Certified CYK parsing of context-free languages

Denis Firsov, Tarmo Uustalu

Institute of Cybernetics at TUT, Tallinn, Estonia

Abstract

We describe our work on certified parsing for context-free grammars. In our development we implement the Cocke–Younger–Kasami parsing algorithm and prove it correct using the Agda dependently typed programming language.

1. Introduction

In previous work [4], we implemented a certified parser-generator for regular expressions based on the Boolean matrix representation of finite-state automata using the dependently typed programming language Agda [7]. Here we want to show that a similar thing can be done for a wider class of languages.

We decided to implement the Cocke–Younger–Kasami (CYK) parsing algorithm for context-free grammars [11], because of its simple and elegant structure. The original algorithm is based on multiplication of matrices over sets of nonterminals. We digress slightly from this classical approach and use matrices over sets of parse trees. By this we immediately achieve soundness of the parsing function and also eliminate the final step of parse tree reconstruction.

We first develop a simple functional version that is easily seen to be correct. Then we add memoization, thanks to whom the imperative CYK algorithm is actually usable and efficient. We show that the memoizing version computes the same results, but without the excessive recomputation of intermediate results. The memoized version achieves $O(n^3)$ time complexity for non-ambiguous grammars and can be exponential for ambiguous ones since the algorithm is complete – it generates all possible parse trees.

Valiant [9] showed how to modify the CYK algorithm so as to use Boolean matrix multiplication (by viewing sets of non-terminals as binary words of some fixed lengths). We avoid this approach because of the details of this lower-level encoding that obfuscate the higher-level structure of the algorithm.

The paper is structured as follows. We first present the naive version of the algorithm, show it correct and terminating. Then we show how memoization can be introduced systematically, maintaining the correctness guarantee. Since we have chosen to represent matrices as association lists, our development is heavily based on manipulation of lists and reasoning about them. As we explain in the last section, we structure it using the

Email addresses: `denis@cs.ioc.ee` (Denis Firsov), `tarmo@cs.ioc.ee` (Tarmo Uustalu) list monad and some theorems about lists. To avoid notational clutter, in the paper we employ an easy-to-read unofficial list comprehension syntax for monadic code instead of the monad operations.

The Agda development is available online at `http://cs.ioc.ee/~denis/cert-cfg`.

2. The algorithm

2.1. Context-free grammars

We will work with context-free grammars in Chomsky normal form.

Let `N` and `T` be two globally fixed types with decidable equality for nonterminals and terminals respectively. We define an abbreviation `String = List T`.

Normal-form rules are elements of the variant type `Rule`.

```
data Rule : Set where
  _⟶_   : N → T → Rule       -- from nonterminal to terminal
  _⟶_•_ : N → N → N → Rule  -- from nonterminal to two nonterminals
```

We also define an abbreviation `Rules = List Rule`.

A context-free grammar is a record of type `Grammar` specifying the rules and the start nonterminal. In addition we must know if the empty string is to be accepted.

```
record Grammar : Set where
  field
    nullable : Bool           -- does grammar accept empty string?
    Rs       : Rules          -- list of rules
    S        : N              -- start nonterminal
    S-axiom₁ : ∀ {A B} → (A ⟶ S • B) ∉ Rs
    S-axiom₂ : ∀ {A B} → (A ⟶ B • S) ∉ Rs
```

The two last fields state that the start nonterminal is never encountered on the right-hand side of the rule.

Henceforth, we assume one fixed grammar `G` in the context, so we use the fields of the grammar record directly, i.e. `Rs` and `nullable` instead of `Rs G` and `nullable G` for some `G`.

2.2. Parsing relation

Before describing the parsing algorithm, we must define correctly constructed parse trees.

We define the parsing relation for our fixed grammar `G` as an inductive predicate:.

```
data _[_,_)▶_ (s : String) : ℕ → ℕ → N → Set where
  empt : ∀{i} → nullable ≡ true → s[ i, i )▶ S
  sngl : ∀{i A} → (A ⟶ charAt i s) ∈ Rs
           → s[ i, suc i )▶ A
  cons : ∀ {i j k A B C} → (A ⟶ B • C) ∈ Rs
           → s[ i, j )▶ B
           → s[ j, k )▶ C
           → s[ i, k )▶ A
```

(Arguments enclosed in curly braces are implicit. The type checker will try to figure out the argument value for you. If the type checker cannot infer an implicit argument, then it must be provided explicitly.)

The proposition `s[ i, j )▶ A` states that the substring of `s` from the `i`-th position (inclusive) to the `j`-th (exclusive) is derivable from nonterminal `A`. Proofs of this proposition are parse trees.

In particular, the string `s` is in the language of the grammar if it is derivable from `S`, i.e., we have a proof of `s[ 0, length s )▶ S`.

1. If `nullable` is true, then `empt` constructs a parse tree for the empty word. Note that in this case both indices are equal.
2. If the `i`-th terminal in the string `s` is `x` and `A ⟶ x ∈ Rs` for some `A` then the constructor `sngl` builds a parse tree for terminal `x`.
3. If `t₁` is a parse tree starting from `B` and `t₂` is a parse tree from `C` and there is a rule `A ⟶ B • C ∈ Rs` for some `A`, then the constructor `cons` combines those trees into a tree starting from `A`.

2.3. Parsing algorithm

The algorithm works with matrices of sets of parse trees where the rows and columns correspond to two positions in some string `s`. The only allowed entries at a position `i`, `j` are parse trees from various nonterminals for the substring of `s` from the `i`-th to the `j`-th position.

We represent matrices as association lists (row, column, entry). Note that we allow multiple entries in the same row and column of a matrix, corresponding to different parse trees for the same substring (and possibly the same nonterminal).

```
Mtrx : String → Set
Mtrx s = List (∃[i : ℕ] ∃[j : ℕ] ∃[A : N] s[ i, j )▶ A)
```

Let `m₁` and `m₂` be two matrices for the same string `s`. Their product of `m₁` and `m₂` (in the ordinary sense of the product of two matrices) is defined as:

```
_*_  : Mtrx s → Mtrx s → Mtrx s
m₁ * m₂ = { (i, k, A, cons _ t₁ t₂) | (i, j, B, t₁) ← m₁,
               (j, k, C, t₂) ← m₂, (A ⟶ B • C) ← Rs }
```

Next we define a function `triples` which, given a natural number `n`, enumerates all pairs of natural numbers which add up to `n`:

```
triples : (n : ℕ) → List (∃[i : ℕ] ∃[j : ℕ] i + j ≡ n)
triples = { (i, n - i, +-eq n i) | i ← [0 ... n] }
  where
    +-eq : (n : ℕ)(i : [0 ... n]) → i + (n - i) ≡ n
    +-eq = ...
```

For a matrix `m` we define raising `m` to the `n`-th power as:

```
pow : Mtrx s → ℕ → Mtrx s
pow m zero = if nullable
               then { (S, i, i, empt _) | i ← [0 ... length s) }
               else []
pow m (suc zero) = m
pow m (suc (suc n)) = { t | (i, j, _) ← triples n,
                          t ← pow m (suc i) * pow m (suc j) }
```

Note that this function is not structurally recursive. The numbers `suc i`, `suc j` returned by `triples` are in fact smaller than `suc (suc n)`, but not by definition, only provably.

The CYK parsing algorithm takes a string `s` and checks if `s` can be derived from the start nonterminal `S`. Our version of the algorithm also returns a list of all possible derivations (trees) of string `s` from all nonterminals.

We record all parse trees of all length-1 substrings of `s` in the matrix `m-init s`:

```
m-init : (s : String) → Mtrx s
m-init s = { (i, suc i, A, sngl _) | i ← [0 ... length s),
                          (A ⟶ charAt i s) ← Rs }
```

As a result, `pow (m-init s) n` contains exactly all parse trees of all length-`n` substrings of `s`. Indeed, the intuition is follows. The empty string is parsed if the grammar is nullable. And any string of length 2 or longer has as its parse trees given by a binary rule and parse trees for shorter strings. We give a formal correctness argument soon.

To find the parse trees of the full string `s` for the start nonterminal `S`, we compute `pow (m-init s) (length s)` and extract the parse trees for `S`.

```
cyk-parse : (s : String) → Mtrx s
cyk-parse s = pow (m-init s) (length s)

cyk-parse-main : (s : String) → List (s[ 0, length s )▶ S)
cyk-parse-main s =  { (_, _, A, t) ← cyk-parse s, A == S }
```

3. Correctness

Correctness of the algorithm means that it defines the same parse trees as the parsing relation. We break it down into soundness and completeness.

Soundness in the sense that `pow (m-init s) n` produces good parse trees of substrings of `s` is immediate by typing. With minimal reasoning we can also conclude

```
sound : (s : String)(i j : ℕ)(A : T)(t : s[ i, j )▶ A)(n : ℕ) →
        (i, j, A, t) ∈ pow (m-init s) n → j ≡ n + i
```

i.e., only substrings of length `n` are derived at stage `n`.

To prove completeness, we need to show that `triples n` contains all possible combinations of natural numbers `i` and `j` such that `i + j ≡ n`

```
triples-complete : (i j n : ℕ) →
        (prf : i + j ≡ n) → (i, j, prf) ∈ triples n
```

This property is proved by induction on `n`.

It is easily proved that a proof of `s[ i, j )▶ A` with `A` not equal to `S` parses a non-empty substring of `s`.

```
compl-help : (s : String) (i j : ℕ) (A : T) →
        s[ i, j )▶ A → A ≢ S → ∃[n : ℕ] j ≡ suc n + i
```

Now, we are ready to show that the parsing algorithm is complete.

```
complete : (s : String) (i n : ℕ) (A : T) →
        (t : s[ i, n + i )▶ A) → (i, n + i, A, t) ∈ pow (m-init s) n
```

The proof is by induction on the parse tree `t`. Let us analyze the possible cases:

- If `t = empt prf` for some `prf` of type `nullable ≡ true`, then `n = 0` and the first defining equation of the function `pow` applies:

  ```
  pow (m-init s) 0 = [(S, i, i, empt _) | i ← [0 ... length s)].
  ```

  which clearly contains `t`.

- If `t = sngl p` for some `p` of type `A ⟶ charAt i s ∈ Rs`, then `n = 1` and `pow (m-init s) 1 = m-init s`. By definition, `m-init s` contains all possible derivations of single terminals found in `s`.

- In the third case `t = cons p` t_1 t_2 for some `p` of type `A ⟶ B • C ∈ Rs`, t_1 of type `s[ i, j )▶ B`, t_2 of type `s[ j, n + i )▶ C`.

  - If `B` or `C` are equal to `S`, then we get contradiction with S-axiom$_1$ or S-axiom$_2$ respectively.
  - If neither `B` or `C` is equal to `S`, then by `compl-help` we get `j = suc c + i` for some `c` and `n + i = suc d + suc c + i`, for some `d`. By the induction hypothesis we get that (`B, i, suc c + i,` t_1) ∈ `pow (m-init s) (suc c)` and (`suc c + i, suc d + suc c + i, C,` t_2)∈ `pow (m-init s) (suc d)`. Hence, from the definition of multiplication and `A ⟶ B • C ∈ Rs` we conclude that

    ```
    (i, suc d + suc i + i, A, t) ∈ pow (m-init s) (suc c) ∗
                                   pow (m-init s) (suc d)
    ```

    From `n + i = suc d + suc c + i` we get `n = suc (suc (c + d))` and by `triples-complete` we know that `(c, d, refl) ∈ triples (c + d)` where `refl : c + d ≡ c + d`. Since `pow` computes and unions all the products of pairs returned by `triples (c + d)` we conclude that

    ```
    pow (m-init s) (suc c) ∗ pow (m-init s) (suc d)
                                     ⊆  pow (m-init s) n.
    ```

    which completes the proof.

Note that this proof of correctness makes explicit the induction principles employed and other details which are usually left implicit in textbook expositions of the algorithm.

In our Agda development, we have implemented the algorithm together with the completeness proofs just shown. The most interesting part of implementation is the design of data structures together with some useful invariants which support smooth formal proofs.

4. Termination

For the logic of Agda to be consistent all functions must be terminating. This is statically checked by Agda's termination checker. So, it is the duty of a programmer to provide sufficiently convincing arguments.

The definition of `pow` given above is not recognized by Agda as terminating, even if it actually terminates.

The reason is that Agda accepts recursive calls on definitionally structurally smaller arguments of an inductive type. In our case, however, a call of `pow` on `suc (suc n)` leads to calls on `suc i` and `suc j` where `(i, j, prf) ∈ Triples n`, i.e., to calls on provably smaller numbers (and not on, say, just `suc n` or `n`).

To make our definition acceptable not only to Agda's type-checker, but also the termination-checker, we have to explain Agda that we make recursive calls along a well-founded relation.

Classically we can say that a relation is well-founded, if it contains no infinite descending chains. An adequate constructive version uses the notion of *accessibility*.

An element `x` of a set `A` is called accessible with respect to some relation `_≺_` if all elements related to `x` are accessible. Crucially, this definition is to be read inductively.

```
data Acc {A : Set}(_≺_: A → A → Set)(x : A) : Set where
  acc : ((y : A) → y ≺ x → Acc _≺_ y) → Acc _≺_ x
```

A relation can be said to be well-founded if all elements in the carrier set are accessible.

```
Well-founded : {A : Set}(_≺_: A → A → Set) → Set
Well-founded = (x : A) → Acc _≺_ x
```

Now we can define the less-than relation `_<_` on natural numbers:

```
data _<_ (m : ℕ) : ℕ → Set where
  <-base : m < suc m
  <-step : {n : ℕ} → m < n → m < suc n
```

And we can prove that relation `_<_` is well-founded.

```
<-wf : Well-founded _<_
```

Finally, we can summarize everything and give a definition of `pow` that is structurally recursive on the proof of accessibility, and by doing so, discharge the obligations of the termination checker:

```
pow' : {s : String } → Mtrx s → (n : ℕ) → Acc _<_ n → Mtrx s
pow' m zero accn = if nullable
                      then { (S, i, i, empt _) | i ← [0 ... length s) }
                      else []
pow' m (suc zero) accn = m
pow' m (suc (suc n)) (acc acf) = { t | (i, j, prf) ← triples n,
     t ← (pow' m (suc i) (acf (suc i) (<-lem₁ prf))) *
          (pow' m (suc j) (acf (suc j) (<-lem₂ prf))) }
    where
      <-lem₁ : ∀{i j n} → i + j ≡ n → suc i < suc (suc n)
      <-lem₂ : ∀{i j n} → i + j ≡ n → suc j < suc (suc n)

pow : {s : String} → Mtrx s → (n : ℕ) → Mtrx s
pow m n = pow' m n (<-wf n)
```

5. Memoization

Our implementation of the algorithm is well-founded recursive on the less-than relation. Without memoization, it involves excessive recomputation of the matrices `pow m n` and hence fails to faithfully represent the efficient imperative version of the algorithm.

To implement a properly memoized version of `pow` function, we need to arrange for tabulation of the powers of `m`.

We introduce a type of memo tables. A memo table can record some powers of `m` as entries; we allow only valid entries.

```
MemTbl : {s : String} → Mtrx s → Set
MemTbl {s} m = (n : ℕ) → Maybe (∃[m' : Mtrx s] m' ≡ pow m n)
```

We introduce a function `pow-tbl` that is like `pow`, except that it expects to get some element `tbl` of `MemTbl m` as an argument. Instead of making recursive calls, it looks up matrices in the given memo table `tbl`. If the required matrix is not there, it falls back to `pow`. At this stage we do not worry about where to get a memo table from; we just assume that we have one given.

```
pow-tbl : {s : String} → (m : Mtrx s) → ℕ  →
    MemTbl m → Mtrx s
pow-tbl m zero tbl = if nullable
                      then { (S, i, i, empt _) | i ← [0 ... length s) }
                      else []
pow-tbl m (suc zero)    tbl = m
pow-tbl m (suc (suc n)) tbl = { t | (i, j, _) ← triples n,
                        t ← mt (suc i) * mt (suc j) }
    where
        mt n = maybe (pow m n) fst (tbl n)

        maybe : B → (A → B) → Maybe A → B
        maybe b f nothing  = b
        maybe b f (just r) = f r
```

The next step is to prove that pow and pow-tbl compute propositionally equal results.

```
pow≡pow-tbl : {s : String} → (m : Mtrx s) → (n : ℕ) →
    (tbl : MemTbl m) → pow-tbl m n tbl ≡ pow m n
```

The proof is easy. Recall that the only difference between the functions pow and pow-tbl is that the function pow calls itself while function pow-tbl first tries to retrieve the result from the memo table tbl. Let us analyze the possible cases:

- If tbl n returns nothing, then mt n returns the result of pow m n.
- If tbl n returns just p, then mt n is a pair of a matrix m' and proof that m' equals to pow m n.

Hence the functions pow and pow-tbl are extensionally equal.

Now we have to find a way to actually build memo tables with intermediate results together with the proofs that they coincide with the matrices returned by pow.

We implement a function which iteratively computes the powers pow m n of an argument matrix m, where i ≤ n ≤ i + j for given i and j, remembering all intermediate results.

```
pow-mem : {s : String} → (m : Mtrx s) → ℕ → ℕ
  → MemTbl m
  → Mtrx s
pow-mem m i zero    tbl = pow-tbl m i tbl
pow-mem m i (suc j) tbl = pow-mem m (suc i) j tbl' where
       tbl' p = if p == i
                    then just (pow-tbl m i tbl, pow≡pow-tbl m i tbl)
                    else tbl p)
```

The function pow-mem calls itself with ever more filled memo tables starting from lower powers. Observe how the theorem pow≡pow-tbl is now used to ensure the correctness of each new memo table tbl'.

Finally, the function for CYK parsing can be defined as follows:

```
cyk-parse-mem : (s : String) → Mtrx s
cyk-parse-mem s =
  pow-mem (m-init s) 0 (length s) (λ _ → nothing)
```

6. Comprehending list monad

In the definitions above, for the sake of clarity we used list comprehensions. They give a good intuition about properties of the functions defined. Agda does not support such syntax, but we can explicate the monad structure on lists and use that to faithfully translate the comprehension syntax into Agda. The way of translating comprehensions into monadic code was described in [10]. Basically, we follow the trail, but instead of "join" we use "bind" operator.

First, we define "bind" and "return" operators:

```
_>>=_ : {X Y : Set} → List X → (X → List Y) → List Y
_>>=_ xs f = foldl (λ res el → res ++ f el) [] xs

return : {X : Set} → X → List X
return x = [ x ]
```

Second, we prove the monad laws:

- Left identity:

  ```
  left-id : {X Y : Set} →  (x : X) →  (f : X → List Y)
        → return x >>= f ≡ f x
  ```

- Right identity:

  ```
  right-id : {X : Set} → (xs : List X)
        →  xs >>= return ≡ xs
  ```

- Associativity:

  ```
  assoc : {X Y Z : Set} → (xs : List X) → (f : X → List Y)
    → (g : Y → List Z)
    → (xs >>= f) >>= g ≡ xs >>= (λ x → f x >>= g)
  ```

Finally, we can define the translation from comprehensions to monadic code:

- For the base case we have:

  ```
  { t | x ← xs } = xs >>= (λ x → return t)
  ```

- And for the step case:

  ```
  { t | p ← ps, q } = ps >>= (λ p → { t | q })
  ```

To be independent from the concrete implementation of `_>>=_` we prove following theorems:

– The elements of lists defined by a comprehension can be traced back to where they originate from. This theorem provides a generic way for proving properties about the elements of a comprehension:

  ```
  list-monad-th : {X Y : Set}(y : Y)(xs : List X)(f : X → List Y)
    → y ∈ xs >>= f
    → ∃[x : X]  x ∈ xs × y ∈ f x
  ```

– We also need to use that a comprehension does not miss anything:

  ```
  list-monad-ht : {X Y : Set}(y : Y)(xs : List X)(f : X → List Y)
    → (x : X) → x ∈ xs → e ∈ f x
    → y ∈ (xs >>= f)
  ```

– If f and g are extensionally equal (i.e. propositonally equal on all arguments), then we can change one for the other, a sort of congruence property:

```
>>=cong  : ∀ {X Y : Set} → (f g : X → List Y) → (xs : List X)
  → (∀ x → f x ≡ g x) → xs >>= f ≡ xs >>= g
```

– The next property (a corollary from the associativity law) shows that "bind" applied to a concatenation is a concatenation of "bind" applied to the two lists:

```
>>=split : {X Y : Set} → (xs ys : List X) → (f : X → List Y)
  →  (xs ++ ys) >>= f ≡ (xs >>= f) ++ (ys >>= f)
```

The main proofs in our work reason about list comprehensions only wit the monad laws and properties of the "bind" operator like those just outlined. This makes them modular and concise.

7. Related works

The formal verification of parsers seems to be an interesting and challenging topic for the developers of certified software.

Barthwal and Norrish [2] formalize SLR parsing using the HOL4 proof assistant. They construct an SLR parser for context-free grammars, and prove it to be sound and complete. Formalization of SLR parser is done in over 20 000 lines of code, which is a rather big development. However, SLR parsers handle only unambiguous grammars (SLR grammars are a subset of LR(1) grammars).

Parsing Expression Grammars (PEGs) are a relatively recent formalism for specifying recursive descent parsers. Koprowski and Binsztok [**?**] formalize the semantics of PEGs in Coq. They check context-free grammars for well-formedness. Well-formedness ensures that the grammar is not left-recursive. Under this assumption, they prove that a non-memoizing interpreter is terminating. Soundness and completeness of the interpreter are shown easily, because the PEG interpreter is a functional representation of the semantics of PEGs.

An $LR(1)$ parser is a finite-state automaton, equipped with a stack, which uses a combination of its current state and one lookahead symbol in order to determine which action to perform next. Jourdan, Pottier and Leroy [5] present a validator which, when applied to a context-free grammar G and an automaton A, checks that A and G agree. The validation process is independent of which technique was used to construct A. The validator is implemented and proved to be sound and complete using the Coq proof assistant. However, there is no guarantees of termination of interpreter. Termination is ensured by supplying some large constant (fuel) to the interpreter.

Danielsson and Norell [3] implement a library of parser combinators with termination guarantees in the dependently typed functional programming language Agda [7]. They use dependent types to add type indices to the parser type, and use these indices to ensure that left recursion is not possible.

None of the previously mentioned works can treat all context-free grammars. Ridge [8] demonstrates how to construct sound and complete parser implementations directly from grammar specifications, for all context-free grammars, based on combinator parsing. He constructs a generic parser generator and shows that generated parsers are sound and complete. The formal proofs are mechanized using the HOL4 theorem prover. The time complexity of the memoized version of implemented parser is $O(n^5)$.

8. Conclusions and Future Work

We have shown that with careful design, programming with dependent types is a powerful tool for implementing non-trivial algorithms together with correctness proofs.

Since the CYK algorithm handles only grammars in normal form, we plan to extend our work to grammars in general form. One possible way of doing it is to implement a verified normalization algorithm of context-free grammars, i.e., conversion context-free grammars from general form to normal form. In the constructive setting, proofs of soundness and completeness of this procedure will be functions between parse trees in the general and normal-form grammars. So, one can use the CYK implementation of this paper to produce parse trees for grammars in normal form and then convert them to trees for grammars in general form by using the soundness proof of the normalization algorithm.

Acknowledgements. This work was supported by the Estonian Centre of Excellence in Computer Science, EXCS, a project funded by the European Regional Development Fund, ERDF, and by the Estonian Research Council grant no. 9475.

References

[1] The Agda Team. The Agda Wiki. `http://wiki.portal.chalmers.se/agda/`, 2013.

[2] A. Barthwal and M. Norrish. Verified, executable parsing. In *Proceedings of the 18th European Symposium on Programming Languages and Systems, ESOP '09*, Lecture Notes in Computer Science, pages 160–174, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] N. A. Danielsson. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 285–296, New York, NY, USA, 2010. ACM.

[4] D. Firsov. Certified parsing of regular languages. Master's thesis, Tallinn University of Technology, 2012.

[5] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In *Programming Languages and Systems – 21st European Symposium on Programming, ESOP 2012*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416. Springer, 2012.

[6] A. Koprowski and H. Binsztok. TRX: A formally verified parser interpreter. In *Proceedings of the 19th European Symposium on Programming (ESOP '10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 345–365. Springer, 2010.

[7] U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, and S. D. Swierstra, editors, *AFP 2008*, volume 5832 of *LNCS*, pages 230–266. Springer, 2009.

[8] T. Ridge. Simple, functional, sound and complete parsing for all context-free grammars. In J.-P. Jouannaud and Z. Shao, editors, *CPP*, volume 7086 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 2011.

[9] L. G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–314, Apr. 1975.

[10] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM.

[11] D. Younger. Recognition and parsing of context-free languages in time $O(n^3)$. *Information and Control*, 10(2):189–208, 1967.