DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Formal Verification of an Earley Parser

Martin Rau

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Formal Verification of an Earley Parser

Formale Verifikation eines Earley Parsers

Author:	Martin Rau
Supervisor:	Tobias Nipkow
Advisor:	Tobias Nipkow
Submission Date:	15.06.2023

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.06.2023 Martin Rau

Acknowledgments

I owe an enormous debt of gratitude to my family which always suported me throughout my studies. Thank you. I also would like to thank Prof. Tobias Nipkow for introducing me to the world of formal verification through Isabelle and for supervising both my Bachelor's and my Master's thesis. It was a pleasure to learn from and to work with you.

Abstract

TODO: Abstract

Contents

1 Snippets

1.1 Earley

Context-free grammars have been used extensively for describing the syntax of programming languages and natural languages. Parsing algorithms for context-free grammars consequently play a large role in the implementation of compilers and interpreters for programming languages and of programs which understand or translate natural languages. Numerous parsing algorithms have been developed. Some are general, in the sense that they can handle all context-free grammars, while others can handle only subclasses of grammars. The latter, restricted algorithms tend to be much more efficient The algorithm described here seems to be the most efficient of the general algorithms, and also it can handle a larger class of grammars in linear time than most of the restricted algorithms.

A language is a set of strings over a finite set of symbols. We call these terminal symbols and represent them by lowercase letters: a, b, c. We use a context-free grammar as a formal device for specifying which strings are in the set. This grammar uses another set of symbols, the nonterminals, which we can think of as syntactic classes. We use capitals for nonterminals: A, B, C. String of either terminals or non-terminals are represented by greek letters: alpha, beta, gamma. The empty string is epsilon. There is a finite set of productions or rewriting rules of the form A -> alpha. The nonterminal which stands for sentence is called the root R of the grammar. The productions with a particular nonterminal A on their left sides are called the alternatives of A. We write alpha => beta if exists gamma, delta, ny, A such taht a = gamma A delta and beta = gamma ny delta and A -> ny is a production. We write alpha =>* beta if exists alpha0, alpha1, ... alpham (m > =0) such that alpha = alpha0 => alpha1 => ... => alpham = beta The sequence alphai is called a derivation of beta from alpha. A sentential form is a string alpha such the the root R =>* alpha. A sentence is a sentential form consisting entirely of terminal symbols. The language defined by a grammar L(G) is the set of its sentences. We may represent any sentential form in at least one way as a derivation tree or parse tree reflecting the steps made in deriving it. The degree of ambiguity of a sentence is the number of its distinct derivation trees. A sentence is unambiguous if it has degree 1 of ambiguity. A grammar is unambiguous if each of its sentences is unambiguous. A grammar is reduced if every nonterminal appears in some derivation of some sentence. A recognizer is an algorithm which takes a input a string and either accepts or rejects it depending on whether or not the string is a sentence of the grammer. A parser is a recogizer which also outputs the set of all legal derivation trees for the string.

1.2 Scott

The Computer Science community has been able to automatically generate parsers for a very wide class of context free languages. However, many parsers are still written manually, either using tool support or even completely by hand. This is partly because in some application areas such as natural language processing and bioinformatics we don not have the luxury of designing the language so that it is amendable to know parsing techniques, but also it is clear that left to themselves computer language designers do not naturally write LR(1) grammars. A grammar not only defines the syntax of a language, it is also the starting point for the definition of the semantics, and the grammar which facilitates semantics definition is not usually the one which is LR(1). Given this difficulty in constructing natural LR(1) grammars that support desired semantics, the general parsing techniques, such as the CYK Younger [**Younger:1967**], Earley [**Earley:1970**] and GLR Tomita [**Tomita:1985**] algorithms, developed for natural language processing are also of interest to the wider computer science community. When using grammars as the starting point for semantics definition, we distinguish between recognizers which simply determine whether or not a given string is in the language defined by a given grammar, and parserwhich also return some form of derivation of the string, if one exists. In their basic form the CYK and Earley algorithms are recognizers while GLR-style algorithms are designed with derivation tree construction, and hence parsing, in mind.

There is no known liner time parsing or recognition algorithm that can be used with all context free grammars. In their recognizer forms the CYK algorithm is worst case cubic on grammars in Chomsky normal form and Earley's algorithm is worst case cubic on general context free grammers and worst case n2 on non-ambibuous grammars. General recognizers must, by definition, be applicable to ambiguous grammars. Tomita's GLR algorithm is of unbounded polynomial order in the worst case. Expanding general recognizers to parser raises several problems, not least because there can be exponentially many or even infinitely many derivations for a given input string. A cubic recognizer which was modified to simply return all derivations could become an unbounded parser. Of course, it can be argued that ambiguous grammars reflect ambiguous semantics and thus should not be used in practice. This would be far too extreme a position to take. For example, it is well known that the if-else statement in hthe AnSI-standard grammar for C is ambiguous, but a longest match resolution results in a linear time parser that attach the else to the most recent if, as specified by the ANSI-C semantics. The ambiguous ANSI-C grammar is certainly practical for parser implementation. However, in general ambiguity is not so easily handled, and it is well known that grammar ambiguity is in fact undecidable Hopcroft *et al* [**Hopcroft:2006**], thus we cannot expect a parser generator simply to check for ambiguity inthe grammar and report the problem back to the user. Another possiblity is to avoid the issue by just returning one derivation. However, if only one derivation is returned then this creates problems for a user who wants all derivations and, even in the case where only one derivation is required, there is the issue of ensuring that it is the required derivationthat is returned. A truely general parser will reutrn all possible derivations in some form. Perhaps the most well known representation is the shared packed parse foreset SPPF described and used by Tomita [**Tomita:1985**]. Tomita's description of the representation does ont allow for the infinitely many derivations which arise from grammars which contain cycles, the source adapt the SPPF representation to allow these. Johnson [**Johnson:1991**] has shown that Tomita-style SPPFs are worst case unbounded polynomial size. Thus using such structures will alo turn any cubic recognition technique into a worst case unbounded polynomial parsing technique. Leaving aside the potential increase in complexity when turning a recogniser into a parser, it is clear that this proccess is often difficult to carry out correctly. Earley gave an algorithm for constructing derivations of a string accepted by his recognizer, but this was subsequently shown by Tomita [**Tomita:1985**] to return spurious derivations in certain cases. Tomita's original version of his algorithm failed to terminate on grammars with hidden left recursio and, as remarked above , had no mechanism for contructing complete SPPFs for grammers with cycles.

A shared packed parse forest SPPF is a representation designed to reduce the space required to represent multiple derivation trees for an ambiguous sentence. In an SPPF, nodes which have the same tree below them are shared and nodes which correspond to different derivations of the same substring from the same non-terminal are combined by creating a packed node for each family of children. Nodes can be packed only if their yields correspond to the same portion of the input string. Thus, to make it easier to determine whether two alternates can be packed under a given node, SPPF nodes are labelled with a triple (x,i,j) where $a_{j+1} \ldots a_i$ is a substring matched by x. To obtain a cubic algorithm we use binarised SPPFs which contain intermediate additional nodes but which are of worst case cubic size. (EXAMPlE SPPF running example???)

We can turn earley's algorithm into a correct parser by adding pointers between items rather than instances of non-terminals, and labelling th epointers in a way which allows a binariesd SPPF to be constructed by walking the resulting structure. However, inorder to construct a binarised SPPF we also have to introduce additional nodes for grammar rules of length greater than two, complicating the final algorithm.

1.3 Aycock

Earley's parsing algorithm is a general algorithm, capable of parsing according to any context-free grammar. General parsing algorithms like Earley parsing allow unfettered expression of ambiguous grammar contructs which come up often in practice (REFERENCE).

In terms of implementation, the Earley sets are built in increasing order as the input is read. Also, each set is typically represented as a list of items. This list representation of a set is particularly convenient, because the list of items acts as a work queue when building the sets: items are examined in order, applying the transformations as necessary: items added to the set are appended onto the end of the list.

At any given point i in the parse, we have two partially constructed sets. Scanner may add items to S_{i+1} and S_i may have items added to it by Predictor and Completer. It is this latter possibility, adding items to S_i while representing sets as lists, which causes grief with epsilon-rules. When Completer processes an item A -> dot, j which corresponds to the epsilon-rule A -> epsiolon, it must look through S_j for items with the dot before an A. Unfortunately, for epsilon-rule items, j is always equal to i. Completer is thus looking through the partially constructed set S_i. Since implementations process items in S_i in order, if an item B -> alpha dot A beta, k is added to S_i after Completer has processed A -> dot, j, Completer will never add B -> αA dot β, k to S_i. In turn, items resulting directly and indirectly from B -> αA dot β, k will be omitted too. This effectively prunes protential derivation paths which might cause correct input to be rejected. (EXAMPLE) Aho *et al* **[Aho:1972]** propose the stay clam and keep running the Predictor and Completer in turn until neither has anything more to add. Earley himself suggest to have the Completer note that the dot needed to be moved over A, then looking for this whenever future items were added to S_i. For efficiency's sake the collection of on-terminals to watch for should be stored in a data structure which allows fast access. Neither approach is very satisfactory. A third solution **[Aycoack:2002]** is a simple modification of the Predictor based on the idea of nullability. A non-terminal A is said to be nullable if A derives star epsilon. Terminal symbols of course can never be nullable. The nullability of non-terminals in a grammar may be precomputed using well-known techniques **[Appel:2003]** **[Fischer:2009]** Using this notion the Predictor can be stated as follows: if A -> αdot B β, j is in S_i, add B -> dot γ, i to S_i for all rules B -> γ. If B is nullable, also add A -> αB dot β, j to S_i. Explanation why I decided against it. Involves every grammar can be rewritten to not contain epsilon productions. In other words we eagerly move the dot over a nonterminal if that non-terminal can derive epsilon and effectivley disappear. The source implements this precomputation by constructing a variant of a LR(0) deterministic finite automata (DFA). But for an earley parser we must keep track of which parent pointers and LR(0) items belong together which leads to complex and inelegant implementations [**McLean:1996**]. The source resolves this problem by constructing split epsilon DFAs, but still need to adjust the classical earley algorithm by adding not only predecessor links but also causal links, and to construct the split epsilon DFAs not the original grammar but a slightly adjusted equivalent grammar is used that encodes explicitly information that is crucial to reconstructing derivations, called a grammar in nihilist normal form (NNF) which might increase the size of the grammar whereas the authors note empirical results that the increase is quite modest (a factor of 2 at most).

Example: S -> AAAA, A -> a, A -> E, E -> epsilon, input a S_0 S -> dot AAAA,0, A -> dot a, 0, A -> dot E, 0, E -> dot, 0, A -> E dot, 0, S -> A dot AAA, 0 S_1 A -> a dot, 0, S -> A dot AAA, 0, S -> AA dot AA, 0, A -> dot a, 1, A -> dot E, 1, E -> dot, 1, A -> E dot, 1, S -> AAA dot A, 0

2 Introduction

2.1 Motivation

some introduction about parsing, formal development of correct algorithms: an example based on earley's recogniser, the benefits of formal methods, LocalLexing and the Bachelor thesis.

2.2 Structure

2.3 Related Work

Tomita [**Tomita:1987**] presents an generalized LR parsing algorithm for augmented context-free grammars that can handle arbitrary context-free grammars.

Izmaylova *et al* [**Izmaylova:2016**] develop a general parser combinator library based on memoized Continuation-Passing Style (CPS) recognizers that supports all context-free grammars and constructs a Shared Packed Parse Forest (SPPF) in worst case cubic time and space.

Obua *et al* [**Obua:2017**] introduce local lexing, a novel parsing concept which interleaves lexing and parsing whilst allowing lexing to be dependent on the parsing process. They base their development on Earley's algorithm and have verified the correctness with respect to its local lexing semantics in the theorem prover Isabelle/HOL. The background theory of this Master's thesis is based upon the local lexing entry [**LocalLexing-AFP**] in the Archive of Formal Proofs.

Lasser *et al* [**Lasser:2019**] verify an LL(1) parser generator using the Coq proof assistant.

Barthwal *et al* [**Barthwal:2009**] formalize background theory about context-free languages and grammars, and subsequently verify an SLR automaton and parser produced by a parser generator.

Blaudeau *et al* [**Blaudeau:2020**] formalize the metatheory on Parsing expression grammars (PEGs) and build a verified parser interpreter based on higher-order parsing combinators for expression grammars using the PVS specification language and verification system. Koprowski *et al* [**Koprowski:2011**] present TRX: a parser interpreter formally developed in Coq which also parses expression grammars.

Jourdan *et al* [**Jourdan:2012**] present a validator which checks if a context-free grammar and an LR(1) parser agree, producing correctness guarantees required by verified compilers.

Lasser *et al* [**Lasser:2021**] present the verified parser CoStar based on the ALL(*) algorithm. They proof soundness and completeness for all non-left-recursive grammars using the Coq proof assistant.

2.4 Contributions

3 Earley's Algorithm

We present a slightly simplified version of Earley's original recognizer algorithm **[Earley:1970]**, omitting Earley's proposed look-ahead since it's primary purpose is to increase the efficiency of the resulting recognizer. Throughout this thesis we are working with a running example. The considered grammar is a tiny excerpt of a toy arithmetic expression grammar: $\mathcal{G} ::= S \rightarrow x \mid S \rightarrow S + S$ and the input is $\omega = x + x + x$.

Intuitively, Earley's recognizer works in principle like a top-down parser carrying along all possible parses simultaneously in an efficient manner. In detail, the algorithm works as follows: it scans the input $\omega = a_0, \ldots, a_n$, constructing $n + 1$ Earley bins B_i which are sets of Earley items. An inital bin B_0 and one bin B_{i+1} for each symbol a_i of the input. In general, an Earley item $A \rightarrow \alpha \bullet \beta, i, j$ consists of four parts: a production rule of the grammar which we are currently scanning, a bullet signalling how much of the production's right-hand side we have recognized so far, an origin i describing the position of ω where we started scanning, and an end j indicating the position of ω we are currently considering next for the remaining right-hand side of the production rule. Note that there will be only one set of earley items or only one bin B and we say an item is conceptually part of bin B_j if it's end is the index j. Table 3.1 lists the items for our example grammar. Bin B_4 contains for example the item $S \rightarrow S + \bullet S, 2, 4$. Or, we are scanning the rule $S \rightarrow S + S$, have recognized the substring from 2 to 4 (the first index being inclusive the second one exclusive) of ω by $\alpha = S+$, and are trying to scan $\beta = S$ from position 4 in ω.

The algorithm initializes B by applying the *Init* operation. It then proceeds to execute the *Scan*, *Predict* and *Complete* operations listed in figure 3.1 until there are no more new items being generated and added to B. Next we describe these four operations in detail:

1. The *Init* operation adds items $S \rightarrow \bullet\alpha, 0, 0$ for each production rule containing the start symbol S on its left-hand side.

   For our example *Init* adds the items $S \rightarrow \bullet x, 0, 0$ and $S \rightarrow \bullet S + S, 0, 0$.

2. The *Scan* operation applies if there is a terminal to the right side of the bullet, or items of the form $A \rightarrow \alpha \bullet a\beta, i, j$, and the j-th symbol of ω matches the terminal symbol following the bullet. We add one new item $A \rightarrow \alpha a \bullet \beta, i, j + 1$ to B

moving the bullet over the scanned terminal symbol.

Considering our example, bin B_3 contains the item $S \rightarrow S \bullet +S, 2, 3$, the third element of ω is the terminal $+$, so we add the item $S \rightarrow S + \bullet S, 2, 4$ to the conceptual bin B_4.

3. The *Predict* operation is applicable to an item when there is a non-terminal to the right of the bullet or items of the form $A \rightarrow \alpha \bullet B\beta, i, j$. It adds one new item $B \rightarrow \bullet\gamma, j, j$ to the bin for each alternate $B \rightarrow \gamma$ of that non-terminal.

   E.g. for the item $S \rightarrow S + \bullet S, 0, 2$ in B_2 we add the two items $S \rightarrow \bullet x, 2, 2$ and $S \rightarrow \bullet S + S, 2, 2$ corresponding to the two alternates of S. The bullet is set to the beginning of the right-hand side of the production rule, the origin and end are set to $j = 2$ to indicate that we are starting to scan in the current bin and have not scanned anything so far.

4. The *Complete* operation applies if we process an item with the bullet at the end of the right side of its production rule. For an item $B \rightarrow \gamma\bullet, j, k$ we have successfully scanned the substring ω[j..k) and are now going back to the origin bin B_j where we predicted this non-terminal. There we look for any item of the form $A \rightarrow \alpha \bullet B\beta, i, j$ containing a bullet in front of the non-terminal we completed, or the reason we predicted it on the first place. Since we scanned the predicted non-terminal successfully, we are allowed to move over the bullet, resulting in one new item $A \rightarrow \alpha B \bullet \beta, i, k$. Note in particular the origin and end indices.

   Looking back at our example, we can add the item $S \rightarrow S + S\bullet, 0, 5$ for two different reasons corresponding conceptually to the two different ways we can derive ω. When processing $S \rightarrow x\bullet, 4, 5$ we find $S \rightarrow S + \bullet S, 0, 4$ in the origin bin B_4 which conceptually corresponds to recognizing $(x + x) + x$. We ,add 'the same item again while applying the *Complete* operation to $S \rightarrow S + S\bullet, 2, 5$ and $S \rightarrow S + \bullet S, 0, 2$ which corresponds to recognizing the input as $x + (x + x)$.

If the algorithm encounters an item of the form $S \rightarrow \alpha, 0, |\omega| + 1$, it returns *true*, otherwise it returns *false*. For the tiny arithmetic expression grammar we generate the item $S \rightarrow S + S\bullet, 0, 5$ and return the correct answer *true*, since there exist derivations for $\omega = x + x + x$, e.g. $S \Rightarrow S + S \Rightarrow x + S \Rightarrow x + S + S \overset{*}{\Rightarrow} x + x + x$ or $S \Rightarrow S + S \Rightarrow S + x \Rightarrow S + S + x \overset{*}{\Rightarrow} x + x + x$.

To proof the correctness of Earley's recognizer algorithm we need to show the following theorem:

$$S \rightarrow \alpha\bullet, 0, |\omega| + 1 \in B \text{ iff } S \overset{*}{\Rightarrow} \omega$$

It follows from the following three lemmas:

1. Termination: there only exist a finite number of Earley items
2. Soundness: for every generated item there exists an according derivation:
   $A \rightarrow \alpha \bullet \beta, i, j \in B$ implies $A \overset{*}{\Rightarrow} \omega[i..j)$
3. Completeness: for every derivation we generate an according item:
   $A \overset{*}{\Rightarrow} \omega[i..j)$ implies $A \rightarrow \alpha \bullet \beta, i, j \in B$

$$\textsc{Init}\quad \frac{}{\mathrm{S} \rightarrow \bullet\alpha,\ 0,\ 0} \qquad \textsc{Scan}\quad \frac{\mathrm{A} \rightarrow \alpha\bullet\mathrm{a}\ \beta,\ \mathrm{i},\ \mathrm{j} \qquad \omega[\mathrm{j}] = \mathrm{a}}{\mathrm{A} \rightarrow \alpha\mathrm{a}\ \bullet\beta,\ \mathrm{i},\ \mathrm{j+1}} \qquad \textsc{Predict}\quad \frac{\mathrm{A} \rightarrow \alpha\bullet\mathrm{B}\ \beta,\ \mathrm{i},\ \mathrm{j} \qquad \mathrm{B} \rightarrow \gamma \in \mathcal{G}}{\mathrm{B} \rightarrow \bullet\gamma,\ \mathrm{j},\ \mathrm{j}}$$

$$\textsc{Complete}\quad \frac{\mathrm{A} \rightarrow \alpha\bullet\mathrm{B}\ \beta,\ \mathrm{i},\ \mathrm{j} \qquad \mathrm{B} \rightarrow \gamma\bullet,\ \mathrm{j},\ \mathrm{k}}{\mathrm{A} \rightarrow \alpha\mathrm{B}\ \bullet\beta,\ \mathrm{i},\ \mathrm{k}}$$

Figure 3.1: Earley inference rules

Table 3.1: Earley items for the grammar $\mathcal{G}$: $S \rightarrow x$, $S \rightarrow S + S$

B_0	B_1	B_2
$S \rightarrow \bullet x, 0, 0$	$S \rightarrow x\bullet, 0, 1$	$S \rightarrow S + \bullet S, 0, 2$
$S \rightarrow \bullet S + S, 0, 0$	$S \rightarrow S \bullet + S, 0, 1$	$S \rightarrow \bullet x, 2, 2$
		$S \rightarrow \bullet S + S, 2, 2$
B_3	B_4	B_5
$S \rightarrow x\bullet, 2, 3$	$S \rightarrow S + \bullet S, 2, 4$	$S \rightarrow x\bullet, 4, 5$
$S \rightarrow S + S\bullet, 0, 3$	$S \rightarrow S + \bullet S, 0, 4$	$S \rightarrow S + S\bullet, 2, 5$
$S \rightarrow S \bullet + S, 2, 3$	$S \rightarrow \bullet x, 4, 4$	$S \rightarrow S + S\bullet, 0, 5$
$S \rightarrow S \bullet + S, 0, 3$	$S \rightarrow \bullet S + S, 4, 4$	$S \rightarrow S \bullet + S, 4, 5$
		$S \rightarrow S \bullet + S, 2, 5$
		$S \rightarrow S \bullet + S, 0, 5$

4 Earley's Algorithm Formalization

4.1 Draft

- Introduce background theory about CFG and Isabelle syntax
- explain the auxiliary definitions until earley_recognized, the small ones incorporated into text, the big ones as definitions
- explain Init, Scan, Predict, Complete REFERENCE and relate them back to the previous chapter
- explain fixpoint iteration REFERENCE and iteration over all bins
- illustrate the running example in this algorithm
- explain wellformedness proof
- explain soundness definitions and proof
- explain monotonicity and absorption proofs
- explain completeness proof, this one in great detail!
- explain finiteness proof

4.2 Background Theory

type-synonym *'a rule* = *'a* × *'a list*
type-synonym *'a rules* = *'a rule list*
type-synonym *'a sentential* = *'a list*

datatype *'a cfg* =
  CFG
    ($\mathfrak{N}$: *'a list*)
    ($\mathfrak{T}$: *'a list*)
    ($\mathfrak{R}$: *'a rules*)
    ($\mathfrak{S}$: *'a*)

definition *derives1* :: *'a cfg* ⇒ *'a sentential* ⇒ *'a sentential* ⇒ *bool* **where**
  derives1 cfg u v ≡
    $\exists\ x\ y\ N\ \alpha.$
      $u = x$ @ $[N]$ @ y
    $\wedge\ v = x$ @ α @ y
    $\wedge\ (N, \alpha) \in$ *set* ($\mathfrak{R}$ *cfg*)

definition *derivations1* :: *'a cfg* ⇒ (*'a sentential* × *'a sentential*) *set* **where**
  derivations1 cfg = { (*u*,*v*) | *u v*. *derives1 cfg u v* }

definition *derivations* :: *'a cfg* ⇒ (*'a sentential* × *'a sentential*) *set* **where**
  derivations cfg = (*derivations1 cfg*)^∗

definition *derives* :: *'a cfg* ⇒ *'a sentential* ⇒ *'a sentential* ⇒ *bool* **where**
  derives cfg u v ≡ (*u*, *v*) ∈ *derivations cfg*

fun *slice* :: *nat* ⇒ *nat* ⇒ *'a list* ⇒ *'a list* **where**
  slice - - [] = []
| *slice* - *0* (*x*#*xs*) = []
| *slice 0* (*Suc b*) (*x*#*xs*) = *x* # *slice 0 b xs*
| *slice* (*Suc a*) (*Suc b*) (*x*#*xs*) = *slice a b xs*

lemma *slice-append*:
  assumes $a \leq b$ $b \leq c$
  shows *slice a b xs* @ *slice b c xs* = *slice a c xs*
definition *disjunct-symbols* :: *'a cfg* ⇒ *bool* **where**
  disjunct-symbols cfg ⟷ *set* ($\mathfrak{N}$ *cfg*) ∩ *set* ($\mathfrak{T}$ *cfg*) = {}

definition *wf-startsymbol* :: *'a cfg* ⇒ *bool* **where**
  wf-startsymbol cfg ⟷ $\mathfrak{S}$ *cfg* ∈ *set* ($\mathfrak{N}$ *cfg*)

definition *wf-rules* :: *'a cfg* ⇒ *bool* **where**
$\textit{wf-rules cfg} \equiv \forall (N, \alpha) \in \textit{set } (\mathfrak{R}\ \textit{cfg}).\ N \in \textit{set } (\mathfrak{N}\ \textit{cfg}) \wedge (\forall s \in \textit{set } \alpha.\ s \in \textit{set } (\mathfrak{N}\ \textit{cfg}) \cup \textit{set } (\mathfrak{T}\ \textit{cfg}))$

definition *distinct-rules* :: *'a cfg* ⇒ *bool* **where**
$\textit{distinct-rules cfg} = \textit{distinct } (\mathfrak{R}\ \textit{cfg})$

definition *wf-cfg* :: *'a cfg* ⇒ *bool* **where**
$\textit{wf-cfg cfg} \longleftrightarrow \textit{disjunct-symbols cfg} \wedge \textit{wf-startsymbol cfg} \wedge \textit{wf-rules cfg} \wedge \textit{distinct-rules cfg}$

definition *is-terminal* :: *'a cfg* ⇒ *'a* ⇒ *bool* **where**
$\textit{is-terminal cfg s} \equiv s \in \textit{set } (\mathfrak{T}\ \textit{cfg})$

definition *is-nonterminal* :: *'a cfg* ⇒ *'a* ⇒ *bool* **where**
$\textit{is-nonterminal cfg s} \equiv s \in \textit{set } (\mathfrak{N}\ \textit{cfg})$

definition *is-symbol* :: *'a cfg* ⇒ *'a* ⇒ *bool* **where**
$\textit{is-symbol cfg s} \longleftrightarrow \textit{is-terminal cfg s} \vee \textit{is-nonterminal cfg s}$

definition *wf-sentential* :: *'a cfg* ⇒ *'a sentential* ⇒ *bool* **where**
$\textit{wf-sentential cfg s} \equiv \forall x \in \textit{set s}.\ \textit{is-symbol cfg x}$

definition *is-sentence* :: *'a cfg* ⇒ *'a sentential* ⇒ *bool* **where**
$\textit{is-sentence cfg s} \equiv \forall x \in \textit{set s}.\ \textit{is-terminal cfg x}$

4.3 Definitions

definition *rule-head* :: *'a rule* ⇒ *'a* **where**
rule-head = *fst*

definition *rule-body* :: *'a rule* ⇒ *'a list* **where**
rule-body = *snd*

datatype *'a item* =
Item
(*item-rule*: *'a rule*)
(*item-dot* : *nat*)
(*item-origin* : *nat*)
(*item-end* : *nat*)

type-synonym *'a items* = *'a item set*

definition *item-rule-head* :: *'a item* ⇒ *'a* **where**
item-rule-head x = *rule-head* (*item-rule x*)

definition *item-rule-body* :: *'a item* $\Rightarrow$ *'a sentential* **where**
item-rule-body x = *rule-body* (*item-rule x*)

definition *item-α* :: *'a item* $\Rightarrow$ *'a sentential* **where**
item-α x = *take* (*item-dot x*) (*item-rule-body x*)

definition *item-β* :: *'a item* $\Rightarrow$ *'a sentential* **where**
item-β x = *drop* (*item-dot x*) (*item-rule-body x*)

definition *init-item* :: *'a rule* $\Rightarrow$ *nat* $\Rightarrow$ *'a item* **where**
init-item r k = *Item r 0 k k*

definition *is-complete* :: *'a item* $\Rightarrow$ *bool* **where**
is-complete x $\equiv$ *item-dot x* $\geq$ *length* (*item-rule-body x*)

definition *next-symbol* :: *'a item* $\Rightarrow$ *'a option* **where**
next-symbol x $\equiv$ *if is-complete x then None else Some* ((*item-rule-body x*) ! (*item-dot x*))

definition *inc-item* :: *'a item* $\Rightarrow$ *nat* $\Rightarrow$ *'a item* **where**
inc-item x k = *Item* (*item-rule x*) (*item-dot x* + *1*) (*item-origin x*) *k*

definition *bin* :: *'a items* $\Rightarrow$ *nat* $\Rightarrow$ *'a items* **where**
bin I k = { *x* . *x* $\in$ *I* $\wedge$ *item-end x* = *k* }

definition *wf-item* :: *'a cfg* $\Rightarrow$ *'a sentential* => *'a item* $\Rightarrow$ *bool* **where**
wf-item cfg inp x $\equiv$
item-rule x $\in$ *set* ($\mathfrak{R}$ *cfg*) $\wedge$
item-dot x $\leq$ *length* (*item-rule-body x*) $\wedge$
item-origin x $\leq$ *item-end x* $\wedge$
item-end x $\leq$ *length inp*

definition *wf-items* :: *'a cfg* $\Rightarrow$ *'a sentential* $\Rightarrow$ *'a items* $\Rightarrow$ *bool* **where**
wf-items cfg inp I $\equiv$ $\forall$ *x* $\in$ *I*. *wf-item cfg inp x*

definition *is-finished* :: *'a cfg* $\Rightarrow$ *'a sentential* $\Rightarrow$ *'a item* $\Rightarrow$ *bool* **where**
is-finished cfg inp x $\equiv$
item-rule-head x = $\mathfrak{S}$ *cfg* $\wedge$
item-origin x = *0* $\wedge$
item-end x = *length inp* $\wedge$
is-complete x

definition *earley-recognized* :: *'a items* $\Rightarrow$ *'a cfg* $\Rightarrow$ *'a sentential* $\Rightarrow$ *bool* **where**
earley-recognized I cfg inp $\equiv$ $\exists$ *x* $\in$ *I*. *is-finished cfg inp x*

```
definition Init :: 'a cfg ⇒ 'a items where
  Init cfg = { init-item r 0 | r. r ∈ set (ℜ cfg) ∧ fst r = (𝔖 cfg) }

definition Scan :: nat ⇒ 'a sentential ⇒ 'a items ⇒ 'a items where
  Scan k inp I =
    { inc-item x (k+1) | x a.
        x ∈ bin I k ∧
        inp!k = a ∧
        k < length inp ∧
        next-symbol x = Some a }

definition Predict :: nat ⇒ 'a cfg ⇒ 'a items ⇒ 'a items where
  Predict k cfg I =
    { init-item r k | r x.
        r ∈ set (ℜ cfg) ∧
        x ∈ bin I k ∧
        next-symbol x = Some (rule-head r) }

definition Complete :: nat ⇒ 'a items ⇒ 'a items where
  Complete k I =
    { inc-item x k | x y.
        x ∈ bin I (item-origin y) ∧
        y ∈ bin I k ∧
        is-complete y ∧
        next-symbol x = Some (item-rule-head y) }

fun funpower :: ('a ⇒ 'a) ⇒ nat ⇒ ('a ⇒ 'a) where
  funpower f 0 x = x
| funpower f (Suc n) x = f (funpower f n x)

definition natUnion :: (nat ⇒ 'a set) ⇒ 'a set where
  natUnion f = ⋃ { f n | n. True }

definition limit  :: ('a set ⇒ 'a set) ⇒ 'a set ⇒ 'a set where
  limit f x = natUnion (λ n. funpower f n x)

definition E-step :: nat ⇒ 'a cfg ⇒ 'a sentential ⇒ 'a items ⇒ 'a items where
  E-step k cfg inp I = I ∪ Scan k inp I ∪ Complete k I ∪ Predict k cfg I

definition E :: nat ⇒ 'a cfg ⇒ 'a sentential ⇒ 'a items ⇒ 'a items where
  E k cfg inp I = limit (E-step k cfg inp) I

fun ℰ :: nat ⇒ 'a cfg ⇒ 'a sentential ⇒ 'a items where
```

$\mathcal{E}$ *0 cfg inp* = *E 0 cfg inp* (*Init cfg*)
| $\mathcal{E}$ (*Suc n*) *cfg inp* = *E* (*Suc n*) *cfg inp* ($\mathcal{E}$ *n cfg inp*)

definition *earley* :: *'a cfg* ⇒ *'a sentential* ⇒ *'a items* **where**
earley cfg inp = $\mathcal{E}$ (*length inp*) *cfg inp*

4.4 Wellformedness

lemma *wf-Init*:
assumes *x* ∈ *Init cfg*
shows *wf-item cfg inp x*

by definition

lemma *wf-Scan-Predict-Complete*:
assumes *wf-items cfg inp I*
shows *wf-items cfg inp* (*Scan k inp I* ∪ *Predict k cfg I* ∪ *Complete k I*)

by definition

lemma *wf-E-step*:
assumes *wf-items cfg inp I*
shows *wf-items cfg inp* (*E-step k cfg inp I*)

wf-Scan-Predict-Complete by definition

lemma *wf-funpower*:
assumes *wf-items cfg inp I*
shows *wf-items cfg inp* (*funpower* (*E-step k cfg inp*) *n I*)

wf-E-step, by induction on n

lemma *wf-E*:
assumes *wf-items cfg inp I*
shows *wf-items cfg inp* (*E k cfg inp I*)

wf-funpower by definition

lemma *wf-E0*:
shows *wf-items cfg inp* (*E 0 cfg inp* (*Init cfg*))

wf-Init wf-E by definition

lemma *wf-*$\mathcal{E}$:
shows *wf-items cfg inp* ($\mathcal{E}$ *n cfg inp*)

wf-E0 wf-E by induction on n

lemma *wf-earley*:
shows *wf-items cfg inp* (*earley cfg inp*)

wf-$\mathcal{E}$ by definition

4.5 Soundness

definition *sound-item* :: *'a cfg* ⇒ *'a sentential* ⇒ *'a item* ⇒ *bool* **where**
sound-item cfg inp x = *derives cfg* [*item-rule-head x*] (*slice* (*item-origin x*) (*item-end x*) *inp* @ *item-β x*)

definition *sound-items* :: *'a cfg* ⇒ *'a sentential* ⇒ *'a items* ⇒ *bool* **where**
sound-items cfg inp I ≡ ∀ *x* ∈ *I*. *sound-item cfg inp x*

lemma *sound-Init*:
shows *sound-items cfg inp* (*Init cfg*)
lemma *sound-item-inc-item*:
assumes *wf-item cfg inp x sound-item cfg inp x*
assumes *next-symbol x* = *Some a k* < *length inp inp*!*k* = *a item-end x* = *k*
shows *sound-item cfg inp* (*inc-item x* (*k*+*1*))
lemma *sound-Scan*:
assumes *wf-items cfg inp I sound-items cfg inp I*
shows *sound-items cfg inp* (*Scan k inp I*)
lemma *sound-Predict*:
assumes *sound-items cfg inp I*
shows *sound-items cfg inp* (*Predict k cfg I*)
lemma *sound-Complete*:
assumes *wf-items cfg inp I sound-items cfg inp I*
shows *sound-items cfg inp* (*Complete k I*)
lemma *sound-E-step*:
assumes *wf-items cfg inp I sound-items cfg inp I*
shows *sound-items cfg inp* (*E-step k cfg inp I*)
lemma *sound-funpower*:
assumes *wf-items cfg inp I sound-items cfg inp I*
shows *sound-items cfg inp* (*funpower* (*E-step k cfg inp*) *n I*)
lemma *sound-E*:
assumes *wf-items cfg inp I sound-items cfg inp I*
shows *sound-items cfg inp* (*E k cfg inp I*)
lemma *sound-E0*:
shows *sound-items cfg inp* (*E 0 cfg inp* (*Init cfg*))
lemma *sound-$\mathcal{E}$*:
shows *sound-items cfg inp* ($\mathcal{E}$ *k cfg inp*)
lemma *sound-earley*:
shows *sound-items cfg inp* (*earley cfg inp*)
theorem *soundness*:
assumes *earley-recognized* (*earley cfg inp*) *cfg inp*
shows *derives cfg* [$\mathfrak{S}$ *cfg*] *inp*

4.6 Completeness

lemma *Scan-$\mathcal{E}$*:
assumes $i+1 \leq k$ $k \leq$ *length inp* $x \in$ *bin* $(\mathcal{E}$ *k cfg inp*$)$ *i*
assumes *next-symbol* $x =$ *Some a inp*$!i = a$
shows *inc-item* x $(i+1) \in \mathcal{E}$ *k cfg inp*
lemma *Predict-$\mathcal{E}$*:
assumes $i \leq k$ $x \in$ *bin* $(\mathcal{E}$ *k cfg inp*$)$ *i next-symbol* $x =$ *Some N* $(N,\alpha) \in$ *set* $(\mathfrak{R}$ *cfg*$)$
shows *init-item* (N,α) $i \in \mathcal{E}$ *k cfg inp*
lemma *Complete-$\mathcal{E}$*:
assumes $i \leq j$ $j \leq k$ $x \in$ *bin* $(\mathcal{E}$ *k cfg inp*$)$ *i next-symbol* $x =$ *Some N* $(N,\alpha) \in$ *set* $(\mathfrak{R}$ *cfg*$)$
assumes $i =$ *item-origin y* $y \in$ *bin* $(\mathcal{E}$ *k cfg inp*$)$ *j item-rule* $y = (N,\alpha)$ *is-complete y*
shows *inc-item x j* $\in \mathcal{E}$ *k cfg inp*
type-synonym *'a derivation* $=$ $($*nat* $\times$ *'a rule*$)$ *list*

definition *Derives1* :: *'a cfg* $\Rightarrow$ *'a sentential* $\Rightarrow$ *nat* $\Rightarrow$ *'a rule* $\Rightarrow$ *'a sentential* $\Rightarrow$ *bool* **where**
Derives1 cfg u i r v $\equiv$
$\exists$ x y N α.
$u = x$ @ $[N]$ @ y
$\wedge$ $v = x$ @ α @ y
$\wedge$ $(N, \alpha) \in$ *set* $(\mathfrak{R}$ *cfg*$)$
$\wedge$ $r = (N, \alpha)$ $\wedge$ $i =$ *length x*

fun *Derivation* :: *'a cfg* $\Rightarrow$ *'a sentential* $\Rightarrow$ *'a derivation* $\Rightarrow$ *'a sentential* $\Rightarrow$ *bool* **where**
Derivation - a $[]$ $b = (a = b)$
| *Derivation cfg a* $(d\#D)$ $b = (\exists$ x. *Derives1 cfg a* $($*fst d*$)$ $($*snd d*$)$ x $\wedge$ *Derivation cfg x D b*$)$

definition *partially-completed* :: *nat* $\Rightarrow$ *'a cfg* $\Rightarrow$ *'a sentential* $\Rightarrow$ *'a items* $\Rightarrow$ $($*'a derivation* $\Rightarrow$ *bool*$)$ $\Rightarrow$ *bool* **where**
partially-completed k cfg inp I P $\equiv$
$\forall$ *i j x a D*.
$i \leq j$ $\wedge$ $j \leq k$ $\wedge$ $k \leq$ *length inp* $\wedge$
$x \in$ *bin I i* $\wedge$ *next-symbol* $x =$ *Some a* $\wedge$
Derivation cfg $[a]$ *D* $($*slice i j inp*$)$ $\wedge$ *P D* $\longrightarrow$
inc-item x j $\in I$

lemma *fully-completed*:
assumes $j \leq k$ $k \leq$ *length inp*
assumes $x =$ *Item* (N,α) *d i j* $x \in I$ *wf-items cfg inp I*
assumes *Derivation cfg* $($*item-*β *x*$)$ *D* $($*slice j k inp*$)$
assumes *partially-completed k cfg inp I* $(\lambda D'.$ *length* $D' \leq$ *length D*$)$
shows *Item* (N,α) $($*length* $\alpha)$ *i k* $\in I$
lemma *partially-completed-$\mathcal{E}$*:
assumes *wf-cfg cfg*

shows *partially-completed k cfg inp* ($\mathcal{E}$ *k cfg inp*) (λ-. *True*)
lemma *partially-completed-earley*:
assumes *wf-cfg cfg*
shows *partially-completed* (*length inp*) *cfg inp* (*earley cfg inp*) (λ-. *True*)
theorem *completeness*:
assumes *derives cfg* [$\mathfrak{S}$ *cfg*] *inp is-sentence cfg inp wf-cfg cfg*
shows *earley-recognized* (*earley cfg inp*) *cfg inp*
corollary
assumes *wf-cfg cfg is-sentence cfg inp*
shows *earley-recognized* (*earley cfg inp*) *cfg inp* $\longleftrightarrow$ *derives cfg* [$\mathfrak{S}$ *cfg*] *inp*

4.7 Finiteness

lemma *finiteness-UNIV-wf-item*:
shows *finite* { *x* | *x*. *wf-item cfg inp x* }
theorem *finiteness*:
shows *finite* (*earley cfg inp*)

5 Draft

- introduce auxiliary definitions: filter_with_index, pointer, entry in more detail most everything else in text

- overview over earley implementation with linked list and pointers and the mapping into a functional setting

- introduce Init_it, Scan_it, Predict_it and Complete_it, compare them with the set notation and discuss performance improvements (Grammar in more specific form) Why do they all return a list?!

- discus bin(s)_upd(s) functions. Why bin_upds like this -> easier than fold for proofs!

- discuss pi_it and why it is a partial function -> only terminates for valid input and foreshadow how this is done in isabelle

- introduce remaining definitions (analog to sets)

- discuss wf proofs quickly and go into detail about isabelle specifics about termination and the custom induction scheme using finiteness

- outline the approach to proof correctness aka subsumption in both directions

- discuss list to set proofs
- discuss soundness proofs (maybe omit since obvious)

- discuss completeness proof focusing on the complete case shortly explaining scan and predict which don't change via iteration and order does not matter

- highlight main theorems

6 Earley Recognizer Implementation

Table 6.1: Earley items with pointers for the grammar $\mathcal{G}$: $S \to x$, $S \to S + S$

	B_0	B_1	B_2
0	$S \to \bullet x, 0, 0;$	$S \to x\bullet, 0, 1; 0$	$S \to S + \bullet S, 0, 2; 1$
1	$S \to \bullet S + S, 0, 0;$	$S \to S \bullet + S, 0, 1; (0, 1, 0)$	$S \to \bullet x, 2, 2;$
2			$S \to \bullet S + S, 2, 2;$

	B_3	B_4	B_5
0	$S \to x\bullet, 2, 3; 1$	$S \to S + \bullet S, 2, 4; 2$	$S \to x\bullet, 4, 5; 2$
1	$S \to S + S\bullet, 0, 3; (2, 0, 0)$	$S \to S + \bullet S, 0, 4; 3$	$S \to S + S\bullet, 2, 5; (4, 0, 0)$
2	$S \to S \bullet + S, 2, 3; (2, 2, 0)$	$S \to \bullet x, 4, 4;$	$S \to S + S\bullet, 0, 5; (4, 1, 0), (2, 0, 1)$
3	$S \to S \bullet + S, 0, 3; (0, 1, 1)$	$S \to \bullet S + S, 4, 4;$	$S \to S \bullet + S, 4, 5; (4, 3, 0)$
4			$S \to S \bullet + S, 2, 5; (2, 2, 1)$
5			$S \to S \bullet + S, 0, 5; (0, 1, 2)$

6.1 Definitions

fun *filter-with-index'* :: *nat* ⇒ (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ (*'a* × *nat*) *list* **where**
filter-with-index' - - [] = []
| *filter-with-index' i P* (*x#xs*) = (
if P x then (*x,i*) # *filter-with-index'* (*i+1*) *P xs*
else filter-with-index' (*i+1*) *P xs*)

definition *filter-with-index* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ (*'a* × *nat*) *list* **where**
filter-with-index P xs = *filter-with-index' 0 P xs*

datatype *pointer* =
Null
| *Pre nat*
| *PreRed nat* × *nat* × *nat* (*nat* × *nat* × *nat*) *list*

datatype *'a entry* =

Entry
(*item* : *'a item*)
(*pointer* : *pointer*)

type-synonym *'a bin* = *'a entry list*
type-synonym *'a bins* = *'a bin list*

definition *items* :: *'a bin* ⇒ *'a item list* **where**
items b = *map item b*

definition *pointers* :: *'a bin* ⇒ *pointer list* **where**
pointers b = *map pointer b*

definition *bins-eq-items* :: *'a bins* ⇒ *'a bins* ⇒ *bool* **where**
bins-eq-items bs0 bs1 ⟷ *map items bs0* = *map items bs1*

definition *bins-items* :: *'a bins* ⇒ *'a items* **where**
bins-items bs = ⋃ { *set* (*items* (*bs* ! *k*)) | *k*. *k* < *length bs* }

definition *bin-items-upto* :: *'a bin* ⇒ *nat* ⇒ *'a items* **where**
bin-items-upto b i = { *items b* ! *j* | *j*. *j* < *i* ∧ *j* < *length* (*items b*) }

definition *bins-items-upto* :: *'a bins* ⇒ *nat* ⇒ *nat* ⇒ *'a items* **where**
bins-items-upto bs k i = ⋃ { *set* (*items* (*bs* ! *l*)) | *l*. *l* < *k* } ∪ *bin-items-upto* (*bs* ! *k*) *i*

definition *wf-bin-items* :: *'a cfg* ⇒ *'a sentential* ⇒ *nat* ⇒ *'a item list* ⇒ *bool* **where**
wf-bin-items cfg inp k xs ≡ ∀ *x* ∈ *set xs*. *wf-item cfg inp x* ∧ *item-end x* = *k*

definition *wf-bin* :: *'a cfg* ⇒ *'a sentential* ⇒ *nat* ⇒ *'a bin* ⇒ *bool* **where**
wf-bin cfg inp k b ≡ *distinct* (*items b*) ∧ *wf-bin-items cfg inp k* (*items b*)

definition *wf-bins* :: *'a cfg* ⇒ *'a list* ⇒ *'a bins* ⇒ *bool* **where**
wf-bins cfg inp bs ≡ ∀ *k* < *length bs*. *wf-bin cfg inp k* (*bs* ! *k*)

definition *nonempty-derives* :: *'a cfg* ⇒ *bool* **where**
nonempty-derives cfg ≡ ∀ *N*. *N* ∈ *set* (𝔑 *cfg*) ⟶ ¬ *derives cfg* [*N*] []

definition *Init-list* :: *'a cfg* ⇒ *'a sentential* ⇒ *'a bins* **where**
Init-list cfg inp ≡
let rs = *filter* (λ*r*. *rule-head r* = 𝔖 *cfg*) (ℜ *cfg*) *in*
let b0 = *map* (λ*r*. (*Entry* (*init-item r 0*) *Null*)) *rs in*
let bs = *replicate* (*length inp* + *1*) ([]) *in*
bs[*0* := *b0*]

definition *Scan-list* :: *nat* ⇒ *'a sentential* ⇒ *'a* ⇒ *'a item* ⇒ *nat* ⇒ *'a entry list* **where**
Scan-list k inp a x pre ≡
if inp!k = a then
let x' = inc-item x (k+1) in
[Entry x' (Pre pre)]
else []

definition *Predict-list* :: *nat* ⇒ *'a cfg* ⇒ *'a* ⇒ *'a entry list* **where**
Predict-list k cfg X ≡
let rs = filter (λr. rule-head r = X) (𝔑 cfg) in
map (λr. (Entry (init-item r k) Null)) rs

definition *Complete-list* :: *nat* ⇒ *'a item* ⇒ *'a bins* ⇒ *nat* ⇒ *'a entry list* **where**
Complete-list k y bs red ≡
let orig = bs ! (item-origin y) in
let is = filter-with-index (λx. next-symbol x = Some (item-rule-head y)) (items orig) in
map (λ(x, pre). (Entry (inc-item x k) (PreRed (item-origin y, pre, red) []))) is

fun *bin-upd* :: *'a entry* ⇒ *'a bin* ⇒ *'a bin* **where**
bin-upd e' [] = [e']
| *bin-upd e' (e#es) = (*
case (e', e) of
(Entry x (PreRed px xs), Entry y (PreRed py ys)) ⇒
if x = y then Entry x (PreRed py (px#xs@ys)) # es
else e # bin-upd e' es
| - ⇒
if item e' = item e then e # es
else e # bin-upd e' es)

fun *bin-upds* :: *'a entry list* ⇒ *'a bin* ⇒ *'a bin* **where**
bin-upds [] b = b
| *bin-upds (e#es) b = bin-upds es (bin-upd e b)*

definition *bins-upd* :: *'a bins* ⇒ *nat* ⇒ *'a entry list* ⇒ *'a bins* **where**
bins-upd bs k es = bs[k := bin-upds es (bs!k)]

partial-function (*tailrec*) *E-list'* :: *nat* ⇒ *'a cfg* ⇒ *'a sentential* ⇒ *'a bins* ⇒ *nat* ⇒ *'a bins* **where**
E-list' k cfg inp bs i = (
if i ≥ length (items (bs ! k)) then bs
else
let x = items (bs!k) ! i in
let bs' =
case next-symbol x of
Some a ⇒

if is-terminal cfg a then
if k < length inp then bins-upd bs (k+1) (Scan-list k inp a x i)
else bs
else bins-upd bs k (Predict-list k cfg a)
| None ⇒ bins-upd bs k (Complete-list k x bs i)
in E-list′ k cfg inp bs′ (i+1))

definition *E-list* :: *nat ⇒ ′a cfg ⇒ ′a sentential ⇒ ′a bins ⇒ ′a bins* **where**
E-list k cfg inp bs = E-list′ k cfg inp bs 0

fun *ℰ-list* :: *nat ⇒ ′a cfg ⇒ ′a sentential ⇒ ′a bins* **where**
ℰ-list 0 cfg inp = E-list 0 cfg inp (Init-list cfg inp)
| ℰ-list (Suc n) cfg inp = E-list (Suc n) cfg inp (ℰ-list n cfg inp)

definition *earley-list* :: *′a cfg ⇒ ′a sentential ⇒ ′a bins* **where**
earley-list cfg inp = ℰ-list (length inp) cfg inp

6.2 Wellformedness

lemma *wf-bin-bin-upd*:
assumes *wf-bin cfg inp k b wf-item cfg inp (item e) item-end (item e) = k*
shows *wf-bin cfg inp k (bin-upd e b)*
lemma *wf-bin-bin-upds*:
assumes *wf-bin cfg inp k b distinct (items es)*
assumes *∀ x ∈ set (items es). wf-item cfg inp x ∧ item-end x = k*
shows *wf-bin cfg inp k (bin-upds es b)*
lemma *wf-bins-bins-upd*:
assumes *wf-bins cfg inp bs distinct (items es)*
assumes *∀ x ∈ set (items es). wf-item cfg inp x ∧ item-end x = k*
shows *wf-bins cfg inp (bins-upd bs k es)*
lemma *wf-bins-Init-list*:
assumes *wf-cfg cfg*
shows *wf-bins cfg inp (Init-list cfg inp)*
lemma *wf-bins-Scan-list*:
assumes *wf-bins cfg inp bs k < length bs x ∈ set (items (bs!k)) k < length inp next-symbol x ≠ None*
shows *∀ y ∈ set (items (Scan-list k inp a x pre)). wf-item cfg inp y ∧ item-end y = k+1*
lemma *wf-bins-Predict-list*:
assumes *wf-bins cfg inp bs k < length bs k ≤ length inp wf-cfg cfg*
shows *∀ y ∈ set (items (Predict-list k cfg X)). wf-item cfg inp y ∧ item-end y = k*
lemma *wf-bins-Complete-list*:
assumes *wf-bins cfg inp bs k < length bs y ∈ set (items (bs!k))*
shows *∀ x ∈ set (items (Complete-list k y bs red)). wf-item cfg inp x ∧ item-end x = k*
fun *earley-measure* :: *nat × ′a cfg × ′a sentential × ′a bins ⇒ nat ⇒ nat* **where**

earley-measure $(k, cfg, inp, bs)\ i = card\ \{\ x \mid x.\ wf\text{-}item\ cfg\ inp\ x \wedge item\text{-}end\ x = k\ \} - i$

definition *wf-earley-input* :: $(nat \times{} 'a\ cfg \times{} 'a\ sentential \times{} 'a\ bins)\ set$ **where**
wf-earley-input = {
  $(k, cfg, inp, bs) \mid k\ cfg\ inp\ bs.$
    $k \leq length\ inp\ \wedge$
    $length\ bs = length\ inp + 1\ \wedge$
    $wf\text{-}cfg\ cfg\ \wedge$
    $wf\text{-}bins\ cfg\ inp\ bs$
}

lemma *wf-earley-input-Init-list*:
 assumes $k \leq length\ inp\ wf\text{-}cfg\ cfg$
 shows $(k, cfg, inp, Init\text{-}list\ cfg\ inp) \in wf\text{-}earley\text{-}input$
lemma *wf-earley-input-Complete-list*:
 assumes $(k, cfg, inp, bs) \in wf\text{-}earley\text{-}input\ \neg\ length\ (items\ (bs!k)) \leq i$
 assumes $x = items\ (bs!k)!i\ next\text{-}symbol\ x = None$
 shows $(k, cfg, inp, bins\text{-}upd\ bs\ k\ (Complete\text{-}list\ k\ x\ bs\ red)) \in wf\text{-}earley\text{-}input$
lemma *wf-earley-input-Scan-list*:
 assumes $(k, cfg, inp, bs) \in wf\text{-}earley\text{-}input\ \neg\ length\ (items\ (bs!k)) \leq i$
 assumes $x = items\ (bs!k)!i\ next\text{-}symbol\ x = Some\ a$
 assumes $is\text{-}terminal\ cfg\ a\ k < length\ inp$
 shows $(k, cfg, inp, bins\text{-}upd\ bs\ (k{+}1)\ (Scan\text{-}list\ k\ inp\ a\ x\ pre)) \in wf\text{-}earley\text{-}input$
lemma *wf-earley-input-Predict-list*:
 assumes $(k, cfg, inp, bs) \in wf\text{-}earley\text{-}input\ \neg\ length\ (items\ (bs!k)) \leq i$
 assumes $x = items\ (bs!k)!i\ next\text{-}symbol\ x = Some\ a\ \neg\ is\text{-}terminal\ cfg\ a$
 shows $(k, cfg, inp, bins\text{-}upd\ bs\ k\ (Predict\text{-}list\ k\ cfg\ a)) \in wf\text{-}earley\text{-}input$
lemma *wf-earley-input-E-list'*:
 assumes $(k, cfg, inp, bs) \in wf\text{-}earley\text{-}input$
 shows $(k, cfg, inp, E\text{-}list'\ k\ cfg\ inp\ bs\ i) \in wf\text{-}earley\text{-}input$
lemma *wf-earley-input-E-list*:
 assumes $(k, cfg, inp, bs) \in wf\text{-}earley\text{-}input$
 shows $(k, cfg, inp, E\text{-}list\ k\ cfg\ inp\ bs) \in wf\text{-}earley\text{-}input$
lemma *wf-earley-input-$\mathcal{E}$-list*:
 assumes $k \leq length\ inp\ wf\text{-}cfg\ cfg$
 shows $(k, cfg, inp, \mathcal{E}\text{-}list\ k\ cfg\ inp) \in wf\text{-}earley\text{-}input$
lemma *wf-earley-input-earley-list*:
 assumes $k \leq length\ inp\ wf\text{-}cfg\ cfg$
 shows $(k, cfg, inp, earley\text{-}list\ cfg\ inp) \in wf\text{-}earley\text{-}input$
lemma *wf-bins-E-list'*:
 assumes $(k, cfg, inp, bs) \in wf\text{-}earley\text{-}input$
 shows $wf\text{-}bins\ cfg\ inp\ (E\text{-}list'\ k\ cfg\ inp\ bs\ i)$
lemma *wf-bins-E-list*:
 assumes $(k, cfg, inp, bs) \in wf\text{-}earley\text{-}input$

shows *wf-bins cfg inp* (*E-list k cfg inp bs*)
lemma *wf-bins-$\mathcal{E}$-list*:
assumes $k \leq$ *length inp wf-cfg cfg*
shows *wf-bins cfg inp* ($\mathcal{E}$*-list k cfg inp*)
lemma *wf-bins-earley-list*:
assumes *wf-cfg cfg*
shows *wf-bins cfg inp* (*earley-list cfg inp*)

6.3 Soundness

lemma *Init-list-eq-Init*:
shows *bins-items* (*Init-list cfg inp*) = *Init cfg*
lemma *Scan-list-sub-Scan*:
assumes *wf-bins cfg inp bs bins-items bs* $\subseteq$ *I x* $\in$ *set* (*items* (*bs* ! *k*))
assumes $k <$ *length bs* $k <$ *length inp next-symbol x* = *Some a*
shows *set* (*items* (*Scan-list k inp a x pre*)) $\subseteq$ *Scan k inp I*
lemma *Predict-list-sub-Predict*:
assumes *wf-bins cfg inp bs bins-items bs* $\subseteq$ *I x* $\in$ *set* (*items* (*bs* ! *k*)) $k <$ *length bs*
assumes *next-symbol x* = *Some X*
shows *set* (*items* (*Predict-list k cfg X*)) $\subseteq$ *Predict k cfg I*
lemma *Complete-list-sub-Complete*:
assumes *wf-bins cfg inp bs bins-items bs* $\subseteq$ *I y* $\in$ *set* (*items* (*bs* ! *k*)) $k <$ *length bs*
assumes *next-symbol y* = *None*
shows *set* (*items* (*Complete-list k y bs red*)) $\subseteq$ *Complete k I*
lemma *E-list'-sub-E*:
assumes (*k*, *cfg*, *inp*, *bs*) $\in$ *wf-earley-input*
assumes *bins-items bs* $\subseteq$ *I*
shows *bins-items* (*E-list' k cfg inp bs i*) $\subseteq$ *E k cfg inp I*
lemma *E-list-sub-E*:
assumes (*k*, *cfg*, *inp*, *bs*) $\in$ *wf-earley-input*
assumes *bins-items bs* $\subseteq$ *I*
shows *bins-items* (*E-list k cfg inp bs*) $\subseteq$ *E k cfg inp I*
lemma $\mathcal{E}$*-list-sub-*$\mathcal{E}$:
assumes $k \leq$ *length inp wf-cfg cfg*
shows *bins-items* ($\mathcal{E}$*-list k cfg inp*) $\subseteq$ $\mathcal{E}$ *k cfg inp*
lemma *earley-list-sub-earley*:
assumes *wf-cfg cfg*
shows *bins-items* (*earley-list cfg inp*) $\subseteq$ *earley cfg inp*

6.4 Completeness

lemma *impossible-complete-item*:

assumes *wf-cfg cfg wf-item cfg inp x sound-item cfg inp x*
assumes *is-complete x item-origin x* $=$ *k item-end x* $=$ *k nonempty-derives cfg*
shows *False*
lemma *Complete-Un-eq-terminal*:
assumes *next-symbol z* $=$ *Some a is-terminal cfg a wf-items cfg inp I wf-item cfg inp z wf-cfg cfg*
shows *Complete k* $(I \cup \{z\}) =$ *Complete k I*
lemma *Complete-Un-eq-nonterminal*:
assumes *next-symbol z* $=$ *Some a is-nonterminal cfg a sound-items cfg inp I item-end z* $=$ *k*
assumes *wf-items cfg inp I wf-item cfg inp z wf-cfg cfg nonempty-derives cfg*
shows *Complete k* $(I \cup \{z\}) =$ *Complete k I*
lemma *Complete-sub-bins-Un-Complete-list*:
assumes *Complete k I* $\subseteq$ *bins-items bs I* $\subseteq$ *bins-items bs is-complete z wf-bins cfg inp bs wf-item cfg inp z*
shows *Complete k* $(I \cup \{z\}) \subseteq$ *bins-items bs* $\cup$ *set* (*items* (*Complete-list k z bs red*))
lemma *E-list′-mono*:
assumes (*k*, *cfg*, *inp*, *bs*) $\in$ *wf-earley-input*
shows *bins-items bs* $\subseteq$ *bins-items* (*E-list′ k cfg inp bs i*)
lemma *E-step-sub-E-list′*:
assumes (*k*, *cfg*, *inp*, *bs*) $\in$ *wf-earley-input*
assumes *E-step k cfg inp* (*bins-items-upto bs k i*) $\subseteq$ *bins-items bs*
assumes *sound-items cfg inp* (*bins-items bs*) *is-sentence cfg inp nonempty-derives cfg*
shows *E-step k cfg inp* (*bins-items bs*) $\subseteq$ *bins-items* (*E-list′ k cfg inp bs i*)
lemma *E-step-sub-E-list*:
assumes (*k*, *cfg*, *inp*, *bs*) $\in$ *wf-earley-input*
assumes *E-step k cfg inp* (*bins-items-upto bs k 0*) $\subseteq$ *bins-items bs*
assumes *sound-items cfg inp* (*bins-items bs*) *is-sentence cfg inp nonempty-derives cfg*
shows *E-step k cfg inp* (*bins-items bs*) $\subseteq$ *bins-items* (*E-list k cfg inp bs*)
lemma *E-list′-bins-items-eq*:
assumes (*k*, *cfg*, *inp*, *as*) $\in$ *wf-earley-input*
assumes *bins-eq-items as bs wf-bins cfg inp as*
shows *bins-eq-items* (*E-list′ k cfg inp as i*) (*E-list′ k cfg inp bs i*)
lemma *E-list′-idem*:
assumes (*k*, *cfg*, *inp*, *bs*) $\in$ *wf-earley-input*
assumes $i \le j$ *sound-items cfg inp* (*bins-items bs*) *nonempty-derives cfg*
shows *bins-items* (*E-list′ k cfg inp* (*E-list′ k cfg inp bs i*) *j*) $=$ *bins-items* (*E-list′ k cfg inp bs i*)
lemma *E-list-idem*:
assumes (*k*, *cfg*, *inp*, *bs*) $\in$ *wf-earley-input*
assumes *sound-items cfg inp* (*bins-items bs*) *nonempty-derives cfg*
shows *bins-items* (*E-list k cfg inp* (*E-list k cfg inp bs*)) $=$ *bins-items* (*E-list k cfg inp bs*)
lemma *funpower-E-step-sub-E-list*:
assumes (*k*, *cfg*, *inp*, *bs*) $\in$ *wf-earley-input*
assumes *E-step k cfg inp* (*bins-items-upto bs k 0*) $\subseteq$ *bins-items bs sound-items cfg inp* (*bins-items bs*)
assumes *is-sentence cfg inp nonempty-derives cfg*
shows *funpower* (*E-step k cfg inp*) *n* (*bins-items bs*) $\subseteq$ *bins-items* (*E-list k cfg inp bs*)

lemma *E-sub-E-list*:
  assumes $(k, cfg, inp, bs) \in$ *wf-earley-input*
  assumes *E-step k cfg inp* (*bins-items-upto bs k 0*) $\subseteq$ *bins-items bs sound-items cfg inp* (*bins-items bs*)
  assumes *is-sentence cfg inp nonempty-derives cfg*
  shows *E k cfg inp* (*bins-items bs*) $\subseteq$ *bins-items* (*E-list k cfg inp bs*)
lemma $\mathcal{E}$*-sub-*$\mathcal{E}$*-list*:
  assumes $k \leq$ *length inp wf-cfg cfg*
  assumes *is-sentence cfg inp nonempty-derives cfg*
  shows $\mathcal{E}$ *k cfg inp* $\subseteq$ *bins-items* ($\mathcal{E}$*-list k cfg inp*)
lemma *earley-sub-earley-list*:
  assumes *wf-cfg cfg is-sentence cfg inp nonempty-derives cfg*
  shows *earley cfg inp* $\subseteq$ *bins-items* (*earley-list cfg inp*)

6.5 Main Theorem

definition *earley-recognized-list* :: *'a bins* $\Rightarrow$ *'a cfg* $\Rightarrow$ *'a sentential* $\Rightarrow$ *bool* **where**
  earley-recognized-list I cfg inp $\equiv \exists\, x \in$ *set* (*items* (*I* ! *length inp*)). *is-finished cfg inp x*

theorem *earley-recognized-list-iff-earley-recognized*:
  assumes *wf-cfg cfg is-sentence cfg inp nonempty-derives cfg*
  shows *earley-recognized-list* (*earley-list cfg inp*) *cfg inp* $\longleftrightarrow$ *earley-recognized* (*earley cfg inp*) *cfg inp*
corollary *correctness-list*:
  assumes *wf-cfg cfg is-sentence cfg inp nonempty-derives cfg*
  shows *earley-recognized-list* (*earley-list cfg inp*) *cfg inp* $\longleftrightarrow$ *derives cfg* [$\mathfrak{S}$ *cfg*] *inp*

7 Earley Parser Implementation

7.1 Draft

7.2 Pointer lemmas

definition *predicts* :: $'a$ *item* $\Rightarrow$ *bool* **where**
predicts $x \equiv$ *item-origin* $x =$ *item-end* $x \wedge$ *item-dot* $x = 0$

definition *scans* :: $'a$ *sentential* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *item* $\Rightarrow$ $'a$ *item* $\Rightarrow$ *bool* **where**
scans inp k x y $\equiv y =$ *inc-item x k* $\wedge$ $(\exists a.$ *next-symbol* $x =$ *Some a* $\wedge$ *inp*$!(k-1) = a)$

definition *completes* :: *nat* $\Rightarrow$ $'a$ *item* $\Rightarrow$ $'a$ *item* $\Rightarrow$ $'a$ *item* $\Rightarrow$ *bool* **where**
completes k x y z $\equiv y =$ *inc-item x k* $\wedge$ *is-complete z* $\wedge$ *item-origin z* $=$ *item-end x* $\wedge$
$(\exists N.$ *next-symbol* $x =$ *Some N* $\wedge N =$ *item-rule-head z*$)$

definition *sound-null-ptr* :: $'a$ *entry* $\Rightarrow$ *bool* **where**
sound-null-ptr e $\equiv$ *pointer e* $=$ *Null* $\longrightarrow$ *predicts* $($*item e*$)$

definition *sound-pre-ptr* :: $'a$ *sentential* $\Rightarrow$ $'a$ *bins* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *entry* $\Rightarrow$ *bool* **where**
sound-pre-ptr inp bs k e $\equiv \forall$ *pre. pointer e* $=$ *Pre pre* $\longrightarrow$
$k > 0 \wedge$ *pre* $<$ *length* $($*bs*$!(k-1)) \wedge$ *scans inp k* $($*item* $($*bs*$!(k-1)!$*pre*$))$ $($*item e*$)$

definition *sound-prered-ptr* :: $'a$ *bins* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *entry* $\Rightarrow$ *bool* **where**
sound-prered-ptr bs k e $\equiv \forall$ *p ps* k' *pre red. pointer e* $=$ *PreRed p ps* $\wedge$ $(k',$ *pre*, *red*$) \in$ *set* $($*p*$\#$*ps*$) \longrightarrow$
$k' < k \wedge$ *pre* $<$ *length* $($*bs*$!k') \wedge$ *red* $<$ *length* $($*bs*$!k) \wedge$ *completes k* $($*item* $($*bs*$!k'!$*pre*$))$ $($*item e*$)$ $($*item*
$($*bs*$!k!$*red*$))$

definition *sound-ptrs* :: $'a$ *sentential* $\Rightarrow$ $'a$ *bins* $\Rightarrow$ *bool* **where**
sound-ptrs inp bs $\equiv \forall k <$ *length bs*. $\forall e \in$ *set* $($*bs*$!k)$.
sound-null-ptr e $\wedge$
sound-pre-ptr inp bs k e $\wedge$
sound-prered-ptr bs k e

definition *mono-red-ptr* :: $'a$ *bins* $\Rightarrow$ *bool* **where**
mono-red-ptr bs $\equiv \forall k <$ *length bs*. $\forall i <$ *length* $($*bs*$!k)$.
$\forall k'$ *pre red ps. pointer* $($*bs*$!k!i) =$ *PreRed* $(k',$ *pre*, *red*$)$ *ps* $\longrightarrow$ *red* $< i$

lemma *sound-ptrs-bin-upd*:
assumes *sound-ptrs inp bs* $k <$ *length bs es* $=$ *bs*!*k distinct* (*items es*)
assumes *sound-null-ptr e sound-pre-ptr inp bs k e sound-prered-ptr bs k e*
shows *sound-ptrs inp* (*bs*[*k* := *bin-upd e es*])
lemma *mono-red-ptr-bin-upd*:
assumes *mono-red-ptr bs* $k <$ *length bs es* $=$ *bs*!*k distinct* (*items es*)
assumes $\forall k'$ *pre red ps. pointer e* $=$ *PreRed* (k', *pre*, *red*) *ps* $\longrightarrow$ *red* $<$ *length es*
shows *mono-red-ptr* (*bs*[*k* := *bin-upd e es*])
lemma *sound-mono-ptrs-bin-upds*:
assumes *sound-ptrs inp bs mono-red-ptr bs* $k <$ *length bs b* $=$ *bs*!*k distinct* (*items b*) *distinct* (*items es*)
assumes $\forall e \in$ *set es. sound-null-ptr e* $\wedge$ *sound-pre-ptr inp bs k e* $\wedge$ *sound-prered-ptr bs k e*
assumes $\forall e \in$ *set es.* $\forall k'$ *pre red ps. pointer e* $=$ *PreRed* (k', *pre*, *red*) *ps* $\longrightarrow$ *red* $<$ *length b*
shows *sound-ptrs inp* (*bs*[*k* := *bin-upds es b*]) $\wedge$ *mono-red-ptr* (*bs*[*k* := *bin-upds es b*])
lemma *sound-mono-ptrs-E-list*$'$:
assumes (*k*, *cfg*, *inp*, *bs*) $\in$ *wellformed-bins*
assumes *sound-ptrs inp bs sound-items cfg inp* (*bins-items bs*)
assumes *mono-red-ptr bs*
assumes *nonempty-derives cfg wf-cfg cfg*
shows *sound-ptrs inp* (*E-list*$'$ *k cfg inp bs i*) $\wedge$ *mono-red-ptr* (*E-list*$'$ *k cfg inp bs i*)
lemma *sound-mono-ptrs-E-list*:
assumes (*k*, *cfg*, *inp*, *bs*) $\in$ *wellformed-bins*
assumes *sound-ptrs inp bs sound-items cfg inp* (*bins-items bs*)
assumes *mono-red-ptr bs*
assumes *nonempty-derives cfg wf-cfg cfg*
shows *sound-ptrs inp* (*E-list k cfg inp bs*) $\wedge$ *mono-red-ptr* (*E-list k cfg inp bs*)
lemma *sound-ptrs-Init-list*:
shows *sound-ptrs inp* (*Init-list cfg inp*)
lemma *mono-red-ptr-Init-list*:
shows *mono-red-ptr* (*Init-list cfg inp*)
lemma *sound-mono-ptrs-*$\mathcal{E}$*-list*:
assumes $k \leq$ *length inp wf-cfg cfg nonempty-derives cfg wf-cfg cfg*
shows *sound-ptrs inp* ($\mathcal{E}$*-list k cfg inp*) $\wedge$ *mono-red-ptr* ($\mathcal{E}$*-list k cfg inp*)
lemma *sound-mono-ptrs-earley-list*:
assumes *wf-cfg cfg nonempty-derives cfg*
shows *sound-ptrs inp* (*earley-list cfg inp*) $\wedge$ *mono-red-ptr* (*earley-list cfg inp*)

7.3 Trees and Forests

datatype *'a tree* =
Leaf 'a
| *Branch 'a 'a tree list*

fun *yield-tree* :: *'a tree* $\Rightarrow$ *'a sentential* **where**
yield-tree (*Leaf a*) = [*a*]
| *yield-tree* (*Branch - ts*) = *concat* (*map yield-tree ts*)

fun *root-tree* :: *'a tree* $\Rightarrow$ *'a* **where**
root-tree (*Leaf a*) = *a*
| *root-tree* (*Branch N -*) = *N*

fun *wf-rule-tree* :: *'a cfg* $\Rightarrow$ *'a tree* $\Rightarrow$ *bool* **where**
wf-rule-tree - (Leaf a) $\longleftrightarrow$ *True*
| *wf-rule-tree cfg (Branch N ts)* $\longleftrightarrow$ (
  ($\exists\, r \in$ *set* ($\mathfrak{R}$ *cfg*). *N* = *rule-head r* $\wedge$ *map root-tree ts* = *rule-body r*) $\wedge$
  ($\forall\, t \in$ *set ts. wf-rule-tree cfg t*))

fun *wf-item-tree* :: *'a cfg* $\Rightarrow$ *'a item* $\Rightarrow$ *'a tree* $\Rightarrow$ *bool* **where**
wf-item-tree cfg - (Leaf a) $\longleftrightarrow$ *True*
| *wf-item-tree cfg x (Branch N ts)* $\longleftrightarrow$ (
  N = *item-rule-head x* $\wedge$ *map root-tree ts* = *take* (*item-dot x*) (*item-rule-body x*) $\wedge$
  ($\forall\, t \in$ *set ts. wf-rule-tree cfg t*))

definition *wf-yield-tree* :: *'a sentential* $\Rightarrow$ *'a item* $\Rightarrow$ *'a tree* $\Rightarrow$ *bool* **where**
wf-yield-tree inp x t $\equiv$ *yield-tree t* = *slice* (*item-origin x*) (*item-end x*) *inp*

datatype *'a forest* =
FLeaf 'a
| *FBranch 'a 'a forest list list*

fun *combinations* :: *'a list list* $\Rightarrow$ *'a list list* **where**
combinations [] = [[]]
| *combinations* (*xs#xss*) = [*x#cs . x <− xs, cs <− combinations xss*]

fun *trees* :: *'a forest* $\Rightarrow$ *'a tree list* **where**
trees (*FLeaf a*) = [*Leaf a*]
| *trees* (*FBranch N fss*) = (
  let tss = (*map* (λ*fs. concat* (*map* (λ*f. trees f*) *fs*)) *fss*) *in*
  map (λ*ts. Branch N ts*) (*combinations tss*)
)

7.4 A Single Parse Tree

partial-function (*option*) *build-tree'* :: *'a bins* $\Rightarrow$ *'a sentential* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *'a tree option* **where**
build-tree' bs inp k i = (
  let e = *bs!k!i in* (

```
  case pointer e of
    Null ⇒ Some (Branch (item-rule-head (item e)) [])
  | Pre pre ⇒ (
      do {
        t ← build-tree' bs inp (k−1) pre;
        case t of
          Branch N ts ⇒ Some (Branch N (ts @ [Leaf (inp!(k−1))]))
        | - ⇒ None
      })
  | PreRed (k', pre, red) - ⇒ (
      do {
        t ← build-tree' bs inp k' pre;
        case t of
          Branch N ts ⇒
            do {
              t ← build-tree' bs inp k red;
              Some (Branch N (ts @ [t]))
            }
        | - ⇒ None
      })
 ))
```

```
definition build-tree :: 'a cfg ⇒ 'a sentential ⇒ 'a bins ⇒ 'a tree option where
 build-tree cfg inp bs ≡
  let k = length bs − 1 in (
  case filter-with-index (λx. is-finished cfg inp x) (items (bs!k)) of
    [] ⇒ None
  | (-, i)#- ⇒ build-tree' bs inp k i)
```

```
fun build-tree'-measure :: ('a bins × 'a sentential × nat × nat) ⇒ nat where
 build-tree'-measure (bs, inp, k, i) = foldl (+) 0 (map length (take k bs)) + i
```

```
definition wf-tree-input :: ('a bins × 'a sentential × nat × nat) set where
 wf-tree-input = {
  (bs, inp, k, i) | bs inp k i.
    sound-ptrs inp bs ∧
    mono-red-ptr bs ∧
    k < length bs ∧
    i < length (bs!k)
 }
```

```
lemma wf-tree-input-pre:
 assumes (bs, inp, k, i) ∈ wf-tree-input
 assumes e = bs!k!i pointer e = Pre pre
```

shows $(bs, inp, (k-1), pre) \in$ *wf-tree-input*
lemma *wf-tree-input-prered-pre*:
assumes $(bs, inp, k, i) \in$ *wf-tree-input*
assumes $e = bs!k!i$ *pointer* $e = PreRed\ (k', pre, red)\ ps$
shows $(bs, inp, k', pre) \in$ *wf-tree-input*
lemma *wf-tree-input-prered-red*:
assumes $(bs, inp, k, i) \in$ *wf-tree-input*
assumes $e = bs!k!i$ *pointer* $e = PreRed\ (k', pre, red)\ ps$
shows $(bs, inp, k, red) \in$ *wf-tree-input*
lemma *build-tree′-termination*:
assumes $(bs, inp, k, i) \in$ *wf-tree-input*
shows $\exists N\ ts.$ *build-tree′ bs inp k i* $=$ *Some* $(Branch\ N\ ts)$
lemma *wf-item-tree-build-tree′*:
assumes $(bs, inp, k, i) \in$ *wf-tree-input*
assumes *wf-bins cfg inp bs*
assumes $k <$ *length bs* $i <$ *length* $(bs!k)$
assumes *build-tree′ bs inp k i* $=$ *Some t*
shows *wf-item-tree cfg* $(item\ (bs!k!i))\ t$
lemma *wf-yield-tree-build-tree′*:
assumes $(bs, inp, k, i) \in$ *wf-tree-input*
assumes *wf-bins cfg inp bs*
assumes $k <$ *length bs* $i <$ *length* $(bs!k)$ $k \leq$ *length inp*
assumes *build-tree′ bs inp k i* $=$ *Some t*
shows *wf-yield-tree inp* $(item\ (bs!k!i))\ t$
theorem *wf-rule-root-yield-tree-build-tree*:
assumes *wf-bins cfg inp bs sound-ptrs inp bs mono-red-ptr bs length bs* $=$ *length inp* $+ 1$
assumes *build-tree cfg inp bs* $=$ *Some t*
shows *wf-rule-tree cfg t* $\wedge$ *root-tree t* $= \mathfrak{S}$ *cfg* $\wedge$ *yield-tree t* $=$ *inp*
corollary *wf-rule-root-yield-tree-build-tree-earley-list*:
assumes *wf-cfg cfg nonempty-derives cfg*
assumes *build-tree cfg inp* $($*earley-list cfg inp*$) =$ *Some t*
shows *wf-rule-tree cfg t* $\wedge$ *root-tree t* $= \mathfrak{S}$ *cfg* $\wedge$ *yield-tree t* $=$ *inp*
theorem *correctness-build-tree-earley-list*:
assumes *wf-cfg cfg is-sentence cfg inp nonempty-derives cfg*
shows $(\exists t.$ *build-tree cfg inp* $($*earley-list cfg inp*$) =$ *Some t*$) \longleftrightarrow$ *derives cfg* $[\mathfrak{S}\ cfg]$ *inp*

7.5 All Parse Trees

fun *insert-group* :: $('a \Rightarrow 'k) \Rightarrow ('a \Rightarrow 'v) \Rightarrow 'a \Rightarrow ('k \times 'v\ list)\ list \Rightarrow ('k \times 'v\ list)\ list$ **where**
insert-group K V a $[] = [(K\ a, [V\ a])]$
| *insert-group K V a* $((k, vs)\#xs) = ($
if $K\ a = k$ *then* $(k, V\ a\ \#\ vs)\ \#\ xs$
else $(k, vs)\ \#$ *insert-group K V a xs*

```
)

fun group-by :: ('a ⇒ 'k) ⇒ ('a ⇒ 'v) ⇒ 'a list ⇒ ('k × 'v list) list where
  group-by K V [] = []
| group-by K V (x#xs) = insert-group K V x (group-by K V xs)

partial-function (option) build-trees' :: 'a bins ⇒ 'a sentential ⇒ nat ⇒ nat ⇒ nat set ⇒ 'a forest
list option where
  build-trees' bs inp k i I = (
    let e = bs!k!i in (
    case pointer e of
      Null ⇒ Some ([FBranch (item-rule-head (item e)) []])
    | Pre pre ⇒ (
        do {
          pres ← build-trees' bs inp (k−1) pre {pre};
          those (map (λf.
            case f of
              FBranch N fss ⇒ Some (FBranch N (fss @ [[FLeaf (inp!(k−1))]]))
            | - ⇒ None
          ) pres)
        })
    | PreRed p ps ⇒ (
        let ps' = filter (λ(k', pre, red). red ∉ I) (p#ps) in
        let gs = group-by (λ(k', pre, red). (k', pre)) (λ(k', pre, red). red) ps' in
        map-option concat (those (map (λ((k', pre), reds).
          do {
            pres ← build-trees' bs inp k' pre {pre};
            rss ← those (map (λred. build-trees' bs inp k red (I ∪ {red})) reds);
            those (map (λf.
              case f of
                FBranch N fss ⇒ Some (FBranch N (fss @ [concat rss]))
              | - ⇒ None
            ) pres)
          }
        ) gs))
      )
  ))

definition build-trees :: 'a cfg ⇒ 'a sentential ⇒ 'a bins ⇒ 'a forest list option where
  build-trees cfg inp bs ≡
    let k = length bs − 1 in
    let finished = filter-with-index (λx. is-finished cfg inp x) (items (bs!k)) in
    map-option concat (those (map (λ(-, i). build-trees' bs inp k i {i}) finished))
```

fun $\mathit{build\text{-}forest'\text{-}measure} :: ({'a}\ \mathit{bins} \times {'a}\ \mathit{sentential} \times \mathit{nat} \times \mathit{nat} \times \mathit{nat\ set}) \Rightarrow \mathit{nat}$ **where**
$\mathit{build\text{-}forest'\text{-}measure}\ (bs, inp, k, i, I) = \mathit{foldl}\ (+)\ 0\ (\mathit{map\ length}\ (\mathit{take}\ (k{+}1)\ bs)) - \mathit{card}\ I$

definition $\mathit{wf\text{-}trees\text{-}input} :: ({'a}\ \mathit{bins} \times {'a}\ \mathit{sentential} \times \mathit{nat} \times \mathit{nat} \times \mathit{nat\ set})\ \mathit{set}$ **where**
$\mathit{wf\text{-}trees\text{-}input} = \{$
$(bs, inp, k, i, I) \mid bs\ inp\ k\ i\ I.$
$\mathit{sound\text{-}ptrs}\ inp\ bs\ \wedge$
$k < \mathit{length}\ bs\ \wedge$
$i < \mathit{length}\ (bs!k)\ \wedge$
$I \subseteq \{0..{<}\mathit{length}\ (bs!k)\}\ \wedge$
$i \in I$
$\}$

lemma $\mathit{wf\text{-}trees\text{-}input\text{-}pre}$:
assumes $(bs, inp, k, i, I) \in \mathit{wf\text{-}trees\text{-}input}$
assumes $e = bs!k!i\ \mathit{pointer}\ e = \mathit{Pre}\ \mathit{pre}$
shows $(bs, inp, (k{-}1), \mathit{pre}, \{\mathit{pre}\}) \in \mathit{wf\text{-}trees\text{-}input}$
lemma $\mathit{wf\text{-}trees\text{-}input\text{-}prered\text{-}pre}$:
assumes $(bs, inp, k, i, I) \in \mathit{wf\text{-}trees\text{-}input}$
assumes $e = bs!k!i\ \mathit{pointer}\ e = \mathit{PreRed}\ p\ ps$
assumes $ps' = \mathit{filter}\ (\lambda(k', \mathit{pre}, \mathit{red}).\ \mathit{red} \notin I)\ (p\#ps)$
assumes $gs = \mathit{group\text{-}by}\ (\lambda(k', \mathit{pre}, \mathit{red}).\ (k', \mathit{pre}))\ (\lambda(k', \mathit{pre}, \mathit{red}).\ \mathit{red})\ ps'$
assumes $((k', \mathit{pre}), \mathit{reds}) \in \mathit{set}\ gs$
shows $(bs, inp, k', \mathit{pre}, \{\mathit{pre}\}) \in \mathit{wf\text{-}trees\text{-}input}$
lemma $\mathit{wf\text{-}trees\text{-}input\text{-}prered\text{-}red}$:
assumes $(bs, inp, k, i, I) \in \mathit{wf\text{-}trees\text{-}input}$
assumes $e = bs!k!i\ \mathit{pointer}\ e = \mathit{PreRed}\ p\ ps$
assumes $ps' = \mathit{filter}\ (\lambda(k', \mathit{pre}, \mathit{red}).\ \mathit{red} \notin I)\ (p\#ps)$
assumes $gs = \mathit{group\text{-}by}\ (\lambda(k', \mathit{pre}, \mathit{red}).\ (k', \mathit{pre}))\ (\lambda(k', \mathit{pre}, \mathit{red}).\ \mathit{red})\ ps'$
assumes $((k', \mathit{pre}), \mathit{reds}) \in \mathit{set}\ gs\ \mathit{red} \in \mathit{set}\ \mathit{reds}$
shows $(bs, inp, k, \mathit{red}, I \cup \{\mathit{red}\}) \in \mathit{wf\text{-}trees\text{-}input}$
lemma $\mathit{build\text{-}trees'\text{-}termination}$:
assumes $(bs, inp, k, i, I) \in \mathit{wf\text{-}trees\text{-}input}$
shows $\exists fs.\ \mathit{build\text{-}trees'}\ bs\ inp\ k\ i\ I = \mathit{Some}\ fs \wedge (\forall f \in \mathit{set}\ fs.\ \exists N\ fss.\ f = \mathit{FBranch}\ N\ fss)$
lemma $\mathit{wf\text{-}item\text{-}tree\text{-}build\text{-}trees'}$:
assumes $(bs, inp, k, i, I) \in \mathit{wf\text{-}trees\text{-}input}$
assumes $\mathit{wf\text{-}bins}\ \mathit{cfg}\ inp\ bs$
assumes $k < \mathit{length}\ bs\ i < \mathit{length}\ (bs!k)$
assumes $\mathit{build\text{-}trees'}\ bs\ inp\ k\ i\ I = \mathit{Some}\ fs$
assumes $f \in \mathit{set}\ fs$
assumes $t \in \mathit{set}\ (\mathit{trees}\ f)$
shows $\mathit{wf\text{-}item\text{-}tree}\ \mathit{cfg}\ (\mathit{item}\ (bs!k!i))\ t$
lemma $\mathit{wf\text{-}yield\text{-}tree\text{-}build\text{-}trees'}$:
assumes $(bs, inp, k, i, I) \in \mathit{wf\text{-}trees\text{-}input}$

assumes *wf-bins cfg inp bs*
assumes $k <$ *length bs* $i <$ *length* $(bs!k)$ $k \leq$ *length inp*
assumes *build-trees' bs inp k i I* $=$ *Some fs*
assumes $f \in$ *set fs*
assumes $t \in$ *set* $($*trees f*$)$
shows *wf-yield-tree inp* $($*item* $(bs!k!i))$ t
theorem *wf-rule-root-yield-tree-build-trees*:
assumes *wf-bins cfg inp bs sound-ptrs inp bs length bs* $=$ *length inp* $+ 1$
assumes *build-trees cfg inp bs* $=$ *Some fs* $f \in$ *set fs* $t \in$ *set* $($*trees f*$)$
shows *wf-rule-tree cfg t* $\wedge$ *root-tree t* $= \mathfrak{S}$ *cfg* $\wedge$ *yield-tree t* $=$ *inp*
corollary *wf-rule-root-yield-tree-build-trees-earley-list*:
assumes *wf-cfg cfg nonempty-derives cfg*
assumes *build-trees cfg inp* $($*earley-list cfg inp*$)$ $=$ *Some fs* $f \in$ *set fs* $t \in$ *set* $($*trees f*$)$
shows *wf-rule-tree cfg t* $\wedge$ *root-tree t* $= \mathfrak{S}$ *cfg* $\wedge$ *yield-tree t* $=$ *inp*
theorem *soundness-build-trees-earley-list*:
assumes *wf-cfg cfg is-sentence cfg inp nonempty-derives cfg*
assumes *build-trees cfg inp* $($*earley-list cfg inp*$)$ $=$ *Some fs* $f \in$ *set fs* $t \in$ *set* $($*trees f*$)$
shows *derives cfg* $[\mathfrak{S}$ *cfg*$]$ *inp*
theorem *termination-build-tree-earley-list*:
assumes *wf-cfg cfg nonempty-derives cfg derives cfg* $[\mathfrak{S}$ *cfg*$]$ *inp*
shows $\exists$*fs. build-trees cfg inp* $($*earley-list cfg inp*$)$ $=$ *Some fs*

7.6 A Word on Completeness

8 Usage

definition ε*-free* :: *'a cfg* $\Rightarrow$ *bool* **where**
ε*-free cfg* $\longleftrightarrow$ $(\forall\, r \in set\ (\mathfrak{R}\ cfg).\ rule\text{-}body\ r \neq [])$

lemma ε*-free-impl-non-empty-deriv*:
$\varepsilon\text{-}free\ cfg \Longrightarrow N \in set\ (\mathfrak{R}\ cfg) \Longrightarrow \neg\ derives\ cfg\ [N]\ []$
datatype $t = x \mid plus$
datatype $n = S$
datatype $s = Terminal\ t \mid Nonterminal\ n$

definition *nonterminals* :: *s list* **where**
nonterminals = [*Nonterminal S*]

definition *terminals* :: *s list* **where**
terminals = [*Terminal x*, *Terminal plus*]

definition *rules* :: *s rule list* **where**
rules = [
(*Nonterminal S*, [*Terminal x*]),
(*Nonterminal S*, [*Nonterminal S*, *Terminal plus*, *Nonterminal S*])
]

definition *start-symbol* :: *s* **where**
start-symbol = *Nonterminal S*

definition *cfg* :: *s cfg* **where**
cfg = *CFG nonterminals terminals rules start-symbol*

definition *inp* :: *s list* **where**
inp = [*Terminal x*, *Terminal plus*, *Terminal x*, *Terminal plus*, *Terminal x*]

lemma *wf-cfg*:
shows *wf-cfg cfg*
lemma *is-sentence-inp*:
shows *is-sentence cfg inp*
lemma *nonempty-derives*:
shows *nonempty-derives cfg*
lemma *correctness*:

shows *earley-recognized-list* (*earley-list cfg inp*) *cfg inp* $\longleftrightarrow$ *derives cfg* [$\mathfrak{S}$ *cfg*] *inp*

9 Conclusion

9.1 Summary

9.2 Future Work

Different approaches:

(1) SPPF style parse trees as in Scott et al -> need Imperative/HOL for this

Performance improvements:

(1) Look-ahead of k or at least 1 like in the original Earley paper. (2) Optimize the representation of the grammar instead of single list, group by production, ... (3) Keep a set of already inserted items to not double check item insertion. (4) Use a queue instead of a list for bins. (5) Refine the algorithm to an imperative version using a single linked list and actual pointers instead of natural numbers.

Parse tree disambiguation:

Parser generators like YACC resolve ambiguities in context-free grammers by allowing the user the specify precedence and associativity declarations restricting the set of allowed parses. But they do not handle all grammatical restrictions, like 'dangling else' or interactions between binary operators and functional 'if'-expressions.

Grammar rewriting:

Adams *et al* [**Adams:2017**] describe a grammar rewriting approach reinterpreting CFGs as the tree automata, intersectiong them with tree automata encoding desired restrictions and reinterpreting the results back into CFGs.

Afroozeh *et al* [**Afroozeh:2013**] present an approach to specifying operator precedence based on declarative disambiguation rules basing their implementation on grammar rewriting.

Thorup [**Thorup:1996**] develops two concrete algorithms for disambiguation of grammars based on the idea of excluding a certain set of forbidden sub-parse trees.

Parse tree filtering:

Klint *et al* [**Klint:1997**] propose a framework of filters to describe and compare a wide range of disambiguation problems in a parser-independent way. A filter is a function that selects from a set of parse trees the intended trees.

List of Figures

List of Tables