program location once it was parsed.

Our intended solution to handle macros was to expand the macro body and insert the resulting block at the location of the macro expansion. This led to the idea that program locations could not simply be integers (unless we wanted to re-assign every program location on every macro expansion), so we implemented program locations as Python tuples. Each statement's program location represented two things: how deeply nested into the program its containing block was, and the position at which it fell within that block. The first statement at the top level in the program had location (0), and the second had location (1). If the second statement was a function definition, for example, any statements within that function's body block would have locations of (1, 0), (1, 1), and so on. Each layer of nesting would extend the length of the tuple by one element, assigning the last element to zero for the first statement at that level. The default Python sort order on tuples happens to order these program locations correctly (`(0) < (1) < (1, 0) < (1, 1) < (2)`).

Conditional execution and `goto` were implemented as special `jump` statements that were inserted into the program as needed. These would cause program evaluation to immediately restart at a specific program location. We realized that a real call stack would be necessary to ensure the ability to return out of a function correctly and that the implementation of the evaluator would be cleaner as a virtual machine instead. Rather than complicate the language implementation further, we decided to remove `goto` from the language and implement `while` in its place.

6.1.2 Feature Blocks

As part of the extensibility goals of the language, we wanted an instructor (or other language maintainer) to be able to add features to the language that were more complex than our macro system would allow. Our original plan was to structure the interpreter in a way that was inspired by the strategy outlined in [12]. The language would be broken up into feature sets, and each feature set would require a parser and evaluator. Each feature set would be implemented by a feature block, and these blocks could then be composed in different ways to form interpreters for different languages.

At the time, we were still experimenting with existing parser generators, so we only implemented the evaluator in this style. Due to the recursive nature of stacked evaluators and Python's lack of tail-call optimization, any non-trivial program caused us to exceed Python's recursion limit. One choice we had was to implement this in a language that provided tail call optimization, but we decided to re-structure the evaluator in a more iterative style in order to support `goto` (which was still in the language at that point).

The ability to toggle features eventually returned in the form of the settings file, but adding entirely new features to the interpreter still requires an understanding of the other features already present.

6.1.3 Operator Mini-Language

Our original extensibility features included a mini-language (similar to the macro mini-language) for defining new operators from within the language itself. The original syntax used a keyword `operator` and required specifying both the string to use as the operator symbol and the function to use as its evaluator. We used the arity of the function to determine whether to create a unary or binary operator (or error, in the case of a higher arity function). This automatic operator arity detection goes against our learnability goals, so we split the `operator` keyword into `unary_operator` and `binary_operator`.

In this formulation, each new operator defined was at a lower precedence than any previously defined operators. This allowed a user to define operators without knowing the existing precedence hierarchy, at the expense of forcing a specific ordering of operator definitions. However, this made it impossible to add a new operator to a pre-existing precedence level or define two new operators at the same precedence level. We considered an extension to the mini-language in which the precedence level was included in the operator definition, but chose not to implement it because this required the user to design an entire precedence hierarchy before defining a collection of operators.

Operators appear to be a special-case of macros, so it seems possible to use the macro system to regain this function without its own mini-language. However, this is only true for prefix operators. In the case of infix and suffix operators, the macro begins by looking for an `expr`, which causes infinite left recursion in the parser. Although there are ways of automatically re-writing rules to avoid left recursion [13], we cannot guarantee this will work for the non-regular rules in our grammar. Ultimately, we decided that instructors would be better equipped to define custom operators for the language, so we moved all operator definition into the settings file.

6.2 Future Work

We note two important improvements to be made on this work. First, there are still many opportunities to improve the interpreter framework, both in terms of usability and efficiency. Second, the language design can be improved with feedback from real programmers at any experience level.

Modules There is currently no notion of modularity among files being executed by the interpreter. All files are run in the same global environment with no separation between them, which can cause unexpected problems if variable names are reused across files. This can be remedied by implementing a module system with a new `import` statement.

Temporary Macros One problem with macros is that there is no notion of a temporary macro. Once a macro has been defined, it exists in the grammar until the program finishes executing. One way to accomplish this is by adding a new type of statement that allows for the enabling and disabling of an existing macro. Another way would be to provide a mechanism for restricting the use of a macro to a certain scope.

Feature Blocks Now that `goto` has been removed from the language and the parser can be modularized, we can return to the original interpreter structure with feature blocks. Each block can define grammar rules and data structures along with registering evaluator functions for specific AST node types. This allows for easy addition, removal, or modifcation of entire language features.

Evaluation Our primary goal was to make a language that was easy for students to learn, but we have not shown that how this language compares to other languages in that respect. Future work can include experiments or surveys to determine how properties of this language affect the rate and quality of learning.

Appendix A

Sample Grammars

In this appendix, we present some sample grammars that can be defined using our parser generator library (discussed in Chapter 3).

A.1 EBNF Grammar

As an example, we have included a sketch of the implementation of a parser for EBNF grammars (Figure A-1). This is not the most concise specification of EBNF, but we include this form of it to show some common usage patterns.

Letters and digits are recognized by common regular expression patterns, but the regular expression rules for matching any of a set of special characters that are also reserved in regular expressions are complicated. Here we make a single `Raw` string matcher for each special character and combine them into a `Choice` rule. Identifiers cannot contain whitespace, and terminals should include all characters within their enclosing quotes, so we disable whitespace skipping within those rules. `rhs` is an example of a self-recursive rule, so defining it as a simple `Choice` rule would fail, as the argument to `Choice` cannot be evaluated until `rhs` has been defined. The current workaround for this is to create a function that then makes a rule (with all names resolvable by the time it is called) and then yield all results from a parse of that rule.

This specification does not use any callback functions, so all the data attributes on returned `Match`es are strings or lists of strings. To check whether a string is

Figure A-1: An EBNF grammar

```
from generator import *

letter = Regex(r'[a-zA-Z]')
digit = Regex(r'\d')
symbol = Choice([Raw(s) for s in '''[]{}()<>'"=|.,;'''])
character = Choice([letter, digit, symbol, Raw('_')])
identifier = Sequence(letter, Star(Choice([letter, digit, Raw('_')])),
                      skip_whitespace=False)
terminal = Choice([
        Sequence([Raw("'"), Plus(character), Raw("'")],
                 skip_whitespace=False),
        Sequence([Raw('"'), Plus(character), Raw('"')],
                 skip_whitespace=False),
])
lhs = identifier

def rhs(source, position):
    rule = Choice([
            identifier,
            terminal,
            Sequence([Raw('['), rhs, Raw(']')]),
            Sequence([Raw('{'), rhs, Raw('}')]),
            Sequence([Raw('('), rhs, Raw(')')]),
            Sequence([rhs, Raw('|'), rhs]),
            Sequence([rhs, Raw(','), rhs]),
    ])
    yield from parse(rule, source, position)

rule = Sequence([lhs, Raw('='), rhs, Raw(';')])
ebnf = Star(rule)
```

recognized by this grammar, we would use the generator library's `parse` function and check whether or not it returned a valid match:

```
next(parse(ebnf, "x = 'Hello';", 0)) is not None
```

A.2 Palindromes

Although arbitrary-length palindromes cannot be matched by traditional regular expressions, many programming languages (Perl and Ruby, for example) provide extensions to regular expressions that allow for such a "regular expression" to exist. Python is not one of these languages, so an external library must be used to access that kind of extension. In Figure A-2, we demonstrate a parser rule that would recognize arbitrary-length palindromes that contain no whitespace.

The implementation of `is_palindrome` makes this rule extremely slow. It naively checks for palindromes by verifying that the first and last characters are the same and then recurses into the middle of the string. This creates a very large call stack, and creates many slices of the string in memory. However, we include this as an example of rules that would be difficult (or impossible) to define in other parser generators.

Figure A-2: A palindrome grammar

```
from generator import *

def is_palindrome(x):
    if len(x) < 2: return True
    if x[0] == x[-1]: return is_palindrome(x[1:-1])
    return False

def palindrome(source, position):
    for match in parse(Regex(r'\S*'), source, position):
        for i in range(len(match.text), -1, -1):
            pal = match.text[:i]
            if is_palindrome(pal):
                yield Match(pal, position, pal)
```

A.3 Stateful Parser

Many parser generators support parser states to simplify the implementation of separate "modes" of parsing. This parser generator also supports modes as a special case of general parser state. We include an example of a stateful grammar (Figure A-3) that only accepts words that have been explicitly marked as acceptable (using the `:word` syntax). This example is somewhat contrived; any grammars that truly need state tend to be too large or complicated to use as an example.

This parser stores its state in the `accepted_words` list. The callback function for `new_word` adds each new word to this list. Each time `known_word` is used to generate matches, it creates a new `Choice` from all the acceptable words at that time.

Figure A-3: A stateful grammar

```
from generator import *

accepted_words = []

def new_word(source, position):
    def callback(data):
        _, word = data
        accepted_words.append(word)
        return data
    rule = Sequence([Raw(':'), Regex('\S+', callback=lambda x: x.group())],
        callback=callback)
    yield from parse(rule, source, position)

def known_word(source, position):
    rule = Choice([Raw(word) for word in accepted_words])
    yield from parse(rule, source, position)

line = Choice([new_word, known_word])
grammar = SeparatedStar(line, Regex(r'\s*'))
```

Appendix B

Example Programs

In this appendix, we present some sample programs in our language. Unless otherwise stated, these programs assume that the interpreter was started with the settings file in Appendix C.2.

B.1 Prime

The program in Figure B-1 defines some helper functions, building up to a function that determines whether a number is prime.

B.2 Switch

The program in Figure B-2 defines a macro that implements a `switch` statement. Besides syntax, the differences between this macro a C-like switch statement is that cases do not "fall through" and the executed case is the last one that matches, rather than the first.

Figure B-1: A prime-checking program

```
not = function(a) if a then False else True end end
neither = function(a,b) if a then False else not(b) end end

modulo = function(num, modulus)
    if num >= modulus then
        modulo(num-modulus, modulus)
    else
        num
    end
end

divides? = function(m, n)
    modulo(m, n) == 0
end

prime? = function(x)
    divisor_found = False
    if x < 2 then
        divisor_found = True
    else
        i = 2
        while neither(i >= x, divisor_found) do
            divisor_found = divides?(x, i)
            i = i + 1
        end
    end
    not(divisor_found)
end

print(prime?(3))
```

Figure B-2: A program using the `switch` macro

```
macro_block switch (i, found, idx, value, case)
syntax
    $'switch' value_block ($'case' cases $'do' bodies *)
    ($'default' default ?) $'end'
transform
    @found = False
    @i = 0
    @value = @value_block
    while @i < size(@cases) do
        @case = @cases[@i]
        if @value == @case then
            @idx = @i
            @found = True
        end
        @i = @i + 1
    end

    if @found then
        @bodies[@idx]
    else
        @default[0]
    end
end

x = 10
y = 20

switch x
case 1 do
    print($'is one')
case y do
    print($'is y')
case 10 do
    print($'ten!')
default
    print($'none')
end
```

Appendix C

Example Settings

In this appendix, we present some examples of settings files that can be used to perform modifications to the language at interpreter startup.

C.1 Default Settings

This section describes the default settings of the interpreter.

Figure C-1: The default settings file

```
DefaultSettings = {
    'partial_application': False,
    'constant_function_arguments': False,
    'redeclaration_is_an_error': True,
    'declaration_required': False,
    'canonical_string_delimiter': None,
    'unquote_operator': '@',
    'binary_operations': [],
    'unary_operations': [],
    'grammar_post_create': [],
}
```

By default, functions cannot be called with be partially applied and the names bound to their arguments can be reassigned. Re-declaring a name binding results in an error, regardless of whether it was declared as a constant or a variable, but declaration is not required. There is no canonical string delimiter, so strings will display with the delimiters used when they were defined. The unquote operator is `@`, and there are no additional operators defined. There are no additional modifications to the grammar performed after it is created.

C.2 Basic Settings

This settings file defines one set of missing features (assignment and math operators) for the base language. These settings are applied by merging them with the default settings above, concatenating any settings that are lists, and overwriting any settings that are single values.

```
from src import generator
from src.util import Node


def add_assignment(grammar):
    def assignment(source, position):
        matcher = generator.sequence([
                generator.choice(grammar.assignable),
                generator.raw('='),
                grammar.expr,
        ],
            skip_whitespace=grammar.ws_one_line,
            callback=lambda seq: Node('assignment', seq[0], seq[2]),
        )
        yield from generator.parse(matcher, source, position)

    def declare_variable(source, position):
        matcher = generator.sequence([
                generator.choice(grammar.assignable),
                generator.raw(':='),
                grammar.expr,
        ],
            skip_whitespace=grammar.ws_one_line,
            callback=lambda seq: Node('declare_variable', seq[0], seq[2]),
        )
        yield from generator.parse(matcher, source, position)

    def declare_constant(source, position):
        matcher = generator.sequence([
                generator.choice(grammar.assignable),
                generator.raw('::='),
                grammar.expr,
        ],
            skip_whitespace=grammar.ws_one_line,
            callback=lambda seq: Node('declare_constant', seq[0], seq[2]),
        )
        yield from generator.parse(matcher, source, position)

    grammar.assignable = [grammar.identifier, grammar.unquote]

    grammar.assignment = assignment
    grammar.statement_types.insert(0, assignment)

    grammar.declare_variable = declare_variable
    grammar.statement_types.insert(0, declare_variable)

    grammar.declare_constant = declare_constant
    grammar.statement_types.insert(0, declare_constant)

    grammar.reserved_substrings |= set(['=', ':=', '::='])
```

```
settings = {
    'binary_operations': [
        (0, '==', 'left', lambda l,r: l==r),
        (0, '!=', 'left', lambda l,r: l!=r),
        (1, '>=', 'left', lambda l,r: l>=r),
        (1, '<=', 'left', lambda l,r: l<=r),
        (1, '>', 'left', lambda l,r: l>r),
        (1, '<', 'left', lambda l,r: l<r),
        (2, '+', 'left', lambda l,r: l+r),
        (2, '-', 'left', lambda l,r: l-r),
        (3, '*', 'left', lambda l,r: l*r),
        (3, '/', 'left', lambda l,r: l/r),
    ],
    'unary_operations': [
        (2, '-', lambda x: -x),
    ],
    'grammar_post_create': [add_assignment],
}
```

Bibliography

[1] Benedict du Boulay, Tim O'Shea, and John Monk. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 14(3):237–249, 1981.

[2] `https://cacm.acm.org/blogs/blog-cacm/176450`, April 2017.

[3] Roy D Pea. Logo programming and problem solving. 1987.

[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113–128, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.

[6] Linda McIver and Damian Conway. Seven deadly sins of introductory programming language design. In *Proceedings of the 1996 International Conference on Software Engineering: Education and Practice (SE:EP '96)*, SEEP '96, pages 309–, Washington, DC, USA, 1996. IEEE Computer Society.

[7] David Joyner, Ondřej Čertík, Aaron Meurer, and Brian E. Granger. Open source computer algebra systems: Sympy. *ACM Commun. Comput. Algebra*, 45(3/4):225–234, January 2012.

[8] Piraye Bayman and Richard E. Mayer. A diagnosis of beginning programmers' misconceptions of basic programming statements. *Commun. ACM*, 26(9):677–679, September 1983.

[9] `http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm#climbing`, April 2016.

[10] `http://www.engr.mun.ca/~theo/Misc/pratt_parsing.htm`, April 2017.

[11] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968.

[12] Guy L. Steele, Jr. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 472–492, New York, NY, USA, 1994. ACM.

[13] Robert C. Moore. Removing left recursion from context-free grammars. In *Proceedings of the 1st North American Chapter of the Association for Computational Linguistics Conference*, NAACL 2000, pages 249–255, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.