Readtable-Macro Transducer-Chain Parsing

MSc Thesis *(Afstudeerscriptie)*

written by

Eli T. Drumm
(born June 27th, 1989 in New Orleans, Louisiana, USA)

under the supervision of **Dr Bruno Loff** and **Dr Leen Torenvliet**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
July 4th, 2016	Prof Dr Ronald M. de Wolf (chair)
	Dr Bruno Loff
	Dr Leen Torenvliet
	Prof Dr J.J. Vinju
	Prof Dr Jan van Eijck

Institute for Logic, Language and Computation

Abstract

While extending the semantics of programming languages is commonplace, extending their syntax is usually not possible. In this thesis, we are concerned with the problem of parsing programming languages that are simultaneously syntactically extensible and human-readable. The quintessential extensible programming language, Lisp, can be extended by two separate macro systems that allow user-defined syntactic features to be integrated naturally and directly into the parser. However, the default syntax of Lisp is notoriously unreadable. Other languages, such as Python, are highly readable, but are designed and parsed in ways that leave the syntax essentially frozen from the user's perspective.

We will propose a new parsing method, called *Readtable-Macro Transducer-Chain parsing*, that is inspired by Lisp's reader algorithm. This algorithm is naively extended to add support for readable syntactic structures (such as infix notation), while retaining the spirit of simplicity and modularity that is its characteristic.

As a proof-of-concept, we implement a compiler for a new programming language called *Anoky* — essentially an alternative syntax for Python.

Contents

Chapter 1

Towards a Readable, Extensible Syntax

Extensibility is an indispensable part of modern programming languages. Variable definitions, user-defined types and classes, libraries, modules and import systems, interfaces with other languages, are all extensions in some form or another.

Our concern in this thesis is *syntactic extensibility* of programming languages. This is not a new idea. Many languages incorporate features that give users limited ways to modify the default syntax, for example defining custom operators. Usually, though, syntax is one of the unprogrammable parts of a programming language. A programmer (or group, or community) may want or need to add or customize

- new control structures or abbreviations for frequently-used patterns (`foreach`, `parallel-for`, ...)
- new notation for a new kind of data-type (GObjects, ...)
- new notation for literals (regexps, ...)
- annotations for a certain purpose (documentation, contracts, ...)
- alternative parsing at the top level (for literate programming, mathematical notebooks, ...)
- support for embedding data, markup, or code written in domain-specific languages (SQL, HTML, ...)

But unless the language has implemented an explicit extension mechanism for that particular situation, there is no way to do it, other than maybe writing a compiler into the desired language.

A language's syntax constrains how its users can express ideas, data, etc. and influences how they think. Not all programmers will have the ability or resources to create their own language according to their personal ideals and way of thinking, but almost everyone has ideas, preferences, moments of "I love this programming language, but...".

The reasons for the commonly-seen inextensibility of syntax are mostly technical. Parsing is a difficult problem; creating an accurate, reasonably fast parser for a given language requires specialized knowledge, and the design of programming languages and syntax has historically been restricted to those with the know-how: CS researchers, talented hobbyists and visionaries, professional software engineers.

Modifying a working parser to correctly parse a new bit of syntax can be a challenging experience, as even small extensions will sometimes require a lot of work. Even using automatic parser-generating tools is not the user-friendly experience it should be.

As a result, programming language design is usually restricted to a small group of developers, and only they get to decide in what direction the programming language gets to grow[1]. This is unfortunate, because it misses the opportunity to use the distributed work of the entire community to enrich the language with new features at the level of syntax [Ste12].

The motivation underlying our work is the need for parsing methods — algorithms, paradigms, software, etc. — that are compatible with extensible syntax. To this end, we will present a new parsing method, called *Readtable-Macro Transducer-Chain parsing*.

In the next section, 1.1, we lay out the properties that we would like to have in any extensible parsing method. Then in section 1.2 we turn our attention to Lisp, the quintessential extensible language. We discuss the factors that contribute to the intrinsic extensibility of the Lisp family, followed by descriptions of the two main ways to extend the syntactic behavior of Lisp: macros and reader macros. Section 1.3 is devoted to the topic of readability. Using Python as an example, we focus on two features of Python's syntax that influence its readability, and which are relevant for the choice of parsing method, and contrast Python's example with Lisp. Finally, in section 1.3.3, we briefly describe the "readable S-expressions" project [Whe], perhaps the most noteworthy attempt to improve Lisp's written syntax.

1.1 What we are looking for

To focus our discussion, let us start by identifying some desirable properties of a parsing method that supports extensible syntax.

Simplicity of the algorithm

The algorithm should be simple to understand. Complexity is not an automatic disqualifier, and is unavoidable to some extent given the inherent difficulty of parsing. It is enough if the complexity of the parsing method is compartmentalized in a way that allows users to extend the programming language without having learn the entire algorithm, or keeping the entire algorithm in mind.

[1] Some programming languages have democratic procedures for proposing changes to their language, such as Python's PEP system[War+00], but this is no substitute for direct extensibility.

Simplicity of the extension process

Modifying the language should be simple. Small tweaks should be quick to implement. More significant changes should be within reach. The user should not have to devote significantly more time to learning how the parser interface works than actually working on the language.

Modularity

The user should be able to load syntactic extensions written by other users in a modular fashion. Ideally, there will be some natural way to define what a "module" is and what "loading" or "importing" entails in the context of the given parsing method. The parser should require little or no additional configuration in order for this to work, and there should be little potential for conflicts among different imported features.

Compositionality

If our method can parse languages L_1 and L_2, it should be possible to produce a language L_3 that combines both languages. L_3 may depend on the parser, the user, or both.

A compositional parsing method allows the user to combine languages coherently. There should be a way for the user or designer to specify how to use multiple languages in the same file, for example by way of meta-rules that specify how to put the two syntactic structures together.

Generality

We want to give the user as much freedom as possible to extend the syntax in arbitrary ways. Because we cannot anticipate the specific needs of users, the parser should be designed so that it is customizable in any direction.

This means, for example, that the interface between user and parser has to allow for more expressiveness than toggling prefabricated extensions from a built-in set.

Readability

Representations and layouts that make a language easier for humans to read can frequently require more sophisticated parsing techniques. An extensible parser should be able to parse the kind of syntactic structures that make programming languages readable.

Speed and efficiency

The parser should be as fast and efficient as possible. Speed can be sacrificed to *some* extent in order to satisfy the above properties, but it cannot be entirely neglected.

The quintessential example of an extensible programming language is Lisp, which we discuss in depth in the following section. Lisp and its parser, the Lisp reader, are simple, general, and modular.

On the other side, we have Python, a language designed to be readable. Python is extensible in many ways but not when it comes to syntax. We will look at readability in section 1.3.

1.2 Extensibility in Lisp

Lisp is really a family of programming languages; the most popular of the Lisp dialects in use today are Common Lisp, Scheme, Racket, Emacs Lisp, and Clojure. For the most part, we can talk about *Lisp* in a monolithic sense because the relatives all share similar principles, including a similar approach to parsing and extensibility. When we need to be more specific, we will use the Common Lisp Hyper Spec [Pit96] as a reference.

1.2.1 Why is Lisp so extensible?

There are several reasons, which we will outline here.

Lisp's syntax is generic

Parsed Lisp code consists of *forms*, which are either *atoms* or *compound forms*. Atoms are either *symbols* or *self-evaluating objects* (objects that evaluate to their own value, e.g. numbers, strings), and compound forms are defined inductively as lists of other forms.

Every instance of parsed lisp code is represented in this way (by atoms and compound forms), regardless of whether the parsed code represents the application of a control-structure, function call, or anything else. In this way we can say that the syntax of Lisp is *generic*. Notably, this allows for a written representation for the language, called *symbolic expressions* or *S-expressions*, so simple that it can be completely described in one sentence: compound forms are enclosed in parentheses and atoms are separated by whitespace.

```
(a 100 (b (c "d") e) f)
```

This choice of syntax, among other advantages, makes parsing very simple. Genericity is also useful in simplifying the evaluation model and other core language functionality.

Lisp's syntax is homoiconic

By reading a line of written code we immediately know the abstract structure of the parsed code. For example, we can identify `(print (+ 1 4))` immediately as a compound form with two elements: the symbol `print` and another compound form made up of three atoms.

This property of a language has been termed *homoiconicity*. In Lisp this manifests as a direct correspondence between a form's abstract tree structure and its representation as an S-expression. Every pair of parentheses corresponds to a compound form, the outermost parentheses correspond to the root of the syntax tree, and the elements of a form are written in the same order.

As a contrast, consider the equivalent Python code `print(1 + 4)`. If we know Python, we know we are looking at a function call, the function has one argument, and this argument is a binary arithmetic expression. We might

guess that the abstract syntax tree for this code is something like in the Lisp example, but to be sure we have to look up the documentation for Python's abstract syntax trees. We will find that these trees, even for small examples, are significantly more complicated than the simple inductive definition of Lisp forms.

It is easy to manipulate Lisp code using Lisp

A generic syntax makes the internal representation of code very simple. A homoiconic syntax makes it easy to identify the internal representation of code when looking at its written representation. It is then also easy to write programs that perform various operations on this internal representation. This is called *meta-programming*.

```
> (setq x '(+ 5 5))
> (eval x)
10
> (setf (nth 0 x) '*)
> (eval x)
25
```

Above we set the variable `x` to the form with three atoms (one symbol, `+`, and two numbers). Then we evaluate it, change the first atom, and evaluate it again.

The reader and evaluation algorithms are simple

Because both the written and abstract syntax is so generic, procedures for processing arbitrary Lisp code mostly come down to a few case statements and recursion. This includes core functionality like the `read` and `eval` procedures.

The user does not have to understand most details and subroutines in the implementation in order to understand the evaluation model, or to extend it. Additionally, these functions are implemented in Lisp itself; so extending Lisp does not require learning a new language or configuration system.

These features give us almost everything we are looking for

These features together provide us with a simple, general, and modular way of extending Lisp. Generic syntax, homoiconicity, and the code-manipulating abilities of lisp make Lisp macros powerful and expressive. Most extensions that are commonly desired can be implemented by Lisp macros (the same could not be said of C macros, for example).

Full generality follows because almost all parsing tasks, even those very different from Lisp's methods, can be introduced via reader macros (we will see an example of this in section 1.3.3). Macros and reader macros are modular by nature: it is easy to import macros and reader macros from different libraries, often even in cases when the writers of said libraries did not coordinate. And, remarkably, using both of these features is simple, owing to the simplicity of the reader and evaluation algorithms.

What is missing from Lisp, alas, is readability. We will discuss this in section 1.3. For the remainder of this section, let us describe macros and reader macros in greater detail.

1.2.2 Syntactic Macros

Lisp code can be created and manipulated as data. Lisp macros are functions that operate on Lisp code. A macro is fed a Lisp form, performs some operations, and returns some other Lisp form.

As an example,

```
(defmacro plus3 (x) `(+ 3 ,x))
```

defines a macro called `plus3` that takes one parameter, `x`. To use this macro, we can write, for instance, `(plus3 3)`.

Lisp distinguishes macros from functions in the way macros are integrated into the evaluation process and how they are used as a result of this. The general idea is that macros are used to transform Lisp code in place during evaluation. When evaluating code, some symbols will be associated with a macro function. Lisp will maintain these associations in an implementation-dependent way [Pit96, "Accessor MACRO-FUNCTION"]. If the head of a compound form is a symbol associated with a macro, the respective macro function will be called with the given unevaluated arguments and output some code; this process is called *macro expansion*. The Lisp code it outputs will then be evaluated in place of the original form.

Note that Lisp macros are not limited to simple replacement operations. The macro function can execute arbitrary computation when expanding forms. Compare this to C macros: Although they are both used to process code before it is executed, Lisp macros are fundamentally different than those in C. A macro in C is part of a separate preprocessor that modifies source code before it is given to the "real" C compiler. This preprocessor has limited power, as it can only do simple text replacement operations.

In Lisp, macros allow programmers to be more concise and to write clear code that is expanded to longer and less readable code that performs the same computation. But macros are not merely a convenience, they are a fundamental component of both the use and implementation of the language.

Using macros, we can define new control structures that cannot be expressed easily with functions. In fact, every control structure in Common Lisp can be defined as a macro built up from 25 primitives, called *special forms* [Pit96, "3.1.2.1.2.1 Special Forms"]. For example `tagbody` and `go` can be used to define `loop`, which can be used to define `do` [Pit96, "Macro DO, DO*"].

Or consider, to give another example, the Common Lisp Object System (CLOS). It adds object-oriented programming to Lisp, and it is largely implemented using macros. Features like class definitions, method definitions, *etc.* can be accessed via a special-purpose syntax that feels natural to the programmer. Yet CLOS can be imported just like any other package. This is impossible to do in most programming languages.

1.2.3 Reader macros

Reader macros are distinct from macros in the previous section. Both types of macros can extend syntax, but their function and mode of operation is completely different. Whereas macros operate on Lisp code during evaluation, a *reader macro* operates on the input characters as they are read into Lisp.

Certain characters of text are classified as *macro characters*. The Lisp reader maintains a *readtable* that keeps track of the type and properties of every character it might encounter; for macro characters, the readtable will maintain associated reader macro functions.

When a macro character is encountered while reading input, the reader delegates processing of the input to the reader macro function, which is then responsible for reading and processing the input stream until it returns a form. If the input stream is not exhausted, the calling reader may continue from that point. The most familiar macro characters in Lisp are `(` and `)`. The reader macro function for `(` accumulates a list of objects — by calling the default reader repeatedly via the `read` function — until it finds a closing parenthesis. It then returns a list with all the forms that were read.

To extend the written syntax of a Lisp program, we may write a reader macro that converts text into the Lisp forms the text is supposed to represent (according to the custom syntax being implemented). The generality of reader macros comes from the fact that the active reader has complete control over the stream, and — being a programmable procedure like any other — can parse the input using any desired parsing method. We are free to write a reader macro which processes input that looks absolutely nothing like Lisp, provided the end result is still a form.

One common application for reader macros is adding notation. See [Hoy08, chapter 4] for an example of adding Perl-like regular expression syntax to Lisp (i.e., being able to use special purpose notation for regular expressions, as in `/abc/`).[2]

1.3 Readability

Like extensibility, readability is an uncontroversially desirable property for a programming language. Yet its desirability relative to other features, and even the assessment of what readable code should look like, are matters of frequent disagreement.

Without controlled, empirical studies, it is hard to justify a claim that Language 1 or Notation X is more readable than Language 2 or Notation Y, and, even then, it is hard to argue against personal preference.

However, Python is generally regarded as an example of a readable programming language. In the following section (1.3.1) we discuss two central aspects of Python's syntax that help make Python highly readable, and that are relevant to our discussion of parsing methods.

1.3.1 Why is Python so readable?

Readability is a core design principle of Python [Ros09].

A host of reasons contribute to the readability of Python, many of which are not syntactic in nature. For example, its sparing use of punctuation, its carefully chosen lexical scoping rules, or its culturally maintained notational Catholicism. But the two central factors that concern us are Python's use of

[2] In section 4.4 we will extend our own programming language to support Perl-like regular-expression syntax. Incidentally, we will also use reader macros to add S-expression syntax to our own language in §4.3.1).

indentation for grouping blocks of code, and its use of infix and other specialized syntax for various constructs.

Indentation

The original proposal for using indentation to denote program structure is found in Landin's landmark paper *The Next 700 Programming Languages* [Lan66]. There, Landin introduces what he calls the *offside rule*:

> The southeast quadrant that just contains the phrase's first symbol must contain the entire phrase, except possibly for bracketed subsegments.

This formalizes the intuitive idea that we can group together parts of a program (lines, statements, *etc.*) by indenting them at least as much as the first part.

Landin presents the offside rule as an alternative format for writing code that can be used *alongside* standard whitespace-insensitive notation, and this is how most programming languages make use of it.

Indentation and other whitespace is consistently used in practice to add readability to written programs in whitespace-insensitive languages. However, there are relatively few programming languages where indentation is an integral part of the syntax. Aside from Python, the exceptions tend to be special-purposes languages for data and markup (YAML, Haml, reStructuredText, Markdown, *etc*) which have a specific reason to prioritize readability.

In languages where indentation has no actual bearing on the meaning of the code, a program with misleading indentation will compile just as well. Some compilers can even verify that the layout of the written code matches its parsed structure, even though the two are technically independent (e.g. GCC's `-Wmisleading-indentation` option).

The central argument for whitespace sensitivity is this: Indentation is the best way to make code readable for humans, and if we can program a parser that also uses indentation to discern grouping, using explicit markings on top of the whitespace is unnecessary at best. Insisting on two grouping notations requires additional effort to maintain consistency between the two, and adds to the visual clutter, actively lowering readability.

Making indentation a part of the syntax allows programmers to write code by sticking to common layout conventions which they would (or should) likely have been using anyway. Additionally, because there is less punctuation there are fewer permutations of prescribable punctuation placements, which increases stylistic homogeneity across the community of users, and decreases the amount of visual noise.[3]

Infix and special syntax

Almost all programming languages in use today support infix notation (the extant members of the Lisp family are the rare exceptions). Almost all of them make use to some extent of special syntax for keywords and core features of the language. Python is solidly in both categories.

[3] `https://docs.python.org/3.5/faq/design.html#why-does-python-use-indentation-for-grouping-of-statements`

Without wading into a discussion of whether some syntaxes are more "natural" in some sense, it is certainly the case that most humans are more accustomed to reading expressions as infix — from their education and culture generally — than other forms of notation. Even if it is only sugar, this syntax is essential.

Infix notation is not restricted to mathematical expressions. Python makes use of this fact in its notation for comprehensions, as the examples in the following section will illustrate.

For our purposes, we must then realize the following: The parsing method should be powerful enough to properly parse infix and special-purpose syntax.

1.3.2 Python vs. Lisp by example

Python is the most prominent modern language where indentation is a core component of the syntax and forced on the programmer. This takes away some programmer freedom but ensures that the resulting code is more readable. Written Lisp code, on the other hand, has no indentation specification and as a result is notoriously parenthesis-heavy, which is unsurprising given that a pair of parentheses enclose every non-atomic piece of syntax.

Here are two examples that illustrate the difference.

```
(defun f (a1 a2)
  (let ((x 2) (y 3))
    (+ (* a1 x) (g a2 y))))
```

```
def f(a1, a2):
    x = 2
    y = 3
    return a1 * x + g(a2, y)
```

In the Python example, function calls, assignment targets, statement boundaries, etc. are all immediately recognizable. The Lisp example is full of syntactic noise, and the important elements are harder to discern. Furthermore, the prefix notation forces us to look around the individual forms to understand the semantics.

Python, like most programming languages, has special-purpose syntax for essential language constructs, and leverages infix as well as postfix notations. This is illustrated in the example below. The syntax in the Lisp version is, as we would expect, mostly uniform. It lacks flexibility in notation and demonstrates many of the same problems as in the previous example.

```
(define (odds n)
        (map (lambda (x) (+ (* x 2) 1))
             (iota n)))

; returns all odd factors of 3 from between 0 and 40
(filter (lambda (x) (= (mod x 3) 0)) (odds 20))
```

```
def odds(n):
    return [ x * 2 + 1   for x in range(n) ]

[x   for x in odds(20)   if x % 3 == 0]
```

1.3.3 Alternative syntaxes for Lisp

Despite its history, there is no reason why Lisp's code can only be rendered in its own trademark symbolic expressions, or more generally why a written syntax must respect the genericity of the syntax tree it represents.

The questions of how to go about improving or replacing Lisp's syntax, what alternative should supplement or supplant it, and whether such change would really be an improvement at all, have come up several times in the history of the Lisp family [Whe, page "Rationale"]. Most are proposals aimed at improving S-expressions generally, although a few extended relatives in the Lisp family have fully adopted a more standard written syntax [4].

M-expressions

During the development of Lisp, S-expressions were initially considered as a mere transitional representation for programs that would one day be written in a notation modeled after FORTRAN, called *M-expressions*.

But at some point in Lisp's development, its programmers found that using S-expressions directly had some advantages. As McCarthy describes,

> a new generation of programmers appeared who preferred internal notation to any FORTRAN-like or ALGOL-like notation that could be devised. [McC79]

By the time Landin introduced the original offside rule, M-expressions were considered only *an intermediate result in hand-preparing LISP programs* [Lan66]. In the time since, the S-expression has retained its status as *the* written syntax for languages in the Lisp family.

Gabriel and Steele [SG93] speculate that because manipulating Lisp code is central to Lisp programming, alternative syntaxes have not caught on precisely because they don't express the internal syntactic structure, i.e. they are not homoiconic. Yet, they also speculate that inventing and implementing new syntaxes will always be an enticing open problem because of how easy Lisp makes this Indeed, several projects have attempted to provide a better alternative. The most complete such attempt comes from the *Readable S-expressions* project.

Readable S-expressions

One recent attempt to improve the written representation of Lisp code comes from the "Readable Lisp S-expressions" project [Whe], which defines a new syntax for Lisp, called "sweet expressions". It adds rudimentary support for infix notation, prefix notation, and grouping via indentation.

The authors of the project are keenly aware of the uniqueness of S-expressions and offer a detailed rationale for each particular syntactic extension they propose, as well as a comprehensive rebuttal to the idea that Lisp's syntax is fundamentally unimprovable. The stated objective of the project is to increase the readability of Lisp with alternatives to S-expressions that retain their genericity and homoiconicity.

To illustrate, let us rewrite the two examples of the previous section using sweet expressions.

[4] `http://opendylan.org/`

```
define odds(n)
    map
        lambda (x) {{x * 2} + 1}
        iota(n)

; returns all odd factors of 3 from between 0 and 40
filter
    lambda (x) {{x mod 3} = 0}
    odds(20)
```

```
defun f(a1, a2)
    let ((x 2) (y 3))
        {{a1 * x} + g(a2, y)}
```

Chapter 2

Parsing with Extensibility in Mind

In this chapter we will overview several grammar-based parsing techniques, and discuss some issues that these methods suffer from.

2.1 A brief review of parsing methods

A context-free grammar (CFG) is a set of terminal symbols T, nonterminal symbols N of which one is specified as the start symbol S, and derivation rules $A \to \alpha$ where $A \in N$ and $\alpha \in (T \cup N)^*$. If $A \to \alpha$, we can say A *derives* or *generates* α. Rules $A \to \alpha$, $A \to \beta$, etc. can be written at once as $A \to \alpha \mid \beta \mid ...$ A *derivation* is starts from a sequence of terminals and nonterminals and at each step replaces one nonterminal by applying a rule. For example, if we have a rule $A \to xB$, one step in a derivation is to rewrite $wACz$ as $wxBCz$, where the A has been replaced. We are interested in derivations starting with the start symbol and ending with *sentences*, sequences containing only terminal symbols. Such a derivation can be represented as a tree, called a *parse tree* — the leaves are terminals and the intermediate nodes are non-terminals.

The language generated by a given grammar consists of all the sequences of terminal symbols that can be generated from the start symbol. The goal of a parser is, for a given grammar, when given a sequence of terminal symbols, find a parse tree representing its derivation of the sequence by the grammar, or signal when such a derivation does not exist.

2.1.1 Deterministic linear-time parsing

The two main traditions in programming language parsing originate in methods developed in the 1960s. These are deterministic methods, in that the languages they recognize can be generated by unambiguous grammars.

LL

LL parsers start at the left side of the input and progress rightwards, building a tree for the sentence from the top down. The "top" is the start symbol of

the grammar, and the parser's goal is to find a derivation of the sequence of terminals it reads as input.

At a nonterminal A, the parser has to consider the possible rules $A \to \alpha$ that could be applied. An LL parser must be able to decide which rule to apply based on the symbol being read. To apply rule $A \to xBy$, for example, the parser reads an input symbol and compares it with x. If that matches, it tries to match B using the procedure available for B. Finally, if the procedure for B is found to match at that point in the input, the parser reads one more symbol from the input and compares with y. Failure to match is signaled to the calling procedure, and success creates a new intermediate node in the parse tree. If the start rule fails to match, then the input is rejected, otherwise we have completed a derivation.

Clearly such a method is unable to parse left-recursive rules, such as $A \to Ax$. More generally, the parser may encounter situations where it cannot conclusively determine which rule needs to be invoked. For example in the two rules $A \to B|C$, where $B \to xyB'$ and $C \to xzC'$, it cannot be determined just by reading the next symbol, whether to invoke the procedure for B or for C (though the grammar may possibly be refactored to make this decision possible). The deterministic LL parsers used in parsing programming languages cannot make guesses, but they can be provided with a certain amount of *lookahead*, most commonly one symbol. The *lookahead* is the number of unconsumed symbols based on which the parser can decide which rule to apply. In the example above, a lookahead of 2 would allow the parser to decide between B and C.

A grammar that can be parsed by an LL parser with one symbol of lookahead is said to be an $LL(1)$ grammar; more generally, an $LL(k)$ grammar can be recognized with k symbols of lookahead. $LL(k)$ also refers to a parser with this much lookahead. The LL parsing method is a *recursive descent* algorithm: the recognition process is separated into procedures for each nonterminal production. During the parse, these procedures will recursively call each other to parse the input. This makes the method relatively easy to understand and implement by hand.

However, not all context-free languages are representable as an $LL(1)$ (or even LL) grammar [Ros70]. Even when such a grammar does exist for a language, finding it can be difficult.

LR

The original LR algorithm (along with the minimalist abbreviation scheme) was given by Knuth in 1965 [Knu65]. An LR parser moves through the elements in its input from left to right, gradually filling out the parse tree structure of the input. Unlike LL, the LR parsers operate "bottom-up": smaller syntactic structures are identified first, these isolated subtrees are then grouped into successively larger trees, until the parser reaches the last symbol and completes the tree (if at all possible).

The algorithm does this by making a single pass through the input, and taking constant time at each step, hence being able to parse the input in linear time. Formally, the parser is a pushdown automaton that maintains a stack of symbols. Based on its internal state, and on a lookahead window over the input, the automaton must decide whether to *shift* — pushing the next input symbol onto the stack — or to *reduce*: replacing the symbols on top of the stack with

a single nonterminal symbol (which adds a new intermediate node to the parse tree).

This strategy can recognize strictly more grammars than the LL method, but it still cannot recognize all unambiguous grammars, much less all context-free grammars. As in the case of LL grammars, even when an LR grammar exist it may be difficult to produce.

2.1.2 Parsing Expression Grammars

Parsing Expression Grammars (PEGs) were introduced by [For04] where they are presented as *an alternative, recognition-based formal foundation for language syntax*. They are similar in appearance to CFGs: a PEG is defined by a sets of terminals, nonterminals, and rules, and can be written in a EBNF-like format. Unlike a traditional grammar, however, the rules of a PEG are not productions; instead, they are interpreted as instructions for recognizing symbols in the grammar.

The most salient contrast between PEGs and CFGs is that each non-terminal prioritizes the rules associated with it in a given order. Following [For04], let us replace the notation `|`, used for CFGs above, and separate the different rules in a PEG by the symbol `/`, in decreasing priority.

Then the difference between the CFG rule

$$A \to \texttt{xx} \mid \texttt{x}$$

and the PEG rule

$$A \leftarrow \texttt{xx} \;/\; \texttt{x}$$

is that, in the CFG, there is no precedence attached to the nonterminal A — so there are two possible derivations for the sequence "xx"; by contrast, the PEG rule tells us explicitly to try to find two consecutive `x`s, and then if this fails to find a single `x` — for "xx" the first case succeeds.

More generally, a PEG rule $A \leftarrow \alpha_1 \;/\; \ldots \;/\; \alpha_k$ is parsed by attempting to match each α_i in order; terminals in α_i are matched by direct comparison with the input, and non-terminals are matched by calling their associated matching procedure. This gives a recursive-descent procedure for recognizing any PEG, but a naive implementation may result in a parser that requires too much time to recognize the language. Intuitively, the rule $A \leftarrow B \;/\; C$ can be described as "attempt to match B, and if this fails, attempt to match C". But determining that B does not match at the current position might require parsing a good amount of the input, far more than a few symbols of lookahead can provide. Some grammars would require exponential time to be recognized in this naive manner.

But actually, a method called *packrat parsing* can, for PEG with m rules, parse a sequence of n symbols in $\mathcal{O}(mn)$ time, and using $\mathcal{O}(mn)$ space. The method is simple: the outcome of attempting to match each rule at each position of the input is memoized, and this ensures that at most $\mathcal{O}(m)$ steps of computation are spent at each position.

2.1.3 Generalized parsing

The idea of the Generalized LR parser was already described in 1974 [Lan74], but was not presented in full detail until a decade later [Tom85]. Generalized

versions of LL parsers were only developed relatively recently [SJ10; SJ13].

The GLL and GLR algorithms both share two ideas: they generalize the stack structure to a graph structure, and they remember the outcome of previous parsing steps, so they do not need to be executed more than once.

In the case of GLL, it is the *call-stack* which is generalized to a *call-graph*: each node in the graph corresponds to a pair $u = (L, k)$, where L is an execution point in one of the parsing subroutines (we may think of it as a *goto* label), and k is a position in the input stream. There will be an edge from (L', k') to (L, k) if executing from L at position k resulted in a call to L' at position k'.

The memoization is a set of triples (L, k, k'), where L is an execution point that is known to return a successful match (of its associated terminal or non-terminal) ending at position k', if it begins execution at position k. This is stored to quickly resolve any later calls to L at position k. In this aspect it is similar to the packrat parsing algorithm outlined in section 2.1.2, with the difference that triples (L, k, k') can appear for more than a single value of k'.

The parsing is then carried out in an *asynchronous* fashion. In an LL parser, the rule $A \to Bc$, invoked at position k, would result in a direct call to the procedure for B at k, followed by scanning the input to check for c at whichever position B returned.

The issue is that now procedure B can return multiple times, as there can be more than one match for B at position k. So the procedure for A is broken into two execution points L_{A_0} and L_{A_1}. If we enter point L_{A_0} at input position k, before we request procedure B to be executed at k, we add an edge from (L_{A_1}, k) to (L_{B_0}, k), and schedule an execution of L_{B_0} which, upon a successful match, should itself schedule execution to continue at L_{A_1}, starting at the position after the last consumed input symbol. Then every time that B returns a match, execution will proceed from point L_{A_1}, where we try to match the input against non-terminal c; if this match is successful, then A must schedule its own caller to continue its execution; a successful match will be memoized, so that any future calls to A at the same position will be immediately answered.

Generalized LR and LL are capable of parsing all context-free languages, much like the classic CFG parsing algorithms used for parsing natural languages (e.g. CYK, Earley). These algorithms are not linear-time in the worst case, and any $\mathcal{O}(n^{3-\varepsilon})$-time algorithm for CFL parsing, would give a $\mathcal{O}(n^{3-\frac{\varepsilon}{3}})$-time algorithm for for boolean matrix multiplication [Lee02].[1] But since the "generalized" machinery only kicks in when the parser encounters an ambiguous state, for grammars that are *mostly* in LR or LL, the runtime of a GLR or GLL parser will be acceptably close to linear.

2.1.4 Lisp's reader algorithm

Lisp's approach to parsing differs significantly from the grammar-based methods described so far. Instead of formally defining the Lisp language and providing this definition to a compatible parsing algorithm, the algorithm itself, the *Lisp reader*, is defined explicitly.

A good source for learning about this algorithm is Guy Steele's book "Common Lisp, the Language" [Ste90, "22. Input/Output"]. A complete specification

[1] Which suggests that any such algorithms will be based on fast matrix multiplication. It is currently a (vaguely defined) open problem to find a $\mathcal{O}(n^{3-\varepsilon})$-time "combinatorial" algorithm for boolean matrix multiplication, see, e.g. [Yu15].

of the default reader algorithm can be found in the Common Lisp Hyper Spec [Pit96, "2.2 Reader Algorithm"]. Steele describes the reader algorithm in the following terms:

> The reader is organized as a recursive-descent parser. Broadly speaking, the reader operates by reading a character from the input stream and treating it in one of three ways. Whitespace characters serve as separators but are otherwise ignored. Constituent and escape characters are accumulated to make a token, which is then interpreted as a number or symbol. Macro characters trigger the invocation of functions (possibly user-supplied) that can perform arbitrary parsing actions, including recursive invocation of the reader.

As mentioned in section 1.2.3, the reader algorithm keeps track of character types using a readtable. Common Lisp classifies characters as one of six types: *macro*, *constituent*, *single escape*, *multiple escape*, *whitespace*, and *invalid*. For each macro character, the readtable keeps an associated macro function.

The reader constructs objects of Lisp code directly. Constituent and escape characters are grouped together and interpreted as symbols or numbers, and macro characters delegate the parsing of strings, compound-forms, and other objects, to the corresponding reader macro function. This function can freely process the input, and is not restricted to having a specific form or behavior (as in the case of the grammar-based methods we have seen).

2.2 The benefits and disadvantages of each method

We now revisit the various parsing methods in light of the criteria described in section 1.1.

2.2.1 LL and LR

LL methods are simple to understand and implement by hand in many cases, but are not general, as they are unable to parse many natural syntactic structures. **LR** methods are less restrictive, but only marginally, and it is difficult to follow their execution.

These nongeneralized grammar-based methods make it very difficult or impossible to find a natural specification of a language that has a form compatible with the parser. Often a non-LL/LR grammar can be factored into an equivalent grammar that is LL/LR, but this process distorts the structure of the grammar, and by the time a grammar is factorized into compliance, it may barely resemble its original form.

Furthermore, even small changes to the language can require a disproportionately large refactoring of the grammar. The effect of small changes to the grammar, in turn, can be non-obvious: the Python Developer's Guide, for example, reports that "it took well over a year before someone noticed that adding the floor division operator (//) broke the parser module." [2], and now recommends a carefully curated process to be followed when making any changes to the grammar. So LL/LR methods are not compositional, modular or simple to extend.

[2] `https://docs.python.org/devguide/grammar.html`

However, both algorithms are are very fast.

2.2.2 GLR and GLL

GLR/GLL parsers run in linear time on LR/LL grammars, and the algorithm can be optimized to run fast for programming language grammars [MN04] (which are "mostly" LR/LL). However there is no formal guarantee on the runtime, and there is always a chance that clarity of the grammar must be sacrificed if more speed is needed.

The algorithms are significantly more complicated than their nongeneralized analogues. Although GLL is presented as an improvement on GLR in this respect — combining the power of generalized techniques and the intuitiveness of top-down parsing — it is still not a simple algorithm.

The generalized methods are strongly compositional. Because unambiguity is not preserved by grammatical union (or even a decidable property of context-free grammars), combining two grammars must be done carefully. If we want to include Grammar 2 within Grammar 1, we may do so while preserving non-ambiguity by enforcing a non-ambiguous way of delimiting the region where Grammar 2 should be used. Some GLR/GLL-based parsing tools have a module system: an SDF module, for example, can contain grammar rules that add productions from nonterminals in an existing grammar, and full grammars can be written in a modular fashion[3].

It is also worth noting that general CFG parsers do not fully satisfy the generality criterion, since some programming languages are in fact not context-free. One approach to parsing context-sensitive syntactic constructs is to extend CFGs with additional expressive power.[4] Using such a system can also allow using natural grammars that would normally be ambiguous. However, the algorithms for parsing an extended CFG are no longer guaranteed to be $\mathcal{O}(n^3)$.

2.2.3 PEGs

The packrat parsing algorithm runs in time linear on the length of the input, though the linear factor is dependent on the number of grammar rules. The algorithm also uses a linear amount of space, which would have been prohibitive when the LR algorithm was invented, but which is considered OK nowadays, when memory availability far surpasses the size of the files one typically wishes to parse.

The algorithm is simple to understand, and its behavior is very directly prescribed by the corresponding grammar rules.

The class of PEG-parsable languages has been shown to be incomparable with context-free languages, i.e., there are context-free languages that cannot be parsed by PEGs, and vice-versa [For04; Lee02].

Because we can modify the rules of a PEG without fear of rendering the grammar unambiguous or unparsable in linear-time, it is easy to implement

[3] `http://www.meta-environment.org/doc/books/syntax/sdf/sdf.html`

[4] To parse indentation in programming languages, for example, Adams [Ada13] captures block structure by annotating nonterminals, and Afroozeh and Izmaylova [AI15] use more general data-dependent grammars that add arbitrary constraints to productions.

a module system for PEG parsers[5]. Extending a PEG in this way is usually a straightforward process, although some care must be taken so that a given extension does not interfere with another (ensuring, for example, that no higher-precedence rule matches a prefix of a lower-precedence one).

PEGs have been used as a formalism for several projects providing modular syntax, most notably the (now defunct) Fortress programming language[6].

2.2.4 Lisp's readtable-macro algorithm

The Lisp reader algorithm, like other recursive-descent methods, is simple. The parser is written in Lisp and extensible in Lisp, and is simple to extend.

The algorithm is intrinsically modular; importing syntax consists of add new reader macros and characters to the readtable, which can be easily done in Lisp with built-in functions. Changes in the readtable takes effect instantly, meaning the syntax can be modified on-the-fly by reader macros or within a REPL.

Because reader macros can use arbitrary Lisp computation, the algorithm is as general as possible. It is also compositional, in the sense that any reader macro can be used as a component of different readtables, and will not interfere with the way the rest of the readtable is defined. It also allows for a modular mechanism for extending the parser. However, although the parser can be changed in arbitrary ways, some parsing features do not fit well with the readtable-macro parsing algorithm. So it would not be easy or natural to extend the parser with features like infix operators, custom operator syntax, or indentation. And so the resulting written syntax is notoriously unreadable.

2.2.5 A feature-comparison chart

The analysis above is summarized in the following table:

	LL	LR	PEG	GLR/GLL	Lisp RM
simplicity of algorithm	+	-	+	-	+
simplicity of extension	-	-	+	+	++
modularity	-	-	+	++	+
compositionality	-	-	+	++	+
generality	-	-	-	+	++
readability	+	+	+	+	-
speed/efficiency	++	++	+	-	+

Table 2.1: The relative strengths and weaknesses of the discussed methods.

[5] And in fact has been done, for example in the *Rats!* parsing library (`https://cs.nyu.edu/rgrimm/xtc/rats-intro.html`).

[6] `https://projectfortress.java.net/`

2.3 Transducer chains in Natural Language Processing

One approach that has been applied to several problems in Natural Language Processing (NLP) is to decompose complex transformations into a series of simpler ones.

The essential notion of a "simple transformation" can be captured by a *finite-state transducer* (FST). The traditional FST is an automata that transforms strings into strings: the machine has a tape and a head, and its rules are mappings of states and local symbols on the tape to states and actions (move tape head, write symbol). The rules are such that the tape head only moves in one direction, meaning the transducer processes the input and writes back output in one pass over the tape.

Finite-state transducers have found various applications in NLP, including morphological and lexical analysis [Kar01], but have not seen wide adoption in natural-language syntactic processing.

A classic example due to Chomsky [Cho57, Ch. 3] illustrates that English is not a regular language, in that it cannot be recognized by a finite-state automaton. The problem is that ambiguity is ubiquitous in natural-language syntax [see MS99, p.410, for an illustrative example], but finite-automata based methods are too weak to express it. Context-free grammars, on the other hand, are an excellent formalism for capturing ambiguity.

Chains of FSTs, called *cascades* in the NLP literature, have also been successfully applied to several problems across NLP (for some examples, see [FM04, §3]). Since FSTs are closed under composition, chaining two or more together does not result in increased transduction power. However, cascades facilitate breaking down complicated problems into discrete subtasks or levels. Taken together, the outputs at each point in the chain can contain more information than the final output of the chain by itself. If the chain is constructed appropriately, this can allow for "an efficient, systematic framework for characterizing the relationships between different levels of analysis" [Woo80].

This approach has been successfully applied to syntax tree construction. The CASS parser of [Abn96] constructs a syntax tree for an input string deterministically, by using a cascade of transducers to first "chunk" phrasal constituents together at a low level and then group these chunks together into clauses and sentences. Chunking is central to the philosophy underlying Abney's parser. The parsing method we introduce in Chapter 3 is motivated largely by the same principle, as we will see.

Finite-state *string* transducers can be generalized to *finite-state tree transducers* (FSTTs). Instead specifying moving and writing on tapes, the movement pattern of an FSTT through an input tree is fixed. The format of rules varies between different types of FSTTs.

The original *top-down tree transducers* introduced by Rounds [Rou70] and Thatcher [Tha70], for instance, start at the root node and work their way downwards; at every node N, they apply some rule R according to their current state and N[7], which may rearrange the subtrees immediately below the current node, and then recursively process each subtree rooted at the children of N (the start-

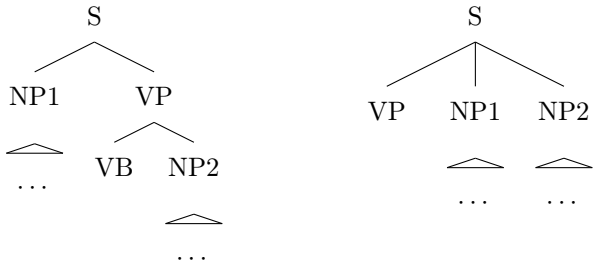
[7] Also common are "extended" top-down tree transducers, that can additionally use nearby descendant nodes in the tree in selecting rules to apply.

ing states for these recursive calls are also specified by the rule R). Although we will not describe this model of tree transducers in formal detail, the following example illustrates the general idea. An example of a rule is:

$$\text{q A}(x_0, x_1) \quad \rightarrow \quad \text{B}(\text{q } x_0, \text{C}(\text{r } x_1)).$$

This rule specifies that, if we are in state q over a node A with exactly two children, we transform the A to a B, create a new node C between B and x_1, and then recursively process x_0 in state q and x_1 in state r. We may write such a rule by drawing two trees; the tree on the left is transformed into the tree on the right:

Because they can transform trees at various depths, tree transducers are able to process syntactic structures that their string-based counterparts cannot. As a result, they have found use in areas where their additional "syntax-based" power is needed. Transformations in machine translation, for example, can often require high-level reorganization of phrases in the input: when translating from English to Arabic, the subject-verb-object word order must be transformed to a verb-subject-object form; whereas string transducers are not suited for this, the transformation can be specified naturally using a FSTT rule as shown in this example from [KG05]:



Although most work on FSTTs has focused on single transducers and on important classes of FSTTs and their properties, cascades of FSTTs have been studied and applied to some NLP tasks. Yamada and Knight [YK01] present a statistical machine translation model for transforming syntax trees in English to Japanese using a 3-transducer cascade of FSTTs. The first level reorders the tree structure, the second level inserts new words, and the third level performs a direct translation at the leaves.

To our knowledge, transducer cascades have not thus far been applied to the problem of parsing programming languages. Research on tree transducer cascades has been almost exclusively concerned with so-called "tree-to-tree" and "tree-to-string" models, as opposed to "string-to-tree" problems like parsing.

Programming language parsing, on the other hand, has focused on methods based on pushdown automata and recursive descent.

It is worth asking what properties a transducer-chain parser for programming languages might have, and what it might look like in practice. In particular, how would transducer-chain parsing compare with the methods surveyed above, under the criteria laid out in §1.1? There are a few plausible reasons why such a method might be extensible:

- The parser would only be as complicated as the individual transducers comprising the chain, so if the transformations can be made simple enough, the algorithm will remain simple.
- The algorithm is modular by design.
- Transducer cascades have been used to parse and translate natural language, which strongly suggests they are powerful enough to parse a readable programming language syntax.

In the next chapter, we present a parsing method for programming languages based on chains of transducers. Chapter 4 will discuss an implementation of the method in Python, and Chapter 5 will conclude the thesis by evaluating the merits and flaws of the method.

Chapter 3

RMTC parsing

The *readtable-macro transducer-chain* method parses written code in two stages. This separation is similar in form and function to the lexer/parser paradigm used by traditional parsing methods, but the stages themselves work quite differently.

The first stage, the *readtable-macro parser*, functions as the lexer component of our compiler[1]. Although the original readtable-macro algorithm is a full-fledged recursive descent parser, our version constructs and returns sequences of tokens instead of syntax trees (which is what Lisp's reader returns).

The output of this stage is a sequence of tokens which is passed to the next stage. This sequence can be seen as the leaves of a two-level tree. Then the second stage, the *transducer-chain parser*, consists of a sequence of simple transformations on trees: a chain of tree transducers. The parser sequentially applies each transducer in the chain, to progressively modify the tree to the desired form. The end result is the syntax tree for the given input.

In section 3.1 we describe our readtable-macro parsing algorithm, which extends the original reader algorithm of Lisp with support for indentation. In section 3.2 we propose a formal model for representing the kind of transformations we are interested in, and give some basic theoretical results for it. We then discuss how the formal model provides a foundation for a transducer-chain parsing method.

3.1 Readtable-Macro Parsing

The Readtable-Macro parsing algorithm we provide here is an indentation-sensitive adaptation of the default reader algorithm of Common Lisp.

To get an idea of how our extended algorithm works, let us imagine we are parsing the nested indentation structure of a given piece of written code. We do this by using a nested chain of coroutines, where each coroutine corresponds to a single indentation level. The top-level coroutine *reads* characters directly from the input and *emits* tokens to the output. Below the top level, each coroutine reads characters from its parent and emits tokens back to that parent. Each coroutine also *yields* characters to its children and *captures* tokens emitted by those children, passing them along to the next level up.

[1] We will frequently refer to our readtable-macro parser as a *tokenizer* in light of this fact.

The flow of characters is such that each coroutine should only be able to read characters appearing at or after the column corresponding to its indentation level, and should only yield those characters to its children which appear at or after *their* indentation level. The following image illustrates the setup.

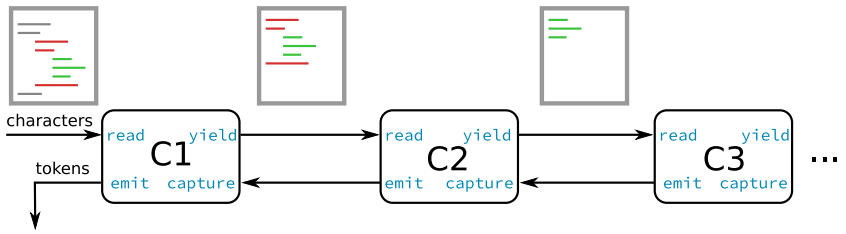


Figure 3.1: The extended readtable–macro parser.

The implementation must be done carefully, because passing characters and tokens up and down the chain — through every coroutine — would be very inefficient. While the indentation level visible by children is at the discretion of the parent coroutine, the processing and bookkeeping that makes this work should be done by a *stream* object, which is passed to the child when it is first invoked.

A stack of column numbers corresponding to indentation levels is kept by the stream object. When a coroutine reads from this stream object, the only characters that are returned to it are those characters which appear at or after the indentation level on top of the stack.

After every newline, the stream object throws away all whitespace up to the current indentation level. Any subsequent character is visible to the invoking coroutine. This coroutine may then decide to *push a new indentation level onto the stream*, i.e., onto the stack of indentation levels held by the stream object. When the stream encounters a non-whitespace character appearing before the current indentation level, it will signal an `EOF` in response to any read request. Presumably, at this point, the subroutine will return, and its parent can pop the indentation stack, to the point when the stream will again return input characters.

By using the stream as a kind of *whitespace filter*, we are then able to extend the original reader algorithm in a way that preserves its overall structure. The reader need only compare the two algorithms to find their obvious overlap.

3.1.1 The readtable

Similarly to the Lisp reader algorithm, we think of the input as organized into sequences of one or more characters belonging to one of several types. For our purposes, we use the following set of sequence types: WHITESPACE, NEWLINE, MACRO, CLOSING, CONSTITUENT, ISOLATED_CONSTITUENT, and PUNCTUATION.

The character sequences, their types, and any additional information associated with a sequence are all stored in a data structure called a *readtable*. For

a given readtable and a given input stream, *reading a readtable sequence from the stream* is the process of reading one character at a time from the input and identifying the longest contiguous sequence appearing in the readtable. The Readtable-Macro algorithm will then read entire *readtable sequences* at a time instead of single characters.

The contents of the readtable have great influence on the parsing procedure, can be changed dynamically during parsing, and will depend on the language one wishes to parse.

MACRO sequences are associated, in the readtable, with reader macro functions. As we will describe below, when the readtable-macro algorithm finds a macro sequence, it will delegate further reading of the input to the corresponding macro function.

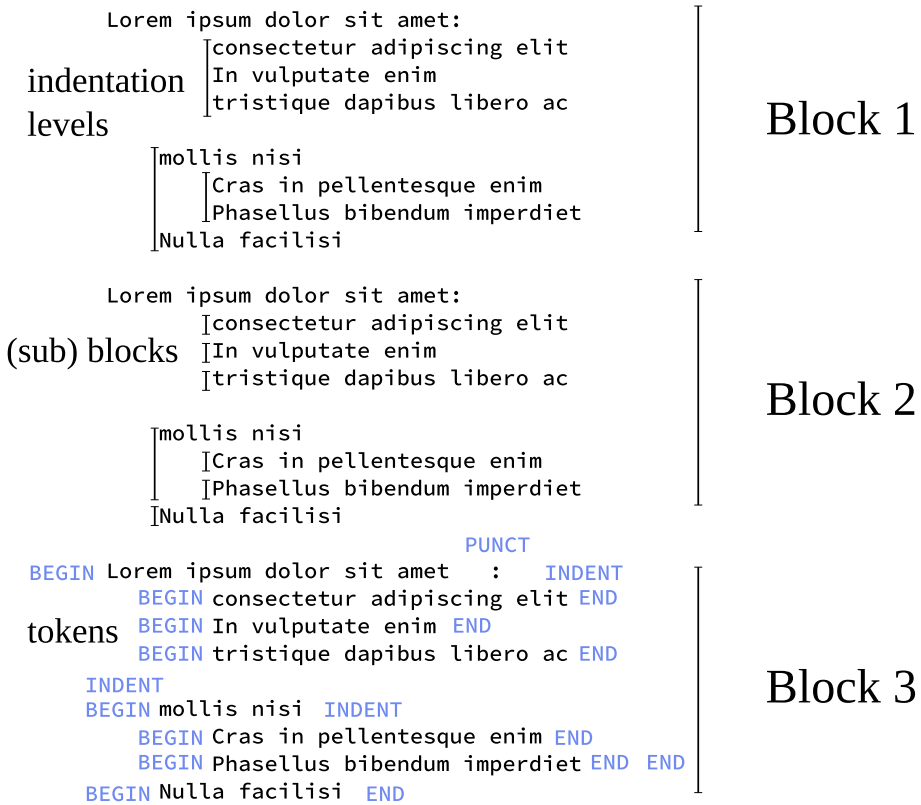
3.1.2 Output

The output of the Readtable-Macro algorithm consists of a sequence of tokens. The specific types are not important — it is not necessary to include a specific type for punctuation, for instance, and other token types (`STRING`, `COMMENT`, etc.) can be created for use by certain read macros.

The Readtable-Macro algorithm will make use of the following token types:

- `CONSTITUENT`, an accumulation of constituent characters.
- `BEGIN` and `END`, mark the beginning and the end of code blocks.
- `PUNCTUATION`, a punctuation token.
- `INDENT`, marks a change in indentation level within a block.

Instead of a long and unintelligible description, we will illustrate the meaning of each token in Figure 3.2, as well as the meaning of *block* and *indentation level*.




```
Lorem ipsum dolor sit amet:
    consectetur adipiscing elit
    In vulputate enim
    tristique dapibus libero ac

    mollis nisi
        Cras in pellentesque enim
        Phasellus bibendum imperdiet
    Nulla facilisi
```

```
Lorem ipsum dolor sit amet:
    consectetur adipiscing elit
    In vulputate enim
    tristique dapibus libero ac

    mollis nisi
        Cras in pellentesque enim
        Phasellus bibendum imperdiet
    Nulla facilisi
```

```
                                PUNCT
BEGIN Lorem ipsum dolor sit amet   :   INDENT
    BEGIN consectetur adipiscing elit END
    BEGIN In vulputate enim END
    BEGIN tristique dapibus libero ac END
INDENT
BEGIN mollis nisi  INDENT
    BEGIN Cras in pellentesque enim END
    BEGIN Phasellus bibendum imperdiet END  END
BEGIN Nulla facilisi  END
END
```

Figure 3.2: Blocks, indentation levels, and a sample output.

3.1.3 The Readtable-Macro algorithm

The readtable-macro algorithm parses one block at a time in three distinct stages. In **Stage 0**, it finds the start of the next block, **Stage 1** tokenizes the first line of the block, and **Stage 2** parses the indented levels within the current block by recursively calling itself.

The Readtable–Macro algorithm (with indentation)

All three stages below loop until the procedure returns. Each loop of the three stages parses a single block.

Stage 0. Advance to the next block

- Read one readtable sequence from the stream at a time, until a non-whitespace sequence is encountered.
- If the first (non-whitespace) sequence is `EOF` or a CLOSING sequence, unread it back to the stream and return.
- Otherwise, unread the first (nw) sequence, and emit a `BEGIN` token.
- If the first (nw) sequence did not appear in the first column, signal an error (unexpected indentation).
- Continue to **Stage 1**.

Stage 1. Parse the first line

Read one readtable sequence from the stream at a time. Depending on the sequence's type, select one of the following cases:

1.1 If the `EOF` was reached emit an `END` token and exit.

1.2 If a CLOSING sequence was read, it is unread, then an `END` token is emitted and the procedure exits.

1.3 WHITESPACE is ignored.

1.4 A NEWLINE causes us to break the loop and go to **Stage 2**.

1.5 A CONSTITUENT sequence causes us to read additional sequences from the stream until we find one that is not a CONSTITUENT. These are concatenated and a `CONSTITUENT` token is emitted.

1.6 An ISOLATED_CONSTITUENT sequence causes us to emit a `CONSTITUENT` token containing only the characters in the read sequence.

1.7 Seeing a PUNCTUATION sequence causes us to emit the corresponding `PUNCTUATION` token.

1.8 A MACRO sequence triggers a recursive call. The readtable associates a reader macro function with each MACRO sequence. We call this reader macro function on the current stream. We forward all the tokens emitted by the reader macro function.

Stage 2. Offside content

We loop the following procedure repeatedly. Each loop parses an indented sub-level.

2.1 Initialize i to $+\infty$; i will hold the current indentation level.

2.2 Read one readtable sequence from the stream at a time, until a non-whitespace sequence is encountered.

2.3 If the `EOF` is reached instead, emit an `END` token and return.

2.4 Otherwise, let s be the first non-whitespace sequence, and let c be the column number of the first character in s; we begin by unreading s. Then:

- 2.4.1 If c is 1, emit an `END` token and break to **Stage 0**.
- 2.4.2 If $c > i$, signal an error... unless we found a CLOSING sequence, in which case we emit an `END` token and return.
- 2.4.3 If $c < i$, emit an `INDENT` token and set c to i.

2.5 At this point $c = i > 1$. We then:

- 2.5.1 Push an indentation level onto the stream at the current position.
- 2.5.2 Recursively call this procedure on the indented input stream, forwarding all emitted tokens. The call must return when the indentation level goes below c, because the indented stream will then signal the `EOF` to the recursively called procedure.
- 2.5.3 Pop the indentation level from the stream, and continue the **Stage 2** loop.

3.2 Parsing with transducers

After tokenization, the resulting sequence of tokens is transformed into a syntax tree by a chain of transducers. Each transducer in the chain performs a simple transformation, similarly to the transducer cascades described in section 2.3.

Our method organizes the blocks and indentation levels into subtrees, and then the various infix operators and special notations are recognized and processed within those subtrees. Parsing this way is similar to how humans parse code themselves: it reflects the process of first recognizing the overall structure of the code first, and then breaking it down locally.

Remarkably, although we devised this parsing strategy independently, a similar strategy has been used to parse the syntactic structures of natural language. Abney's conception of "chunking" [Abn91] is based on the intuition that humans read natural language by identifying small segments of phrases first, and then fitting these segments together from the bottom up. He slightly refines this idea in describing the strategy used by the transducer-chain parser.

> The philosophy is *easy-first parsing* — we make the easy calls first, whittling away at the harder decisions in the process. [...] Easy-first parsing means that we do not build a parse tree systematically

> from bottom to top, but rather recognize those features of structure that we can. Where reliable markers for high level boundaries are recognized, uncertain intermediate level structure can be skipped over [...]

Our transducer-chain parser operates in this way. For programming languages, high-level structure is almost always explicitly marked (by indented blocks and pairs of delimiters), and can be processed first. After this high-level structure is known, the remaining structure can be parsed easily in decreasing order of precedence.

In §4.2.2, we describe how a chain of transducers is put together for parsing a programming language similar to Python. In this section, we are instead interested in giving a precise, formal definition of a machine model to process labeled graphs, which will serve as our conceptual guideline for the kind of operations that can be performed at each level in the chain.

The model can be described roughly as a Turing machine where the tape is replaced by a graph structure. On top of this, we impose a simple restriction on the behavior of the machines. From this restriction it will follow that any fixed chain of (restricted) transformations can be simulated by a random-access machine in linear-time. We then give some theorems and conjectures regarding the computational power of the restricted model. The next section (3.3) will show how a chain of these machines can be used for parsing a toy language (a more complicated chain will be given in §4.2.2).

3.2.1 Finite pointer machines

The model we will propose is a generalization of Turing Machines, with the following modifications:

- The tape, which can be thought of as a graph with a linear structure, is replaced with a general bounded-degree graph.
- The machine has multiple heads pointing to nodes in the graph, which can move around along the graph's edges.
- The instructions can not only change the symbols written at each graph node, they can also change the structure of the graph.

Formally, a *finite pointer machine*[2] (FPM) is specified by a tuple $(Q, q_0, q_f, \Sigma, k, d, \delta)$, where:

- Q is a set of states,
- $q_0 \in Q$ is the initial state,
- $q_f \in Q$ is the final state,
- Σ is a finite alphabet,
- $k \in \mathbb{N}$ is the number of pointers[3],
- $D \in \mathbb{N}$ is a maximum degree, and

[2]The machine is called *finite* because it has a finite control.
[3]Each pointer is analogous to a tape head.

- $\delta : Q \times \Sigma^k \to Q \times$ INSTRUCTIONS is a transition function.

Let G_d denote the class of directed graphs of maximum out-degree $d \leq D$, whose nodes are labeled by symbols in Σ — with the node's edges numbered from 1 to $\leq d$ — and where one node — the *root* — has been singled out. The input to an FPM is a graph in G_d. At any time step, each of the pointers is *pointing* to one node in the graph and can read the symbol stored there.

Let $i, j \in [k]$ denote pointers, and $l, m \in [d]$ denote edge numbers. Then each INSTRUCTION is one among the following:

1. $S(i) := \sigma$, write symbol σ in node i.
2. $i := \mathcal{N}(i, \ell)$, set pointer i to the ℓ-th neighbor of node j.
3. $\mathcal{N}(i, \ell) := \mathcal{N}(j, m)$, point the ℓ-th edge of node i to the m-th neighbor of node j.
4. *new* $\mathcal{N}(i, \ell)$, create a new node u, and point the ℓ-th edge of node i to u.
5. *del* $\mathcal{N}(i, \ell)$, remove the ℓ-th neighbor of node i.

An FPM $\mathcal{M}$ computes a transformation $F : G_d \to G_d$ if for any graph $G \in G_d$, running $\mathcal{M}$ in its initial state with all pointers over the root of G, will cause the computation to halt in the final state with graph $F(G)$, having all pointers pointing to the root of $F(G)$.

3.2.2 The linear-time restriction

We now devise a simple restriction that will ensure that all functions computed by a restricted FPM on a graph G will be computable by a random-access machine (RAM) using time linear in the number of nodes of G.

For a node i, the *offspring* of i are the nodes created in operations of the form $new\mathcal{N}(i, \ell)$. Note that even if the edge from i to the new node is later removed, the node will always be considered to be *offspring* of i.

Each step in an FPM computation is the application of one of the five types of instructions. Each type of instruction involves certain nodes as *participants*. Identifying the nodes that participate in each instruction should be self-explanatory; for example, nodes i and j participate in instructions of the third type ($\mathcal{N}(i, \ell) := \mathcal{N}(j, m)$).

Our restriction will prevent the FPM from being able to spend more than a finite amount of time interacting with each node, or its offspring.

A finite pointer machine $\mathcal{M}$ obeys our restriction if the following holds:

There is some constant $c_{\mathcal{M}}$ such that, on any input given to the machine, the total number of times a given node participates in instructions plus the total number of times its offspring participates in instructions, throughout the entire computation, is at most $c_{\mathcal{M}}$.

We can now define:

Definition 1. *LFPM is the class of transformations computable by some FPM that satisfies our restriction.*

It is not hard to see that the class LFPM is a *syntactic* complexity class, in the following sense:

Theorem 2. *There is an effective enumeration of FPMs satisfying our restriction, such that any LFPM-computable transformation is computed by some machine in the enumeration*[4]*.*

This can be shown by having machines in the enumeration use a working alphabet which encodes a finite counter, and by adding one extra edge in each node to point to an auxiliary node holding the counter (every original node and its offspring will point to the same auxiliary node).

3.2.3 Theorems and conjectures

Let RAM-DTIME(t) denote the class of transformations $F : G_d \to G_d$ which can be computed by a Random-Access Machine in time $O(t(n))$, where n is the number of nodes in the input graph.

Theorem 3. *LFPM* $\subseteq$ *RAM-DTIME(n).*

A RAM can simulate a computation of a given LFPM machine in linear time, as follows. We can store the information for each possible node in a data structure of bounded size. The data structure contains the node's current symbol, and the (finite) list of nodes that it connects to. The machine has a fixed number of pointers. For each pointer, we store the memory position of the data structure for the node to which it is currently pointing. The RAM can simulate each step in constant time — we can check this by verifying for each instruction type. Now clearly the entire computation can be simulated in linear time, as the LFPM executes $\mathcal{O}(n)$ instructions before halting.

Theorem 4. *The class LFPM is closed under composition.*

For any two machines that compute transformations in LFPM, there is a machine that computes the first transformation followed by the second. All of the relevant properties of the two machines are finite. The composed transformation is thus also in LFPM.

Although we did not work out all the details, we are fairly confident that the following is true:

Conjecture 5. *Any transformation $F : G_d \to G_d$ computed in linear time by a multi-tape Turing machine can be computed by an LFPM.*

The broad idea is to first traverse the graph to create a pool of *unused nodes*, which we will use to simulate tape cells. Note that this class includes the LL and LR algorithms.

We also expect something like the following to be true:

Conjecture 6. *RAM-DTIME($\frac{n}{\log n}$)* $\subseteq$ *LFPM.*

The intuition here is that we should be able to use the graph as a sparse array data structure with $\mathcal{O}(\log n)$ update and query time. This data structure will be used to simulate the RAM's memory.

[4]This stands in contrast to *semantic* complexity classes, such as BPP, for which such a computable enumeration does not exist.

3.3 RMTC parsing of a toy language

In this section we will illustrate how the RMTC method would be implemented for the task of parsing a small toy language for simple arithmetic, which we will denote by CA[5].

We begin with an informal description. The language is made of expressions, and blocks of expressions. Expressions can be written on a single line as sequence of numbers, binary operators, and other expressions enclosed in parentheses. For example:

```
(10 + 3) - 4 * (6 + 10) * 5
```

The correct syntax tree for an expression is determined by the usual order of precedence, where parenthesized sub-expressions are grouped first, multiplication and division operations are grouped next from left to right, and finally addition and subtraction operations are grouped, also left to right. Incidentally, this will be the very same strategy used by the transducer chain to parse expressions.

Alternatively, a single operation can also be written using a block notation. We write the operator (the *head*) on the first line, then write its arguments, which can themselves be in block form, on the following lines indented by spaces. For example:

```
-
  10 + 3
  *
    4
    6 + 10
    5
```

Formally, the alphabet for our language consists of the digits 0 through 9, the operator symbols `+`, `-`, `*`, and `/`, `(` and `)`, a single space ␣, and a new-line symbol `\n`. CA is a set of sentences inductively defined as follows:

NUMBER A sentence in CA can be a *number*, which is a sequence of one or more digits.

FACTOR It can also be a *factor*, which is either a number or a sequence '`(`' *e* '`)`', where *e* is an expression (defined below).

TERM A sequence of one or more FACTORs where each pair is separated by a single `*` or `/` symbol is a sentence called a *term*.

EXPR *Expressions* are defined similarly, but as sequences of TERMs separated by `+` and `-` symbols.

[5] for "Calculator Anoky"

BLOCK A *block* is either an expression, or one of the four operator characters, followed by an indented sequence of blocks[6], as in:

```
+
    BLOCK_1
    BLOCK_2
    ...
```

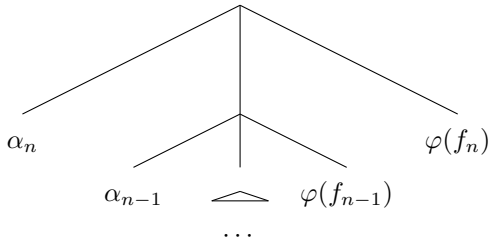
BLOCKSEQ A BLOCKSEQ is a finite sequence of non-indented BLOCKs $b_1 b_2 \ldots$.

3.3.1 Correct parsing of CA

Before we define the readtable and transducer chain for parsing CA, let us define what we mean by a *correct parse* of a CA sentence.

The correct parse $\varphi(s)$ of a sentence s in CA is the labeled tree defined inductively as follows:

- If s is a NUMBER, $\varphi(s)$ is a single node, labeled "NUMBER".
- If a FACTOR is a NUMBER, it should parsed in the same way. If it is a parenthesized EXPR, then $\varphi(\text{`(' } e \text{ `)'}) = \varphi(e)$.
- The correct parse of a TERM, $\varphi(f_0\ \alpha_1\ f_1\ \alpha_2\ \cdots\ \alpha_n\ f_n)$, is the left-associative tree: $(\alpha_n\ (\alpha_{n-1}\ \ldots\ \varphi(f_{n-1}))\ \varphi(f_n))$:



- An EXPR $t_0\ \alpha_1\ t_1\ \alpha_2\ \cdots\ \alpha_n\ t_n$ should be parsed in the same way as a TERM, but with TERMs t_i in place of FACTORs f_i.
- A BLOCK that is an EXPR should be parsed into the same tree.
- An indented BLOCK with operator α and sub-blocks $b_1, b_2, \ldots$ should be parsed into a tree with the operator as the first child and $\varphi(b_1), \varphi(b_2)$, *etc.* are the remaining children.
- A BLOCKSEQ $b_1\ b_2 \ldots$ should be parsed as a tree with children $\varphi(b_1)$, $\varphi(b_2)$, *etc.*

[6]By indented sequence, we mean that each line in each sub-block is preceeded by the same number of ␣ characters.

3.3.2 Defining an RMTC parser

We will now define a parser for CA which uses the readtable-macro transducer-chain method.

We have already described readtable-macro parsing in detail. The default reader of our CA parser uses the extended RM algorithm; we need to specify a set of readtable sequence types and a set of token types, define a readtable, and define any custom reader macro functions.

For the transducer-chain parser, we will need to define the transducers that make up the chain and how each moves through and transforms the parse tree.

The readtable and reader macros

The written code is tokenized with the extended readtable–macro algorithm from §3.1.3. We define a readtable suitable for parsing CA.

sequence	type	macro function
`0`–`9`	MACRO	`NumberTokenizer`
`+`, `-`, `*`, `/`	ISOLATED_CONSTITUENT	
`(`	MACRO	`ExpressionTokenizer`
`)`	CLOSING	
␣	SPACE	
`\n`	NEWLINE	

The `ExpressionTokenizer` reader-macro function first emits a `BEGIN` token, and then executes the **Stage 1** loop from the RM algorithm (§3.1.3 in page 28), with the difference that a CLOSING sequence is checked to see if it is the matching closing parenthesis (otherwise an error is raised).

The `NumberTokenizer` reader-macro function will read sequences until it encounters a sequence that is not a digit. It then concatenates the read digits into a single `NUMBER` token, unreads the terminating non-digit sequence, emits the token, and exits.

Encoding syntax trees in G_6

We will explicitly explain how a syntax tree is represented as a rooted graph of maximum degree 6 (G_6), which clarifies how the transducers we will define below can be implemented by FPMs.

A syntax tree is a type of directed graph with labeled edges. The nodes are labeled using the symbols $\varnothing, +, -, *, /$, NUMBER, BEGIN, END, and INDENT. The edges, or pointers, are labeled *prev*, *next*, *first*, *last*, *parent* and *end*; and we write next(x) for the neighbor of x along its *next* edge. The *children* of a node form a doubly-linked sequence, where each has a *next* and *prev* edge to and from the next node in the sequence. If a node has any children, there is a *first* edge to the first and a *last* edge to the last (which might be the same node), but no other edges. If a node is either the *first* or *last* child of another node, its *parent* edge points to its parent node. The *end* edge will be used to pair BEGIN tokens with their corresponding END tokens.

The syntax tree which is input to the chain

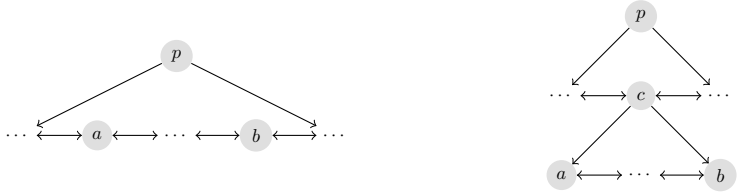
We can then encode the token sequence — which is the input to the transducer chain — as a two-level syntax tree with a root node labeled $\varnothing$ and children corresponding to the tokens emitted by the RM parser. Each child in this tree is labeled with one of the symbols NUMBER, +, -, *, /, BEGIN, END, and INDENT, and each BEGIN token has its *end* edge pointing to its corresponding END token.

We are now ready to describe the different transducers in the chain.

Basic building blocks

Suppose that at some point in our algorithm we have pointers that are pointing to the nodes a, b and c. Then the following operations can be carried out by an FPM in a constant number of steps. We will give some operations in detail, but they should be otherwise self-explanatory.

- *replace* a with b: replaces node a and its sub-tree with node b and *its* sub-tree. In full detail: (1) follow any *pref, next* and *parent* edges in a to find the *next, prev, first* and *last* edges directed at a, and redirect them to b; (2) direct the *prev, next* and *parent* edges of b to the same nodes as the corresponding edges in a, and (3) remove the *prev, next* and *parent* edges from a.
- *remove* a: removes node a, and connects the two adjacent nodes to a, if any, to each other.
- *transpose* a and b: switches two adjacent nodes a and b.
- *wrap* from a to b: create new node c, to be the new parent of all nodes between a and b. As in the following picture:



Top-down transducers defined by an arrangement

The transducers we will define are specified by an *arrangement*. The arrangement defines a condition and a procedure for locally transforming the tree.

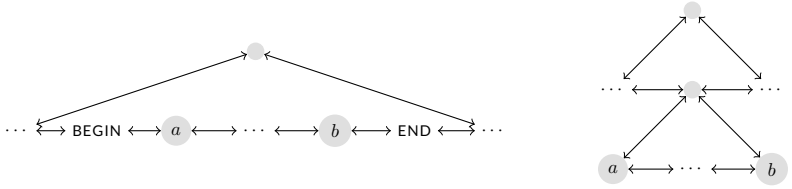
Given an arrangement, the transducer begins at the root node, and traverses the tree top-down, breadth-first, from left to right, as follows. At every traversed node the transducer checks if the associated condition is satisfied at the current node. If the condition is found to hold, then its procedure will be applied to the tree. This procedure will transform the tree, and indicate the node where the scanning of the current level should continue. The transducer proceeds by scanning the current level from that node onwards. After having scanned the level once, it will do it again, this time recursively calling itself on each child.

The transducer chain for CA

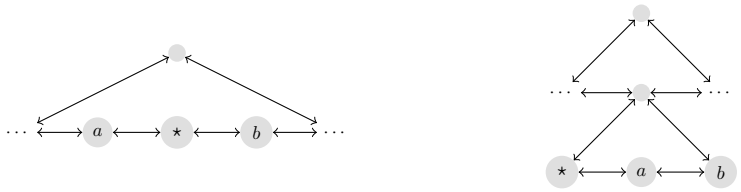
The chain for CA consists of four transducers:

1. PRIMARY
2. MULTIPLICATION
3. ADDITION
4. UNWRAP_EXPRS

The PRIMARY transducer has only one arrangement, which is triggered at every BEGIN node. At such a node, it will remove an INDENT node if one is found at the node after the next node, and wrap everything up to, but excluding, the corresponding END token, and then remove the BEGIN and END. For example:



The MULTIPLICATION transducer has one arrangement: if the current node is labeled with $\star$ or /, wrap prev(x) to next(x) and transpose prev(x) and next(x). The ADDITION transducer works in the same way, with respect to + and -.



The first three transducers process graphs top-down. The final transducer in the chain, UNWRAP_EXPRS, processes trees *bottom-up*. At a node x, a bottom-up transducer performs the same two steps as in the top-down case, but in reverse: the subtrees under each of its children are processed, and then children of x themselves.

We use an UNWRAP_EXPRS transducer to remove superfluous grouping structure in the syntax tree. For example, in written CA, `((((4))))` is a valid sentence, but it should have the same parsed syntax as `(4)`. At a node x, UNWRAP_EXPRS will check if x has exactly one child. If so, it replaces x with its child.

3.3.3 Correctness of the algorithm on valid sentences

The RMTC parser defined above produces some transformation from a given CA sentence into the set of syntax trees. Let ρ be this map. We will show that the parsing algorithm correctly parses every sentence in CA.

Theorem 7. *For every sentence s in CA, $\rho(s) = \varphi(s)$.*

Tokenization

Let $\mathrm{tk}(s)$ be the token sequence obtained by running the tokenizer on the string s. We also define three functions corresponding to the stages of the RM algorithm.

- tk_0 is tk.
- $\mathrm{tk}_1(s)$ is the token sequence resulting of running the algorithm starting at the beginning of the **Stage 1** loop, but not including the `END` token that is emitted when encountering `EOF` (step 1.2).
- $\mathrm{tk}_2(s)$ is the result of running the algorithm on s starting at the beginning of the **Stage 2** loop up to but not including the first `END` token resulting from either `EOF` (step 2.3) or a line starting at the first column (step 2.4.1).

Observation 1. If the RM algorithm is in **Stage 1**, the only situations where it exits the loop are if it reads an `EOF`, a CLOSING sequence, or a NEWLINE sequence. `END` tokens resulting from NEWLINEs and `EOF`s are only found in `BLOCK`s, and if we are parsing the text via the `ExpressionTokenizer` and a CLOSING parenthesis is encountered, then the tokenizer will return control to its parent tokenizer, which must be in **Stage 1**. Therefore, if the algorithm is started in **Stage 1** on a `NUMBER`, `FACTOR`, `TERM`, or `EXPR`, it also ends in **Stage 1**. This means that when s is one of these four types, $\mathrm{tk}_1(s\ t) = \mathrm{tk}_1(s)\ \ \mathrm{tk}_1(t)$.

Observation 2. For sentences s, $\mathrm{tk}(s)$ = `BEGIN` $\mathrm{tk}_1(s)$ `END`. We can see this by following the algorithm as it starts from **Stage 0**. Regardless of the next character in the stream, it triggers a `BEGIN` token and the tokenizer enters the **Stage 1** loop. The next tokens emitted are precisely $\mathrm{tk}_1(s)$. The tokenizer emits an additional `END` token in response to the `EOF` or a CLOSING parenthesis.[7]

Notice this is only true for sentences of CA and not for arbitrary subexpressions of sentences.

Now we can determine what the result of tokenization is for each possible s. In order to carry out the inductive argument, we will need to figure out $\mathrm{tk}_1(s)$ for every sentence type other than `BLOCK` and `BLOCKSEQ`. Observation 2 then gives us $\mathrm{tk}(s)$ for these types.

`NUMBER` $\mathrm{tk}_1(n)$ of a number n, is just `NUMBER`. Started in **Stage 1**, a digit is found, and the `NumberTokenizer` is called immediately. It reads in the number, emits a `NUMBER` token, and returns.

`FACTOR` A `FACTOR` f is either a number or a parenthesized expression. If it is a number, we have shown $\mathrm{tk}_1(f)$ is `NUMBER`.

Tokenization of a parenthesized expression '`(`' e '`)`' is straightforward. Encountering an open parenthesis will cause the current tokenizer to delegate to the `ExpressionTokenizer`. This tokenizer emits a `BEGIN` token and starts to process e in the **Stage 1** loop.[8] The tokens $\mathrm{tk}_1(e)$ are emitted. The tokenizer

[7] The tk_1 function by definition includes everything up to but not including that `END`.

[8] For well-formed sentences in CA, an expression will never contain `\n` or a premature closing parenthesis, which are the only sequences that the `ExpressionTokenizer` treats

then encounters the CLOSING sequence `)`, emits an `END` token, and returns. Therefore,

$$\mathrm{tk}_1(f) = \mathrm{tk}_1(\text{`(' } e \text{ `)'}) = \texttt{BEGIN}\ \mathrm{tk}_1(e)\ \texttt{END}$$

(Note that for $f =$ `(' $e$ `)', $\mathrm{tk}(f) =$ `BEGIN BEGIN` $\mathrm{tk}_1(e)$ `END END`. This double-wrapping will be collapsed later by the UNWRAP_EXPRS transducer.)

TERM If s is a `TERM`, it is a sequence of `FACTOR`s $f_0, \ldots, f_n$ separated by operators $\alpha_1, \ldots, \alpha_n$ where each operator is either `*` or `/`:

$$t \;=\; f_0\ \alpha_1\ f_1\ \alpha_2\ \cdots\ \alpha_n\ f_n.$$

By **Observation 1**, since f_0 is a prefix of t, $\mathrm{tk}_1(f_0)$ is an initial sequence of $\mathrm{tk}_1(t)$ and after tokenizing this prefix, the algorithm is again in **Stage 1**. The first operator α_1 is tokenized into a `CONSTITUENT` token. The rest of t is processed in the same way, alternating between factors and operators. Therefore,

$$\begin{aligned}\mathrm{tk}_1(t) &= \mathrm{tk}_1(f_0\ \alpha_1\ f_1\ \alpha_2\ \cdots\ \alpha_n\ f_n)\\ &= \mathrm{tk}_1(f_0)\ \texttt{CONST}_{\alpha_1}\ \mathrm{tk}_1(f_1)\ \texttt{CONST}_{\alpha_2}\ \cdots\ \texttt{CONST}_{\alpha_n}\ \mathrm{tk}_1(f_n))\end{aligned}$$

(above, `CONST` denotes a `CONSTITUENT` token).

EXPR If e is an `EXPR`, it is a sequence of `TERM`s $t_0, \ldots, t_n$ separated by operators $\alpha_1, \ldots, \alpha_n$ where each operator is either `+` or `-`. The same argument as in the `TERM` case works here. We then have:

$$\begin{aligned}\mathrm{tk}_1(e) &= \mathrm{tk}_1(t_0\ \alpha_1\ t_1\ \alpha_2\ \cdots\ \alpha_n\ t_n)\\ &= \mathrm{tk}_1(t_0)\ \texttt{CONST}_{\alpha_1}\ \mathrm{tk}_1(t_1)\ \texttt{CONST}_{\alpha_2}\ \cdots\ \texttt{CONST}_{\alpha_n}\ \mathrm{tk}_1(t_n)\end{aligned}$$

BLOCK and BLOCKSEQ Let b be a block. b is either an expression e or a single operator α followed by equally-indented subblocks $b_0, \ldots, b_n$.

We have already considered the case above when b is an expression. So, consider b of the form

$$\begin{array}{l}\alpha\\ \quad b_0\\ \quad b_1\\ \quad \vdots\\ \quad b_n\end{array}$$

We will denote the constant whitespace before each subblock as `-->`.[9] We can then write b explicitly as α `\n -->` b_0 `\n -->` b_1 `\n` $\cdots$ `-->` b_n.

differently from the default **Stage 1** loop. Whereas in general a correctness proof would have to treat each reader macro function R individually—working out $\mathrm{tk}_R(x)$ for every x that R might encounter—for CA this is not necessary.

[9] For CA, where all subblocks of a block have the same indentation level, the number of ␣ characters in each "`-->`" is the same for every sub-block. In the general case there could be several indented sub-levels, and the proof would be more complicated.

To determine $\mathrm{tk}(b)$, we start in **Stage 0**. The operator is detected, so a `BEGIN` token is emitted and the algorithm moves to **Stage 1**. α is then read and a `CONSTITUENT` token is emitted. When the NEWLINE sequence is encountered, the algorithm jumps to **Stage 2**. Hence,

$$\mathrm{tk}(b) \;=\; \texttt{BEGIN CONSTITUENT}\ \mathrm{tk}_2(\texttt{--}\!\rightarrow b_0\ \texttt{\textbackslash n}\ \cdots\ \texttt{\textbackslash n}\ \texttt{--}\!\rightarrow b_n)$$

At this point, the tokenizer reads and discards until it finds a non-whitespace sequence, which is the first character of b_0 (an operator, digit, or open parenthesis). It emits an `INDENT` token and pushes a new level onto the stream, then recursively calls itself (starting anew in **Stage 0**) to continue processing the input. To the recursively called tokenization, the input appears to be b_0 `\n` $\cdots$ `\n` b_n, i.e., a `BLOCKSEQ`. If the `EOF` is reached, or the indentation level decreases to 0, the subcall is fed `EOF` (which eventually causes it to return), the indentation level is decreased, and an `END` token is emitted (step 2.3 if `EOF` is found, or step 2.4.1 if indentation level decreased). And so,

$$\mathrm{tk}(b) \;=\; \texttt{BEGIN CONSTITUENT INDENT}\ \mathrm{tk}(b_0\ \texttt{\textbackslash n}\ \cdots\ \texttt{\textbackslash n}\ b_n)\ \texttt{END}$$

If the indentation level has decreased to 0, the block we are tokenizing is itself part of a `BLOCKSEQ`; the algorithm will then reset to **Stage 0** to tokenize the next block (step 2.4.1). It then follows that

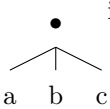
$$\mathrm{tk}(b_0\ \texttt{\textbackslash n}\ \cdots\ \texttt{\textbackslash n}\ b_n) \;=\; \mathrm{tk}(b_0)\ \mathrm{tk}(b_1)\ \cdots\ \mathrm{tk}(b_n)$$

We can then conclude by induction that for a single block:

$$\mathrm{tk}(b) \;=\; \texttt{BEGIN CONSTITUENT INDENT}\ \mathrm{tk}(b_0)\ \mathrm{tk}(b_1)\ \cdots\ \mathrm{tk}(b_n)\ \texttt{END}$$

Transducing

Instead of drawing trees at every step, we will represent a node as a parenthesized list of its children, e.g.



is written ⟦ a b c ⟧.

We define $\mathrm{tr}_1(t)$ to be the result of applying the PRIMARY transducer to the tree t, and likewise tr_2, tr_3, and tr_4 for the MULTIPLICATION, ADDITION, and UNWRAP_EXPRS. We also define tr_0 on token sequences τ such that $\mathrm{tr}_0(\tau)$ is the corresponding tree that is given to the transducer chain: the entire token sequence wrapped under a single top-level node.

Additionally, we define (for convenience) functions $\bar{\mathrm{tr}}_k$, where $\bar{\mathrm{tr}}_k(\tau)$ is the result of applying the transducer chain in order up to the k^{th} transducer. So, $\bar{\mathrm{tr}}_1$ is just tr_1, $\bar{\mathrm{tr}}_2$ is $\mathrm{tr}_2 \circ \mathrm{tr}_1$, $\bar{\mathrm{tr}}_3$ is $\mathrm{tr}_3 \circ \bar{\mathrm{tr}}_2$, and tr_4 is ρ.

PRIMARY The only actions of the PRIMARY transducer are rearrangements triggered by `BEGIN` tokens.

BLOCK and BLOCKSEQ We have already shown that regardless of the type of block, it is tokenized into a sequence starting with BEGIN, ending with END, and potentially with an INDENT token.

So, for all blocks, the INDENT is removed if it exists, everything after the BEGIN and before the END is wrapped under a new node, and finally the BEGIN and END themselves are removed.

$$\bar{\mathrm{tr}}_1(\mathrm{tk}(b)) \;=\; [\![\; \bar{\mathrm{tr}}_1(\texttt{CONSTITUENT}\ \mathrm{tk}(b_0)\ \mathrm{tk}(b_1)\ \cdots\ \mathrm{tk}(b_n)) \;]\!]$$

The PRIMARY transducer then moves on to process each of the children of the new node. Since each $\mathrm{tk}(b_i)$ is a disjoint sequence beginning and ending with a BEGIN/END pair, we can consider the action of this transducer on each of them individually. So,

$$\bar{\mathrm{tr}}_1(\mathrm{tk}(b)) \;=\; [\![\; \bar{\mathrm{tr}}_1(\texttt{CONSTITUENT})\ \bar{\mathrm{tr}}_1(\mathrm{tk}(b_0))\ \bar{\mathrm{tr}}_1(\mathrm{tk}(b_1))\ \cdots\ \bar{\mathrm{tr}}_1(\mathrm{tk}(b_n)) \;]\!]$$

This same consideration applies to BLOCKSEQs (each block in the sequence will be processed individually). It then holds that $\mathrm{tr}_1([\![\; \mathrm{tr}_0(b_1)\,\mathrm{tr}_0(b_2)\ldots\mathrm{tr}_0(b_n) \;]\!])$ is $[\![\; \mathrm{tr}_1(b_1)\,\mathrm{tr}_1(b_2)\ldots\mathrm{tr}_1(b_n) \;]\!]$.

NUMBER For a number n, $\mathrm{tk}_1(n) = \texttt{NUMBER}$, and $\bar{\mathrm{tr}}_1(\mathrm{tk}_1(n)) = \texttt{NUMBER}$.[10] At the block level, $\mathrm{tk}(n) = \texttt{BEGIN NUMBER END}$, so $\bar{\mathrm{tr}}_1(\mathrm{tk}(n)) = [\![\; \texttt{NUMBER} \;]\!]$.

TERM As we have shown, for a term t,

$$\mathrm{tk}_1(t) \;=\; \mathrm{tk}_1(f_0)\ \texttt{CONST}_{\alpha_1}\ \mathrm{tk}_1(f_1)\ \texttt{CONST}_{\alpha_2}\cdots\ \texttt{CONST}_{\alpha_n}\ \mathrm{tk}_1(f_n).$$

Since the only BEGIN/END pairs are within the factors, the PRIMARY transducer may rearrange items within each $\mathrm{tk}_1(f_i)$, but the structure of the TERM is preserved:

$$\bar{\mathrm{tr}}_1(\mathrm{tk}_1(t)) \;=\; \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_0))\ \alpha_1\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_1))\ \alpha_2\cdots\ \alpha_n\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_n))$$

If t is at the block level, the PRIMARY transducer will first rearrange the top-level BEGIN/END, but otherwise acts the same as on TERMs within EXPRs.

$$\begin{aligned} \mathrm{tk}(t) \;&=\; \texttt{BEGIN}\ \mathrm{tk}_1(t)\ \texttt{END} \\ \bar{\mathrm{tr}}_1(\mathrm{tk}(t)) \;&=\; [\![\; \bar{\mathrm{tr}}_1(\mathrm{tk}_1(t)) \;]\!] \end{aligned}$$

EXPR The situation for EXPRs is the same as for TERMs, but with TERMs instead of FACTORs.

FACTOR If a FACTOR f is a NUMBER, see above. If f is a parenthesized EXPR e and appears within a TERM, we have seen $\mathrm{tk}_1(f) = \texttt{BEGIN}\ \mathrm{tk}_1(e)\ \texttt{END}$, and so $\bar{\mathrm{tr}}_1(\mathrm{tk}_1(f)) = [\![\; \bar{\mathrm{tr}}_1(\mathrm{tk}_1(e)) \;]\!]$.

For parenthesized expressions appearing at block level:

$$\mathrm{tk}(f) \;=\; \texttt{BEGIN BEGIN}\ \mathrm{tk}_1(e)\ \texttt{END END}$$

so

$$\bar{\mathrm{tr}}_1(\mathrm{tk}(f)) \;=\; [\![[\![\; \bar{\mathrm{tr}}_1(\mathrm{tk}_1(e)) \;]\!]\!] \;=\; [\![\; \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f)) \;]\!].$$

[10] We are overloading symbols here: a NUMBER is either a syntax type of CA or it is a type of node in a syntax tree corresponding to this type.

MULTIPLICATION

NUMBER $\bar{\mathrm{tr}}_2(\mathrm{tk}_1(n))$ is still just a single node for the number n.

BLOCK If BLOCK b is an EXPR, e,

$$\begin{aligned} \bar{\mathrm{tr}}_1(\mathrm{tk}(b)) &= [\![\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(e))\ ]\!], \qquad \text{and so} \\ \bar{\mathrm{tr}}_2(\mathrm{tk}(b)) &= [\![\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(e))\ ]\!]. \end{aligned}$$

If b is an indented block,

$$\bar{\mathrm{tr}}_1(\mathrm{tk}(b)) = [\![\ \alpha\ \bar{\mathrm{tr}}_1(\mathrm{tk}(sb_1))\ \bar{\mathrm{tr}}_1(\mathrm{tk}(sb_2))\ \cdots\ \bar{\mathrm{tr}}_1(\mathrm{tk}(sb_n))\ ]\!]$$

so

$$\bar{\mathrm{tr}}_2(\mathrm{tk}(b)) = [\![\ \alpha\ \bar{\mathrm{tr}}_2(\mathrm{tk}(sb_1))\ \bar{\mathrm{tr}}_2(\mathrm{tk}(sb_2))\ \cdots\ \bar{\mathrm{tr}}_2(\mathrm{tk}(sb_n))\ ]\!].$$

It is important to note that the α might be a $*$ or $/$, but the MULTIPLICATION transducer will not apply because there is no item before α.

BLOCKSEQ Each block is transduced individually. Thus,

$$\bar{\mathrm{tr}}_2([\![\ \mathrm{tk}(b_1)\ \cdots\ \mathrm{tk}(b_1)\ ]\!]) = [\![\ \bar{\mathrm{tr}}_2(\mathrm{tk}(b_1))\ \cdots\ \bar{\mathrm{tr}}_2(\mathrm{tk}(b_1))\ ]\!].$$

EXPR For EXPR e,

$$\bar{\mathrm{tr}}_1(\mathrm{tk}_1(e)) = \bar{\mathrm{tr}}_1(\mathrm{tk}_1(t_0))\ \alpha_1\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(t_1))\ \alpha_2\ \cdots\ \alpha_n\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(t_n))$$

so

$$\bar{\mathrm{tr}}_2(\mathrm{tk}_1(e)) = \bar{\mathrm{tr}}_2(\mathrm{tk}_1(t_0))\ \alpha_1\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(t_1))\ \alpha_2\ \cdots\ \alpha_n\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(t_n)).$$

At the block level,

$$\begin{aligned} \bar{\mathrm{tr}}_1(\mathrm{tk}(e)) &= [\![\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(e))\ ]\!], \qquad \text{so} \\ \bar{\mathrm{tr}}_2(\mathrm{tk}(e)) &= [\![\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(e))\ ]\!]. \end{aligned}$$

FACTOR If FACTOR f is a number, the MULTIPLICATION transducer leaves it untouched. If f is a parenthesized expression, the MULTIPLICATION transducer will proceed downwards, doing nothing at this level:

$$\bar{\mathrm{tr}}_2(\mathrm{tk}_1(f)) = [\![\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(e))\ ]\!].$$

And at the block level:

$$\begin{aligned} \bar{\mathrm{tr}}_2(\mathrm{tk}(f)) &= [\![[\![\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(e))\ ]\!]\!] \\ &= [\![\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(f))\ ]\!]. \end{aligned}$$

TERM Here is the important case for MULTIPLICATION. We have for a TERM t that

$$\bar{\mathrm{tr}}_1(\mathrm{tk}_1(t)) \;=\; \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_0))\ \alpha_1\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_1))\ \alpha_2\ \cdots\ \alpha_n\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_n))$$

where each f_i is a FACTOR. Now, for each f_i it holds that $\bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_i))$ is a single node:

- If f_i is a NUMBER, $\bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_i))$ is already one node (a leaf).
- If f_i is a parenthesized EXPR (a FACTOR), the PRIMARY transducer will have wrapped everything between the BEGIN and END under a single new node.

The MULTIPLICATION transducer begins from the left. $\bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_0))$ is not $*$ or $/$, so it moves on. When it encounters α_1, it sees that there is a node before and after the current node and the current node is $*$ or $/$, so it transforms

$$\bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_0))\ \alpha_1\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_1))\ \alpha_2\ \cdots\ \alpha_n\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_n))$$

into

$$[\![\ \alpha_1\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_0))\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_1))\ ]\!]\ \alpha_2\ \cdots\ \alpha_n\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_n)).$$

It then proceeds to the next symbol, α_2, which leads to another transformation. The node before the current node in this case is the new node created in the first transformation.

$$[\![\ \alpha_2\ [\![\ \alpha_1\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_0))\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_1))\ ]\!]\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_2))\ ]\!]\ \alpha_3 \\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_3))\ \cdots\ \alpha_n\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_n))$$

This continues until the entire term is completed. The result of this is

$$[\![\ \alpha_n\ [\![\ \alpha_{n-1}\ [\![\ \cdots [\![\ \alpha_2\ [\![\ \alpha_1\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_0))\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_1))\ ]\!]\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_2))]\!] \\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_3))\ ]\!]\ \cdots]\!]\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_{n-1}))\ ]\!]\ \bar{\mathrm{tr}}_1(\mathrm{tk}_1(f_n))\ ]\!].$$

Finally, the children are recursively processed by the MULTIPLICATION transducer. The final result:

$$\bar{\mathrm{tr}}_2(\mathrm{tk}_1(t)) \;=[\![\ \alpha_n\ [\![\ \alpha_{n-1}\ [\![\ \cdots\ [\![\ \alpha_2\ [\![\ \alpha_1\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(f_0))\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(f_1))\ ]\!] \\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(f_2))\ ]\!]\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(f_3))\ ]\!]\ \cdots\ ]\!]\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(f_{n-1}))\ ]\!]\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(f_n))\ ]\!].$$

ADDITION The same arguments as with the MULTIPLICATION transducer work in this case. The nontrivial case here is EXPR rather than TERM, where by the same reasoning,

$$\bar{\mathrm{tr}}_2(\mathrm{tk}_1(e)) \;=\; \bar{\mathrm{tr}}_2(\mathrm{tk}_1(t_0))\ \alpha_1\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(t_1))\ \alpha_2\ \cdots\ \alpha_n\ \bar{\mathrm{tr}}_2(\mathrm{tk}_1(t_n))$$

becomes

$$\bar{\mathrm{tr}}_3(\mathrm{tk}_1(e)) \;=[\![\ \alpha_n\ [\![\ \alpha_{n-1}\ [\![\ \cdots\ [\![\ \alpha_2\ [\![\ \alpha_1\ \bar{\mathrm{tr}}_3(\mathrm{tk}_1(t_0))\ \bar{\mathrm{tr}}_3(\mathrm{tk}_1(t_1))\ ]\!] \\ \bar{\mathrm{tr}}_3(\mathrm{tk}_1(t_2))\ ]\!]\ \bar{\mathrm{tr}}_3(\mathrm{tk}_1(t_3))\ ]\!]\ \cdots\ ]\!]\ \bar{\mathrm{tr}}_3(\mathrm{tk}_1(t_{n-1}))\ ]\!]\ \bar{\mathrm{tr}}_3(\mathrm{tk}_1(t_n))\ ]\!].$$

UNWRAP_EXPRS Finally, the bottom-up UNWRAP_EXPRS transducer is applied. At this point, all syntax types have been transduced into single tree, with one root node. NUMBERs are still one node. TERMs were processed during MULTIPLICATION, EXPRs were processed during ADDITION, and FACTORs are either numbers or a node containing an EXPR.

Any of these occurring at the block level will be wrapped under an extra node. When UNWRAP_EXPRS processes these, it will remove this unnecessary node. For example, if one of the subblocks of a block b is a single number n, that subblock will be parsed into $\bar{\mathrm{tr}}_3(\mathrm{tk}(b))$, which will be a subtree consisting of a single node with a single child, n. That is, $\bar{\mathrm{tr}}_3(\mathrm{tk}(n)) = [\![\ n\ ]\!]$. UNWRAP_EXPRS will proceed upwards, processing the child node n before its parent. It finds that n is an only child, and replaces the parent with its child. So, $\bar{\mathrm{tr}}_4(\mathrm{tk}(n)) = n$. The case is analogous for the other types.

We will show below that $\bar{\mathrm{tr}}_4(\mathrm{tk}(s)) = \bar{\mathrm{tr}}_4(\mathrm{tk}_1(s)) = \varphi(s)$ for any sentence s in CA, except for BLOCK and BLOCKSEQ sentences, where we will show only that $\bar{\mathrm{tr}}_4(\mathrm{tk}(s)) = \varphi(s)$. This concludes the proof of Theorem 7.

NUMBER For a NUMBER n, $\bar{\mathrm{tr}}_4(\mathrm{tk}(n)) = \mathrm{tr}_4([\![\,\mathrm{tk}_1(n)\,]\!]) = \mathsf{NUMBER} = \varphi(n)$, and $\bar{\mathrm{tr}}_4(\mathrm{tk}_1(n)) = \mathsf{NUMBER}$ also.

FACTOR $\bar{\mathrm{tr}}_4(\mathrm{tk}(n)) = \bar{\mathrm{tr}}_4(\mathrm{tk}_1(n)) = \mathsf{NUMBER}$ as above.

If f is '(' e ')', then $\bar{\mathrm{tr}}_3(\mathrm{tk}(f)) = [\![[\![\ \bar{\mathrm{tr}}_3(\mathrm{tk}_1(e))\ ]\!]\!] = [\![\ \bar{\mathrm{tr}}_3(\mathrm{tk}_1(f))\ ]\!]$. Here UNWRAP_EXPRS applies twice, first removing the lower wrapping node, then removing the top node. Hence $\bar{\mathrm{tr}}_4(\mathrm{tk}(f)) = \bar{\mathrm{tr}}_4(\mathrm{tk}_1(e))$. This equals $\varphi(e)$ by induction hypothesis, which in turn is exactly $\varphi(f)$. The result for $\bar{\mathrm{tr}}_4(\mathrm{tk}_1(f))$ is identical, although only one unwrapping is carried out.

TERM If t is $f_0\ \alpha_1\ f_1\ \alpha_2\ \cdots\ \alpha_n\ f_n$, then

$$\bar{\mathrm{tr}}_4(\mathrm{tk}_1(t)) = [\![\ \alpha_n\ [\![\ \cdots\ [\![\ \alpha_1\ \bar{\mathrm{tr}}_4(\mathrm{tk}_1(f_0))\ \bar{\mathrm{tr}}_4(\mathrm{tk}_1(f_1))\ ]\!]\ \cdots\ ]\!]\ \bar{\mathrm{tr}}_4(\mathrm{tk}_1(f_n))\ ]\!]$$

By induction hypothesis this equals $[\![\ \alpha_n\ [\![\ \cdots\ [\![\ \alpha_1\ \varphi(f_0)\ \varphi(f_1)\ ]\!]\ \cdots\ ]\!]\ \varphi(f_n)\ ]\!]$, which in turn is exactly $\varphi(t)$. The outcome of $\bar{\mathrm{tr}}_4(\mathrm{tk}(t))$ is the same, after tr_4 unwraps the outermost node.

EXPR The same reasoning for TERM applies to EXPR.

BLOCK If b is an indented block $\alpha\ b_1 \ldots b_n$, then:

$$\begin{aligned}\bar{\mathrm{tr}}_4(\mathrm{tk}(b)) &= [\![\ \alpha\ \bar{\mathrm{tr}}_4(\mathrm{tk}_1(b_1))\ \cdots\ \bar{\mathrm{tr}}_4(\mathrm{tk}_1(b_n))\ ]\!] \\ &\overset{\text{IH}}{=} [\![\ \alpha\ \varphi(b_0)\ \cdots\ \varphi(b_n)\ ]\!] = \varphi(b)\end{aligned}$$

BLOCKSEQ If b is a block sequence $b_1 \ldots b_n$, then:

$$\begin{aligned}\bar{\mathrm{tr}}_4(\mathrm{tk}(b)) &= [\![\ \bar{\mathrm{tr}}_4(\mathrm{tk}_1(b_1))\ \cdots\ \bar{\mathrm{tr}}_4(\mathrm{tk}_1(b_n))\ ]\!] \\ &\overset{\text{IH}}{=} [\![\ \varphi(b_0)\ \cdots\ \varphi(b_n)\ ]\!] = \varphi(b)\end{aligned}$$

3.3.4 A limitation of RMTC parsing

In this subsection, we will illustrate the difficulties encountered when trying to extend the syntax of CA with a unary - (minus) operator. We will find we need to provide some way of explicitly disambiguating the binary - from its unary analogue.

The problem

The natural way of introducing a unary minus operator is by adding a new UNARY_MINUS transducer to the transducer chain. Thinking naively, we may first think that the unary minus should have a higher precedence than addition, otherwise the sentence $A + -B$ would be parsed as ⟦⟦+ A − ⟧ B ⟧. So it should be grouped by a transducer earlier in the chain than ADDITION. We should then insert the UNARY_MINUS transducer into the chain between PRIMARY and MULTIPLICATION[11], but defining the conditions for applying the unary minus arrangement is not obvious.

We recall that the binary minus operator is processed by the ADDITION transducer. This transducer has an arrangement rule that will be applied if the current node is -, there is a node immediately before the current node, and there is a node immediately after. As a first guess, then, we might stipulate that the unary minus transducer should be applied if the current node is - and there is a next node. So `CONSTITUENT_` x should be wrapped into ⟦ `CONSTITUENT_` x ⟧.

However, this presents a problem if we consider an expression such as `y - x`. As the unary minus has higher precedence, the UNARY_MINUS transducer appears earlier in the chain than the ADDITION transducer. But in this case, the new transducer applies when it finds the minus operator that has a next node. The outcome is then ⟦ `y` ⟦ `CONSTITUENT_ x` ⟧⟧.

We can remedy this first problem by specifying that the unary minus rearrangement should only apply if there is a following node *and* there is *not* a previous node. Here we also immediately run into a problem. Consider the input `y - - x`, which should be parsed into ⟦ `CONSTITUENT_ y` ⟦ `CONSTITUENT_ x` ⟧ ⟧. With the new rule, UNARY_MINUS will now skip both minus signs, as they both have a previous node; the output we are left with is ⟦ `y CONSTITUENT_ CONSTITUENT_ x` ⟧.

One approach at this point would be to further specify that the unary minus arrangement should only apply if there is a following node *and* there is not a previous node *unless* this previous node is also an operator that has a lower (or equal) precedence. However, this requires us to keep some sort of precedence table, and may lead to problems if another transducer were inserted between the ADDITION and UNARY_MINUS transducers.

This kind of problem arises when we apply the RMTC method to parse overloaded syntax. This is not a rare feature in programming languages, and other parsing methods are able to handle it. Python, in fact, does correctly parse `-3`, `4 - 3`, `4 - - 3`, and even expressions like `- - 3` and `4 - - - 3`, with or without spacing.

The solution we ended up adopting for Anoky (the programming language presented in the next chapter), was to disambiguate unary minus with binary

[11] For CA it is easy to find the right position in the chain to insert new transducers; we would expect this to be a more delicate matter for real languages.

minus by requiring that binary minuses be surrounded by whitespace, and unary minuses have no whitespace between the operator - and the operand.

This kind of difficulty seems to be unavoidable with our parsing method, and may be considered a serious limitation.

Chapter 4

Anoky

The Anoky language is an alternative syntax for the semantics of Python. Written Anoky code is parsed using the RMTC parsing method, and a Python syntax tree is generated and compiled directly from this code by interfacing with Python's built-in `ast` module.

We begin this chapter by describing Anoky syntax trees, and comparing them to their Lisp equivalents. We describe the four stages of the Anoky compiler: tokenization, transducer-chain parsing, macro expansion, and code generation. Finally, we provide some examples of using Anoky in practice, including how to extend its parser.

4.1 Anoky syntax

Anoky uses a generic syntax tree, similar to Lisp forms described in section 1.2.1. This is done with the intention of leveraging the benefits of Lisp syntax, benefits which were also described in that section.

The syntactic units of written Lisp code are forms, which are defined inductively as being either atoms or compound forms (lists of forms). Anoky's syntax follows this generic example, while being only slightly less ascetic. There are two atomic types and two compound types:

- Identifiers play the same role as Lisp's symbols[1].
- Literals play the same role as Lisp's self-evaluating objects[2].
- Forms play the same role as Lisp's compound forms[3].
- Seqs have no syntactic equivalent in Lisp.

Identifiers

Identifiers play the same role in Anoky syntax as *symbols* in Lisp. The role of identifiers is to name objects. These objects can exist at compile-time — as in the case of macros and special forms — or at run-time — as in the case of variables, function arguments, etc.

[1] `http://clhs.lisp.se/Body/26_glo_s.htm#symbol`
[2] `http://clhs.lisp.se/Body/26_glo_s.htm#self-evaluating_object`
[3] `http://clhs.lisp.se/Body/26_glo_c.htm#compound_form`

Literals

Anoky *literals* are elements of code that represent a value. A *literal* plays same role in Anoky as a Lisp *self-evaluating* object. In Lisp, written code is sometimes parsed directly into so-called *self-evaluating* objects, which evaluate to themselves. For example `(+ 1 3)` is read in as a form with three elements: the first is a symbol and the other are integers. In Anoky, the analogous written code (`1 + 3` or `\#(+ 1 3)`) will parse `1` and `3` into literals containing the integer values, instead of the bare integers themselves. Wrapping all values with literals ensures that all code objects inherit a single parent class. This is currently used to store positional information, but may later make it easy to extend code with other metadata.

Nodes: forms and seqs

Nodes are ordered sequences of *elements*. Anoky nodes play a similar role as lists do in Lisp syntax. However, the underlying data structure of a node is a *doubly*-linked list instead of a singly-linked list. [4]

In Anoky, a node is almost always one of two types, a *form* or a *seq*.

Anoky's forms fulfill the same role as compound forms in Lisp. The first element of a form is its *head* the remaining elements are *arguments* or just *args*. A form is intended to represent the application (in some sense) of the head, parameterized by the args, if any exist.

A seq[5] represents an ordered sequence of two or more code elements. The elements of a seq are called *items*. A seq does not carry with it the notion that the first element is being applied to the rest. Including seqs as a type of code in Anoky means the distinction between sequences that represent a head applied to arguments and sequences that represent ordered data is moved from the semantics to the syntax of the language.

A UML diagram

Figure 4.1 shows the UML diagram for the various types described above

4.2 Overview of the Anoky compiler

The Anoky compiler takes as input a sequence of characters, and outputs a Python syntax tree. The compilation process is organized into four distinct stages:

Tokenization A readtable-macro parser processes the sequence of characters into a sequence of tokens.

Transducer-chain parsing A chain of transducers takes this sequence of tokens and produce an Anoky syntax tree.

[4] Lisp technically implements a list as nested pairs with a nil terminator. Doubly linked lists allow for transducers to proceed through the elements of a node just as easily in either direction.

[5] The term *seq* was chosen to avoid conflict with *list* or *tuple*, which are part of Python's semantics.

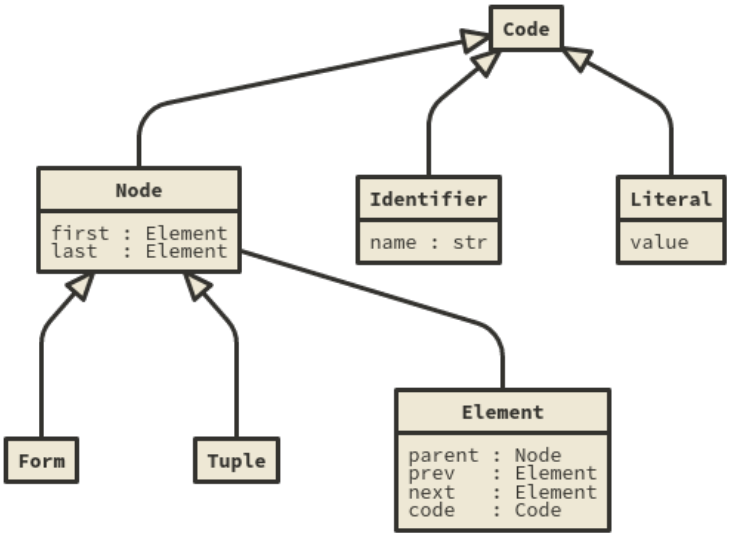


Figure 4.1: UML for code types.

Macro expansion A compile-time macro system processes this tree.

Code generation A code-generation system translates the macro-expanded tree into a Python syntax tree.

The resulting Python syntax tree can be compiled and executed using the facilities that Python provides.

4.2.1 Tokenization

The Anoky tokenizer uses the extended readtable-macro parsing algorithm described in 3.1. As described previously, the algorithm has a readtable that classifies characters or sequences of characters as one of a number of types. Sequences are classified in the Anoky readtable as follows:

- Operators (e.g. `+`, `*`, `<=`, `/`) are classified as ISOLATED_CONSTITUENT sequences.
- `)`, `]`, and `}` are CLOSING sequences.
- `,` and `:` are PUNCTUATION sequences.
- Spaces, tabs, and other non-printing characters are WHITESPACE sequences.
- The newline and carriage return are NEWLINE sequences.
- MACRO sequences will be listed below.
- By default, any other sequence of characters is classified as a CONSTITUENT.

The `#`, `(`, `[`, `{`, `#(`, `‘` (backtick), `’` (single quote), `’’’` (3 times single quote), `"` (double quote) and `"""` (3 times double quote) sequences are MACRO sequences in the readtable. They are associated with reader macro functions as described below.

- `#` As in Python, the hash character signals a comment. The comment tokenizer will emit a `BEGIN_MACRO`, collect every following line with a higher indentation level into a `COMMENT` token, emit an `END_MACRO`, and return.

- `[`, `{`, `(`, and `‘` Opening delimiters are tokenized by a similar procedure. In each case, the procedure first emits a `BEGIN_MACRO` token. It then finds the first non-whitespace, non-newline sequence, pushes the indentation level at that point, and recursively calls the default tokenizer. When the default tokenizer returns, the procedure emits an `END_MACRO` token, pops the stream, and returns.

- `’`, `"`, `’’’`, `"""` Like Python, Anoky allows quotation with either single quotation marks or double quotation marks. The reader macro function for these consumes characters and collects them as a string until it encounters a matching closing quotation mark, at which point it returns the collected string and exits.

  When the string tokenizer receives a dispatch, it emits a `BEGIN_MACRO` token and begins consuming characters from the stream. It accumulates everything into a single value until it encounters the designated closing delimiter, whereupon it wraps the string value in a `STRING` token, emits this, emits an `END_MACRO`, and returns.

- `#(` As we have seen, by dispatching to a new tokenizer the current tokenizer relinquishes control over the output token stream, and this can be used to embed DSLs, literate documentation, and other forms of nested syntax. We use this feature of reader macros now to supplement Anoky with an alternative written notation, namely S-expressions. A hash character immediately followed with an open parenthesis dispatches to the “Lisp mode” tokenizer until the matching close parenthesis. Newlines and indentation are ignored.

4.2.2 Transducer-chain parsing

The end result of tokenization is a sequence of tokens. We can think of this sequence of tokens as a two-level tree with one root node whose children are the tokens of the sequence. This tree is then processed by a chain of transformations. We will say that each such transformation is a *tree transducer*, or simply a *transducer*. The transducer chain transforms the simple two-level tree into a fully-fledged Anoky syntax tree: made up of identifiers and literals at the leaves and forms and seqs at the inner nodes.

Most transducers that make up the transducer chain of the Anoky parser scan the tree from top to bottom: each level of the tree is scanned from left to right (or reversely in some cases), and the transducers manipulate the scanned nodes according to so-called *arrangement rules*; then the level is scanned again, recursively applying the transducer at each node.[6]

[6]While in practice we use the call stack, and FPMs were defined with a finite state and no

The Anoky transducer chain

1. CONSTITUENT
2. PRIMARY
3. INFIX-SPECIAL-OPS
4. PUNCTUATION
5. UNARY-MINUS
6. EXPONENTIATION
7. TILDE
8. MULTIPLICATION
9. ADDITION
10. STAR-ARGUMENTS
11. BIT-SHIFT
12. BITWISE-AND
13. BITWISE-XOR
14. BITWISE-OR
15. AS
16. COMPARISONS
17. NOT
18. AND
19. OR
20. ASSIGNMENT

The function of many of the transducers in the chain is straightforward. For example, the ADDITION transducer walks down the tree and groups `+` and `-` signs surrounded by an element on each side into a new node at that location with the sign as its head.

The transducers corresponding to binary operators (MULTIPLICATION, EXPONENTIATION, BIT-SHIFT, BITWISE-*, AND, OR) are all parsed in a similar fashion, and other binary operators (ASSIGNMENT) are parsed similarly but right-to-left instead of left-to-right. The transducers handling unary operators (TILDE, NOT, STAR-ARGUMENTS) follow an analogous strategy. Chains of operators (COMPARISONS) are only slightly more sophisticated.

Parts of the chain mirror Python's own operator precedence table[7].

Let us now describe the remaining transducers:

CONSTITUENT

The first transducer goes through the token sequence and replaces `CONSTITUENT` tokens with identifiers or literals. This transducer attempts to interpret a constituent as a Python integer. If it succeeds, the token is replaced with an integer literal; if it fails, the token will be replaced by an identifier.

PRIMARY

Before it deals with individual operators and keywords, the parser needs to process the large-scale structure of the code. A single "primary" transducer is

stack, this use can be completely avoided by using the graph structure as the stack. When we are doing the second scan and we want to recursively apply the transducer one level down, we restart execution at that level, and this execution is expected to *return* to its parent level and continue the scan; once this second scan finishes, the transducer *returns* to the level above, so that *it* can continue its scan.

[7] `https://docs.python.org/3/reference/expressions.html#operator-precedence`

responsible for doing this.

Essentially, this task consists of translating the implicit structure in the various tokens (most notably `BEGIN`, `END`, `BEGIN_MACRO`, and `END_MACRO`) into Forms and Seqs. In Anoky there are many ways of representing the same Form or Seq — this will be described in section §4.3. As a result, the rules that govern the PRIMARY transducer include many cases, and we will not describe them extensively.

The attribute accessor operator (the *dot*, `.`) must also be handled by this transducer, as it has a similar level of precedence. Indeed, `a.b(c)` parses as `((. a b)c)` and not `(. a (b c))`.

INFIX-SPECIAL-OPS

Keywords such as **`return`** and **`yield`** have special grouping behavior. The INFIX-SPECIAL-OPS transducer performs the necessary rearrangements so that these are not parsed as regular syntax.

When it finds a return statement, the transducer will wrap everything after the return keyword into one form. This ensures, for example, that **`return`** `a b, c: d, e` is parsed as `(`**`return`**` (a (b, c)d e)` and not `(`**`return`**` ((a, b), c)d e` (which would be the default parse if "return" was a normal function).

PUNCTUATION

After the form head, commas serve to separate groups of arguments. Each group is wrapped together into a seq. All the seqs between the head of a form and a colon are further wrapped into a single seq.

For example, `(head a1 a2, b, c1 c2: ...)` becomes `head ((a1, a2), b, (c1, c2))...)` The PUNCTUATION transducer scans each node from left to right and applies these groupings.

Certain form heads will trigger a different punctuation arrangement, designed especially for them. The `for` form, for example, gives a special meaning to the first occurrence of the `in` identifier, so that `(for a, b in c: ...)` becomes `for ((a, b), c)...)`.

4.2.3 Macro expansion

After the transducer chain completes its parse, control is handed to Anoky's macro expansion system. This system is meant to play the same role as Lisp macros, although all macro expansion in Anoky happens at compile-time (Lisp's macro expansion happens at evaluation time[8], even though modern Lisp compilers know how to optimize this in some cases, by applying the macro at compile time).

Macro expansion works via a recursive procedure which modifies the code destructively. The process is governed by a certain default procedure, called the *default macro expander*. The action of this procedure on an element is dependent on the type the code being expanded, and also on the contents of two tables, called *macro table* and *identifier-macro table*, which associate macro functions with certain identifiers. The behavior of the default macro expander can be described as follows:

[8] See `http://clhs.lisp.se/Body/03_aba.htm`.

- When encountering a literal, it does nothing.
- When encountering a seq, it expands each child element.
- When encountering an identifier, it checks if it appears in the *identifier-macro table*, and if this is the case it delegates macro expansion to the macro function associated with this identifier. Otherwise, it does nothing.
- When encountering a form, it checks if the form's head is an identifier appearing in the *macro table*. If this is the case, macro expansion is delegated to the macro function associated with this identifier. If the form's head is not an identifier, or it is an identifier which does not appear in the macro table, then each child element of the form is recursively macro expanded (by the default macro expander).

An important point to note is that if a macro function is applied to a form, expansion does not by default continue downwards after the macro function returns. Macro functions have complete control over the macro expansion process within the form where they are applied, and this includes control over how (and whether) the form's children are macro-expanded. This means that macros which want these children to be expanded must themselves explicitly call the default expander.

As example of why this is a desirable feature, consider the `quote` special form.[9] The ability to `quote` code underlies one of the most powerful features of Lisp (see section 1.2.1). The `quote` special form should generate Python code that represents the code being quoted, instead of whatever meaning that code usually has. For example, the form `(+ a b)` will usually generate the Python code `a + b`, but the form `(quote (+ a b))` will generate instead the Python code

```
Form(Symbol('+'), Symbol('a'), Symbol('b')).
```

To see why special care is needed in macro-expanding quoted Anoky code, consider expanding `(quote\ (f a\ b\ \~{}(g\ c)))`. If the default expander continued expanding all code after encountering a `quote` form, then the `f`, `a`, *etc.* would be expanded (if they appeared in the macro or identifier macro tables), which is not what we want. The point of quoting code is that we want to return *that* code. However, we *do* want the `g` and the `c` to be checked by the expander, since that code appears within the *unquote* operator `\~`. Thus, the macro expansion process for `quote` proceeds through the syntax tree and does nothing unless it encounters a form headed by the unquote operator `\~{}`, at which point it recursively calls the default macro expander.

4.2.4 Code generation

At this point the Anoky code is fully parsed and macro-expanded. In order to compile and execute it, we convert the Anoky syntax tree into a Python syntax tree, which can then be compiled and executed.

Python makes it possible to work directly with syntax trees via the standard library module `ast`. The module provides an interface to dynamically build a Python syntax tree node by node (starting with nodes corresponding to

[9] A special form is both a macro and a code generator (it knows how to generate its own Python code).

numbers, strings, etc., and then grouping those within nodes for lists, tuples, statements, etc.). The built-in `compile` function can compile a Python syntax tree into Python bytecode, which can then be stored and/or executed.

The code generation process is very similar to how macro expansion works. The behavior of the default code generator depends on list of so-called *special forms* (analogous to the macro tables of the default macro expander), and on a context variable which identifies the current *code generation domain*. This domain is dependent on whether the code is being used in a statement, an expression, an assignment target, or a deletion statement.

- When the default code generator encounters a literal, it converts it to a Python literal (an `ast.Str` or `ast.Num` node).
- When it encounters an identifier, it checks if the identifier is `True`, `False`, or `None`, which result in `ast.NameConstant` nodes being generated; for all other identifiers, it will generate a Python identifier (`ast.Name` node).
- When it encounters a seq, it will generate a Python tuple (`ast.Tuple` node), and will recursively call itself to generate the elements of the tuple.
- When it encounters a form, it checks if the head of the form names a special form and in this case it delegates generation to it. Otherwise it generates an `ast.Call` node and recursively calls itself to generate nodes for each of the given arguments.

The special forms

A *special form*[10] is a function that corresponds to a syntactic construction in Python. It generates a Python `ast` node from its elements according to its own unique rules. In order to cover the entire Python AST specification, we require roughly one special form for each of the possible node types. The default special forms table includes include special forms for

- all of the operators, such as `+` and `+=`,
- control flow statements, such as **`if`**,
- Python keywords, such as **`class`** and `global`, and
- some of the default macros in Anoky, such as `quote`.

Except for the Anoky macros, each of these is responsible for generating a specific Python node[11], for example the `+` special form generates an `ast.Add` node.

4.3 Writing Anoky code

Now that we have broadly outlined how the Anoky compiler is structured, let us show how to write Anoky code.

[10] We adapt this terminology from Lisp. A "special form" is a form that is treated in a nonstandard way in the evaluation process; for Anoky, the analogous process is Python code generation.

[11] The Python documentation for the `ast` module does not give many details about the different types of nodes or how to generate full trees from scratch; the best reference is `https://greentreesnakes.readthedocs.io`.

4.3.1 Lisp mode

Via the *Lisp-mode* reader macro, it is possible to write Anoky code in a way that makes the tree-structure very obvious. The macro is triggered by the character sequence `#(`.

The Lisp-mode reader macro completely ignores indentation. Forms are parenthesis-delimited with whitespace-separated args, and seqs are parenthesis-delimited with comma-separated items (a seq must have two or more elements, so a comma must always be present).

```
#(head form_arg_1 form_arg_2 (seq_item_1, seq_item_2))
```

For example, here is how we use the Lisp-mode macro to write code for defining the Fibonacci function:

```
#(def (fib n)
    (= (a, b) (1, 1))
    (for i (range (- n 1))
        (= (a, b) (b, (+ a b))))
    (return a))
```

As described in §1.3.2, the downside to this way of writing code is that it can become hard to read, and lacks convenient notational features such as indentation (unless we keep the indentation in sync manually), infix operators, etc. These problems are already apparent in the simple example shown above, and they become very tiresome in more complex programs.

However, as its name suggests, Lisp mode provides Anoky with a homoiconic syntax that can be used wherever this is beneficial. It is additionally useful in explaining the syntax of parsed Anoky code without going into the full detail of §4.1.

4.3.2 Blocks and indentation levels

Let us show by example how written Anoky code is structured by whitespace.

```
1  f(a, b, c):
2      d(d1, d2):
3          e(e1)
4      g(g1, g2, g3)
5  h: h1
```

In the example above, lines 1-5, 2-4 and 3 form indentation levels, and lines 1-4, 2-3, 3, 4 and 5 form blocks. Formally, we specify the following:

- By *line* we mean a single line of written code.
- An *indentation level* is a group of contiguous lines, where the first non-whitespace character of the first line appears at column c (henceforth we will say the line *starts* at column c), and all subsequent lines are either empty, or start at column c or greater.
- A *block* is defined in the same way, but where subsequent non-empty lines must start at a column strictly greater than c.

If a block has indentation levels within it, their starting column must then be strictly greater than the starting column of the block's first line. See Figure 3.2 for another example.

4.3.3 How blocks and indentation levels get parsed into forms and seqs

Lists of arguments and lists of forms

Most syntactic constructs of programming languages follow one of two patterns. We either have a certain operation applied to a list of expressions, or we have a certain meta-operation, possibly parameterized by a list of expressions, applied to a list of statements.

Anoky has two *parsing modes*, so to speak, suitable for each of these two situations. In *list-of-args mode*, the blocks in a given indentation level are treated as parts of a comma-separated sequence of code elements. This provides a suitable way of factoring several function arguments over multiple lines. For example, the inner indentation level of the following block is parsed in list-of-args mode:

```
func
   arg1 arg2, arg3
   "very long arg"
   arg5
```

The form generated by the above code, when written in Lisp mode, is:

```
#(func (arg1, arg2) arg3 "very long arg" arg5)
```

In *list-of-forms mode*, each block in a given indentation level is processed as a form, meaning that if it contains more than one element, it is parsed into a form having the first element as a head. For example, the inner indentation level of the following block is parsed in list-of-args mode:

```
macro:
   arg1 arg2, arg3
   "very long arg"
   arg5
```

Notice that the only difference, when compared with the previous example, is the use of the colon `:`. This is how one indicates if the following indentation level should be parsed in list-of-args or list-of-forms mode.

The form generated by the above code, when written in Lisp mode, is:

```
#(macro (arg1 arg2 arg3) "very long arg" arg5)
```

The difference is that the `arg1 arg2, arg3` block is parsed into the form `#(arg1 arg2 arg3)`, instead of the seq `#(arg1, arg2)` and the identifier `arg3`.

Typically we will use list-of-args mode to parse long lists expressions passed as arguments to a funtion, and list-of-forms mode to parse lists of statements passed as arguments to a macro or special form. But there is nothing forcing us to do so, as long as we understand what syntactic structure results from the code we have written.

One thing should be noted: a colon or an extra indentation will always cause the containing block to be parsed as a form, even in list-of-args mode. So the difference between list-of-args and list-of-forms mode is how single-line blocks get parsed. For example:

```
func
   list of, args
   but this is a: form
   and this, is
   also a, form
```

```
#(func (list, of) args
    (but (this, is, a) form)
    (and this is (also, a) form))
```

Before-colon grouping and double indentation

A common programming pattern is illustrated by the following code (first example is Lisp, second is python):

```
(do ((temp-one 1 (1+ temp-one))
       (temp-two 0 (1- temp-two)))
      ((> (- temp-one temp-two) 5) temp-one)) ; => 4

if y > 2:
    return (x + 2) * y
```

In both examples we have a certain meta-operation (`do` and **`if`**) parameterized by expressions and applied statements. We would ideally like to parse the expressions in list-of-args mode, and the statements in list-of-forms mode.

Anoky provides two special syntaxes for writing code in this pattern. The easy way is by listing items between the head of a form and a colon.

```
head a1, a2, a3:
  f a4, a5
  g a6

#(head (a1, a2, a3) (f a4 a5) (g a6))
```

The head, items list, and colon must all be on the same line.

Sometimes we would like to break down the list of arguments. Even a short list can become visually cumbersome when restricted to one line. Anoky has a special *double indentation* syntax to help with this case. If a block has two different indentation levels within it, the first indentation level will be parsed in list-of-args mode, and the second indentation level (whose starting column must necessarily be smaller) will be parsed in list of forms mode.

```
head  a1, a2
      b1 b2, b3
   c c1, c2

#(head (a1, a2, (b1, b2), b3)
   (c c1 c2))
```

Self-delimited nodes

Finally, Anoky supports writing certain forms a la Python where non-head args are given as a comma-separated list in parentheses appearing after the head. There cannot be any whitespace between the head and open parenthesis, unlike in Python (although this is already a de facto stylistic rule).

For one thing, this allows small forms to be written concisely on one line. In simple cases we can also use a colon to do this (`f: a, b`). Delimited forms offer additional precision outside of Lisp mode, e.g. `f(a)(g(b(c)(e)))`.

Operators and keywords have their own rules in Python syntax and their own representation in Python ASTs. In Anoky, though, the syntax for `=` and **`while`** is the same as for any function. `x = 3 + y + z` can be written `=(x, +(3, y, z)`.

Seqs also can be explicitly delimited by including some whitespace before the open parentheses.

For both forms and seqs written in this way, the code inside the parentheses must belong to the a single indentation level, which is parsed in list-of-args mode. For example:

```
func( a, b
      c: c1 )

#(func a b (c c1))

( d e, f
  g
    g1, g2 )

#((d, e), f, (g g1 g2))
```

Special syntax

Certain forms, such as the assignment `=` and the `for` operators:

```
a = func:
   arg1
   arg2

#(= a (func arg1, arg2)
```

The `for` operator uses the identifier `in` as a separator for its first arg:

```
for a, b in c:
    d

#(for ((a, b), c) d)
```

It is easy to modify the chain to allow for such custom behavior. All it requires is that the custom behavior be parsed with higher precedence than the default.

Multiple realization of code

From lines 13 and 14 of "The Zen of Python"[12]:

> There should be one — and preferably only one — obvious way to do it. Although that way may not be obvious at first unless you're Dutch.

This imperative makes a statement both about variability in code that performs a given function and the number of options to express a given piece of parsed code in written code.

In Anoky, by contrast, there are often plenty of ways to write the same thing.[13] The programmer is encouraged to write code in the most readable way, but certain circumstances may call for code to be expressed in other representations, and Anoky's syntax allows for this.

[12] the `this` in `import this`

[13] Incidentally, we aren't Dutch!

```
f(a, b, c)

#(f a b c)

f a, b, c     # if in list-of-forms

f:
  a
  b
  c
```

4.4 Extending Anoky

Extensions to an RMTC parser can take a variety of forms: defining new tokenizers, modifying the readtable, defining new arrangement rules, modifying the transducer chain, adding arrangement rules to tokenizers already in the chain, etc. As discussed in §1.1, though, the power and flexibility of an extensible parsing method also needs to be accessible. How can a user actually make these modifications, and how simple is the process in practice?

In the case of Anoky, new arrangement rules, tokenizers, macros, etc. can be written as Python modules and added to the appropriate directories in the source. But it is also possible to use Anoky to extend itself, by writing new rules and tokenizers directly as Anoky modules. Additionally, users can modify the readtable and transducer chain of an Anoky program as it is compiled or during an interactive session.

For a concrete example of extending Anoky's syntax we will adapt an example from [Hoy08], in which a syntax for regular expressions as in Perl is added to Lisp using a reader macro. In Anoky this is slightly more complicated than in Lisp, but still straightforward. To extend Anoky in this way, we will do the following:

- Define a reader macro, which entails defining a new class `RegexpTokenizer` extending the `Tokenizer` class. The `run` method of this class will proceed through the input and collect everything into a single string until it finds an unescaped `/` character, at which point it returns the accumulated string as a `STRING` token and exits.
- Associate this tokenizer in the readtable with the MACRO sequence `r/`.
- Define an arrangement rule, `RegexpArrangement`, that transforms nodes containing token sequences output by the regexp tokenizer into a single node containing Anoky code that compiles the regular expression given by the `STRING` token emitted by the reader macro. For this we make use of Python's standard `re` module. Specifically, this removes the `BEGIN_MACRO` and `END_MACRO` tokens from the sequence, leaving only the token containing the regular expression, then replaces this token with code that, when executed, imports the regular expression module `re` from Python's standard library and compiles the original contents of the token.
- We insert this arrangement rule into the PRIMARY transducer in the chain.

We also define a custom infix operator for matching regular expressions, also inspired by Perl. To do this,

- We add our operator to the readtable as an ISOLATED_CONSTITUENT.
- We define a new transducer that identifies processes our operator as a binary operator.
- We insert this transducer into the active transducer chain.
- We specify how our operator should work by defining a new macro and adding it to the active macro table.

Because Anoky's syntax is very similar to Python's, we must emphasize that all the code below is written in Anoky, parsed via the RMTC method which was implemented in Python, and compiled into Python before being executed.

4.4.1 Adding a syntax for regular expressions

Defining and adding a tokenizer

To define a new tokenizer, we define a new class that inherits from the base tokenizer class.

```
class RegexpTokenizer(Tokenizer):
```

A new instance of this class will be created whenever the default tokenizer encounters the `r/` readtable sequence. When this instance is initialized, it will check that it has indeed been started on the right sequence, and then record the position (row and column number) of this sequence.

```
    def __init__( self, context, opening_delimiter,
                  opening_delimiter_position,
                  opening_delimiter_position_after ):

        Tokenizer.__init__(self, context)
        assert opening_delimiter == "r/"
        self.opening_delimiter_position = opening_delimiter_position
        self.opening_delimiter_position_after =
            opening_delimiter_position_after
```

The rest of the definition of the tokenizer consists in writing its reader macro function. We bind the stream locally, check that we are not already looking at an `EOF` character, then push the current indentation level onto the stream.

```
    def run(self):
        stream = self.context.stream

        if stream.next_is_EOF():
            yield Tokens.ERROR:
                "r/"
                self.opening_delimiter_position
                self.opening_delimiter_position_after
                "No characters found after opening delimiter 'r/'."
            return

        stream.push()

        seen_escape = False
```

Before we start consuming characters of the regexp, we emit a `BEGIN_MACRO` token, which is paramaterized by its opening delimiter (in this case `r/`) and position.

```
        begin_macro_token = Tokens.BEGIN_MACRO:
            "r/"
            self.opening_delimiter_position
            self.opening_delimiter_position_after

        yield begin_macro_token
```

We now create an empty string that will be used to store the regexp.

```
        value = ""
        value_first_position = stream.copy_absolute_position()
```

The main loop of the macro function reads in characters from the input and accumulates them into the `value` string. This proceeds until it finds a closing character `/`, that is not preceded by the escape character `\\`; after we find this, it will emit its accumulated regexp as a `STRING` token, emit an `END_MACRO` token with a link to the `BEGIN_MACRO` token emitted previously, pop the indentation level from the stream, and return. If we run into an `EOF` character, we immediately pop the stream and exit with an error.

```
        while True:
            if stream.next_is_EOF():
                stream.pop()
                yield Tokens.ERROR
                        "", stream.copy_absolute_position(),
                        stream.copy_absolute_position(),
                        % "Expected closing regexp-delimiter '/', \
                           matching opening delimiter 'r/' \
                           at position %s.":
                           self.opening_delimiter_position.nameless_str
                return

            char = stream.read()
            if char == '\\':
                if seen_escape:
                    seen_escape = False
                    value += '\\'
                else: seen_escape = True
            else:
                if seen_escape:
                    if char == '/': value += char
                    else: value += '\\' + char
                    seen_escape = False
                else:
                    if char == '/':
                        closing_delimiter_first_position =
                            stream.absolute_position_of_unread()
                        yield Tokens.STRING:
                            value
                            value_first_position
                            closing_delimiter_first_position
                        yield Tokens.END_MACRO:
                            begin_macro_token,
                            '/'
                            closing_delimiter_first_position
                            stream.copy_absolute_position()
                        stream.pop()
                        return
```

```
                    else:
                        value += char

        stream.pop()
```

Defining a new arrangement rule

We define a new class that inherits from the base arrangement rule class. Initialization only consists of setting its name.

```
class RegexpArrangement(ArrangementRule):

    def __init__(self):
        ArrangementRule.__init__(self, "Regexp")
```

At every step in walking the tree, a transducer will call the `applies` method of each of its arrangement rules, which returns `True` if the arrangement should be applied at the current location. In our case, the regexp arrangement should be applied whenever it is over a series of tokens produced by the tokenizer defined above, so we define `applies` to check that the current element is a `BEGIN_MACRO` token and that its opening delimiter is `r/`.

```
    def applies(self, element):
        return is_token(element, Tokens.BEGIN_MACRO, token_text="r/")
```

The actual rearrangement operation is performed by the `apply` method.

```
    def apply(self, begin_macro):
        end_macro = begin_macro.end
        parent = begin_macro.parent
```

If the regexp is empty, as in `r//`, its value is `None`.

```
        if begin_macro.next is end_macro:
            parent.remove(end_macro)
            begin_macro.code = `None`
            return begin_macro.next
```

Otherwise, we have something like `r/abc/`, where "abc" is tokenized into a STRING token, and is the only token between the `BEGIN_MACRO` and `END_MACRO` tokens.

```
        assert is_token(begin_macro.next, Tokens.STRING)
        regexp = begin_macro.next
        assert regexp.next is end_macro
```

If we have gotten this far, we then remove the `BEGIN_MACRO` and `END_MACRO` tokens and replace the `STRING` token's original code with the code `__compile_regexp__("abc")`. Notice here we use the tilde for code interpolation (i.e. *unquoting*). This inserts the actual regexp from the `STRING` token directly into the new code.

```
        parent.remove(begin_macro)
        parent.remove(end_macro)
        regexp.code = `__compile_regexp__(~regexp.value)`
        return regexp.next
```

Outside the definition of the class, we have to define the `compile_regexp` function. Given a string, this function will import Python's `re` module if it is not already imported, and then use the `compile` function defined in that module to compile the regexp contained in its input string. We then add this to the top-level dictionary for the module.

```
def compile_regexp(regexp_str):
    import re
    return re.compile(regexp_str)

__builtins__["__compile_regexp__"] = compile_regexp
```

Adding an arrangement rule to a transducer

Finally, we add this arrangement rule to the PRIMARY transducer.

```
__parser__.get_transducer("Primary").insert_rule
        "Strings", RegexpArrangement()
```

Defining and adding a new transducer

We also define a new transducer that allows us to use `=~` as a binary operator for regexp matching.

We specify that `=~` should be interpreted as an operator by adding it as an ISOLATED_CONSTITUENT sequence in the readtable.

```
__parser__.set_readtable_entry("=~", type=RT.ISOLATED_CONSTITUENT)
```

We define a new transducer, REGEXP-MATCH-OPERATOR, and specify that it should apply the default left-to-right binary operator arrangement rule: whenever it finds a `=~` surrounded by two elements.

```
RegexpMatchOperatorTransducer = TopDownTreeTransducer:
    "Regexp Match Operator"
    Arrangement [LeftRightBinaryOperator({'=~'})]
```

We then add this transducer to the chain between the AS and COMPARISONS transducers.

```
__parser__.insert_transducer_before
    "Compare", RegexpMatchOperatorTransducer
```

Now we have to actually specify how the match operator should work.

We can use `rawmacro` to define a macro for `=~` from within Anoky itself and automatically add it to the current macro table. `rawmacro` is both a macro and a special form: as a macro, it continues expansion of its child elements, and as a special form, it generates the Python code that defines a new macro subclass whose `expand` method will performs the actions defined by the Anoky code in its body.

```
rawmacro =~(element, EC):
    parent = element.parent
    regexp = element.code[1].code
    arg = element.code[2].code

    match = `(~regexp).match(~arg)`
    parent.replace(element, match)
```

Example usage

Let us illustrate what is happening by going through the entire compilation chain for the following example:

```
>>> if r/(.*)abc(.*)/ =~ "testabcxyz": print("yes!")
```

First the Readtable-Macro algorithm produces the following sequence of tokens:

```
BEGIN
    CONSTITUENT(if)
    BEGIN_MACRO(r/)
        STRING((.*)abc(.*))
    END_MACRO(/)
    CONSTITUENT(=~)
    BEGIN_MACRO(")
        STRING(testabcxyz)
    END_MACRO(")
    PUNCTUATION(:)
    CONSTITUENT(print)
    BEGIN_MACRO(()
        BEGIN
            BEGIN_MACRO(")
                STRING(yes!)
            END_MACRO(")
        END
    END_MACRO())
END
```

The transducer chain then converts this to the following form:

```
(if   (=~ (__compile_regexp__ "(.*)abc(.*)")
          "testabcxyz")
    (print "yes!"))
```

During the macro-expansion phase, this gets converted to:

```
(if   ((. (__compile_regexp__ "(.*)abc(.*)") match)
       "testabcxyz")
   (print "yes!"))
```

The code generation stage will produce a syntax tree, which corresponds to the following Python code:

```
if __compile_regexp__('(.*)abc(.*)').match('testabcxyz'):
    print('yes!')
```

This code is compiled, and then executed. The outcome is the expected:

```
yes!
```

Chapter 5

Conclusion

After implementing the Anoky compiler, designing the language, adding assorted features, etc. we are in a position to assess the RMTC parsing method on our criteria described in section 1.1.

The simplicity of the algorithm was a core motivation in developing and realizing the method. The parsing method has two stages: readtable–macro parsing and the transducer chain. Both stages are composed of smaller components: respectively reader macros, and simple tree transducers. In both cases, the low-level components are themselves composed of simple operations. In both cases, the smaller components are combined in a simple and intuitive way: recursive descent, and sequential application. So the algorithm *is* conceptually very simple.

When writing new components and adding them to the readtable or chain, the user does not need to rework other parts of the language. In our extension to the Anoky parser (§4.4), we added support for regular expressions by writing only code that related to our new syntax.

What is unclear is how much a user will need to know before they are able to apply these modifications. We expect the process can be made much simpler than it currently is, by implementing the appropriate syntactic sugar, but it seems unavoidable that users need to understand, at least to some extent, how the default transducers process the tree.

The extension we presented in §4.4 illustrates that, as expected, RMTC can be extended in a modular fashion. Implementing a better interface for extending Anoky in such a way should be straightforward.

Currently, there is no way to control the scope of transducers. This presents a problem for composing RMTC-parsed languages: regardless of how we combine the chains of two RMTC parsers, every transducer in the new chain will apply to all code in a tree, not just the code of the respective language. It may be possible to remedy this by extending the transducer-chain algorithm without adding significant complexity, but in its present form the method is not compositional.

RMTC incorporates Lisp's reader-macro mechanism, and so is fully general in the same sense. However, because it includes native support for parsing indentation, infix syntax, and other syntactic features, it is a suitable parsing method for a readable programming language. Anoky attests to this by mimicking Python's written syntax.

We anticipated speed would not be one of the RMTC's strong points relative

to other methods, given that each transducer in the chain must scan the entire syntax tree (i.e. we expect the linear factor in the algorithm's time bound to be fairly large). However, we cannot say for certain. Our implementation of the method is extremely slow; this could be simply because Python is a fairly slow language, and/or because we did not try to optimize it properly.

Abney's parser performs well in this regard, as each transducer is a finite-state lexical analyzer [Abn96]. We have shown (§3.2.3) that our FPM transducers run in linear time, but it is not clear if the runtime of an individual transducer in practice can be made small enough to render the method competitive with grammar-based parsers.

We may now compare RMTC parsing with the methods previously surveyed.

	LL	LR	PEG	GLR/GLL	Lisp RM	RMTC
simplicity of algorithm	+	-	+	-	+	+
simplicity of extension	-	-	+	+	++	+
modularity	-	-	+	++	+	++
compositionality	-	-	+	++	+	-
generality	-	-	-	+	++	++
readability	+	+	+	+	-	+
speed/efficiency	++	++	+	-	+	?

Table 5.1: The relative strengths and weaknesses of the discussed methods.

Bibliography

[Abn91] Steven P. Abney. "Parsing By Chunks". In: *Studies in Linguistics and Philosophy*. Springer Science + Business Media, 1991, pp. 257–278. DOI: 10.1007/978-94-011-3474-3_10. URL: http://dx.doi.org/10.1007/978-94-011-3474-3_10.

[Abn96] Steven Abney. "Partial parsing via finite-state cascades". In: *Natural Language Engineering* 2.4 (Dec. 1996), pp. 337–344. DOI: 10.1017/s1351324997001599. URL: http://dx.doi.org/10.1017/S1351324997001599.

[Ada13] Michael D. Adams. "Principled parsing for indentation-sensitive languages". In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '13*. Association for Computing Machinery (ACM), 2013. DOI: 10.1145/2429069.2429129. URL: http://dx.doi.org/10.1145/2429069.2429129.

[AI15] Ali Afroozeh and Anastasia Izmaylova. "One parser to rule them all". In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) - Onward! 2015*. Association for Computing Machinery (ACM), 2015. DOI: 10.1145/2814228.2814242. URL: http://dx.doi.org/10.1145/2814228.2814242.

[Cho57] Noam Chomsky. *Syntactic Structures*. Mouton Publishers, 1957. ISBN: 9027933855.

[FM04] N. Friburger and D. Maurel. "Finite-state transducer cascades to extract named entities in texts". In: *Theoretical Computer Science* 313.1 (Feb. 2004), pp. 93–104. DOI: 10.1016/j.tcs.2003.10.007. URL: http://dx.doi.org/10.1016/j.tcs.2003.10.007.

[For04] Bryan Ford. "Parsing expression grammars". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '04*. Association for Computing Machinery (ACM), 2004. DOI: 10.1145/964001.964011. URL: http://dx.doi.org/10.1145/964001.964011.

[Hoy08] Doug Hoyte. *Let Over Lambda*. LULU PR, Apr. 11, 2008. 384 Seiten. ISBN: 1435712757. URL: http://letoverlambda.com/index.cl.

[Kar01] Lauri Karttunen. “Applications of Finite-State Transducers in Natural Language Processing”. In: *Revised Papers from the 5th International Conference on Implementation and Application of Automata*. CIAA ’00. London, UK, UK: Springer-Verlag, 2001, pp. 34–46. ISBN: 3-540-42491-1. URL: http://dl.acm.org/citation.cfm?id=647267.721691.

[KG05] Kevin Knight and Jonathan Graehl. “An Overview of Probabilistic Tree Transducers for Natural Language Processing”. In: *Computational Linguistics and Intelligent Text Processing*. Springer Science and Business Media, 2005, pp. 1–24. DOI: 10.1007/978-3-540-30586-6_1. URL: http://dx.doi.org/10.1007/978-3-540-30586-6_1.

[Knu65] Donald E. Knuth. “On the translation of languages from left to right”. In: *Information and Control* 8.6 (1965), pp. 607–639. DOI: 10.1016/s0019-9958(65)90426-2. URL: http://dx.doi.org/10.1016/S0019-9958(65)90426-2.

[Lan66] P. J. Landin. “The next 700 programming languages”. In: *Communications of the ACM* 9.3 (1966), pp. 157–166. DOI: 10.1145/365230.365257. URL: http://dx.doi.org/10.1145/365230.365257.

[Lan74] Bernard Lang. “Deterministic techniques for efficient non-deterministic parsers”. In: *Automata, Languages and Programming*. Springer Science and Business Media, 1974, pp. 255–269. DOI: 10.1007/3-540-06841-4_65. URL: http://dx.doi.org/10.1007/3-540-06841-4_65.

[Lee02] Lillian Lee. “Fast context-free grammar parsing requires fast boolean matrix multiplication”. In: *Journal of the ACM* 49.1 (2002), pp. 1–15.

[McC79] John McCarthy. “History of Lisp”. In: (Feb. 12, 1979). URL: http://www-formal.stanford.edu/jmc/history/lisp/lisp.html.

[MN04] Scott McPeak and George C. Necula. “Elkhound: A Fast, Practical GLR Parser Generator”. In: *Lecture Notes in Computer Science*. Springer Science and Business Media, 2004, pp. 73–88. DOI: 10.1007/978-3-540-24723-4_6. URL: http://dx.doi.org/10.1007/978-3-540-24723-4_6.

[MS99] Christopher Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT University Press Group Ltd, July 11, 1999. ISBN: 0262133601. URL: http://www.ebook.de/de/product/3755409/christopher_s_manning_foundations_of_statistical_natural_language_processing.html.

[Pit96] Kent Pitman, ed. *The Common Lisp Hyper Spec*. LispWorks Ltd., 1996. URL: http://www.lispworks.com/documentation/HyperSpec/Front/.

[Ros09] Guido van Rossum. *The History of Python: Python’s Design Philosophy*. Jan. 13, 2009. URL: http://python-history.blogspot.com/2009/01/pythons-design-philosophy.html.

[Ros70] D. J. Rosenkrantz. “Properties of Deterministic Top-Down Grammars”. In: *Information and Control* 17 (1970), pp. 226–256.

[Rou70] William C. Rounds. “Mappings and grammars on trees”. In: *Mathematical Systems Theory* 4.3 (Sept. 1970), pp. 257–287. DOI: 10.1007/bf01695769. URL: http://dx.doi.org/10.1007/BF01695769.

[SG93] Guy L. Steele and Richard P. Gabriel. “The Evolution of Lisp”. In: *ACM SIGPLAN Notices* 28.3 (1993), pp. 231–270. URL: https://www.dreamsongs.com/Files/HOPL2-Uncut.pdf.

[SJ10] Elizabeth Scott and Adrian Johnstone. “GLL Parsing”. In: *Electronic Notes in Theoretical Computer Science* 253.7 (Sept. 2010), pp. 177–189. DOI: 10.1016/j.entcs.2010.08.041. URL: http://dx.doi.org/10.1016/j.entcs.2010.08.041.

[SJ13] Elizabeth Scott and Adrian Johnstone. “GLL parse-tree generation”. In: *Science of Computer Programming* 78.10 (Oct. 2013), pp. 1828–1844. DOI: 10.1016/j.scico.2012.03.005. URL: http://dx.doi.org/10.1016/j.scico.2012.03.005.

[Ste12] Guy L. Steele. *Growing a Language*. Keynote at 1998 ACM OOPSLA conference. 2012. URL: https://www.youtube.com/watch?v=_ahvzDzKdB0.

[Ste90] Guy L. Steele. *Common Lisp the Language*. 1990.

[Tha70] James W. Thatcher. “Generalized^2 sequential machine maps”. In: *Journal of Computer and System Sciences* 4.4 (Aug. 1970), pp. 339–367. DOI: 10.1016/s0022-0000(70)80017-4. URL: http://dx.doi.org/10.1016/S0022-0000(70)80017-4.

[Tom85] Masaru Tomita. *Efficient Parsing for Natural Language*. Springer, Sept. 30, 1985. 228 Seiten. ISBN: 0898382025. URL: http://www.ebook.de/de/product/6602078/masaru_tomita_efficient_parsing_for_natural_language.html.

[War+00] Barry Warsaw et al. *PEP 1 – PEP Purpose and Guidelines*. 2000. URL: https://www.python.org/dev/peps/pep-0001/.

[Whe] David A. Wheeler. *Readable Lisp S-Expressions*. Published online at http://readable.sourceforge.net/.

[Woo80] William A. Woods. “Cascaded ATN Grammars”. In: *Computational Linguistics* 6.1 (Jan. 1980), pp. 1–12. ISSN: 0891-2017.

[YK01] Kenji Yamada and Kevin Knight. “A syntax-based statistical translation model”. In: *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics - ACL ’01*. Association for Computational Linguistics (ACL), 2001. DOI: 10.3115/1073012.1073079. URL: http://dx.doi.org/10.3115/1073012.1073079.

[Yu15] Huacheng Yu. “An improved combinatorial algorithm for boolean matrix multiplication”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2015, pp. 1094–1105.

Changes in revised thesis

- The set of criteria in section 1.1 has been restructured and the criteria themselves made more precise.
- The evaluation of parsing methods (§2.2) is now a more uniform overview of the strengths and weaknesses of each parsing method with respect to our criteria.
- The Lisp reader algorithm is now included in our description and review of parsing methods (§2.1,2.2).
- A section (§2.3) has been added on the use of transducers and cascades in natural language processing.
- The connection with Abney's strategy and parser has been included (§3.2).
- The vague overview of parsing with FPMs at the conclusion of chapter 3 has been expanded into an illustration of an RMTC parser for a small language, a proof of correctness of the method on this language, and an example of its limitation.
- The introduction to section 4.4 has been expanded and clarified.
- An review of RMTC parsing with respect to our criteria and the methods discussed in section 2.2 has been added as a conclusion (§5).