TYPED, ALGEBRAIC PARSER IN IDRIS2

by

Vamsi Krishna Bellam

May 14, 2025

A Thesis submitted to the
Faculty of the Graduate School of
the University at Buffalo, The State University of New York
in partial fulfillment of the requirements for the
degree of

Master of Science

Department of Computer Science and Engineering

Acknowledgments

I would like to express my sincere gratitude to my advisor, **Dr. Andrew Hirsch**, for his invaluable guidance and support throughout this research. I would like to thank **Dr. Qianchuan Ye** for his insightful suggestion during my thesis defense. I am thankful to the **Department of Computer Science and Engineering at the University at Buffalo** for giving me this opportunity to pursue this work.

I would like to thank my friends for their immense support. Most of all, I'm deeply grateful to my parents and my sister. Their constant encouragement and belief in me gave me the strength to keep going.

Abstract

Parsing transforms raw input into a structured representation using a set of rules. For example, in compilers, a parser converts a stream of characters into an intermediate representation like a Parse Tree or an Abstract Syntax Tree. Parser combinators are a popular way of building parsers, which use the tools in the host programming language to build small parsers and compose them into more complex parsers. Parser combinators are usually built using a recursive descent approach with backtracking and multiple lookahead tokens, resulting in poor worst-case time complexities. In this thesis, we have implemented a parser combinator library in Idris2 using a typed, algebraic approach. Internally, the library uses a custom type system defined for context-free expressions to identify ambiguous grammars. Furthermore, a typed grammar is deterministically parsed with no backtracking and a single lookahead token, guaranteeing linear-time performance. To demonstrate the practical application of the library, we have implemented parsers for S-expression, JSON, and the IMP programming language. These examples showcase the expressiveness of parser combinators and the robustness of the type checker without sacrificing performance.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Parser Generators

Parser generators are one of the most popular ways of writing parsers in the programming domain. As the name suggests, parser generators accept a formal grammar as input and automatically produce source code for the corresponding parser. Typically, the parsing process involves three distinct stages. First, a set of lexical tokens is defined in a parser file, using a domain-specific language different from the host language. Second, the rules for parsing the string to match the respective tokens are defined in a lexer file. The grammar rules are specified in the Backus-Naur Form [12] to correspond to programming language structures. Finally, the parser generator produces code in the host language and can be used directly for parsing. The output of parsers is ideally some sort of structured data type, such as Abstract Syntax Tree [9] in the host programming language.

Internally, most parser generators use table-driven bottom-up parsing techniques like shift-reduce [12] and its variations. These techniques offer significant efficiency benefits, particularly due to their deterministic nature, avoidance of backtracking, and ability to statically detect ambiguities in the provided grammar. Some examples of parser generators include Menhir [26] in Ocaml, and Happy [11] in Haskell. These tools are battle-tested and

widely used in industry; The official parsers for both the Ocaml compiler [24] and the Haskell GHC compiler [10] are implemented using them.

Despite their strengths, parser generators exhibit certain limitations. One among them is using a separate domain-specific language to define the lexer and parser rules. Though these DSLs have a clear declarative reading, they restrict usage of host language abstractions and features such as modularity, type systems, higher-order composition, etc.

1.2 Parser Combinators

Many modern programming languages support higher-order functions in which a function can accept another function as an argument and return a function as a result. This is a powerful feature for building more abstractions and composability, and it helps to keep the code clean. Combinators are very similar, but instead of functions, they accept parsers as input and can produce parsers as output. In this approach, small parsers are defined and then composed with combinators like choice, sequence, repetition, etc, to build complex parsers. Recursive descent is a top-down parsing approach [12], where each rule in the grammar is implemented as a recursive function. Parser combinators use backtracking to deal with ambiguous grammars, although these can result in exponential time complexity in the worst case. An alternative approach uses lookahead tokens to decide which rules to use in a grammar; parses using this strategy are called predictive parsers and are limited to LL grammars.

Angstrom [7] in OCaml, Parsec [19] in Haskell are some of the popular parser combinator libraries. The high-performance web server Http/af [8] in OCaml uses Angstrom to parse HTTP requests and responses. Similarly, Pandoc[21], one of the popular tools for universal markup conversion, uses Parsec for parsing a variety of formats. Despite the low performance compared to the parser generators, parser combinators excel in building modular and compositional parsers. Performance can be improved with the use of better predictive algorithms,

the use of laziness, and meta-programming techniques. Furthermore, in the languages with advanced type systems like Idris2 [5], Agda [22], and Rocq [29], parsers correctness can be formally verified.

1.3 State Of Parsers in Idris2

In this section, we examine the ecosystem of parsers available in Idris2. Unlike many other programming languages, Idris2 currently doesn't have parser generator tools. Most of the available libraries are designed in a parser-combinator style.

1.3.1 Data.String.Parser

Data.String.Parser [28] is a simple parser combinator framework provided by the Idris2 contrib library, primarily used for parsing string inputs. It is inspired by Haskell's Attoparsec Zepto library [23]. Internally, the parser operates on a given string input and, upon success, returns a tuple consisting of the parsed result and the position at which parsing concluded. In case of failure, an error is produced. The parsers are built as small combinators by implementing the Monad and Applicative interfaces. These combinators are used in a recursive descent style to construct bigger and complex parsers. There is no special mechanism in the framework to avoid ambiguous grammars (like left recursion), and the parser gets trapped in infinite recursion.

1.3.2 Lightyear

Lightyear [27] is a parser combinator library in Idris2 inspired by Haskell's Parsec. It builds upon the well-established observation that parsers form an instance of Monads [14]. Writing parsers in monadic style has a lot of practical benefits, such as simplifying lexical analysis, facilitating the implementation of layout-sensitive syntax (e.g., the offside rule), and enabling the use of comprehension syntax for more readable and concise definitions. Parsec[18], de-

signed as one of the industrial-strength monadic parser combinators focused on efficiency. It supports parsing context-sensitive grammars with infinite lookahead tokens. Parsec combinators don't backtrack by default, but use `try` for explicit backtracking. Lightyear follows the same parsing strategy but inverts backtrack behaviour. Lightyear parsers backtrack by default, but use `commitTo` combinator to explicitly commit to a branch.

For example, if we take an alternate parser, `a <|> b`,

In Parsec,

1. If `a` succeeds against an input, the result is returned.
2. If `a` fails without consuming the input, then parser b is tried.
3. If `a` fails after consuming the input, then it returns a failure message without trying `b`. But, if we need to try b, even after consuming some part of input in a, we need backtracking, and it is handled with `try`

But in lightyear, it takes a different approach.

1. If `a` succeeds against an input, the result is returned.
2. If `a` fails without consuming the input, then parser `b` is tried.
3. If `a` fails after consuming the input, then it backtracks to the input and tries `b`. But, if we want to avoid backtracking and continue on the same branch after consuming some prefix, `commitTo` combinator is used.

In both of these libraries, the user is given control over the use of backtracking. The use of backtracking can lead to exponential time complexity. Neither library incorporates a dedicated mechanism to detect or eliminate left recursion. Consequently, issues such as infinite recursion or parser failure due to left-recursive rules can only be identified at runtime, when the parser is executed on actual input.

1.3.3 TParsec

TParsec [2] is a direct port of Agdarsec, originally developed in Agda, a dependently typed functional language like Idris2. The parser combinators used in languages like Haskell, OCaml don't guarantee termination, especially when writing parsers for recursive grammars. The motivation behind these libraries is to use the more advanced type system and totality checker available in a dependently typed language to enforce constraints at the type level. The parsers are built on top of indexed sets and inductive relations [1]; these help in convincing the totality checker that inputs are always consumed. In left-recursive grammars, a recursive call is attempted without consuming any input; such definitions are statically rejected by the compiler, as they violate the constraints enforced by the type system.

While the advanced usage of type systems and tracking input sizes helps in writing parsers that are guaranteed to terminate, the core idea is to build the parsers using the recursive descent with backtracking. TParsec focuses mainly on correctness, particularly termination guarantees, rather than raw performance. Furthermore, this is originally written in Idris1, and as of now, there is no active available port for Idris2.

1.3.4 Text.Lexer and Text.Parser

This is a higher-order parser combinator library from the official Idris2 contrib package and follows a two-stage process

1. Lexer - transforms a string into a list of tokens.

2. Parser - consumes a list of tokens to produce a structured result, typically an abstract syntax tree (AST).

The purpose of a lexer is to scan the input of strings to produce a list of tokens and simplify the parsing process while dealing with whitespaces or comments. The core of the Lexer is the **Recognizer** combinator, a Generalized Algebraic Data Type(GADT) indexed

with a boolean. The index specifies whether the recognizer can consume input or not, which helps the Idris2 totality checker in verifying that lexers are safe and terminating. Lexers are built using the constructors of **Recognizer** in a combinator-style fashion. Lexers are constructed by composing **Recognizer** constructors in a combinator-style fashion. Once the individual recognizers are defined, a *TokenMap* is created to associate recognizers with corresponding tokens. The actual lexing is performed by the **lex** function, which takes a *TokenMap* and an input string, returning a list of matched tokens. Input is processed linearly, and at each step, the lexer attempts to match against the rules defined in the map. If a match succeeds, the corresponding token is emitted. If a recognizer fails after partially consuming input, backtracking occurs to attempt other alternatives in the *TokenMap*.

The core of the Parser is a **Grammar** combinator structure, similar to a **Recognizer** in Lexer. This is also a GADT indexed with three parameters: the type of tokens, a boolean to specify whether the language can be empty or not, and the output type of a parser. First, we build smaller primitives using the constructors from the **Grammar** data type, and then we build larger ones using the recursive descent approach. As in Parsec, parsers only backtrack if no input has been consumed. If backtracking is desired after partial consumption, the **try** combinator must be used explicitly. The library provides the necessary constructs to do controlled backtracking and lookahead tokens to deal with ambiguous grammars, but it is mostly left to the user and doesn't enforce any constraints on the grammar.

Due to the complexity in understanding **Recognizer** and **Grammar** data types, it is typically used in larger projects. It is an actively maintained library as part of the Idris2 ecosystem. The Idris2 compiler, written in Idris2 (i.e., it is bootstrapped), uses $Text.Lexer$ and $Text.Parser$ for its parsing logic.

1.3.5 idris2-parser

idris2-parser describes itself as a library for writing provably total lexers and parsers [13]. Unlike the other libraries, this is not a combinator-first library. Although it provides a

domain-specific language of combinators in the style of existing libraries, the primary way to build parsers is through explicit mutual recursion. The motivation to choose this style is to have greater control over error messages, improved performance compared to combinators, and to make manual writing of parsers as nice as possible. Similar to the official parsing library in Idris2, this also separates the lexing and parsing into two different stages for better error handling and dealing with spaces, comments, etc.

The lexer and parser DSLs are implemented in a similar way to $Text.Lexer$ and $Text.Parser$ from the contrib library using **Recognizer** and **Grammar** combinator structures. The main interest of this library lies in writing the lexers and parsers using mutually recursive functions, which must be total. But proving the recursive functions as total requires some sort of well-founded relation encoded at the type level to satisfy the compiler. For this purpose, the library provides a few utility functions on the bounded lists, which help establish proofs that input is consumed during parsing.

Writing parsers in idris2-parser feels more like a handwritten parser using language constructs like $if-else$, higher-order functions. But the main complexity lies in proving the type checker that the functions are always total. Left-recursive grammars are disallowed by construction, as they violate totality constraints by failing to demonstrate input consumption. However, other forms of ambiguity must be addressed explicitly by the user.

1.4 Problem Definition

One of the main advantages of using parser generators is their performance and the ability to express grammars in a declarative style close to Backus–Naur Form. In traditional parser combinators, the parsers are treated as first-class values in a programming language and can be composed with other language features to build more complex parsers. While this composability offers significant expressive power, such libraries are typically implemented using recursive descent with backtracking, which can lead to exponential time complexity.

It is good to have the power of abstractions from parser combinators and the performance of parser generators. In this thesis, we have used the typed, algebraic approach [16] to build a parser combinator library in Idris2. In contrast to the above libraries, our library builds an internal type system to reject unambiguous grammars before parsing any input. The static type information gathered from the grammars is then used in the parsing algorithm to avoid backtracking with a single lookahead token, which guarantees the linear-time performance.

Chapter 2

Library Overview and Implementation

2.1 API Overview

Before delving into the implementation details, we present a high-level overview of the combinator API interface.

```
mutual
  public export
  data GrammarType : {n : Nat}
                  -> (ct : Vect n Type)
                  -> (a : Type)
                  -> (tagType : Type -> Type)
                  -> Type
    where
      Eps : a -> GrammarType ct a tagType

      Tok : tagType a -> GrammarType ct a tagType

      Bot : GrammarType ct a tagType
```

```
    Seq : Grammar ct a tagType
       -> Grammar ct b tagType
       -> GrammarType ct (a, b) tagType


    Alt : Grammar ct a tagType
       -> Grammar ct a tagType
       -> GrammarType ct a tagType

    Map : {a : Type}
       -> Grammar ct a tagType
       -> (a -> b)
       -> GrammarType ct b tagType

    Fix : {a : Type}
       -> Grammar (a :: ct) a tagType
       -> GrammarType ct a tagType

    Var : Var a ct -> GrammarType ct a tagType

public export
record Grammar (ct : Vect n Type) (a : Type) (tagType : Type -> Type) where
  constructor MkGrammar
  lang : LangType (TokenType tagType)
  gram : GrammarType ct a tagType
```

This is a mutually recursive type definition. We begin by examining the `Grammar` data type, which is a record with two fields:

1. `lang` - encodes the type of language that is described by grammar.

2. `gram` - A value of type `GrammarType`, which is used to construct and compose grammars.

In addition to these fields, the `Grammar` type is indexed by three parameters: `ct`, `a`, and `tagType`. Here, `ct` represents a context that tracks the return types of the parser; `a` is the return type of the parser; and `tagType` is a type constructor used to create tags (in other words, it determines the type of the tags used in the grammar).

The `GrammarType` is a generalized algebraic data type (GADT) indexed by the same three parameters as `Grammar`.

1. `Eps v` - A grammar that parses an empty token.
2. `Tok t` - A grammar that parses matching the token with tag `t`.
3. `Bot` - A grammar for the parser that never succeeds.
4. `Seq l r` - A sequential combinator that parses the tokens with a prefix parsed by `l` and a suffix by `r` grammars.
5. `Alt l r` - An alternative combinator that parses the tokens parsed by either `l` and `r` grammars.
6. `Map g f` - A map combinator that generates a grammar by applying function `f` to the given grammar `g`.
7. `Fix g` - A fix combinator for constructing the recursive grammars using the variables represented in de Bruijn [3] fashion.
8. `Var i` - Variable used in combination `Fix` to build recursive grammars.

Along with the main grammar data types, the library also provides two other important functions.

```
export
typeCheck : {a : Type}
```

```
          -> {tagType : Type -> Type}
          -> {auto _ : Tag tagType}
          -> Grammar Nil a tagType
          -> Either String (Grammar Nil a tagType)
typeCheck gram = typeof [] gram
```

The `typeCheck` function takes a grammar as input, performs type checking, and returns either a type error or a successfully type-checked grammar.

```
public export
generateParser : {a: Type}
              -> {tagType : Type -> Type}
              -> {auto _ : Tag tagType}
              -> Grammar Nil a tagType
              -> Parser a tagType
```

`generateParser` function takes the type-checked grammar as input and builds the actual parser.

The type of `Parser a tagType` is a function that takes a list of tokens and returns either an error or a pair of parsed values, remaining tokens.

We begin by constructing grammars using the grammar data types provided in the combinator style. Once the grammar is defined, it is passed to the `typeCheck` function, which performs type checking. If the grammar is well-typed, we then use the result to build the corresponding parser.

While constructing grammars using these types can be somewhat verbose, we present a simplified and more ergonomic API in the examples section. This reduced API demonstrates how grammars can be composed more concisely and intuitively.

Note about the syntax - Arguments defined in `{}` are called *implicit* arguments, and those with `auto` keyword are called *auto-implicit* arguments. Implicit arguments are generally

inferred by the type checker. Auto-implicit arguments go one step further and are inferred from the proof search context. Therefore, these arguments usually do not need to be explicitly supplied during function application, unless required by the type checker.

2.2 Context Free Expressions

We begin by recalling the definitions of formal grammars and formal languages.

A formal language is the set of all possible finite-length words defined over the set of alphabet.

A formal grammar is defined as valid strings from the formal language according to the syntax of the language. Usually, the formal grammar is defined using a set of production rules.

Chomsky defined the formal grammar using the four tuples - $G = (N, \Sigma, P, S)$.

1. N is the set of non-terminal symbols.

2. Σ is the set of terminal symbols disjoint from N.

3. P is the production rules, and is of the form $(\Sigma \cup N)^* \, N \, (\Sigma \cup N)^* \; \rightarrow \; (\Sigma \cup N)^*$.

4. $S \in N$ is a start symbol.

Chomsky's hierarchy [6] classifies formal grammars into four types: unrestricted grammars, context-sensitive grammars, context-free grammars, and regular grammars.

Our main interest is in context-free grammars(CFGs), which generate context-free languages. A CFG is also defined in 4 tuples $G = (N, \Sigma, P, S)$, where N, Σ, and S retain the same interpretation as in the above definition. The production rule P in this case is restricted in the form $N \rightarrow \; (\Sigma \cup N)^*$. Each rule has a single non-terminal on the left-hand side and a sequence of terminals and non-terminals on the right-hand side. The application of a rule is independent of the surrounding context, which gives rise to the term context-free.

Context-free expressions are observed as an extension to regular expressions with a least fixed-point operator and the variables [20]. Below, we describe the formal definitions of the context-free expression from the typed, algebraic approach to parsing [16].

$$g \ ::= \ \bot \ \mid \ g \vee g' \ \mid \ \varepsilon \ \mid \ c \ \mid \ g \cdot g' \ \mid \ x \ \mid \ \mu x.\, g$$

Given a finite set of characters σ and a set of variables V, context-free expressions are interpreted as a language. $\bot$ represents empty language, $g \vee g'$ is the union of two languages, ε is the language with empty string, c is the language with one element, $g \cdot g'$ is a concatenation of two languages. These constructs closely resemble those found in regular expressions. x and $\mu x.\, g$ are variable reference and least fixed point operator, respectively. Concepts such as variable scoping, binding, and the distinction between free and bound variables are analogous to those in programming languages like Idris2. The concept of the variable and fixed-point operators captures the notion of non-terminal in formal grammars, and is as expressive as Backus-Naur form[12]. For the semantics and untyped equational theory for the context-free expressions, refer to the typed, algebraic approach [16]. The syntax of these expressions closely resembles the first-order combinator API provided by our library.

For example, consider a context-free grammar, $S ::= \varepsilon \mid aS$. The language contains an empty string and any number of occurrences of the character a in it. The same can be expressed in the context-free expressions as $\mu x.\, (\varepsilon \mid ax)$.

Now, let's take the following simple examples in context-free grammars,

1. $S ::= ab \mid ac$
2. $S ::= \varepsilon \mid Sa$

In the first grammar, if we encounter the character a, it is unclear which production to choose, as both branches begin with the same symbol. In the second grammar, expanding the non-terminal in the second branch introduces ambiguity as well—we may choose ε, or we may continue expanding the non-terminal indefinitely. Both cases exemplify ambiguous

grammars. In the first example, we can use backtracking to avoid ambiguity, but it might become exponential in time complexity. In the second example, the parser may enter into infinite recursion without making progress.

In the next section, we describe the type system to restrict the ambiguous grammars in the context-free expression.

2.3 Type System for Context Free Expressions

First, we need a few properties on languages to disallow ambiguous grammars and help during the parsing.

1. *Null* - A boolean to represent whether the language allows an empty string or not.
2. *First* - represents the set of characters that any string will have in a language.
3. *Follow* - represents the set of characters that can follow the last character of a string in a language.

These three properties together form a type of language.

```
public export
record LangType (token : Type) where
  constructor MkLangType
  null : Bool
  first : SortedSet token
  follow : SortedSet token
  guarded : Bool
```

In Idris2, we can define them as a record with the required fields. We have *null* of type boolean, $first$, and $follow$ are of type sets over a token. The parsers are not limited to parsing strings, we can parse any type of tokens. Hence, the record is indexed with

the generic token with type `Type`. The only constraint on the token type is that it must implement the `Ord` interface, as this is required by `SortedSet` to support token comparison.

We have an extra field called *guarded* to specify whether the record is in the normal or a guarded context. We will discuss more about *guarded* field during the type checking of grammars in the later section.

Now, we will define the types for each of the possible grammars that can be constructed with our API. Once we have the types, then we will move on to type checking.

```
export
tok : {auto _ : Ord token} -> token -> LangType token
tok c =
  MkLangType
    { null  = False
    , first = singleton c
    , follow = empty
    , guarded = True
    }
```

`tok c` is the type of language with a single token c and there is no empty token, hence the null is `False`.

```
export
eps : {auto _ : Ord token} -> LangType token
eps =
  MkLangType
    { null  = True
    , first = empty
    , follow = empty
    , guarded = True
    }
```

```
export
bot : {auto _ : Ord token} -> LangType token
bot =
  MkLangType
    { null  = False
    , first = empty
    , follow = empty
    , guarded = True
    }
```

`eps` is the type of language with just an empty token, hence the null is `True`. Whereas `bot` is the type of an empty language. Both of them have no other tokens in languages, so the empty first and follow sets.

```
export
seq : {auto _ : Show token}
   -> {auto _ : Ord token}
   -> LangType token
   -> LangType token
   -> Either String (LangType token)
seq t1 t2 =
  if apart t1 t2 then
    Right(
      MkLangType
        {
          null  = t1.null && t2.null
        , first = if t1.null then union t1.first t2.first else t1.first
        , follow =
            if t2.null then union t2.follow (union t2.first t1.follow)
```

```
            else t2.follow
        , guarded = t1.guarded
        }
    )
  else Left ("""
              The Languages
              \{show t1}
              \{show t2}
              are not apart!
             """)

  where
    apart : LangType token -> LangType token -> Bool
    apart t1 t2 = not (t1.null) && (intersection t1.follow t2.first == empty)
```

If we try to parse a string against $g \cdot g'$, then we have to divide the parsed string into two pieces $s1$ and $s2$ such that $s1$ is in g and $s2$ is in g'. This decomposition should be unique.

So, before building a type for this sequential grammar, we need to check whether the types of the two concatenating languages are apart.

We say the two languages are apart when the prefix language is not nullable and there are no common tokens between the tokens followed in the prefix language and the tokens starting in the suffix language. If we find that two languages are not apart, we just return an error. Hence, the type is `Either String (LangType token)`.

Once we know the languages are apart, then we build the type of $g \cdot g'$. The resultant language has an empty string only if both languages have an empty token. The first and follow sets are determined depending upon whether the language has an empty token or not.

```
export
alt : {auto _ : Show token}
```

```
    -> {auto _ : Ord token}
    -> LangType token
    -> LangType token
    -> Either String (LangType token)
alt t1 t2 =
  if disjoint t1 t2 then
    Right(
      MkLangType
        { null  = t1.null || t2.null
        , first = union t1.first t2.first
        , follow = union t1.follow t2.follow
        , guarded = t1.guarded && t2.guarded
        }
    )
  else
    Left ("""
              The Languages
              \{show t1}
              \{show t2}
              are not disjoint!
          """)
  where
    disjoint : LangType token -> LangType token -> Bool
    disjoint t1 t2 =
      not (t1.null && t2.null) && (intersection t1.first t2.first == empty)
```

If we try to parse a string against $g \vee g'$, then we have to decide whether to parse with g or g'. This is one ambiguities we discussed earlier.

So, before building a type for this alternate grammar, we need to check whether the types

of the two languages are disjoint.

We say the two languages are disjoint when neither language has an empty token and neither language don't starts with the same set of tokens.

Once we know the languages are disjoint, then we build the type of $g \vee g'$. The resultant language has an empty string if any of the languages have an empty token. The first and follow sets are simply the union of respective sets from both languages.

```
export
fix : {auto _ : Ord token}
   -> (f : Either String (LangType token) -> Either String (LangType token))
   -> Either String (LangType token)
fix f = fixHelper $ Right min


  where
    fixHelper : Either String (LangType token) -> Either String (LangType token)
    fixHelper t =
      let t' = f t in
      if t' == t then t else fixHelper t'


    min : LangType token
    min =
      MkLangType
        { null  = False
        , first = empty
        , follow = empty
        , guarded = False
        }
```

For the fix, we have to find the least fixed point of f. So, we start with the minimum language and then try to expand the language by applying the function f till it stops

changing.

This is the implementation of a type system on context-free in Idris2, for the formal type systems and proofs refer to [16].

2.4 Token Type and Tag Interface

Before going into the details of type checking. Let's discuss about the Token and Tag.

```
public export
data Token : (tagType : Type -> Type) -> Type where
  Tok : {a : Type} -> (tag : tagType a) -> a -> Token tagType


public export
data TokenType : (tagType : Type -> Type) -> Type where
  TokType : {a : Type} -> (tag : tagType a) -> TokenType tagType
```

`Token tagType` is a container that:

1. Carries a `tag` of type `tagType a`
2. And a value of that type `a`

`TokenType tagType` is pretty much similar to Token, but it carries just the tag and not the value.

```
public export
data Cmp : Type -> Type -> Type where
  Leq : Cmp a b
  Eql : Cmp a a
  Geq : Cmp a b


public export
```

```
interface Tag (tagType : Type -> Type) where
  compare : tagType a -> tagType b -> Cmp a b
  show    : tagType a -> String
```

This is a type-indexed comparison result, not just *Ordering* like in Idris2. `Cmp` encodes not only comparison results but also type relations.

For example, `Eql` only constructs when both types are equal. This is very useful when comparing tokens during the parsing, and the reason we need this is discussed in section 2.6

`Tag` is an interface on `tagType`, a type constructor that takes some type `a` and returns of type `tagType a`.

1. `compare` defines a comparison relation over type-indexed tags.

2. `show` gives a string representation for printing.

2.5 Type Checking

Once we build the grammars with our API, we need to check whether the grammar is ambiguous or not. For this, we will be using the type system mentioned in section 2.3.

The whole typing rules, along with the proofs for weakening, transfer, and soundness, are given here [16].

```
export
typeof : {a : Type }
      -> {ct : Vect n Type}
      -> {tagType : Type -> Type}
      -> {auto _ : Tag tagType}
      -> (env : Vect n (LangType (TokenType tagType)))
      -> Grammar ct a tagType
      -> Either String (Grammar ct a tagType)
```

First, let's see the type of `typeof` function. There are two arguments: `env` is a context to store type information of a language, and the second one is a grammar to type check. The result will be either the typed checked grammar, or if type checking fails, it returns an error.

The typing requires two contexts, one for ordinary variables and the other for guarded variables. Guarded variables are variables that must occur to the right of a non-empty string not containing that variable. This helps in identifying the left-recursive grammars. But instead of maintaining two contexts, we can have another field called *guarded* in the type of language to see whether the variable is guarded or not.

```
typeof env (MkGrammar  _ (Eps x)) = Right (MkGrammar eps (Eps x))

typeof env (MkGrammar _ (Tok c)) = Right (MkGrammar (tok (TokType c)) (Tok c))

typeof env (MkGrammar _ Bot) = Right (MkGrammar bot Bot)
```

Building the type information for `Eps`, `Bot` is simply updating the `lang` field with type information by using the respective function from the above type system module.

For `Tok c`, first we build the token type from tag type `c` and use the `tok` function to construct the type information of the grammar.

```
typeof env (MkGrammar _ (Alt g1 g2)) =
  do
    g1' <- typeof env g1
    g2' <- typeof env g2
    altRes <- alt (g1'.lang) (g2'.lang)
    Right (MkGrammar altRes (Alt g1' g2'))

typeof env (MkGrammar _ (Map g f)) =
  do
    g' <- typeof env g
```

```
    Right (MkGrammar g'.lang (Map g' f))

typeof env (MkGrammar  _ (Seq g1 g2)) =
  do
    g1' <- typeof env g1
    g2' <- typeof (map addGaurd env) g2
    seqRes <- seq (g1'.lang) (g2'.lang)
    Right (MkGrammar seqRes (Seq g1' g2'))
```

For the grammar type of `Alt g1 g2`, first type check the $g1$, $g2$ under the same context. After that, verify both grammars are disjoint and build the type for `Alt` using the `alt` from the above type system.

The type information of `Map g f` is just the type information of g, we just type check g and update the *lang* of the grammar with new type information.

The interesting case is `Seq g1 g2`. We first type check the $g1$, $g2$, and verify that the grammars are apart. As we already know that $g1$ should not be empty, which means every variable can be in a guarded context for $g2$; Hence, we update the context while type checking $g2$.

```
typeof env (MkGrammar _ (Fix g)) =
  do
    l <- fix (\lt => do
                       lt' <- lt
                       res <- (typeof (lt' :: env) g)
                       Right (res.lang))
    (if (not l.guarded) then (Left "Error!")
     else
      do
        g' <- typeof (l :: env) g
```

```
        Right (MkGrammar g'.lang  (Fix g')))

typeof env (MkGrammar _ (Var x)) =
  Right (MkGrammar (index (varToFin x) env) (Var x))
```

Type checking `Var x` is just looking up the variable at x in the context. But the x is a de Bruijn variable, so we convert the variable to Nat.

2.6 Parsing Algorithm

In this section, we will go over the algorithm to parse the typed grammar without backtracking.

```
public export
Parser : Type -> (Type -> Type) -> Type
Parser a tagType  =
  List (Token tagType) -> Either String (a, List (Token tagType))
```

This defines a type alias `Parser` which takes two parameters:

1. `a : Type` is the type of the result the parser will produce.
2. `tagType : Type -> Type` is a higher-kinded type representing the tag associated with each token.

This function takes:

1. A list of tagged tokens (`List (Token tagType)`) as input.
2. Returns either:
   (a) `Left String`: a parsing failure with an error message, or

(b) `Right (a, rest)`: a successful parse result of type `a`, and the remaining unparsed tokens.

```
parse : {a : Type}
     -> {ct : Vect n Type}
     -> {tagType : Type -> Type}
     -> {auto _ : Tag tagType}
     -> Grammar ct a tagType
     -> ParseEnv tagType ct
     -> Parser a tagType
parse (MkGrammar _ (Eps g)) penv = eps g
parse (MkGrammar _ (Tok c)) penv = token c
parse (MkGrammar _ Bot) penv = bot
parse (MkGrammar _ (Seq g1 g2)) penv =
  let p1 = parse g1 penv
      p2 = parse g2 penv
  in
  seq p1 p2
parse (MkGrammar _ (Alt g1 g2)) penv =
  let p1 = parse g1 penv
      p2 = parse g2 penv
  in
    alt g1.lang p1 g2.lang p2
parse (MkGrammar _ (Map g f)) penv = map f $ parse g penv
parse (MkGrammar _ (Fix g)) penv = fix (\p => parse g (p :: penv))
parse (MkGrammar _ (Var var)) penv = lookup var penv
```

`parse` function takes a typed grammar, a parser environment to store parser types(as grammar also contains the variables that can refer to context), and then returns a parser. The function pattern matches the grammar structure and calls the respective combinator to

construct the parser.

```
bot : Parser a tagType
bot _ = Left "Impossible"


eps : a -> Parser a tagType
eps v rest = Right (v, rest)
```

`bot` is a parser that never succeeds, so it takes the input and returns an error. `eps` takes a default value, a list of tokens, and returns the tuple of value and input list of tokens.

```
token : {a : Type}
     -> {tagType : Type -> Type}
     -> {auto _ : Tag tagType}
     -> (tag : tagType a)
     -> Parser a tagType
token tag [] = Left "Expected \{Token.show tag}, reached end of the stream"
token tag (Tok tag' value :: rest) =
  case (compare tag tag') of
       Eql => Right(value, rest)
       _ => Left "Expected \{Token.show tag}, got \{Token.show tag'}"
```

`token` builds the parser to parse a token with tag `tag`. If the input token list is empty, the parser fails immediately. Otherwise, it compares the expected tag `tag` with the tag in the token `tag'` using the Tag interface's compare. This is where the `compare` from the Tag interface differs from `compare` from the Ord interface. The `compare` from the Tag interface guarantees that the tags are of the same type, and the value that tags carry is also of the same type.

```
alt : {tagType : Type -> Type}
   -> {auto _ : Tag tagType}
```

```
    -> LangType (TokenType tagType)
    -> Parser a tagType
    -> LangType (TokenType tagType)
    -> Parser a tagType
    -> Parser a tagType
alt l1 p1 l2 p2 cs =
  case head' cs of
    Nothing =>  if l1.null then p1 cs
                else if l2.null then p2 cs
                else Left "Unexpected end of stream"
    Just (Tok tag v) =>
      if contains (TokType tag) l1.first then
        p1 cs
      else if contains (TokType tag) l2.first then
        p2 cs
      else if l1.null then
        p1 cs
      else if l2.null then
        p2 cs
      else
        Left "No Progress possible, unexpected token - \{Token.show tag}"
```

`alt` is most interesting of all the combinators. It uses the static type information on the languages formed by the grammar and decides which parser to use. If there is empty input, it checks which parser can parse an empty token. Otherwise, it sees which parser grammar starts with the given tag `tag` and will parse with that parser.

As we can see, it uses just a single lookahead token.

```
seq : Parser a tagType -> Parser b tagType -> Parser (a, b) tagType
seq p1 p2 cs =
```

```
  do
    (a, rest) <- p1 cs
    (b, rest) <- p2 rest
    Right ((a, b), rest)

map : (a -> b) -> Parser a tagType -> Parser b tagType
map f p cs =
  do
    (a, rest) <- p cs
    Right (f a , rest)

fix : (Parser a tagType -> Parser a tagType) -> Parser a tagType
fix f input = f (fix f) input
```

`seq` parses the input sequentially in order with the given parsers. `map` parses the input with the given parser and then applies the function. `fix` is the classic fixed-point combinator. As we know, Idris2 is a strict language unlike Haskell, hence an extra argument (`input`) is added to the usual fix implementation to avoid an infinite call sequence.

Full implementation of the library along with documentation is here [4].

Chapter 3

Examples

Before presenting the examples, we first examine the reduced API to avoid redundant code.

3.1 Reduced API

```
export
eps : {ct : Vect n Type}
   -> {tagType : Type -> Type}
   -> {auto _ : Tag tagType}
   -> a
   -> Grammar ct a tagType
eps v = MkGrammar bot (Eps v)
```

For example, if we want to build a grammar that parses an empty token, we need to write `MkGrammar bot (Eps v)`. However, this syntax is verbose and cumbersome when used repeatedly. To address this, we define a helper function that abstracts this pattern. Therefore, we can use `eps v` instead of more verbose `MkGrammar bot (Eps v)`.

```
export
bot : {ct : Vect n Type}
```

```
    -> {tagType : Type -> Type}
    -> {auto _ : Tag tagType}
    -> Grammar ct a tagType
bot = MkGrammar bot Bot

export
tok : {ct : Vect n Type}
    -> {tagType : Type -> Type}
    -> {auto _ : Tag tagType}
    -> (tagType a)
    -> Grammar ct a tagType
tok tag = MkGrammar bot (Tok tag)
```

Similarly, we can use `bot` to build a parser that always fails and `tok tag` to build a parser that parses the tokens of `tag` type.

```
export
infixl 6 <|>
export
(<|>) : {ct : Vect n Type}
    -> {tagType : Type -> Type}
    -> {auto _ : Tag tagType}
    -> Grammar ct a tagType
    -> Grammar ct a tagType
    -> Grammar ct a tagType
(<|>) a b = MkGrammar bot (Alt a b)

export
infixl 6 >>>
export
```

```
(>>>) : {ct : Vect n Type}
   -> {tagType : Type -> Type}
   -> {auto _ : Tag tagType}
   -> Grammar ct a tagType
   -> Grammar ct b tagType
   -> Grammar ct (a, b) tagType
(>>>) a b = MkGrammar bot (Seq a b)

export
infixl 6 $$
export
($$) : {a : Type}
   -> {ct : Vect n Type}
   -> {tagType : Type -> Type}
   -> {auto _ : Tag tagType}
   -> Grammar ct a tagType
   -> (a -> b)
   -> Grammar ct b tagType
($$) a f = MkGrammar bot (Map a f)
```

For convenience and clear readability, infix operators are used for grammars that have two arguments. `<|>` is used for building alternate grammars, `a <|> b` is very concise and readable than `MkGrammar bot (Alt a b)`. In the same way, `>>>` is used for sequencing the grammars and **$$** for the map.

```
export
fix : {a : Type}
   -> {ct : Vect n Type}
   -> {tagType : Type -> Type}
   -> {auto _ : Tag tagType}
```

```
    -> Grammar (a :: ct) a tagType
    -> Grammar ct a tagType
fix x = MkGrammar bot (Fix x)


export
var : {a : Type}
    -> {ct : Vect n Type}
    -> {tagType : Type -> Type}
    -> {auto _ : Tag tagType}
    -> Var a ct
    -> Grammar ct a tagType
var x = MkGrammar bot (Var x)
```

Finally, we will use `fix x` and `var x` instead of long definitions.

3.2 Helper Functions

Let's define a few helper combinators that can be used across examples.

```
export
star : {a : Type}
     -> {n : Nat}
     -> {ct : Vect n Type}
     -> {tagType : Type -> Type}
     -> {auto _ : Tag tagType}
     -> Grammar ct a tagType
     -> Grammar ct (List a) tagType
star g = fix (star' g)
  where
```

```
star' : Grammar ct a tagType -> Grammar (List a :: ct) (List a) tagType
star' g = eps [] <|> (wekeanGrammar g >>> var Z $$ (\(x, xs) => x :: xs))
```

First, we will go over the Kleene star function. The combinator `star g` builds that grammar that parses either an empty list or it first parses `g` and then sequentially parses `star g` (basically zero or more instances of g). As we mentioned earlier, we will use de Bruijn indices to refer to the recursive definitions of the functions. Here, `var Z` refers to the zeroth grammar in the context. We can observe from the type definition of `star'`, the zeroth element in context just refers to `star g` definition(hence the prepended element has type `List a`).

```
export
plus : {a : Type}
    -> {n : Nat}
    -> {ct : Vect n Type}
    -> {tagType : Type -> Type}
    -> {auto _ : Tag tagType}
    -> Grammar ct a tagType
    -> Grammar ct (List a) tagType
plus g = (g >>> star g) $$ (\(x, xs) => x :: xs)
```

`plus g`, parses one or more instances of `g`, which is just a sequence of `g` and `star g`. As `star g` returns a list, we will prepend the result of `g` to that list.

```
export
any : {ct : Vect n Type}
   -> {tagType : Type -> Type}
   -> {auto _ : Tag tagType}
   -> List (Grammar ct a tagType)
   -> Grammar ct a tagType
```

```
any gs = foldl (<|>) bot gs


export
maybe : {a : Type}
     -> {tagType : Type -> Type}
     -> {auto _ : Tag tagType}
     -> {ct : Vect n Type}
     -> Grammar ct a tagType
     -> Grammar ct (Maybe a) tagType
maybe p = any [p $$ Just, eps Nothing ]


export
between : {a, b, c : Type}
       -> {ct : Vect n Type}
       -> {tagType : Type -> Type}
       -> {auto _ : Tag tagType}
       -> Grammar ct a tagType
       -> Grammar ct b tagType
       -> Grammar ct c tagType
       -> Grammar ct b tagType
between left p right = (left >>> p >>> right) $$ (\((_, b), _) => b)
```

These are some frequently used combinators and are self-explanatory by the name, but here is a short description.

1. `any gs` - An n-ary version of `<|>`.
2. `maybe p` - Tries to parse with `p` and returns a result on success or nothing.
3. `between a b c` - Sequentially parses `a, b, c` in order and then returns between result ignoring ends.

Let's define a few more grammars for parsing characters as tokens. As we have already seen, the parser takes a list of `Token tagType`, first, we will give tags to represent characters.

```
export
data CharTag : Type -> Type where
  CT : Char -> CharTag Char

public export
Tag CharTag where
  compare (CT x) (CT y) =
    case (compare x y) of
      LT => Leq
      EQ => Eql
      GT => Geq

  show (CT c) = show c
```

Here, we use `CharTag` GADT with one constructor `CT` that carries a value of type `Char`.

`CharTag` will be `tagType` and `CT` is the actual `tag` for a character type. Then we implement the `Tag` interface for our char tag type. The `compare` is straightforward, we just compare the actual character that the tag carries.

```
export
char : {ct : Vect n Type} -> Char -> Grammar ct Char CharTag
char c = tok (CT c) $$ always c

export
charSet : {ct : Vect n Type} -> String -> Grammar ct Char CharTag
charSet str =  charSet' (unpack str)
  where
```

```
charSet' : List Char -> Grammar ct Char CharTag
charSet' [] = bot
charSet' (x :: xs) = char x <|> charSet' xs
```

`char c` build a grammar that accepts the character `c`. For example, `char 'a'` parses only if the first token of input matches with character `'a'`.

`charSet str` builds the grammar that accepts any of the characters in the given string.

Below, we present a few more helper grammars built on top of `char` and `charSet`.

```
export
digit : {ct : Vect n Type} -> Grammar ct Char CharTag
digit = charSet "0123456789"

export
lower : {ct : Vect n Type} -> Grammar ct Char CharTag
lower = charSet "abcdefghijklmnopqrstuvwxyz"

export
upper : {ct : Vect n Type} -> Grammar ct Char CharTag
upper = charSet "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

export
word : {n : Nat} -> {ct : Vect n Type}  -> Grammar ct (List Char) CharTag
word = (upper >>> star lower) $$ (\(c, cs) => c :: cs)
```

`digit` is a grammar that can parse any digit. `lower` for parsing lower case alphabets and `upper` for upper case alphabets.

Similarly, `word` parses an uppercase letter followed by zero or more lowercase letters.

```
export
whitespace : {n : Nat} -> {ct : Vect n Type} -> Grammar ct Char CharTag
```

```
whitespace = charSet " \t\n\r"


export
skipSpace : {a : Type}
         -> {n : Nat}
         -> {ct : Vect n Type}
         -> Grammar ct a CharTag
         -> Grammar ct a CharTag
skipSpace g = (whitespace >>> g) $$ snd
```

`whitespace` parses whitespace, tab, newline, or carriage return. Then `skipSpace g` skips if there is any whitespace before `g`.

3.3 S-expressions

In this section, we will build a parser to parse S-expressions. It is better to separate the process of lexing and parsing. By having it in separate stages, it becomes easier to handle things like whitespace, comments, etc.

Whenever we want to build a parser, the suggested flow is

1. Define a data type to represent tokens.
2. Define grammars to parse each token.
3. Combine all the token grammars to write a lexer grammar.
4. Write a parser grammar following the grammar rules.
5. Type check the lexer grammar and use the typed grammar to generate a parser that takes a string and returns a list of tokens.

6. Type check the parser grammar and use the typed grammar to generate a parser that takes a list of tokens and returns a result(usually an intermediate representation like AST).

The parsers are generated from the built grammars using the `generateParser` function from the library.

For this example, we describe an S-expression as either a word or a list of S-expressions between left and right parentheses.

```
export
data SToken : Type -> Type where
  Symbol : SToken String
  LParen : SToken ()
  RParen : SToken ()
```

Tokens are constructed using the `Token` constructor from the library. To build the tokens, we need to define `tags`. We represent the tags as a GADT, where the tag type is `SToken` and tags are just `Symbol`, `LParen`, `RParen` constructors.

So, the type of s-expression tokens is `Token SToken` and `Tag` interface is implemented for the tokens(for code refer Appendix A.1).

```
symbol : {n : Nat} -> {ct : Vect n Type} -> Grammar ct (Token SToken) CharTag
symbol = word $$ (\s => (Tok Symbol (pack s)))

lparen : {n : Nat} -> {ct : Vect n Type} -> Grammar ct (Token SToken) CharTag
lparen = char '(' $$ always (Tok LParen ())

rparen : {n : Nat} -> {ct : Vect n Type} -> Grammar ct (Token SToken) CharTag
rparen = char ')' $$ always (Tok RParen ())
```

`symbol` recognizes a `word` and builds a token with `Symbol` tag and parsed string as value. `lparen` recognizes a left parenthesis and builds a token with `LParen` tag and unit value. Similarly, `rparen` recognizes a right parenthesis and builds a token with `RParen` tag and unit value.

`always p` just returns a function which always return `p` irrespective of the argument.

```
sexpToken : Grammar Nil (Token SToken) CharTag
sexpToken = fix sexpToken'
  where
    sexpToken' : Grammar [Token SToken] (Token SToken) CharTag
    sexpToken' =
      any
        [ symbol
        , lparen
        , rparen
        , skipSpace (var Z)
        ]
```

We can use the above grammars to build a lexer. The lexer for S-expressions is any of the symbol, left parenthesis, or right parenthesis. Whitespace between tokens can be skipped with `skipSpace (var Z)`, which basically parses a whitespace and then refers to the lexer recursively using the variable.

```
public export
data Sexp = Sym String | Sequence (List Sexp)
```

We are going to represent the parsed s-expressions as an ADT `Sexp`. This is almost the same as the grammar rules of S-expression, except this is represented in Idris2 data types.

```
sexpression : Grammar Nil Sexp SToken
sexpression = fix sexpression'
```

```
  where
    sexpression' : Grammar [Sexp] Sexp SToken
    sexpression' =
      (tok Symbol $$ Sym) <|>
      ((between (tok LParen) (plus (var Z)) (tok RParen)) $$ Sequence)
```

`sexpression` is the grammar for s-expression parser, which parses either a `Symbol` token or a list of s-expressions(at least one) between parentheses.

As we already know from the tag types, a token with `Symbol` tag returns a value of type `String`. So, we map the parsed string to `Sym` constructor.

In the same way, the `plus (var Z)` returns a list of parsed s-expressions, which is mapped to `Sequence` constructor.

3.4 JSON

We are going to use the following grammar rules, similar to one mentioned here [25], to build the JSON parser.

```
value   ::= object | array | number | stringLiteral
            | "null" | "true" | "false"
obj     ::= "{" [members] "}"
arr     ::= "[" [values] "]"
members ::= member {"," member}
member  ::= stringLiteral ":" value
values  ::= value {"," value}
number  ::= integerValue | decimalValue
```

As usual, the first step is to define the tokens. Similar to the S-expressions, we are going to use a GADT to represent the tags for JSON tokens and then build a grammar to parse each token.

```
public export
data Number = Decimal Double | In Int

data JsonToken : Type -> Type where
  TNull : JsonToken ()
  TTrue : JsonToken ()
  TFalse : JsonToken ()
  TNumber : JsonToken Number
  TString : JsonToken String
  TLBrace : JsonToken ()
  TRBrace : JsonToken ()
  TLBracket : JsonToken ()
  TRBracket : JsonToken ()
  TColon : JsonToken ()
  TComma : JsonToken ()
```

`JsonToken` GADT represents the tag type, and each constructor represents the tag.

`Tag` interface is implemented for the tokens(for code refer Appendix A.3).

For tokens with tags like `TNumber`, `TString` also specifies what value types they are going to return while parsing in the type of their tag itself. If tokens don't have any value to carry, we represent them via the unit type.

```
lbracket : {ct : Vect n Type} -> Grammar ct (Token JsonToken) CharTag
lbracket = char '[' $$ always (Tok TLBracket ())

rbracket : {ct : Vect n Type} -> Grammar ct (Token JsonToken) CharTag
rbracket = char ']' $$ always (Tok TRBracket ())

lbrace : {ct : Vect n Type} -> Grammar ct (Token JsonToken) CharTag
lbrace = char '{' $$ always (Tok TLBrace ())
```

```
rbrace : {ct : Vect n Type} -> Grammar ct (Token JsonToken) CharTag
rbrace = char '}' $$ always (Tok TRBrace ())

comma : {ct : Vect n Type} -> Grammar ct (Token JsonToken) CharTag
comma = char ',' $$ always (Tok TComma ())

colon : {ct : Vect n Type} -> Grammar ct (Token JsonToken) CharTag
colon = char ':' $$ always (Tok TColon ())
```

`char '['` parses left bracket. In the `lbracket`, after parsing the left bracket, builds the token with `TLBracket` tag, and as we don't need any value for the left bracket, we use `()`.

A note about the type `Grammar ct (Token JsonToken) CharTag` - this means it can take `CharTag` as input and returns `Token JsonToken` as result.

We convert the string to `CharTag` tokens when passing to a parser, and then it outputs the `JsonToken` tokens. It is not a tedious process to convert a string to `CharTag` tokens, we have a helper function 'toTokens' that can build character tokens from a string.

Similarly, `rbracket`, `lbrace`, `rbrace`, `comma`, and `colon` build the grammars to parse right bracket, left and right braces, comma separator, and colon respectively.

```
nullp : {ct : Vect n Type} -> Grammar ct (Token JsonToken) CharTag
nullp = (char 'n' >>> char 'u' >>> char 'l' >>> char 'l') $$
        always (Tok TNull ())

truep : {ct : Vect n Type} -> Grammar ct (Token JsonToken) CharTag
truep = (char 't' >>> char 'r' >>> char 'u' >>> char 'e') $$
        always (Tok TTrue ())

falsep : {ct : Vect n Type} -> Grammar ct (Token JsonToken) CharTag
```

```
falsep = (char 'f' >>> char 'a' >>> char 'l' >>> char 's' >>> char 'e') $$
           always (Tok TFalse ())
```

`nullp` builds the grammar that recognizes the four characters `n, u, l, l` in sequence and builds the token with tag `TNull` to represent the null in JSON.

`truep`, `falsep` builds the grammars that recognize true, false values, and builds tokens with respective tags.

```
fullstringp : {n : Nat}
            -> {ct : Vect n Type}
            -> Grammar ct (Token JsonToken) CharTag
fullstringp = (char '"' >>> star (compCharSet "\"") >>> char '"') $$
                (\((_, s), _) => Tok TString (pack s))

number : {n : Nat}
       -> {ct : Vect n Type}
       -> Grammar ct (Token JsonToken) CharTag
number = (plus digit >>> maybe (char '.' >>> plus digit)) $$ toNumber
  where
    toNumber : (List Char, Maybe (Char, List Char)) -> Token JsonToken
    toNumber (num, Nothing) = Tok TNumber (In $ cast $ pack num)
    toNumber (num, (Just (dot, frac))) =
      Tok TNumber (Decimal $ cast $ pack (num ++ [dot] ++ frac))
```

The interesting grammars are `fullstrinp` and `number`

First, `fullstringp` builds a grammar to recognize a string. A string is anything between two double quotes. So, we first recognize one double quote, next we sequence with `star (compCharSet "\"")` which recognizes one or more any ASCII characters other than a double quote. And then we sequence it to recognize for end double quote. Once it recognizes the

given thing in order, then it just ignores the end double quotes and builds a token with `TString` tag and recognized string value.

`star (compCharSet "\"")` returns the recognized string as a list of characters, hence we need to pack using `pack` to convert to a string while building a token.

In most of the programming we deal the integers and floats separately and have separate type for them.

For that, a number can be an integer or a floating-point number. So, we represent the number with `Number` type, which either contains a double or an integer value.

`number` first recognizes one or more digits(using plus), then it uses `maybe` to see if there is a dot and one or more digits. The `maybe` grammar can return either a value or nothing. Depending on that value, we build the token for the number. For example, if `maybe` returns nothing, then the recognized number is an integer. We build the token with `TNumber` and int value(`cast` converts the recognized number, which is in string, to an integer in Idris).

```
jsonToken : Grammar Nil (Token JsonToken) CharTag
jsonToken = fix jsonToken'
  where
    jsonToken' : Grammar [Token JsonToken] (Token JsonToken) CharTag
    jsonToken' =
      any
        [ lbracket
        , rbracket
        , lbrace
        , rbrace
        , comma
        , colon
        , nullp
        , truep
        , falsep
```

```
      , fullstringp
      , number
      , skipSpace (var Z)
      ]
```

The lexer grammar just assembles the grammars for all JSON tokens and skips any whitespace between them.

A JSON token can be any of the tokens, one or more whitespace characters followed by any of the tokens. Recognizing one or more whitespace characters is a recursive grammar, hence the use of fix and var.

```
public export
data JsonValue =
    JNull
  | JBool Bool
  | JNumber Number
  | JString String
  | JArray (List JsonValue)
  | JObject (List (String, JsonValue))
```

We have a lexer that can parse the raw string and construct JSON tokens. Now, we need to figure out a way to represent JSON values in Idris. We use an ADT to represent the JSON in Idris.

1. `JNull` represents the null.
2. `JBool Bool` represents the boolean values.
3. `JNumber` represents the number and can have either a double or integer value.
4. `JArray (List JsonValue)` represents JSON array in a list.

5. `JObject (List (String, JsonValue))` represents JSON object in an association list.

```
member : {a : Type}
      -> {n : Nat}
      -> {ct : Vect n Type}
      -> Grammar ct a JsonToken
      -> Grammar ct (String, a) JsonToken
member x = (tok TString >>> tok TColon >>> x) $$ (\((key, _), val) => (key, val))

json : Grammar Nil JsonValue JsonToken
json = fix json'
  where
    json' : Grammar [JsonValue] JsonValue JsonToken
    json' =
      let object = (between
                      (tok TLBrace)
                      (sepByComma (member (var Z)))
                      (tok TRBrace)) $$
                    JObject
          array = (between (tok TLBracket) (sepByComma (var Z)) (tok TRBracket))
                  $$ JArray
          number = tok TNumber $$ JNumber
          string = tok TString $$ JString
          null = tok TNull $$ always JNull
          true = tok TTrue $$ always (JBool True)
          false = tok TFalse $$ always (JBool False) in
      any [object, array, number, string, null, true, false]
```

Implementing the actual parser is quite straightforward, it looks almost the same as the grammar rules, but in a programming language. That's one of the advantages of representing

context-free grammars as context-free expressions.

As we already saw from the grammar, we have to parse one or more values, which automatically suggests using the fix and var to refer to the non-terminal. `json` is any of the following JSON values: object, array, number, string, null, true, and false.

`number`, `string`, `null`, `true`, `false` are simply parsing the respective tokens and then mapping them to the respective representation of JSON value.

In the `array`, the JSON values are separated by commas between the brackets, and the same is defined using the `between` helper function. `sepByComma g` is a helper function that parses comma-separated values of any grammar `g` (refer to Appendix A.2 for definition). In the case of an array, the values are any JSON value, so the recursive definition is referred to via the variable `var Z`(first element in context, which is grammar for JSON).

In the `object`, the members are separated by commas between the braces.

And the `member` (`var Z` is a helper combinator to parse the string followed by a colon before a JSON value.

3.5 IMP Language

For the detailed parser implementation of the IMP language, refer to Appendix A.4.

Chapter 4

Future Work

Our library provides the combinator API in a first-order style, which simplifies type checking. However, a key drawback of this approach is the need to manually refer to recursive definitions via variables. This requires careful tracking of types within the context. When multiple values in the context share similar types, it becomes essential to ensure that the correct variable is used to reference the appropriate definition.

The original paper [16] also uses a first-order API for grammar construction and type checking, but its OCaml library [17] exposes a higher-order API. Internally, the library translates the higher-order API to first-order using the approach described in [3]. While most combinators were straightforward to translate, we encountered difficulties with the `fix` combinator. We were not able to encode enough type information to satisfy the type checker. In the original OCaml library, this limitation was circumvented by using an unsafe variant of the combinator.

Although Idris2 provides mechanisms for writing unsafe code, such as `believe_me`, we intentionally avoid this approach. We explored various ways of representing types to encode sufficient information for the type checker, but were ultimately unable to arrive at a working solution. As such, future work includes identifying a more robust method for translating higher-order representations to first-order while preserving type safety.

In addition, we want to explore the meta-programming capabilities available in Idris2 to enhance the performance of parser combinators. In particular, we are interested in techniques such as multi-staging [15] as used in the original paper [16].

Chapter 5

Conclusion

In this thesis, we presented a typed, algebraic parser combinator library for Idris2, offering a deterministic and performant alternative to traditional backtracking-based combinators. Our approach enables early detection of ambiguities by leveraging a type system for context-free expressions. The parsing algorithm from the library uses type information about grammars to avoid backtracking and ensure linear-time parsing with a single-token lookahead. We demonstrated the expressiveness and practical utility of the system by implementing parsers for S-expressions, JSON, and the IMP language. These examples showcased how the type system not only prevents ambiguous constructions but also aids in constructing complex parsers through compositional design.

Appendix A

Related Code

A.1 Tag Implementation for SToken

```
Tag SToken where
  compare Symbol Symbol = Eql
  compare Symbol _ = Leq
  compare _ Symbol = Geq

  compare LParen LParen = Eql
  compare LParen _ = Leq
  compare _ LParen = Geq

  compare RParen RParen = Eql
  compare RParen _ = Leq
  compare _ RParen = Geq

  show Symbol = "Symbol"
  show LParen = "LParen"
  show RParen = "RParen"
```

A.2 Separated By Comma Combinator

```
sepByComma : {a : Type}
          -> {n : Nat}
          -> {ct : Vect n Type}
          -> Grammar ct a JsonToken
          -> Grammar ct (List a) JsonToken
sepByComma g = fix (sepByComma' g)


  where
    sepByComma' : Grammar ct a JsonToken
               -> Grammar (List a :: ct) (List a) JsonToken
    sepByComma' g =
        eps [] <|> ((wekeanGrammar g >>> maybe (tok TComma >>> var Z)) $$ toList)


      where
        toList : (a, Maybe ((), List a)) -> List a
        toList (x, Nothing) = [x]
        toList (x, (Just y)) = x :: (snd y)
```

A.3 Tag Implementation for JsonToken

```
Tag JsonToken where
  compare TNull TNull = Eql
  compare TNull _ = Leq
  compare _ TNull = Geq


  compare TTrue TTrue = Eql
```

```
compare TTrue _ = Leq
compare _ TTrue = Geq

compare TFalse TFalse = Eql
compare TFalse _ = Leq
compare _ TFalse = Geq

compare TNumber TNumber = Eql
compare TNumber _ = Leq
compare _ TNumber = Geq

compare TString TString = Eql
compare TString _ = Leq
compare _ TString = Geq

compare TLBrace TLBrace = Eql
compare TLBrace _ = Leq
compare _ TLBrace = Geq

compare TRBrace TRBrace = Eql
compare TRBrace _ = Leq
compare _ TRBrace = Geq

compare TLBracket TLBracket = Eql
compare TLBracket _ = Leq
compare _ TLBracket = Geq

compare TRBracket TRBracket = Eql
compare TRBracket _ = Leq
```

```
  compare _ TRBracket = Geq

  compare TColon TColon = Eql
  compare TColon _ = Leq
  compare _ TColon = Geq

  compare TComma TComma = Eql
  compare TComma _ = Leq
  compare _ TComma = Geq



  show TNull = "TNull"
  show TTrue = "TTrue"
  show TFalse = "TFalse"
  show TNumber = "TNumber"
  show TString = "TString"
  show TLBrace = "TLBrace"
  show TRBrace = "TRBrace"
  show TLBracket = "TLBracket"
  show TRBracket = "TRBracket"
  show TColon = "TColon"
  show TComma = "TComma"
```

A.4 Imp Language Parser

```
module Examples.Imp

import Data.Vect
```

```
import Data.String

import Grammar
import Var
import Parser
import Token

import Examples.Utils

%hide Prelude.Ops.infixr.(<|>)

{-

Arithmetic Expressions

a ::= n | X | a0 + a1 | a0 - a1 | a0 * a1

Boolean Expressions

b ::= true | false | a0 = a1 | a0 <= a1 | !b | b0 && b1 | b0 || b1 | (b)

Commands

c ::= skip | X := a | c0;c1 | if b then c0 else c1 done
      | while b do c done | (c)

-}

keywords : Vect 9 String
```

```
keywords =
  ["if"
  , "then"
  , "else"
  , "true"
  , "false"
  , "skip"
  , "while"
  , "do"
  , "done"
  ]


data Aop = APlus | AMinus | AMult
data Acmp = ALte | AEq
data Bop = BAnd | BOr


data IToken : Type -> Type where
  IInt : IToken Int
  ILoc : IToken String
  IPlus : IToken Aop
  IMinus : IToken Aop
  IMult : IToken Aop
  ITrue : IToken ()
  IFalse : IToken ()
  IEqual : IToken Acmp
  ILTE : IToken Acmp
  INot : IToken ()
  IAnd : IToken Bop
  IOr : IToken Bop
```

```
  ISkip : IToken ()
  IAssign : IToken ()
  ISeq : IToken ()
  IIf : IToken ()
  IThen : IToken ()
  IElse : IToken ()
  IDone : IToken ()
  IWhile : IToken ()
  IDo : IToken ()
  ILparen : IToken ()
  IRParen : IToken ()

Tag IToken where
  compare IInt     IInt     = Eql
  compare IInt     _        = Leq
  compare _        IInt     = Geq

  compare ILoc     ILoc     = Eql
  compare ILoc     _        = Leq
  compare _        ILoc     = Geq

  compare IPlus    IPlus    = Eql
  compare IPlus    _        = Leq
  compare _        IPlus    = Geq

  compare IMinus   IMinus   = Eql
  compare IMinus   _        = Leq
  compare _        IMinus   = Geq
```

```
compare IMult    IMult    = Eql
compare IMult    _        = Leq
compare _        IMult    = Geq

compare ITrue    ITrue    = Eql
compare ITrue    _        = Leq
compare _        ITrue    = Geq

compare IFalse   IFalse   = Eql
compare IFalse   _        = Leq
compare _        IFalse   = Geq

compare IEqual   IEqual   = Eql
compare IEqual   _        = Leq
compare _        IEqual   = Geq

compare ILTE     ILTE     = Eql
compare ILTE     _        = Leq
compare _        ILTE     = Geq

compare INot     INot     = Eql
compare INot     _        = Leq
compare _        INot     = Geq

compare IAnd     IAnd     = Eql
compare IAnd     _        = Leq
compare _        IAnd     = Geq

compare IOr      IOr      = Eql
```

```
compare IOr      _        = Leq
compare _        IOr      = Geq

compare ISkip    ISkip    = Eql
compare ISkip    _        = Leq
compare _        ISkip    = Geq

compare IAssign  IAssign  = Eql
compare IAssign  _        = Leq
compare _        IAssign  = Geq

compare ISeq     ISeq     = Eql
compare ISeq     _        = Leq
compare _        ISeq     = Geq

compare IIf      IIf      = Eql
compare IIf      _        = Leq
compare _        IIf      = Geq

compare IThen    IThen    = Eql
compare IThen    _        = Leq
compare _        IThen    = Geq

compare IElse    IElse    = Eql
compare IElse    _        = Leq
compare _        IElse    = Geq

compare IDone    IDone    = Eql
compare IDone    _        = Leq
```

```
compare _        IDone    = Geq

compare IWhile   IWhile   = Eql
compare IWhile   _        = Leq
compare _        IWhile   = Geq

compare IDo      IDo      = Eql
compare IDo      _        = Leq
compare _        IDo      = Geq

compare ILparen  ILparen  = Eql
compare ILparen  _        = Leq
compare _        ILparen  = Geq

compare IRParen  IRParen  = Eql
compare IRParen  _        = Leq
compare _        IRParen  = Geq



show IInt     = "IInt"
show ILoc     = "ILoc"
show IPlus    = "IPlus"
show IMinus   = "IMinus"
show IMult    = "IMult"
show ITrue    = "ITrue"
show IFalse   = "IFalse"
show IEqual   = "IEqual"
show ILTE     = "ILTE"
show INot     = "INot"
```

```
  show IAnd     = "IAnd"
  show IOr      = "IOr"
  show ISkip    = "ISkip"
  show IAssign  = "IAssign"
  show ISeq     = "ISeq"
  show IIf      = "IIf"
  show IThen    = "IThen"
  show IElse    = "IElse"
  show IDone    = "IDone"
  show IWhile   = "IWhile"
  show IDo      = "IDo"
  show ILparen  = "ILparen"
  show IRParen  = "IRParen"



lparen : {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
lparen = char '(' $$ always (Tok ILparen ())

rparen : {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
rparen = char ')' $$ always (Tok IRParen ())

intp : {n : Nat} -> {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
intp = plus digit $$ (\xs => Tok IInt (cast $ pack xs))

stp : {n : Nat} -> {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
stp = (any [lower, upper] >>> star (any [lower, upper, digit]))
      $$ (\((x, xs)) => toToken $ pack (x :: xs))
  where
```

```
    toToken : String -> Token IToken
    toToken "if" = Tok IIf ()
    toToken "then" = Tok IThen ()
    toToken "else" = Tok IElse ()
    toToken "done" = Tok IDone ()
    toToken "true" = Tok ITrue ()
    toToken "false" = Tok IFalse ()
    toToken "skip" = Tok ISkip ()
    toToken "while" = Tok IWhile ()
    toToken "do" = Tok IDo ()
    toToken str = Tok ILoc str

plus : {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
plus = char '+' $$ always (Tok IPlus APlus)

minus : {n : Nat } -> {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
minus = ((char '-') >>> (maybe (plus digit))) $$ toToken

  where
    toToken : (Char, Maybe (List Char)) -> (Token IToken)
    toToken (x, Nothing) = Tok IMinus AMinus
    toToken (x, (Just rest)) = Tok IInt (cast $ pack (x :: rest))

mult : {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
mult = char '*' $$ always (Tok IMult AMult)

equal : {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
equal = char '=' $$ always (Tok IEqual AEq)
```

```
lte : {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
lte = (char '<' >>> char '=') $$ always (Tok ILTE ALte)

not : {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
not = char '!' $$ always (Tok INot ())

and : {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
and = (char '&' >>> char '&') $$ always (Tok IAnd BAnd)

or : {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
or = (char '|' >>> char '|') $$ always (Tok IOr BOr)

assign : {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
assign = (char ':' >>> char '=') $$  always (Tok IAssign ())

seq : {ct : Vect n Type} -> Grammar ct (Token IToken) CharTag
seq = char ';' $$ always (Tok ISeq ())

impToken : Grammar Nil (Token IToken) CharTag
impToken = fix impToken'

  where
    impToken' : Grammar [Token IToken] (Token IToken) CharTag
    impToken' =
      any
        [ lparen
        , rparen
        , intp
        , stp
```

```
        , plus
        , minus
        , mult
        , equal
        , lte
        , not
        , and
        , or
        , assign
        , seq
        , skipSpace (var Z)
        ]


public export
data AExp =
  VInt Int
  | Loc String
  | Plus (AExp, AExp)
  | Minus (AExp, AExp)
  | Mult (AExp, AExp)

Eq AExp where
  (VInt i) == (VInt j) = i == j
  (Loc id1) == (Loc id2) =  id1 == id2
  (Plus (a1, a2)) == (Plus (b1, b2)) = (a1 == b1 && a2 == b2)
  (Minus (a1, a2)) == (Minus (b1, b2)) =  (a1 == b1 && a2 == b2)
  (Mult (a1, a2)) == (Mult (b1, b2)) = (a1 == b1 && a2 == b2)
  _ == _ = False
```

```
export
Show AExp where
  show y = case y of
              (VInt i) => "VInt " ++ show i
              (Loc id) => "Loc " ++ show id
              (Plus x) => "Plus " ++ show' x
              (Minus x) => "Minus " ++ show' x
              (Mult x) => "Mult " ++ show' x
    where
      show' : (AExp, AExp) -> String
      show' (a1, a2) = "(" ++ show a1 ++ ", " ++ show a2 ++ ")"

public export
data BExp =
  VTrue
  | VFalse
  | Eq (AExp, AExp)
  | LTE (AExp, AExp)
  | Not BExp
  | And (BExp, BExp)
  | Or (BExp, BExp)

Eq BExp where
  VTrue == VTrue = True
  VFalse == VFalse = True
  (Eq (a1, a2)) == (Eq (b1, b2)) = (a1 == b1 && a2 == b2)
  (LTE (a1, a2)) == (LTE (b1, b2)) = (a1 == b1 && a2 == b2)
  (Not x) == (Not y) = x == y
```

```
  (And (a1, a2)) == (And (b1, b2)) = (a1 == b1 && a2 == b2)
  (Or (a1, a2)) == (Or (b1, b2)) = (a1 == b1 && a2 == b2)
  _ == _ = False

export
Show BExp where
  show VTrue = "VTrue"
  show VFalse = "VFalse"
  show (Eq x) = "Eq" ++ showParens True (show x)
  show (LTE x) = "LTE " ++ showParens True (show x)
  show (Not x) = "Not " ++ showParens True (show x)
  show (And x) = "And " ++ show' x where
    show' : (BExp, BExp) -> String
    show' (b1, b2) = "(" ++ show b1 ++ ", " ++ show b2 ++ ")"
  show (Or x) = "Or " ++ show' x where
    show' : (BExp, BExp) -> String
    show' (b1, b2) = "(" ++ show b1 ++ ", " ++ show b2 ++ ")"

public export
data Command =
  Skip
  | Assign (String, AExp)
  | Seq (Command, Command)
  | ITE (BExp, Command, Command)
  | While (BExp, Command)

export
Eq Command where
  Skip == Skip = True
```

```
  (Assign x) == (Assign y) = x == y
  (Seq (a1, a2)) == (Seq (b1, b2)) = (a1 == b1 && a2 == b2)
  (ITE (b, c1, c2)) == (ITE (b', c3, c4)) = (b == b' && c1 == c3 && c2 == c4)
  (While (b1, c1)) == (While (b2, c2)) = (b1 == b2 && c1 == c2)
  _ == _ = False

export
Show Command where
  show Skip = "Skip"
  show (Assign x) = "Assign " ++ show x
  show (Seq x) = "Seq " ++ show' x where
    show' : (Command, Command) -> String
    show' (c1, c2) = "(" ++ show c1 ++ ", " ++ show c2 ++ ")"
  show (ITE x) = "ITE " ++ show' x where
    show' : (BExp, Command, Command) -> String
    show' (b, c1, c2) =
      "(" ++ show b ++ ", " ++ show c1 ++ ", " ++ show c2 ++ ")"
  show (While x) = "While " ++ show' x where
    show' : (BExp, Command) -> String
    show' (b, c) = "(" ++ show b ++ ", " ++ show c ++ ")"

paren : {a : Type}
     -> {n : Nat}
     -> {ct : Vect n Type}
     -> Grammar ct a IToken
     -> Grammar ct a IToken
paren p = between (tok ILparen) p (tok IRParen)

arith :  {n : Nat} -> {ct : Vect n Type} -> Grammar ct AExp IToken
```

```
arith =  fix arith'

  where
    arith' : {n : Nat}
          -> {ct' : Vect n Type}
          -> Grammar (AExp :: ct') AExp IToken
    arith' =
      let int = tok IInt $$ VInt
          id = tok ILoc $$ Loc
          toks = any [int, id]
      in (toks >>> maybe (any [tok IPlus, tok IMinus, tok IMult] >>> var Z)) $$
          toAExp

      where
        toAExp : (AExp, Maybe (Aop, AExp)) -> AExp
        toAExp (x, Nothing) = x
        toAExp (x, Just (APlus, z)) = Plus (x, z)
        toAExp (x, Just (AMinus, z)) = Minus (x, z)
        toAExp (x, Just (AMult, Plus(a1, a2))) = Plus ((Mult (x, a1), a2))
        toAExp (x, Just (AMult, Minus(a1, a2))) = Minus (Mult (x, a1), a2)
        toAExp (x, Just (AMult, z)) = Mult (x, z)


bool :  {n : Nat} -> {ct : Vect n Type} -> Grammar ct BExp IToken
bool = fix bool'
  where
    bool' : {n : Nat}
         -> {ct' : Vect n Type}
         -> Grammar (BExp :: ct') BExp IToken
```

```
    bool' =
      let true = tok ITrue $$ always VTrue
          false = tok IFalse $$ always VFalse
          eq =  (arith >>> any [tok IEqual, tok ILTE] >>> arith) $$
                (\((a1, op), a2) => case op of
                                      AEq => Eq (a1, a2)
                                      ALte => LTE (a1, a2))
          te = any [paren (var Z), true, false, eq]
          nt = (tok INot >>> te) $$  (\(_, xs) => Not xs)
          tes = (te >>> (star (any [tok IAnd, tok IOr] >>> any [te, nt]))) $$
                (\(x, xs) =>
                     foldl (\acc, (op , rem) =>
                       case op of
                            BAnd => And (acc, rem)
                            BOr => Or (acc, rem)) x xs)
          ntes = (nt >>> (star (any [tok IAnd, tok IOr] >>> any [te, nt]))) $$
                 (\(x, xs) =>
                       foldl (\acc, (op , rem) =>
                         case op of
                              BAnd => And (acc, rem)
                              BOr => Or (acc, rem)) x xs)
      in
      any [tes, ntes]


command : Grammar Nil Command IToken
command = fix command'
  where
    command' : Grammar [Command] Command IToken
```

```
    command' = (any [baseCommand, paren baseCommand]
                >>> maybe (tok ISeq >>> var Z)) $$
                (\(b, ms) => case ms of
                                  Nothing => b
                                  Just (_, c) => Seq (b, c))
    where
     baseCommand : Grammar [Command] Command IToken
     baseCommand =
      let skip    = tok ISkip $$ always Skip
          assign  = (tok ILoc >>> tok IAssign >>> arith) $$
                    (\((id, _), aexp) => Assign (id, aexp))
          ifelse  = (tok IIf >>> bool >>> tok IThen >>> var Z >>> tok IElse >>>
                     var Z >>> tok IDone) $$
                     (\(((((((_, b), _), c1), _), c2), _)) => ITE (b, c1, c2))
          whiledo = (tok IWhile >>> bool >>> tok IDo >>> var Z >>> tok IDone) $$
                     (\((((((_, b), _), c), _)) => While (b, c))
      in any [skip, assign, whiledo, ifelse]

export
parseArith : String -> Either String AExp
parseArith input = do
  lexedTokens <- lexer impToken input
  parser arith lexedTokens

export
parseBool : String -> Either String BExp
parseBool input = do
  lexedTokens <- lexer impToken input
  parser bool lexedTokens
```

```
export
parseCommand : String -> Either String Command
parseCommand input = do
  lexedTokens <- lexer impToken (trim input)
  parser command lexedTokens
```

Bibliography

[1] Guillaume Allais. "agdarsec – Total Parser Combinators". In: 2017. URL: https://api.semanticscholar.org/CorpusID:91183833.

[2] Guillaume Allais and idris-tparsec contributors. *TParsec — Total Parser Combinators in Idris*. Accessed 30 April 2025. idris-tparsec Project. 2019. URL: https://github.com/gallais/idris-tparsec.

[3] Robert Atkey, Sam Lindley, and Jeremy Yallop. "Unembedding domain-specific languages". In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*. Haskell '09. Edinburgh, Scotland: Association for Computing Machinery, 2009, pp. 37–48. ISBN: 9781605585086. DOI: 10.1145/1596638.1596644. URL: https://doi.org/10.1145/1596638.1596644.

[4] Vamsi Krishna Bellam. *idris2-tap — Typed, Algebraic Parser in Idris2*. 2025. URL: https://github.com/vamsi-bellam/idris2-tap.

[5] Edwin Brady. *Idris 2: Quantitative Type Theory in Practice*. 2021. arXiv: 2104.00480 [cs.PL]. URL: https://arxiv.org/abs/2104.00480.

[6] N. Chomsky. "Three models for the description of language". In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 113–124. DOI: 10.1109/TIT.1956.1056813.

[7] Spiros Eliopoulos and Angstrom contributors. *Angstrom — Parser combinators built for speed and memory efficiency*. Accessed 30 April 2025. Inhabited Type LLC. 2016. URL: https://github.com/inhabitedtype/angstrom.

[8] Spiros Eliopoulos and http/af contributors. *http/af — High-performance web server for OCaml*. Latest tag v0.7.1 (30 Mar 2021); accessed 30 Apr 2025. Inhabited Type LLC. 2016. URL: https://github.com/inhabitedtype/httpaf.

[9] G. Fischer, J. Lusiardi, and J. Wolff von Gudenberg. "Abstract Syntax Trees - and their Role in Model Driven Software Development". In: *International Conference on Software Engineering Advances (ICSEA 2007)*. 2007, pp. 38–38. DOI: 10.1109/ICSEA.2007.12.

[10] GHC development team. *GHC — Glasgow Haskell Compiler*. Mirror of the official repository; accessed 30 April 2025. GHC. 1990. URL: https://github.com/ghc/ghc.

[11] Andy Gill and Simon Marlow. *Introduction — Happy documentation*. Accessed on 30 April 2025. Happy Developers. 2022. URL: https://haskell-happy.readthedocs.io/en/latest/introduction.html.

[12] Dick Grune and Ceriel J. H. Jacobs. *Parsing techniques : a practical guide*. eng. 2nd ed. Monographs in computer science. New York: Springer, 2008. ISBN: 1-281-10822-7.

[13] Stefan Höck and idris2-parser contributors. *idris2-parser — Total lexers and parsers for Idris 2*. Accessed 30 April 2025. idris2-parser Project. 2022. URL: https://github.com/stefan-hoeck/idris2-parser.

[14] Graham Hutton and Erik Meijer. *Monadic parser combinators*. 1996. URL: https://nottingham-repository.worktribe.com/output/1024440.

[15] Oleg Kiselyov. "The Design and Implementation of BER MetaOCaml - System Description". In: *Fuji International Symposium on Functional and Logic Programming*. 2014. URL: https://api.semanticscholar.org/CorpusID:9880167.

[16] Neelakantan R. Krishnaswami and Jeremy Yallop. "A typed, algebraic approach to parsing". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 379–393. ISBN: 9781450367127. DOI: 10.1145/3314221.3314625. URL: https://doi.org/10.1145/3314221.3314625.

[17] Neelakantan R. Krishnaswami, Jeremy Yallop, and ocaml-asp contributors. *asp - Algebraic, staged parsing for OCaml: typed, compositional, and faster than yacc*. Accessed 3 May 2025. asp Project. 2019. URL: https://github.com/yallop/ocaml-asp.

[18] Daan Leijen. "Parsec, a fast combinator parser". In: 2001. URL: https://api.semanticscholar.org/CorpusID:59645871.

[19] Daan Leijen et al. *Parsec — Monadic parser combinators*. Version 3.1.18.0, accessed 30 April 2025. Hackage (Haskell.org). 2001. URL: https://hackage.haskell.org/package/parsec.

[20] Hans Leiss. "Towards Kleene Algebra with Recursion". In: *Annual Conference for Computer Science Logic*. 1991. URL: https://api.semanticscholar.org/CorpusID:1325964.

[21] John MacFarlane. *Pandoc — a universal document converter*. Accessed 30 April 2025. Pandoc Project. 2006. URL: https://pandoc.org/.

[22] Ulf Norell et al. *The Agda Wiki*. Page last updated 15 March 2024; accessed 30 April 2025. Programming Logic Group, Chalmers University of Technology and University of Gothenburg. 2008. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php.

[23] Bryan O'Sullivan and Attoparsec contributors. *attoparsec — Fast combinator parsing for bytestrings and text*. Version 0.14.4, accessed 30 April 2025. Hackage (Haskell.org). 2010. URL: https://hackage.haskell.org/package/attoparsec.

[24] OCaml development team. *OCaml — The core system: compilers, runtime, base libraries*. Accessed 30 April 2025. OCaml. 1996. URL: https://github.com/ocaml/ocaml.

[25] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2008. ISBN: 9780981531601. URL: https://books.google.com/books?id=MFjNhTjeQKkC.

[26] François Pottier and Yann Régis-Gianas. *Menhir — LR(1) parser generator for OCaml*. Accessed on 30 April 2025. INRIA. 2005. URL: https://gallium.inria.fr/~fpottier/menhir/.

[27] Matúš Tejiščák and Lightyear contributors. *Lightyear — Lightweight parser combinators for Idris*. Accessed 30 April 2025. Lightyear Project. 2013. URL: https://github.com/ziman/lightyear.

[28] The Idris 2 Community. *Data.String.Parser — Module documentation (Idris 2 v0.3.0-test)*. Accessed 30 April 2025. Idris 2 Project. 2020. URL: https://www.idris-lang.org/docs/idris2/0.3.0-test/contrib/docs/Data.String.Parser.html.

[29] The Rocq development team. *The Rocq Prover — a trustworthy interactive theorem prover and dependently-typed programming language*. Accessed 30 April 2025. Inria / Rocq Consortium. 1989. URL: https://rocq-prover.org/.