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 Questions:

- How long? Min/Max?
- How to define Earley inductively?
- Nicer notation for all the different kinds of derivations?
- Fix bibliography???

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 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.

2.3 Structure

2.4 Contributions

SNIPPET:

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.

SNIPPET:

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.

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 its 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 that 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 that we are currently considering, a bullet signalling how much of the production's right-hand side we have recognized so far, an origin i describing the position in ω where we started scanning, and an end j indicating the position in 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 considering 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-hand 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 symbol 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-hand side 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-hand 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 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 corresponds to recognizing $(x + x) + x$. We would 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. Soundness: for every generated item there exists an according derivation:
   $A \rightarrow \alpha \bullet \beta, i, j \in B$ implies $A \overset{*}{\Rightarrow} \omega[i..j)\beta$
2. Completeness: for every derivation we generate an according item:
   $A \overset{*}{\Rightarrow} \omega[i..j)\beta$ implies $A \rightarrow \alpha \bullet \beta, i, j \in B$
3. Finiteness: there only exist a finite number of Earley items

Init

$$\frac{}{\text{S} \rightarrow \bullet\alpha\text{, 0, 0}}$$

Scan

$$\frac{\text{A} \rightarrow \alpha\bullet\text{a } \beta\text{, i, j} \qquad \omega\text{[j] = a}}{\text{A} \rightarrow \alpha\text{a } \bullet\beta\text{, i, j+1}}$$

Predict

$$\frac{\text{A} \rightarrow \alpha\bullet\text{B } \beta\text{, i, j} \qquad (\text{B} \rightarrow \gamma) \in \mathcal{G}}{\text{B} \rightarrow \bullet\gamma\text{, j, j}}$$

Complete

$$\frac{\text{A} \rightarrow \alpha\bullet\text{B } \beta\text{, i, j} \qquad \text{B} \rightarrow \gamma\bullet\text{, j, k}}{\text{A} \rightarrow \alpha\text{B } \bullet\beta\text{, i, k}}$$

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

In this chapter we shortly introduce the interactive theorem prover Isabelle/HOL [**Nipkow:2002**] used as the tool for verification in this thesis and recap some of the formalism of context-free grammars and their representation in Isabelle. Finally we formalize the simplified Earley recognizer algorithm presented in Chapter 3; discussing the implementation and the proofs for soundness, completeness, and finiteness. Note that most of the basic definitions of Sections 4.1 and 4.2 are not our own work but only slightly adapted from [**Obua:2017**] [**LocalLexing-AFP**]. All of the proofs in this chapter are our own work.

4.1 Context-free grammars and Isabelle/HOL

Isabelle/HOL [**Nipkow:2002**] is an interactive theorem prover based on a fragment of higher-order logic. It supports the core concepts commonly known from functional programming languages. The notation $t :: \tau$ means that term t has type τ. Basic types include *bool*, *nat*; type variables are written $'a$, $'b$, etc. Pairs are written (a, b); triples are written (a, b, c) and so forth but are internally represented as nested pairs; the nesting is on the first component of a pair. Functions *fst* and *snd* return the first and second component of a pair; the operator $(\times)$ represents pairs at the type level. Most type constructors are written postfix, e.g. $'a\ set$ and $'a\ list$; the function space arrow is $\Rightarrow$; function *set* converts a list into a set. Type synonyms are introduced via the *type_synonym* command. Algebraic data types are defined with the keyword *datatype*. Non-recursive definitions are introduced with the *definition* keyword.

It is standard to define a language as a set of strings over a finite set of symbols. We deviate slightly by introducing a type variable $'a$ for the type of symbols. Thus a string corresponds to a list of symbols and a language is formalized as a set of lists of symbols. We represent a context-free grammar as the datatype *CFG*. An instance *cfg* consists of (1) a list of non-terminals ($\mathfrak{N}$ *cfg*), (2) a list of terminals ($\mathfrak{T}$ *cfg*), (3) a list of production rules ($\mathfrak{R}$ *cfg*), and a start symbol ($\mathfrak{S}$ *cfg*) where $\mathfrak{N}$, $\mathfrak{T}$, $\mathfrak{R}$ and $\mathfrak{S}$ are projections accessing the specific part of an instance *cfg* of the datatype *CFG*. Each rule consists of a left-hand side or *rule-head*, a single symbol, and a right-hand side or *rule-body*, a list of symbols. The productions with a particular non-terminal N on their left-hand sides are called the alternatives of N. We make the usual assumptions about the well-formedness of a context-free grammar: the intersection of the set of terminals and the set of non-terminals is empty; the start symbol is a non-terminal; the rule head of a production is a non-terminal and its rule body consists of only symbols. Additionally, since we are working with a list of productions, we make the assumption that this list is distinct.

type-synonym $'a\ rule = 'a \times 'a\ list$
type-synonym $'a\ rules = 'a\ rule\ list$

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

definition $rule\text{-}head :: 'a\ rule \Rightarrow 'a$ **where**
  $rule\text{-}head = fst$

definition $rule\text{-}body :: 'a\ rule \Rightarrow 'a\ list$ **where**
  $rule\text{-}body = snd$

definition $disjunct\text{-}symbols :: 'a\ cfg \Rightarrow bool$ **where**
  $disjunct\text{-}symbols\ cfg \equiv set\ (\mathfrak{N}\ cfg) \cap set\ (\mathfrak{T}\ cfg) = \{\}$

definition $wf\text{-}startsymbol :: 'a\ cfg \Rightarrow bool$ **where**
  $wf\text{-}startsymbol\ cfg \equiv \mathfrak{S}\ cfg \in set\ (\mathfrak{N}\ cfg)$

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

definition $distinct\text{-}rules :: 'a\ cfg \Rightarrow bool$ **where**
  $distinct\text{-}rules\ cfg \equiv distinct\ (\mathfrak{R}\ cfg)$

definition $wf\text{-}cfg :: 'a\ cfg \Rightarrow bool$ **where**
  $wf\text{-}cfg\ cfg \equiv disjunct\text{-}symbols\ cfg \land wf\text{-}startsymbol\ cfg \land wf\text{-}rules\ cfg \land distinct\text{-}rules\ cfg$

Furthermore, in Isabelle, lists are constructed from the empty list [] via the infix cons-operator (#); the operator (@) appends two lists; $xs\ !\ n$ returns the n-th item of the list xs. Sets follow the standard mathematical notation including the commonly found set builder notation or set comprehensions $\{x \mid P\ x\}$. Sets can also be defined inductively using the keyword *inductive_set*.

Next we formalize the concept of a derivation. We use lowercase letters a, b, c indicating terminal symbols; capital letters A, B, C denote non-terminals; lists of symbols are represented by greek letters: α, β, γ, occasionally also by lowercase letters u, v, w. The empty list in the context of a language is ϵ. A sentential is a list consisting of only symbols. A sentence is a sentential if it only contains terminal symbols. We first define a predicate *derives1 cfg u v* which expresses that we can derive v from u in a single step or the predicate holds if there exist α, β, N and γ such that $u = \alpha$ @ $[N]$ @ β, $v = \alpha$ @ γ @ β and (N, γ) is a production rule. We then can define the set of single-step derivations using *derives1*, and subsequently the set of all derivations given a particular grammar is the reflexive-transitive closure of the set of single-step derivations. Finally, we say v can be derived from u given a grammar *cfg*, or *derives cfg u v* if $(u, v) \in$ *derivations cfg*.

type-synonym *'a sentential* = *'a list*

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

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

definition *is-symbol* :: *'a cfg* ⇒ *'a* ⇒ *bool* **where**
is-symbol cfg s ≡ *is-terminal cfg s* ∨ *is-nonterminal cfg s*

definition *wf-sentential* :: *'a cfg* ⇒ *'a sentential* ⇒ *bool* **where**
wf-sentential cfg s ≡ ∀ *x* ∈ *set s*. *is-symbol cfg x*

definition *is-sentence* :: *'a cfg* ⇒ *'a sentential* ⇒ *bool* **where**
is-sentence cfg s ≡ ∀ *x* ∈ *set s*. *is-terminal cfg x*

definition *derives1* :: *'a cfg* ⇒ *'a sentential* ⇒ *'a sentential* ⇒ *bool* **where**
derives1 cfg u v ≡
  ∃ α β *N* γ.
    u = α @ [*N*] @ β
   ∧ *v* = α @ γ @ β
   ∧ (*N*, γ) ∈ *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*

Potentially recursive but provably total functions that may make use of pattern matching are defined with the *fun* and *function* keywords; partial functions are defined via *partial_function*. Take for example the function *slice* defined below. Term *slice i j xs* computes the slice of a list *xs* between indices *i* (inclusive) and *j* (exclusive), e.g. *slice 2 4* [*a*, *b*, *c*, *d*, *e*] evaluates to [*c*, *d*].

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*

Lemmas, theorems and corollaries are presented using the keywords *lemma*, *theorem*, *corollary* respectively, followed by their names. They consist of zero or more assumptions marked by *assumes* keywords and one conclusion indicated by *shows*. E.g. we can proof a simple lemma about the interaction between the *slice* function and the append operator (@), stating the conditions under which we can split one slice into two.

lemma *slice-append*:
assumes $i \leq j$ $j \leq k$
shows *slice i j xs* @ *slice j k xs* = *slice i k xs*

4.2 The Algorithm

Next we formalize the algorithm presented in Chapter 3. First we define the datatype *item* representing Earley items. For example, the item $S \rightarrow S + \bullet S, 2, 4$ consists of four parts: a production rule (*item-rule*), a natural number (*item-bullet*) indicating the position of the bullet in the production rule, and two natural numbers (*item-origin* inclusive, *item-end* exclusive) representing the portion of the input string ω that has been scanned by the item. Additionally we introduce a few useful abbreviations: the functions *item-rule-head* and *item-rule-body* access the *rule-head* respectively *rule-body* of an item. Functions *item-*α and *item-*β split the production rule body at the bullet, e.g. $S \rightarrow \alpha \bullet \beta$. We call an item *complete* if the bullet is at the end of the production rule body. The next symbol (*next-symbol*) of an item is either *None* if it is complete, or *Some s* where *s* is the symbol in the production rule body following the bullet. An item is finished if the item rule head is the start symbol, the item is complete, and the whole input has been scanned or *item-origin item* = *0* and *item-end item* = $|\omega|$. Finally, we call a set of items *recognizing* if it contains at least one finished item, indicating that this set of items recognizes the input ω.

datatype *'a item* =
Item

$(item\text{-}rule\text{: } 'a\ rule)$
$(item\text{-}bullet : nat)$
$(item\text{-}origin : nat)$
$(item\text{-}end : nat)$

type-synonym $'a\ items = 'a\ item\ set$

definition $item\text{-}rule\text{-}head :: 'a\ item \Rightarrow 'a$ **where**
$item\text{-}rule\text{-}head\ x = rule\text{-}head\ (item\text{-}rule\ x)$

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

definition $item\text{-}\alpha :: 'a\ item \Rightarrow 'a\ sentential$ **where**
$item\text{-}\alpha\ x = take\ (item\text{-}bullet\ x)\ (item\text{-}rule\text{-}body\ x)$

definition $item\text{-}\beta :: 'a\ item \Rightarrow 'a\ sentential$ **where**
$item\text{-}\beta\ x = drop\ (item\text{-}bullet\ x)\ (item\text{-}rule\text{-}body\ x)$

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

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

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

definition $recognizing :: 'a\ items \Rightarrow 'a\ cfg \Rightarrow 'a\ sentential \Rightarrow bool$ **where**
$recognizing\ I\ cfg\ \omega \equiv \exists\, x \in I.\ is\text{-}finished\ cfg\ \omega\ x$

Normally we don't construct items directly via the *Item* constructor but use two auxiliary constructors: the function *init-item* is used by the *Init* and *Predict* operations. It sets the *item-bullet* to 0 or the beginning of the production rule body, initializes the *item-rule*, and indicates that this is an initial item by assigning *item-origin* and *item-end* to the current position in the input. The function *inc-item* returns a new item, moving the bullet over the next symbol (assuming there is one), and setting the *item-end* to the current position in the input, leaving the item rule and origin untouched. It is utilized by the *Scan* and *Complete* operations.

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

definition *inc-item* :: *'a item* ⇒ *nat* ⇒ *'a item* **where**
inc-item x k = *Item* (*item-rule x*) (*item-bullet x* + *1*) (*item-origin x*) *k*

There are different approaches of defining the set of Earley items in accordance with the rules of Figure 3.1. We can take an abstract approach and define the set inductively using Isabelle's inductive sets, or a more operational point of view. We take the latter approach and discuss the reasoning for this decision end the end of this section.

Note that, as mentioned previously, even though we are only constructing one set of Earley items, conceptually all items with the same item end form one Earley bin. Our operational approach is then the following: we generate Earley items bin by bin in ascending order, starting from the 0-th bin which contains all initial items computed by the *Init* operation. The three operations *Scan*, *Predict*, and *Complete* all take as arguments the index of the current bin and the current set of Earley items. For the k-th bin the *Scan* operation initializes the $k+1$-th bin, whereas the *Predict* and *Complete* operations only generate items belonging to the k-th bin. We then define a function *Earley-step* (short for Earley step) that returns the union of the set itself and applying the three operations to a set of Earley items. We complete the k-th bin and initialize the $k+1$-th bin by iterating *Earley-step* until the set of items stabilizes, captured by the *Earley-bin* definition. The function *Earley* then generates the bins up to the n-th bin by applying the *Earley-bin* function first to the initial set of items *Init* and continuing in ascending order bin by bin. Finally, we compute the set of Earley items by applying *Earley* to the length of the input.

definition *bin* :: *'a items* ⇒ *nat* ⇒ *'a items* **where**
bin I k = { *x* . *x* ∈ *I* ∧ *item-end x* = *k* }

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

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

$\quad r \in \mathit{set}\ (\mathfrak{R}\ \mathit{cfg}) \wedge$
$\quad x \in \mathit{bin}\ I\ k \wedge$
$\quad \mathit{next\text{-}symbol}\ x = \mathit{Some}\ (\mathit{rule\text{-}head}\ r)\ \}$

definition $\mathit{Complete} :: \mathit{nat} \Rightarrow {}'a\ \mathit{items} \Rightarrow {}'a\ \mathit{items}$ **where**
$\mathit{Complete}\ k\ I =$
$\{\ \mathit{inc\text{-}item}\ x\ k \mid x\ y.$
$\quad x \in \mathit{bin}\ I\ (\mathit{item\text{-}origin}\ y) \wedge$
$\quad y \in \mathit{bin}\ I\ k \wedge$
$\quad \mathit{is\text{-}complete}\ y \wedge$
$\quad \mathit{next\text{-}symbol}\ x = \mathit{Some}\ (\mathit{item\text{-}rule\text{-}head}\ y)\ \}$

definition $\mathit{Earley\text{-}step} :: \mathit{nat} \Rightarrow {}'a\ \mathit{cfg} \Rightarrow {}'a\ \mathit{sentential} \Rightarrow {}'a\ \mathit{items} \Rightarrow {}'a\ \mathit{items}$ **where**
$\mathit{Earley\text{-}step}\ k\ \mathit{cfg}\ \omega\ I = I \cup \mathit{Scan}\ k\ \omega\ I \cup \mathit{Complete}\ k\ I \cup \mathit{Predict}\ k\ \mathit{cfg}\ I$

fun $\mathit{funpower} :: ({}'a \Rightarrow {}'a) \Rightarrow \mathit{nat} \Rightarrow ({}'a \Rightarrow {}'a)$ **where**
$\mathit{funpower}\ f\ 0\ x = x$
$\mid \mathit{funpower}\ f\ (\mathit{Suc}\ n)\ x = f\ (\mathit{funpower}\ f\ n\ x)$

definition $\mathit{natUnion} :: (\mathit{nat} \Rightarrow {}'a\ \mathit{set}) \Rightarrow {}'a\ \mathit{set}$ **where**
$\mathit{natUnion}\ f = \bigcup\ \{\ f\ n \mid n.\ \mathit{True}\ \}$

definition $\mathit{limit} :: ({}'a\ \mathit{set} \Rightarrow {}'a\ \mathit{set}) \Rightarrow {}'a\ \mathit{set} \Rightarrow {}'a\ \mathit{set}$ **where**
$\mathit{limit}\ f\ x = \mathit{natUnion}\ (\lambda\ n.\ \mathit{funpower}\ f\ n\ x)$

definition $\mathit{Earley\text{-}bin} :: \mathit{nat} \Rightarrow {}'a\ \mathit{cfg} \Rightarrow {}'a\ \mathit{sentential} \Rightarrow {}'a\ \mathit{items} \Rightarrow {}'a\ \mathit{items}$ **where**
$\mathit{Earley\text{-}bin}\ k\ \mathit{cfg}\ \omega\ I = \mathit{limit}\ (\mathit{Earley\text{-}step}\ k\ \mathit{cfg}\ \omega)\ I$

fun $\mathit{Earley} :: \mathit{nat} \Rightarrow {}'a\ \mathit{cfg} \Rightarrow {}'a\ \mathit{sentential} \Rightarrow {}'a\ \mathit{items}$ **where**
$\mathit{Earley}\ 0\ \mathit{cfg}\ \omega = \mathit{Earley\text{-}bin}\ 0\ \mathit{cfg}\ \omega\ (\mathit{Init}\ \mathit{cfg})$
$\mid \mathit{Earley}\ (\mathit{Suc}\ n)\ \mathit{cfg}\ \omega = \mathit{Earley\text{-}bin}\ (\mathit{Suc}\ n)\ \mathit{cfg}\ \omega\ (\mathit{Earley}\ n\ \mathit{cfg}\ \omega)$

definition $\mathcal{E}\mathit{arley} :: {}'a\ \mathit{cfg} \Rightarrow {}'a\ \mathit{sentential} \Rightarrow {}'a\ \mathit{items}$ **where**
$\mathcal{E}\mathit{arley}\ \mathit{cfg}\ \omega = \mathit{Earley}\ (\mathit{length}\ \omega)\ \mathit{cfg}\ \omega$

We follow the operational approach of defining the set of Earley items primarily for two reasons: first of all, we reuse and only slightly adapt most of the basic definitions of this chapter from the work of Obua on *Local Lexing* **[Obua:2017]** **[LocalLexing-AFP]**, which takes the more operational approach and already defines useful lemmas, for example on function iteration. Secondly, the operational approach maps more easily to the list-based implementation of the next chapter that necessarily takes an ordered approach to generating Earley items. Nonetheless, in hindsight, defining the set of Earley items inductively seems to be not only the more elegant approach but also might simplify some of the proofs of this chapter, and is consequently future work worth considering.

4.3 Well-formedness

Due to the operational view of generating the set of Earley items, the proofs of, not only, well-formedness, but also soundness and completeness follow a similar structure: we first proof a property about the basic building blocks, the *Init*, *Scan*, *Predict*, and *Complete* operations. Then, we proof that this property is maintained iterating the function *Earley-step*, and thus holds for the *Earley-bin* operation. Finally, we show that the function *Earley* maintains this property for all conceptual bins and thus for the $\mathcal{E}arley$ definition, or the set of Earley items.

Before we start to proof soundness and completeness of the generated set of Earley items, especially the completeness proof is more involved, we highlight the general proof structure once in detail, for a simpler property: well-formedness of the items, allowing us to concentrate only on the core aspects for the soundness and completeness proofs.

An Earley item is well-formed (*wf-item*) if the item rule is a rule of the grammar; the item bullet is bounded by the length of the item rule body; the item origin does not exceed the item end, and finally the item end is at most the length of the input.

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

definition *wf-items* :: $'a\ cfg \Rightarrow\ 'a\ sentential \Rightarrow\ 'a\ items \Rightarrow bool$ **where**
 $wf\text{-}items\ cfg\ \omega\ I \equiv \forall\, x \in I.\ wf\text{-}item\ cfg\ \omega\ x$

lemma *wf-Init*:
 shows $wf\text{-}items\ cfg\ \omega\ (Init\ cfg)$

lemma *wf-Scan-Predict-Complete*:
 assumes $wf\text{-}items\ cfg\ \omega\ I$
 shows $wf\text{-}items\ cfg\ \omega\ (Scan\ k\ \omega\ I \cup Predict\ k\ cfg\ I \cup Complete\ k\ I)$

lemma *wf-Earley-step*:
 assumes $wf\text{-}items\ cfg\ \omega\ I$
 shows $wf\text{-}items\ cfg\ \omega\ (Earley\text{-}step\ k\ cfg\ \omega\ I)$

Lemmas *wf-Init*, *wf-Scan-Predict-Complete*, and *wf-Earley-step* follow trivially by definition of the respective operations.

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

lemma *wf-Earley-bin*:
assumes *wf-items cfg* ω *I*
shows *wf-items cfg* ω (*Earley-bin k cfg* ω *I*)

lemma *wf-Earley-bin0*:
shows *wf-items cfg* ω (*Earley-bin 0 cfg* ω (*Init cfg*))

We proof the lemma *wf-funpower* by induction on n using lemma *wf-Earley-step*, and lemmas *wf-Earley-bin* and *wf-Earley-bin0* follow immediately using additionally the fact that $x \in \mathit{limit}\ f\ X \equiv \exists\, n.\ x \in \mathit{funpower}\ f\ n\ X$ and lemma *wf-Init*.

lemma *wf-Earley*:
shows *wf-items cfg* ω (*Earley n cfg* ω)

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

Finally, lemma *wf-Earley* is proved by induction on n using lemmas *wf-Earley-bin* and *wf-Earley-bin0*; lemma *wf-$\mathcal{E}$arley* follows by definition of $\mathcal{E}$*arley*.

4.4 Soundness

Next, we proof the soundness of the generated items, or: $A \rightarrow \alpha \bullet \beta, i, j \in B$ implies $A \overset{*}{\Rightarrow} \omega[i..j)\beta$ which is stated in terms of our formalization by the *sound-item* definition below. As mentioned previously, the general proof structure follows the proof for well-formedness. Thus, we only highlight one slightly more involved lemma stating the soundness of the *Complete* operation while stating the remaining lemmas without explicit proof. Additionally, proving lemma *sound-Complete* provides some insight into working with and proving properties about derivations.

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

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

Obua [**Obua:2017**] [**LocalLexing-AFP**] defines derivations at two different abstraction levels. The first representation is as the reflexive-transitive closure of the set of one-step derivations as introduced earlier in this chapter. The second representation is again more operational. He defines a predicate *Derives1 cfg u i r v* that is conceptually analogous to the predicate *derives1 cfg u v* but also captures the rule *r* used for a single rewriting step and the position *i* in *u* where the rewriting occurs.

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

He then defines the type of a *derivation* as a list of pairs representing precisely the positions and rules used to apply each rewrite step. The predicate *Derivation* is defined recursively as follows: *Derivation* α [] β holds only if $\alpha = \beta$. If the derivation consists of at least one rewrite pair (i, r), or *Derivation cfg* α $((i, r)\ \#\ D)$ β, then there must exist a γ such that *Derives1 cfg* α *i r* γ and *Derivation cfg* γ *D* β. Obua then proves that both notions of a derivation are equivalent (lemma *derives-equiv-Derivation*)

type-synonym *'a derivation* = (*nat* × *'a rule*) *list*

fun *Derivation* :: *'a cfg* ⇒ *'a sentential* ⇒ *'a derivation* ⇒ *'a sentential* ⇒ *bool* **where**
Derivation - α [] $\beta = (\alpha = \beta)$
| *Derivation cfg* α $(d\#D)$ $\beta = (\exists\,\gamma.$ *Derives1 cfg* α (*fst d*) (*snd d*) $\gamma \wedge$ *Derivation cfg* γ *D* $\beta)$

lemma *derives-equiv-Derivation*:
shows *derives cfg* α $\beta \equiv \exists\ D.$ *Derivation cfg* α *D* β

Next, we state a small but useful lemma about rewriting derivations using the more operational definition of derivations defined above without explicit proof.

lemma *Derivation-append-rewrite*:
assumes *Derivation cfg* α *D* $(\beta\ @\ \gamma\ @\ \delta)$
assumes *Derivation cfg* γ *E* γ'
shows $\exists F.$ *Derivation cfg* α *F* $(\beta\ @\ \gamma'\ @\ \delta)$

And finally, we proof soundness of the *Complete* operation:

lemma *sound-Complete*:
assumes *wf*: *wf-items cfg* ω *I*
assumes *sound*: *sound-items cfg* ω *I*
shows *sound-items cfg* ω (*Complete k I*)

Proof. Let z denote an arbitrary but fixed item of *Complete k I*. By the definition of the *Complete* operation there exist items x and y such that: $x \in$ *bin I* (*item-origin y*), $y \in$ *bin I k*, *is-complete y*, *next-symbol x* = *Some* (*item-rule-head y*), and z = *inc-item x k*.

Since y is in bin k, it is complete and the set I is sound (assumption *sound*), there exists a derivation E such that

$$\textit{Derivation cfg } [\textit{item-rule-head } y] \; E \; (\textit{slice } (\textit{item-origin } y) \; (\textit{item-end } y) \; \omega)$$

by lemma *derives-equiv-Derivation*. Similarly, since x is in bin *item-origin y* and due to assumption *sound*, there exists a derivation D such that

$$\textit{Derivation cfg } [\textit{item-rule-head } x] \; D \; (\textit{slice } (\textit{item-origin } x) \; (\textit{item-origin } y) \; \omega \; @ \; \textit{item-}\beta \; x)$$

Note that *item-β x* = *item-rule-head y # tl* (*item-β x*) since the next symbol of x is equal to the item rule head of y. Thus, by lemma *Derivation-append-rewrite*, and the definition of D and E, there exists a derivation F such that

$$\begin{aligned} &\textit{Derivation cfg } [\textit{item-rule-head } x] \; F \; (\textit{slice } (\textit{item-origin } x) \; (\textit{item-origin } y) \; \omega) \; @ \\ &\quad \textit{slice } (\textit{item-origin } y) \; (\textit{item-end } y) \; \omega \; @ \; \textit{tl } (\textit{item-}\beta \; x) \end{aligned}$$

Additionally, we know that x and y are well-formed items due to the facts that x is in bin *item-origin y*, y is in bin k, and the assumption *wf-items cfg ω I*. Thus, we can discharge the assumptions of lemma *slice-append* (*item-origin x* $\leq$ *item-origin y* and *item-origin y* $\leq$ *item-end y*) and have

$$\textit{Derivation cfg } [\textit{item-rule-head } x] \; F \; (\textit{slice } (\textit{item-origin } x) \; (\textit{item-end } y) \; \omega \; @ \; \textit{tl } (\textit{item-}\beta \; x))$$

Moreover, since z = *inc-item x k* and the next symbol of x is the item rule head of y, it follows that *tl* (*item-β x*) = *item-β z*, and ultimately *sound-item cfg ω z*, again by the definition of z and lemma *derives-equiv-Derivation*.

□

lemma *sound-Init*:
 shows *sound-items cfg ω* (*Init cfg*)

lemma *sound-Scan*:
 assumes *wf-items cfg ω I*
 assumes *sound-items cfg ω I*
 shows *sound-items cfg ω* (*Scan k ω I*)

lemma *sound-Predict*:

assumes *sound-items cfg* ω *I*
shows *sound-items cfg* ω (*Predict k cfg I*)

lemma *sound-Earley-step*:
assumes *wf-items cfg* ω *I*
assumes *sound-items cfg* ω *I*
shows *sound-items cfg* ω (*Earley-step k cfg* ω *I*)

lemma *sound-funpower*:
assumes *wf-items cfg* ω *I*
assumes *sound-items cfg* ω *I*
shows *sound-items cfg* ω (*funpower* (*Earley-step k cfg* ω) *n I*)

lemma *sound-Earley-bin*:
assumes *wf-items cfg* ω *I*
assumes *sound-items cfg* ω *I*
shows *sound-items cfg* ω (*Earley-bin k cfg* ω *I*)

lemma *sound-Earley-bin0*:
shows *sound-items cfg* ω (*Earley-bin 0 cfg* ω (*Init cfg*))

lemma *sound-Earley*:
shows *sound-items cfg* ω (*Earley k cfg* ω)

lemma *sound-$\mathcal{E}$arley*:
shows *sound-items cfg* ω ($\mathcal{E}$*arley cfg* ω)

Finally, using *sound-$\mathcal{E}$arley* and the definitions of *sound-item*, *recognizing*, *is-finished* and *is-complete* the final theorem follows: if the generated set of Earley items is *recognizing*, or contains a *finished* item, then there exists a derivation of the input ω from the start symbol of the grammar.

theorem *soundness*:
assumes *recognizing* ($\mathcal{E}$*arley cfg* ω) *cfg* ω
shows *derives cfg* [$\mathfrak{S}$ *cfg*] ω

4.5 Completeness

Next, we prove completeness and consequently obtain a concluded correctness proof using theorem *soundness*. The completeness proof is by far the most involved proof of this chapter. Thus, we present it in greater detail, and also slightly deviate from the proof structure of the well-formedness and soundness proofs presented previously. We directly start to prove three properties of the *Earley* function that correspond conceptually to the three different operations that can occur while generating the bins.

We need three simple lemmas concerning the *Earley-bin* function, stated without explicit proof: (1) *Earley-bin k cfg ω I* only (potentially) changes bins k and $k+1$ (lemma *Earley-bin-bin-idem*); (2) the *Earley-step* operation is subsumed by the *Earley-bin* operation, since it computes the limit of *Earley-step* (lemma *Earley-step-sub-Earley-bin*); and (3) the function *Earley-bin* is idempotent (lemma *Earley-bin-idem*).

lemma *Earley-bin-bin-idem*:
 assumes $i \neq k$
 assumes $i \neq k{+}1$
 shows $bin\ (Earley\text{-}bin\ k\ cfg\ \omega\ I)\ i = bin\ I\ i$

lemma *Earley-step-sub-Earley-bin*:
 shows $Earley\text{-}step\ k\ cfg\ \omega\ I \subseteq Earley\text{-}bin\ k\ cfg\ inp\ I$

lemma *Earley-bin-idem*:
 shows $Earley\text{-}bin\ k\ cfg\ \omega\ (Earley\text{-}bin\ k\ cfg\ \omega\ I) = Earley\text{-}bin\ k\ cfg\ \omega\ I$

Next, we proof lemma *Scan-Earley* in detail: it describes under which assumptions the function *Earley* generates a 'scanned' item:

lemma *Scan-Earley*:
 assumes $i{+}1 \leq k$
 assumes $x \in bin\ (Earley\ k\ cfg\ \omega)\ i$
 assumes $next\text{-}symbol\ x = Some\ a$
 assumes $k \leq length\ \omega$
 assumes $\omega!i = a$
 shows $inc\text{-}item\ x\ (i{+}1) \in Earley\ k\ cfg\ \omega$

Proof. The proof is by induction in k for arbitrary i, x, and a:

The base case $k = 0$ is trivial, since we have the assumption $i + 1 \leq 0$.

For the induction step we can assume $i + 1 \leq k + 1$, $k + 1 \leq |\omega|$, $x \in bin\ (Earley\ (k + 1)\ cfg\ \omega)\ i$, $next\text{-}symbol\ x = Some\ a$, and $\omega\ !\ i = a$. Assumptions $x \in bin\ (Earley\ (k + 1)\ cfg\ \omega)\ i$ and $i + 1 \leq k + 1$ imply that $x \in bin\ (Earley\ k\ cfg\ inp)\ i$ by lemma *Earley-bin-bin-idem*.

We then consider two cases:

- $i + 1 \leq k$: We can apply the induction hypothesis using assumptions $k + 1 \leq |\omega|$, $next\text{-}symbol\ x = Some\ a$, $\omega\ !\ i = a$ and additionally $x \in bin\ (Earley\ k\ cfg\ inp)\ i$ and have $inc\text{-}item\ x\ (i + 1) \in Earley\ k\ cfg\ \omega$. The statement to proof follows by lemma *Earley-step-sub-Earley-bin* and the definition of *Earley-step*.

- $k < i + 1$: We have $i = k$, since $i + 1 \leq k + 1$. Thus, we have $inc\text{-}item\ x\ (i + 1) \in Scan\ k\ \omega\ (Earley\ k\ cfg\ \omega)$ using assumptions $k + 1 \leq |\omega|$, $next\text{-}symbol\ x = Some\ a$, $\omega\ !\ i = a$, and additionally $x \in bin\ (Earley\ k\ cfg\ inp)\ i$ by the definition of the *Scan* operation. This in turn implies *inc-item* x $(i + 1) \in$ *Earley-step* k *cfg* ω (*Earley* k *cfg* ω) by lemma *Earley-step-sub-Earley-bin* and the definition of *Earley-step*. Since the function *Earley-bin* is idempotent (lemma *Earley-bin-idem*), we have *inc-item* x $(i + 1) \in$ *Earley* k *cfg* ω by definition of *Earley*. And again, the final statement follows by lemma *Earley-step-sub-Earley-bin* and the definition of *Earley-step*.

□

lemma *Predict-Earley*:
assumes $i \leq k$
assumes $x \in$ *bin* (*Earley* k *cfg* ω) i
assumes *next-symbol* x = *Some* N
assumes $(N,\alpha) \in$ *set* ($\mathfrak{R}$ *cfg*)
shows *init-item* (N,α) $i \in$ *Earley* k *cfg* ω

lemma *Complete-Earley*:
assumes $i \leq j$
assumes $j \leq k$
assumes $x \in$ *bin* (*Earley* k *cfg* ω) i
assumes *next-symbol* x = *Some* N
assumes $(N,\alpha) \in$ *set* ($\mathfrak{R}$ *cfg*)
assumes $y \in$ *bin* (*Earley* k *cfg* ω) j
assumes *item-rule* $y = (N,\alpha)$
assumes i = *item-origin* y
assumes *is-complete* y
shows *inc-item* x $j \in$ *Earley* k *cfg* ω

The proof of lemmas *Predict-Earley* and *Complete-Earley* are similar in structure to the proof of lemma *Scan-Earley* with the exception of the base case that is in both cases non-trivial but can be proven with the help of lemmas *Earley-step-sub-Earley-bin* and *Earley-bin-idem*, the definition of *Earley-bin* and the definitions of *Predict* and *Complete*, respectively.

Next, we give some intuition about the core idea of the completeness proof. Assume there exists an item $N \to \bullet A_0 A_1 \ldots A_n$ in a *complete* (we define what exactly this means) set of items I where A_i are either terminal or non-terminal symbols. Furthermore, assume there exist the following derivations for $i_0 \leq i_1 \leq \cdots \leq i_n \leq i_{n+1}$:

$$
\begin{aligned}
&A_0 \xRightarrow{*} \omega[i_0..i_1) \\
&A_1 \xRightarrow{*} \omega[i_1..i_2) \\
&\ldots \\
&A_n \xRightarrow{*} \omega[i_n..i_{n+1})
\end{aligned}
$$

Then, we have one derivation to move the bullet over each terminal or non-terminal A_i then the item $N \rightarrow A_0 A_1 \ldots A_n \bullet$ should be in I if I is a *complete* set of items.

We formalize this idea as follows: a set I is *partially-completed* if for each non-complete item x in I, the existence of a derivation D from the next symbol of x implies, that the item that can be obtained by moving the bullet over the next symbol of x, is also present in I. The full definition of *partially-completed* below is slightly more involved since we need to keep track of the validity of the indices. Note that the definition also requires that an arbitrary predicate P holds for the derivation D. This predicate is necessary since the completeness proof requires a proof on the length of the derivation D, and thus we limit the *partially-completed* property to derivations that don't exceed a certain length.

Lemma *partially-completed-upto* then formalizes the core idea: if $N \rightarrow \alpha \bullet \beta, i, j$ in a set of items I and there exists a derivation $\beta \overset{D}{\Rightarrow} \omega[j..k)$, then I also contains the complete item $N \rightarrow \alpha\beta\bullet, i, k$. Note that this holds only if $j \leq k$, $k \leq |\omega|$, all items of I are well-formed and most importantly I must be *partially-completed* up to the length of the derivation D.

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 ω *I P* $\equiv$
  $\forall$ *i j x a D.*
   $i \leq j \wedge j \leq k \wedge k \leq$ *length* ω $\wedge$
   $x \in$ *bin I i* $\wedge$ *next-symbol x = Some a* $\wedge$
   Derivation cfg [*a*] *D* (*slice i j* ω) $\wedge$ *P D* $\longrightarrow$
   inc-item x j $\in$ *I*

To proof lemma *partially-completed-upto*, we need two auxiliary lemmas: The first one is about splitting derivations (lemma *Derivation-append-split*): a derivation $\alpha\beta \overset{D}{\Rightarrow} \gamma$, can be split into two derivations E and F whose length is bounded by the length of D, and there exist α' and β' such that $\alpha \overset{E}{\Rightarrow} \alpha'$, $\beta \overset{F}{\Rightarrow} \beta'$ and $\gamma = \alpha' @ \beta'$. The proof is by induction on D for arbitrary α and β and quite technical since we need to manipulate the exact indices where each rewriting rule is applied in α and β, and thus we omit it.

The second one is a, in spirit similar, lemma about splitting slices (lemma *slice-append-split*). The proof is straightforward by induction on the computation of the *slice* function, we also omit it, and move on to the proof of lemmas *partially-completed-upto* and *partially-completed-Earley*.

lemma *Derivation-append-split*:
 assumes *Derivation cfg* ($\alpha @ \beta$) *D* γ
 shows $\exists$ *E F* α' β'. *Derivation cfg* α *E* α' $\wedge$ *Derivation cfg* β *F* β' $\wedge$ $\gamma = \alpha' @ \beta'$ $\wedge$
  length E $\leq$ *length D* $\wedge$ *length F* $\leq$ *length D*

lemma *slice-append-split*:
assumes $i \leq k$
assumes *slice i k xs* $=$ *ys* @ *zs*
shows $\exists j$. *ys* $=$ *slice i j xs* $\wedge$ *zs* $=$ *slice j k xs* $\wedge$ $i \leq b \wedge b \leq k$

lemma *partially-completed-upto*:
assumes *wf-items cfg* ω *I*
assumes $j \leq k$
assumes $k \leq$ *length* ω
assumes $x =$ *Item* (N,α) *b i j*
assumes $x \in I$
assumes *Derivation cfg* (*item-*β *x*) *D* (*slice j k* ω)
assumes *partially-completed k cfg* ω *I* ($\lambda D'$. *length* $D' \leq$ *length D*)
shows *Item* (N,α) (*length* α) *i k* $\in I$

Proof. The proof is by induction on (*item-*β *x*) for arbitrary b, i, j, k, N, α, x, and D:

For the base case we have *item-*β $x = []$ and need to show that *Item* (N, α) $|\alpha|$ *i k* $\in I$:

The bullet of x is right before *item-*β *x*, or *item-*α $x = \alpha$. Thus, the value of the bullet must be equal to the length of α, which implies $x =$ *Item* (N, α) $|\alpha|$ *i j*, since x is a well-formed item and *item-*β $x = []$.

We also know that $j = k$: we have *Derivation cfg* (*item-*β *x*) *D* (*slice j k* ω) and *item-*β *x* $= []$ which in turn implies that *slice j k* $\omega = []$, and thus $j = k$.

Hence, the statement follows from the assumption $x \in I$ and the fact that $x =$ *Item* (N, α) $|\alpha|$ *i j*.

For the induction step we have *item-*β $x = a$ # *as* and need to show that *Item* (N, α) $|\alpha|$ *i k* $\in I$:

Using lemmas *Derivation-append-split* and *slice-append-split* there exists an index j' and derivations E and F such that

$$\textit{Derivation cfg } [a] \; E \; (\textit{slice } j \; j' \; \omega) |E| \leq |D|$$
$$\textit{Derivation cfg as } F \; (\textit{slice } j' \; k \; \omega) |F| \leq |D|$$
$$j \leq j' \qquad j' \leq k$$

Using the facts about derivation E, $j \leq j'$, $j' \leq k$ and the assumptions $k \leq |\omega|$, $x =$ *Item* (N, α) *b i j*, $x \in I$, *next-symbol* $x =$ *Some a* (since *item-*β $x = a$ # *as*) and $x \in$ *bin I j*, we have *inc-item x* $j' \in I$ by the assumption *partially-completed k cfg* ω *I* ($\lambda D'$. $|D'| \leq |D|$). Note that *inc-item x* $j' =$ *Item* (N, α) $(b + 1)$ *i* j', which we will from now on refer to as item x'.

From *partially-completed k cfg* ω *I* ($\lambda D'$. $|D'| \leq |D|$) and $|F| \leq |D|$ follows *partially-completed k cfg* ω *I* ($\lambda D'$. $|D'| \leq |F|$). We also have *as* $=$ *item-*β x' and $x' \in I$.

Hence, we can apply the induction hypothesis for x' using additionally the assumptions *wf-items cfg* ω *I*, $k \leq |\omega|$, and the facts about derivation F from above, and have *Item* (N, α) $|\alpha|$ i $k \in I$, what we intended to show.

□

lemma *partially-completed-Earley*:
assumes *wf-cfg cfg*
shows *partially-completed k cfg* ω (*Earley k cfg* ω) (λ-. *True*)

Proof. Let x, i, a, D, and j be arbitrary but fixed.

By definition of *partially-completed* we can assume $i \leq j$, $j \leq k$, $k \leq |\omega|$, $x \in$ *bin* (*Earley k cfg* ω) i, *next-symbol* x = *Some* a, *Derivation cfg* $[a]$ D (*slice* i j ω), and need to show *inc-item* x $j \in$ *Earley k cfg* ω.

We proof this by complete induction on $|D|$ for arbitrary x, i, a, j, and D, and split the proof into two different cases:

- $D = []$: Since *Derivation cfg* $[a]$ D (*slice* i j ω), we have $[a]$ = *slice* i j ω, and consequently ω ! $i = a$ and $j = i + 1$. Now we discharge the assumptions of lemma *Scan-Earley*, using additionally $x \in$ *bin* (*Earley k cfg* ω) i, *next-symbol* x = *Some* a, and $j \leq |\omega|$, and have *inc-item* x $(i + 1) \in$ *Earley k cfg* ω which finishes the proof since $j = i + 1$.

- $D = d \# D'$: Since *Derivation cfg* $[a]$ D (*slice* i j ω), there exists an α such that *Derives1 cfg* $[a]$ (*fst* d) (*snd* d) α and *Derivation cfg* α D' (*slice* i j ω). From the definition of *Derives1* we see that there exists a non-terminal N such that $a = N$, $(N, \alpha) \in$ *set* ($\mathfrak{R}$ *cfg*), *fst* $d = 0$, and *snd* $d = (N, \alpha)$.

  Let y denote *Item* (N, α) 0 i i. Since we have $i \leq k$, $x \in$ *bin* (*Earley k cfg* ω) i, and *next-symbol* x = *Some* a by assumption, we showed that $a = N$ and $(N, \alpha) \in$ *set* ($\mathfrak{R}$ *cfg*), and y is an initial item, we have $y \in$ *Earley k cfg* ω by lemma *Predict-Earley*.

  Next, we use lemma *partially-completed-upto* to show that we the completed version of item y is also present in the j-th bin of *Earley k cfg* ω since we have a derivation $\alpha \overset{D'}{\Rightarrow} \omega[i..j)$, or *Item* (N, α) $|\alpha|$ i $j \in$ *bin* (*Earley k cfg* ω) j: we have $i \leq j$, $j \leq |\omega|$ by assumption; have proven $y \in$ *Earley k cfg* ω; and have *wf-items cfg* ω (*Earley k cfg* ω) by lemma *wf-Earley*. Additionally, we know *Derivation cfg* (*item-*β y) D' (*slice* i j ω) since *Derivation cfg* $[a]$ D' (*slice* i j ω) and $a = N$, by the definition of item y. Finally, we use the induction hypothesis to show *partially-completed k cfg* ω (*Earley k cfg* ω) (λE. $|E| \leq |D'|$), since $|D'| \leq |D|$ by definition of *partially-completed*, using once again all of our assumptions. This in turn implies *partially-completed j cfg* ω (*Earley k cfg* ω) (λE. $|E| \leq |D'|$) since $j \leq k$ by definition of *partially-completed*.

Now we can use lemma *partially-completed-upto*, and the statement follows from the definition of a bin.

Finally, we prove *inc-item x j* $\in$ *Earley k cfg* ω by lemma *Complete-Earley*: once again we have $i \leq j$, $j \leq k$, and $x \in$ *bin* (*Earley k cfg* ω) *i* by assumption. We also know that *next-symbol x* = *Some N*, due to our assumption *next-symbol x* = *Some a* and *a* = *N*. Moreover, we have $(N, \alpha) \in$ *set* ($\mathfrak{R}$ *cfg*) and most importantly *Item* (N, α) $|\alpha|$ *i j* $\in$ *bin* (*Earley k cfg* ω) *j*, which concludes this proof.

□

Lemma *partially-completed-$\mathcal{E}$arley* follows trivially from *partially-completed-Earley* by definition of *$\mathcal{E}$arley*.

lemma *partially-completed-$\mathcal{E}$arley*:
 assumes *wf-cfg cfg*
 shows *partially-completed* (*length* ω) *cfg* ω (*$\mathcal{E}$arley cfg* ω) (λ-. *True*)

And finally, we can proof completeness of Earley's algorithm, obtaining corollary *correctness-$\mathcal{E}$arley* due to lemma *soundness*.

theorem *completeness*:
 assumes *wf-cfg cfg*
 assumes *is-sentence cfg* ω
 assumes *derives cfg* [$\mathfrak{S}$ *cfg*] ω
 shows *recognizing* (*$\mathcal{E}$arley cfg* ω) *cfg* ω

Proof. We know that there exists an α and a derivation *D* such that ($\mathfrak{S}$ *cfg*, α) $\in$ *set* ($\mathfrak{R}$ *cfg*) and *Derivation cfg* α *D* ω, since *derives cfg* [$\mathfrak{S}$ *cfg*] ω. Let *x* denote the item *Item* ($\mathfrak{S}$ *cfg*, α) *0 0 0*. By definition of *x* and the *Init* operation and *$\mathcal{E}$arley* function, and the fact that *Init cfg* $\subseteq$ *Earley k cfg* ω, we have $x \in$ *$\mathcal{E}$arley cfg* ω, moreover we have *partially-completed* $|\omega|$ *cfg* ω (*$\mathcal{E}$arley cfg* ω) (λ-. *True*) using lemma *partially-completed-$\mathcal{E}$arley* and assumption *wf-cfg cfg*, and thus have *Item* ($\mathfrak{S}$ *cfg*, α) $|\alpha|$ *0* $|\omega|$ $\in$ *$\mathcal{E}$arley cfg* ω by lemmas *partially-completed-upto* and *wf-$\mathcal{E}$arley* and the definition of *partially-completed*. The statement *recognizing* (*$\mathcal{E}$arley cfg* ω) *cfg* ω follows immediately by the definition of *recognizing*, *is-finished*, and *is-complete*.

□

corollary *correctness-$\mathcal{E}$arley*:
 assumes *wf-cfg cfg*
 assumes *is-sentence cfg* ω
 shows *recognizing* (*$\mathcal{E}$arley cfg inp*) *cfg* ω $\longleftrightarrow$ *derives cfg* [$\mathfrak{S}$ *cfg*] ω

4.6 Finiteness

At last, we prove that the set of Earley items is finite. In Chapter 5 we are using this result to prove the termination of an executable version of the algorithm.

Since $\mathcal{E}arley\ cfg\ \omega$ only generates well-formed items (lemma *wf-$\mathcal{E}$arley*) it suffices to prove that there only exists a finite number of well-formed items. Define

$$T = set\ (\mathfrak{R}\ cfg) \times \{0..m\} \times \{0..|\omega|\} \times \{0..|\omega|\}$$

where $m = Max\ \{|rule\text{-}body\ r| \mid r \in set\ (\mathfrak{R}\ cfg)\}$. The set T is finite since there exists only a finite number of production rules and $\{x \mid wf\text{-}item\ cfg\ \omega\ x\}$ is a subset of mapping the *Item* constructor over T (strictly speaking we need to first unpack the quadruple).

lemma *finiteness-UNIV-wf-item*:
 shows *finite* $\{\ x \mid x.\ wf\text{-}item\ cfg\ \omega\ x\ \}$

theorem *finiteness*:
 shows *finite* $(\mathcal{E}arley\ cfg\ \omega)$

5 Earley Recognizer Implementation

5.1 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

Table 5.1: 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			$S \to S \bullet + S, 0, 5; (0, 1, 2)$

5.2 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* $\Rightarrow$ *pointer list* **where**
pointers b = *map pointer b*

definition *bins-eq-items* :: $'a$ *bins* $\Rightarrow$ $'a$ *bins* $\Rightarrow$ *bool* **where**
bins-eq-items bs0 bs1 $\longleftrightarrow$ *map items bs0* = *map items bs1*

definition *bins-items* :: $'a$ *bins* $\Rightarrow$ $'a$ *items* **where**
bins-items bs = $\bigcup$ { *set* (*items* (*bs* ! *k*)) | *k*. *k* < *length bs* }

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

definition *bins-items-upto* :: $'a$ *bins* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *items* **where**
bins-items-upto bs k i = $\bigcup$ { *set* (*items* (*bs* ! *l*)) | *l*. *l* < *k* } $\cup$ *bin-items-upto* (*bs* ! *k*) *i*

definition *wf-bin-items* :: $'a$ *cfg* $\Rightarrow$ $'a$ *sentential* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *item list* $\Rightarrow$ *bool* **where**
wf-bin-items cfg inp k xs $\equiv$ $\forall$ *x* $\in$ *set xs*. *wf-item cfg inp x* $\wedge$ *item-end x* = *k*

definition *wf-bin* :: $'a$ *cfg* $\Rightarrow$ $'a$ *sentential* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *bin* $\Rightarrow$ *bool* **where**
wf-bin cfg inp k b $\equiv$ *distinct* (*items b*) $\wedge$ *wf-bin-items cfg inp k* (*items b*)

definition *wf-bins* :: $'a$ *cfg* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *bins* $\Rightarrow$ *bool* **where**
wf-bins cfg inp bs $\equiv$ $\forall$ *k* < *length bs*. *wf-bin cfg inp k* (*bs* ! *k*)

definition *nonempty-derives* :: $'a$ *cfg* $\Rightarrow$ *bool* **where**
nonempty-derives cfg $\equiv$ $\forall$ *N*. *N* $\in$ *set* ($\mathfrak{N}$ *cfg*) $\longrightarrow$ $\neg$ *derives cfg* [*N*] []

definition *Init-list* :: $'a$ *cfg* $\Rightarrow$ $'a$ *sentential* $\Rightarrow$ $'a$ *bins* **where**
Init-list cfg inp $\equiv$
  let rs = *filter* (λ*r*. *rule-head r* = $\mathfrak{S}$ *cfg*) ($\mathfrak{R}$ *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* $\Rightarrow$ $'a$ *sentential* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ *item* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *entry list* **where**
Scan-list k inp a x pre $\equiv$
  if inp!*k* = *a then*
    let x' = *inc-item x* (*k*+*1*) *in*
    [*Entry* x' (*Pre pre*)]
  else []

definition *Predict-list* :: *nat* $\Rightarrow$ $'a$ *cfg* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ *entry list* **where**
Predict-list k cfg X $\equiv$

```
    let rs = filter (λr. rule-head r = X) (ℜ 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 $\mathcal{E}$*-list* :: *nat* $\Rightarrow$ *'a cfg* $\Rightarrow$ *'a sentential* $\Rightarrow$ *'a bins* **where**
$\mathcal{E}$*-list 0 cfg inp* = *E-list 0 cfg inp* (*Init-list cfg inp*)
| $\mathcal{E}$*-list* (*Suc n*) *cfg inp* = *E-list* (*Suc n*) *cfg inp* ($\mathcal{E}$*-list n cfg inp*)

definition *earley-list* :: *'a cfg* $\Rightarrow$ *'a sentential* $\Rightarrow$ *'a bins* **where**
earley-list cfg inp = $\mathcal{E}$*-list* (*length inp*) *cfg inp*

5.3 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 $\forall$ *x* $\in$ *set* (*items es*). *wf-item cfg inp x* $\wedge$ *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 $\forall$ *x* $\in$ *set* (*items es*). *wf-item cfg inp x* $\wedge$ *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* $\in$ *set* (*items* (*bs!k*)) *k* < *length inp next-symbol x* $\neq$ *None*
shows $\forall$ *y* $\in$ *set* (*items* (*Scan-list k inp a x pre*)). *wf-item cfg inp y* $\wedge$ *item-end y* = *k*+1

lemma *wf-bins-Predict-list*:
assumes *wf-bins cfg inp bs k* < *length bs k* $\leq$ *length inp wf-cfg cfg*
shows $\forall$ *y* $\in$ *set* (*items* (*Predict-list k cfg X*)). *wf-item cfg inp y* $\wedge$ *item-end y* = *k*

lemma *wf-bins-Complete-list*:
assumes *wf-bins cfg inp bs k* < *length bs y* $\in$ *set* (*items* (*bs!k*))
shows $\forall$ *x* $\in$ *set* (*items* (*Complete-list k y bs red*)). *wf-item cfg inp x* $\wedge$ *item-end x* = *k*

fun *earley-measure* :: *nat* $\times$ *'a cfg* $\times$ *'a sentential* $\times$ *'a bins* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
earley-measure (*k*, *cfg*, *inp*, *bs*) *i* = *card* { *x* | *x*. *wf-item cfg inp x* $\wedge$ *item-end x* = *k* } − *i*

definition *wf-earley-input* :: (*nat* $\times$ *'a cfg* $\times$ *'a sentential* $\times$ *'a bins*) *set* **where**

$wf\text{-}earley\text{-}input = \{$
$(k, cfg, inp, bs) \mid k\ cfg\ inp\ bs.$
$k \leq length\ inp\ \wedge$
$length\ bs = length\ inp + 1\ \wedge$
$wf\text{-}cfg\ cfg\ \wedge$
$wf\text{-}bins\ cfg\ inp\ bs$
$\}$

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-earley-input*
shows *wf-bins cfg inp* $($*E-list'* *k cfg inp bs i*$)$

lemma *wf-bins-E-list*:
assumes $(k, cfg, inp, bs) \in$ *wf-earley-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*$)$

5.4 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*

5.5 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 $i \leq j$ *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*)

5.6 Main Theorem

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

theorem *recognizing-list-iff-earley-recognized*:
assumes *wf-cfg cfg is-sentence cfg inp nonempty-derives cfg*
shows *recognizing-list* (*earley-list cfg inp*) *cfg inp* $\longleftrightarrow$ *recognizing* (*earley cfg inp*) *cfg inp*

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

SNIPPET:

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

6 Earley Parser Implementation

6.1 Draft

6.2 Pointer lemmas

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

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

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

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

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

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

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

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

lemma *sound-ptrs-bin-upd*:
assumes $\textit{sound-ptrs inp bs}$ $k < \textit{length bs}$ $es = bs!k$ $\textit{distinct}\ (\textit{items es})$
assumes $\textit{sound-null-ptr e}$ $\textit{sound-pre-ptr inp bs k e}$ $\textit{sound-prered-ptr bs k e}$
shows $\textit{sound-ptrs inp}\ (bs[k := \textit{bin-upd e es}])$

lemma *mono-red-ptr-bin-upd*:
assumes $\textit{mono-red-ptr bs}$ $k < \textit{length bs}$ $es = bs!k$ $\textit{distinct}\ (\textit{items es})$
assumes $\forall\, k'\ \textit{pre red ps}.\ \textit{pointer}\ e = \textit{PreRed}\ (k', \textit{pre}, \textit{red})\ \textit{ps} \longrightarrow \textit{red} < \textit{length es}$
shows $\textit{mono-red-ptr}\ (bs[k := \textit{bin-upd e es}])$

lemma *sound-mono-ptrs-bin-upds*:
assumes $\textit{sound-ptrs inp bs}$ $\textit{mono-red-ptr bs}$ $k < \textit{length bs}$ $b = bs!k$ $\textit{distinct}\ (\textit{items}\ b)$ $\textit{distinct}\ (\textit{items}\ es)$
assumes $\forall\, e \in \textit{set es}.\ \textit{sound-null-ptr e} \wedge \textit{sound-pre-ptr inp bs k e} \wedge \textit{sound-prered-ptr bs k e}$
assumes $\forall\, e \in \textit{set es}.\ \forall\, k'\ \textit{pre red ps}.\ \textit{pointer}\ e = \textit{PreRed}\ (k', \textit{pre}, \textit{red})\ \textit{ps} \longrightarrow \textit{red} < \textit{length}\ b$
shows $\textit{sound-ptrs inp}\ (bs[k := \textit{bin-upds es b}]) \wedge \textit{mono-red-ptr}\ (bs[k := \textit{bin-upds es b}])$

lemma *sound-mono-ptrs-E-list′*:
assumes $(k, \textit{cfg}, \textit{inp}, \textit{bs}) \in \textit{wellformed-bins}$
assumes $\textit{sound-ptrs inp bs}$ $\textit{sound-items cfg inp}\ (\textit{bins-items bs})$
assumes $\textit{mono-red-ptr bs}$
assumes $\textit{nonempty-derives cfg}$ $\textit{wf-cfg cfg}$
shows $\textit{sound-ptrs inp}\ (\textit{E-list}'\ k\ \textit{cfg inp bs}\ i) \wedge \textit{mono-red-ptr}\ (\textit{E-list}'\ k\ \textit{cfg inp bs}\ i)$

lemma *sound-mono-ptrs-E-list*:
assumes $(k, \textit{cfg}, \textit{inp}, \textit{bs}) \in \textit{wellformed-bins}$
assumes $\textit{sound-ptrs inp bs}$ $\textit{sound-items cfg inp}\ (\textit{bins-items bs})$
assumes $\textit{mono-red-ptr bs}$
assumes $\textit{nonempty-derives cfg}$ $\textit{wf-cfg cfg}$
shows $\textit{sound-ptrs inp}\ (\textit{E-list}\ k\ \textit{cfg inp bs}) \wedge \textit{mono-red-ptr}\ (\textit{E-list}\ k\ \textit{cfg inp bs})$

lemma *sound-ptrs-Init-list*:
shows $\textit{sound-ptrs inp}\ (\textit{Init-list cfg inp})$

lemma *mono-red-ptr-Init-list*:
shows $\textit{mono-red-ptr}\ (\textit{Init-list cfg inp})$

lemma *sound-mono-ptrs-$\mathcal{E}$-list*:
assumes $k \leq \textit{length inp}$ $\textit{wf-cfg cfg}$ $\textit{nonempty-derives cfg}$ $\textit{wf-cfg cfg}$
shows $\textit{sound-ptrs inp}\ (\mathcal{E}\textit{-list}\ k\ \textit{cfg inp}) \wedge \textit{mono-red-ptr}\ (\mathcal{E}\textit{-list}\ k\ \textit{cfg inp})$

lemma *sound-mono-ptrs-earley-list*:
assumes $\textit{wf-cfg cfg}$ $\textit{nonempty-derives cfg}$
shows $\textit{sound-ptrs inp}\ (\textit{earley-list cfg inp}) \wedge \textit{mono-red-ptr}\ (\textit{earley-list cfg inp})$

6.3 Trees and Forests

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

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

fun root-tree :: 'a tree ⇒ 'a where
  root-tree (Leaf a) = a
| root-tree (Branch N -) = N

fun wf-rule-tree :: 'a cfg ⇒ 'a tree ⇒ bool where
  wf-rule-tree - (Leaf a) ⟷ True
| wf-rule-tree cfg (Branch N ts) ⟷ (
    (∃ r ∈ set (ℜ cfg). N = rule-head r ∧ map root-tree ts = rule-body r) ∧
    (∀ t ∈ set ts. wf-rule-tree cfg t))

fun wf-item-tree :: 'a cfg ⇒ 'a item ⇒ 'a tree ⇒ bool where
  wf-item-tree cfg - (Leaf a) ⟷ True
| wf-item-tree cfg x (Branch N ts) ⟷ (
    N = item-rule-head x ∧ map root-tree ts = take (item-bullet x) (item-rule-body x) ∧
    (∀ t ∈ set ts. wf-rule-tree cfg t))

definition wf-yield-tree :: 'a sentential ⇒ 'a item ⇒ 'a tree ⇒ bool where
  wf-yield-tree inp x t ≡ 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 ⇒ 'a list list where
  combinations [] = [[]]
| combinations (xs#xss) = [ x#cs . x <− xs, cs <− combinations xss ]

fun trees :: 'a forest ⇒ 'a tree list where
  trees (FLeaf a) = [Leaf a]
| trees (FBranch N fss) = (
    let tss = (map (λfs. concat (map (λf. trees f) fs)) fss) in
    map (λts. Branch N ts) (combinations tss)
  )
```

6.4 A Single Parse Tree

```
partial-function (option) build-tree' :: 'a bins ⇒ 'a sentential ⇒ nat ⇒ nat ⇒ '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 $(bs, inp, k, i) \in \mathit{wf\text{-}tree\text{-}input}$
assumes $e = bs!k!i$ *pointer* $e = \mathit{Pre}\ \mathit{pre}$
shows $(bs, inp, (k-1), pre) \in \mathit{wf\text{-}tree\text{-}input}$

lemma *wf-tree-input-prered-pre*:
assumes $(bs, inp, k, i) \in \mathit{wf\text{-}tree\text{-}input}$
assumes $e = bs!k!i$ *pointer* $e = \mathit{PreRed}\ (k', pre, red)\ ps$
shows $(bs, inp, k', pre) \in \mathit{wf\text{-}tree\text{-}input}$

lemma *wf-tree-input-prered-red*:
assumes $(bs, inp, k, i) \in \mathit{wf\text{-}tree\text{-}input}$
assumes $e = bs!k!i$ *pointer* $e = \mathit{PreRed}\ (k', pre, red)\ ps$
shows $(bs, inp, k, red) \in \mathit{wf\text{-}tree\text{-}input}$

lemma *build-tree′-termination*:
assumes $(bs, inp, k, i) \in \mathit{wf\text{-}tree\text{-}input}$
shows $\exists N\ ts.\ \mathit{build\text{-}tree}'\ bs\ inp\ k\ i = \mathit{Some}\ (\mathit{Branch}\ N\ ts)$

lemma *wf-item-tree-build-tree′*:
assumes $(bs, inp, k, i) \in \mathit{wf\text{-}tree\text{-}input}$
assumes *wf-bins cfg inp bs*
assumes $k < \mathit{length}\ bs\ i < \mathit{length}\ (bs!k)$
assumes $\mathit{build\text{-}tree}'\ bs\ inp\ k\ i = \mathit{Some}\ t$
shows $\mathit{wf\text{-}item\text{-}tree}\ cfg\ (\mathit{item}\ (bs!k!i))\ t$

lemma *wf-yield-tree-build-tree′*:
assumes $(bs, inp, k, i) \in \mathit{wf\text{-}tree\text{-}input}$
assumes *wf-bins cfg inp bs*
assumes $k < \mathit{length}\ bs\ i < \mathit{length}\ (bs!k)\ k \leq \mathit{length}\ inp$
assumes $\mathit{build\text{-}tree}'\ bs\ inp\ k\ i = \mathit{Some}\ t$
shows $\mathit{wf\text{-}yield\text{-}tree}\ inp\ (\mathit{item}\ (bs!k!i))\ t$

theorem *wf-rule-root-yield-tree-build-tree*:
assumes *wf-bins cfg inp bs sound-ptrs inp bs mono-red-ptr bs* $\mathit{length}\ bs = \mathit{length}\ inp + 1$
assumes $\mathit{build\text{-}tree}\ cfg\ inp\ bs = \mathit{Some}\ t$
shows $\mathit{wf\text{-}rule\text{-}tree}\ cfg\ t \wedge \mathit{root\text{-}tree}\ t = \mathfrak{S}\ cfg \wedge \mathit{yield\text{-}tree}\ t = inp$

corollary *wf-rule-root-yield-tree-build-tree-earley-list*:
assumes *wf-cfg cfg nonempty-derives cfg*
assumes $\mathit{build\text{-}tree}\ cfg\ inp\ (\mathit{earley\text{-}list}\ cfg\ inp) = \mathit{Some}\ t$
shows $\mathit{wf\text{-}rule\text{-}tree}\ cfg\ t \wedge \mathit{root\text{-}tree}\ t = \mathfrak{S}\ cfg \wedge \mathit{yield\text{-}tree}\ t = inp$

theorem *correctness-build-tree-earley-list*:
  assumes *wf-cfg cfg is-sentence cfg inp nonempty-derives cfg*
  shows $(\exists t.\ \mathit{build\text{-}tree\ cfg\ inp}\ (\mathit{earley\text{-}list\ cfg\ inp}) = \mathit{Some}\ t) \longleftrightarrow \mathit{derives\ cfg}\ [\mathfrak{S}\ \mathit{cfg}]\ \mathit{inp}$

6.5 All Parse Trees

```
fun insert-group :: ('a ⇒ 'k) ⇒ ('a ⇒ 'v) ⇒ 'a ⇒ ('k × 'v list) list ⇒ ('k × '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 *build-forest′-measure* :: (*'a bins* × *'a sentential* × *nat* × *nat* × *nat set*) ⇒ *nat* **where**
build-forest′-measure (*bs*, *inp*, *k*, *i*, *I*) = *foldl* (+) *0* (*map length* (*take* (*k*+1) *bs*)) − *card I*

definition *wf-trees-input* :: (*'a bins* × *'a sentential* × *nat* × *nat* × *nat set*) *set* **where**
wf-trees-input = {
  (*bs*, *inp*, *k*, *i*, *I*) | *bs inp k i I*.
    sound-ptrs inp bs ∧
    k < *length bs* ∧
    i < *length* (*bs!k*) ∧
    I ⊆ {*0..<length* (*bs!k*)} ∧
    i ∈ *I*
}

lemma *wf-trees-input-pre*:
assumes (*bs*, *inp*, *k*, *i*, *I*) ∈ *wf-trees-input*
assumes *e* = *bs!k!i pointer e* = *Pre pre*
shows (*bs*, *inp*, (*k*−1), *pre*, {*pre*}) ∈ *wf-trees-input*

lemma *wf-trees-input-prered-pre*:
assumes (*bs*, *inp*, *k*, *i*, *I*) ∈ *wf-trees-input*
assumes *e* = *bs!k!i pointer e* = *PreRed p ps*
assumes *ps′* = *filter* (*λ*(*k′*, *pre*, *red*). *red* ∉ *I*) (*p#ps*)
assumes *gs* = *group-by* (*λ*(*k′*, *pre*, *red*). (*k′*, *pre*)) (*λ*(*k′*, *pre*, *red*). *red*) *ps′*
assumes ((*k′*, *pre*), *reds*) ∈ *set gs*
shows (*bs*, *inp*, *k′*, *pre*, {*pre*}) ∈ *wf-trees-input*

lemma *wf-trees-input-prered-red*:
assumes (*bs*, *inp*, *k*, *i*, *I*) ∈ *wf-trees-input*
assumes *e* = *bs!k!i pointer e* = *PreRed p ps*
assumes *ps′* = *filter* (*λ*(*k′*, *pre*, *red*). *red* ∉ *I*) (*p#ps*)
assumes *gs* = *group-by* (*λ*(*k′*, *pre*, *red*). (*k′*, *pre*)) (*λ*(*k′*, *pre*, *red*). *red*) *ps′*
assumes ((*k′*, *pre*), *reds*) ∈ *set gs red* ∈ *set reds*

shows $(bs, inp, k, red, I \cup \{red\}) \in \textit{wf-trees-input}$

lemma *build-trees′-termination*:
assumes $(bs, inp, k, i, I) \in \textit{wf-trees-input}$
shows $\exists fs.\ \textit{build-trees}'\ bs\ inp\ k\ i\ I = \textit{Some}\ fs \land (\forall f \in \textit{set}\ fs.\ \exists N\ fss.\ f = \textit{FBranch}\ N\ fss)$

lemma *wf-item-tree-build-trees′*:
assumes $(bs, inp, k, i, I) \in \textit{wf-trees-input}$
assumes *wf-bins cfg inp bs*
assumes $k < \textit{length}\ bs\ i < \textit{length}\ (bs!k)$
assumes $\textit{build-trees}'\ bs\ inp\ k\ i\ I = \textit{Some}\ fs$
assumes $f \in \textit{set}\ fs$
assumes $t \in \textit{set}\ (\textit{trees}\ f)$
shows $\textit{wf-item-tree}\ cfg\ (\textit{item}\ (bs!k!i))\ t$

lemma *wf-yield-tree-build-trees′*:
assumes $(bs, inp, k, i, I) \in \textit{wf-trees-input}$
assumes *wf-bins cfg inp bs*
assumes $k < \textit{length}\ bs\ i < \textit{length}\ (bs!k)\ k \leq \textit{length}\ inp$
assumes $\textit{build-trees}'\ bs\ inp\ k\ i\ I = \textit{Some}\ fs$
assumes $f \in \textit{set}\ fs$
assumes $t \in \textit{set}\ (\textit{trees}\ f)$
shows $\textit{wf-yield-tree}\ inp\ (\textit{item}\ (bs!k!i))\ t$

theorem *wf-rule-root-yield-tree-build-trees*:
assumes $\textit{wf-bins}\ cfg\ inp\ bs\ \textit{sound-ptrs}\ inp\ bs\ \textit{length}\ bs = \textit{length}\ inp + 1$
assumes $\textit{build-trees}\ cfg\ inp\ bs = \textit{Some}\ fs\ f \in \textit{set}\ fs\ t \in \textit{set}\ (\textit{trees}\ f)$
shows $\textit{wf-rule-tree}\ cfg\ t \land \textit{root-tree}\ t = \mathfrak{S}\ cfg \land \textit{yield-tree}\ t = inp$

corollary *wf-rule-root-yield-tree-build-trees-earley-list*:
assumes *wf-cfg cfg nonempty-derives cfg*
assumes $\textit{build-trees}\ cfg\ inp\ (\textit{earley-list}\ cfg\ inp) = \textit{Some}\ fs\ f \in \textit{set}\ fs\ t \in \textit{set}\ (\textit{trees}\ f)$
shows $\textit{wf-rule-tree}\ cfg\ t \land \textit{root-tree}\ t = \mathfrak{S}\ cfg \land \textit{yield-tree}\ t = inp$

theorem *soundness-build-trees-earley-list*:
assumes *wf-cfg cfg is-sentence cfg inp nonempty-derives cfg*
assumes $\textit{build-trees}\ cfg\ inp\ (\textit{earley-list}\ cfg\ inp) = \textit{Some}\ fs\ f \in \textit{set}\ fs\ t \in \textit{set}\ (\textit{trees}\ f)$
shows $\textit{derives}\ cfg\ [\mathfrak{S}\ cfg]\ inp$

theorem *termination-build-tree-earley-list*:
assumes $\textit{wf-cfg}\ cfg\ \textit{nonempty-derives}\ cfg\ \textit{derives}\ cfg\ [\mathfrak{S}\ cfg]\ inp$
shows $\exists fs.\ \textit{build-trees}\ cfg\ inp\ (\textit{earley-list}\ cfg\ inp) = \textit{Some}\ fs$

6.6 A Word on Completeness

SNIPPET:

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.

7 Usage

definition *ε-free* :: *'a cfg* ⇒ *bool* **where**
ε-free cfg ⟷ (∀ *r* ∈ *set* (𝔑 *cfg*). *rule-body r* ≠ [])

lemma *ε-free-impl-non-empty-deriv*:
ε-free cfg ⟹ *N* ∈ *set* (𝔑 *cfg*) ⟹ ¬ *derives cfg* [*N*] []
datatype *t* = *x* | *plus*
datatype *n* = *S*
datatype *s* = *Terminal t* | *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 *recognizing-list* (*earley-list cfg inp*) *cfg inp* $\longleftrightarrow$ *derives cfg* [$\mathfrak{S}$ *cfg*] *inp*

8 Conclusion

8.1 Summary

8.2 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.