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

# Contents

# 1 Introduction

TODO

## 1.1 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.

## 1.2 Structure

TODO

## 1.3 Contributions

TODO

# 2 Earley Recognizer

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 $\mathcal{G}$: $S ::= x \mid S + S$ and the, rather trivial, input is $\omega = x + x + x$.

Intuitively, an Earley 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 parses 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 productions right-hand side we have recognized so far, an origin $i$ describing the position in $\omega$ where we started parsing, and an end $j$ indicating the position in $\omega$ 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 its end is the index $j$. Table 2.1 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 of $\omega$ (the first index being inclusive the second one exclusive) by $\alpha = S+$, and are trying to parse $\beta = S$ from position 4 in $\omega$.

The algorithm initializes $B$ by applying the *Init* operation. It then proceeds to execute the *Scan*, *Predict* and *Complete* operations listed in Figure 2.1 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 $\omega$ 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 parsed terminal symbol.

Considering our example, bin $B_3$ contains the item $S \rightarrow S \bullet +S, 2, 3$, the third symbol of $\omega$ is the terminal symbol $+$, 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 parse in the current bin and have not parsed 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 parsed the substring $\omega[j..k\rangle$, indices $j$ and $k$ being inclusive respectively exclusive, 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 in the first place. Since we parsed 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 $S \rightarrow S + S\bullet, 0, 5$ to bin $B_5$ for two different reasons corresponding to the two different ways we can derive $\omega$. 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 $S \rightarrow \alpha, 0, |\omega|$, 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. $S \Rightarrow S + S \Rightarrow x + S \Rightarrow x + S + S \Rightarrow^* x + x + x$ or $S \Rightarrow S + S \Rightarrow S + x \Rightarrow S + S + x \Rightarrow^* x + x + x$.

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

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

It follows from the following three lemmas about Earley items:

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

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

$$\textsc{Predict}\quad \frac{\mathrm{A} \rightarrow \alpha\bullet\mathrm{B}\ \beta,\ \mathrm{i},\ \mathrm{j} \qquad (\mathrm{B} \rightarrow \gamma) \ \in \mathcal{G}}{\mathrm{B} \rightarrow \bullet\gamma,\ \mathrm{j},\ \mathrm{j}} \qquad\qquad \textsc{Complete}\quad \frac{\mathrm{A} \rightarrow \alpha\bullet\mathrm{B}\ \beta,\ \mathrm{i},\ \mathrm{j} \qquad \mathrm{B} \rightarrow \gamma\bullet,\ \mathrm{j},\ \mathrm{k}}{\mathrm{A} \rightarrow \alpha\mathrm{B}\ \bullet\beta,\ \mathrm{i},\ \mathrm{k}}$$

Figure 2.1: Earley inference rules

Table 2.1: Earley items for the grammar $\mathcal{G}$: $S ::= x \mid S + S$

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

# 3 Earley Recognizer 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. Then we formalize the simplified Earley recognizer algorithm presented in Chapter 2; discussing the implementation and the proofs for soundness, completeness, and finiteness. Note that most of the basic definitions of Sections 3.1 and 3.2 are not our own work but only slightly adapted from Obua's work on Local Lexing [**Obua:2017**] [**LocalLexing-AFP**]. All of the proofs in this chapter are our own work.

## 3.1 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 $\tau$. Basic types include *bool* and *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, a symbol being either a terminal or a non-terminal. We represent a context-free grammar as the datatype *cfg*. An instance $\mathcal{G}$ consists of (1) a list of non-terminals $(\mathfrak{N}\ \mathcal{G})$, (2) a list of terminals $(\mathfrak{T}\ \mathcal{G})$, (3) a list of production rules $(\mathfrak{R}\ \mathcal{G})$, and a start symbol $(\mathfrak{S}\ \mathcal{G})$ where $\mathfrak{N}$, $\mathfrak{T}$, $\mathfrak{R}$ and $\mathfrak{S}$ are projections accessing the specific part of an instance $\mathcal{G}$ 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** $'a\ cfg =$
$\quad CFG\ (\mathfrak{N} : {}'a\ list)\ (\mathfrak{T} : {}'a\ list)\ (\mathfrak{R} : {}'a\ rules)\ (\mathfrak{S} : {}'a)$

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

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

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

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

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

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

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

Furthermore, in Isabelle, lists are constructed from the empty list [] via the infix cons-operator (#); the operator (@) appends two lists; $|xs|$ denotes the length and $xs\ !\ n$ 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 $\{x \mid P\ x\}$. They 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: $\alpha$, $\beta$, $\gamma$, occasionally also by lowercase letters $u$, $v$, $w$. The empty list in the context of a language is $\epsilon$. A sentential is a list consisting of only symbols. A sentence is a sentential if it only contains terminal symbols. Obua first defines a predicate *derives1* $\mathcal{G}$ $u$ $v$ which expresses that we can derive $v$ from $u$ in a single step or the predicate holds if there exist $\alpha$, $\beta$, $N$ and $\gamma$ such that $u = \alpha \mathbin{@} [N] \mathbin{@} \beta$, $v = \alpha \mathbin{@} \gamma \mathbin{@} \beta$ and $(N, \gamma)$ is a production rule. We introduce some slightly more convenient notation: *derives1* $\mathcal{G}$ $u$ $v$ is written $\mathcal{G} \vdash u \Rightarrow v$ in the following. He then defines 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 $\mathcal{G}$ or *derives* $\mathcal{G}$ $u$ $v$ if $(u, v) \in$ *derivations* $\mathcal{G}$. A slightly more convenient notation is again: *derives* $\mathcal{G}$ $u$ $v = \mathcal{G} \vdash u \Rightarrow^* v$.

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

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

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

**definition** *is-symbol* :: *'a cfg ⇒ 'a ⇒ bool* **where**
  *is-symbol* $\mathcal{G}$ $s \equiv$ *is-terminal* $\mathcal{G}$ $s \lor$ *is-nonterminal* $\mathcal{G}$ $s$

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

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

**definition** *derives1* :: *'a cfg ⇒ 'a sentential ⇒ 'a sentential ⇒ bool* **where**
  *derives1* $\mathcal{G}$ $u$ $v \equiv \exists \alpha\ \beta\ N\ \gamma.$
    $u = \alpha \mathbin{@} [N] \mathbin{@} \beta \land$
    $v = \alpha \mathbin{@} \gamma \mathbin{@} \beta \land$
    $(N, \gamma) \in$ *set* $(\mathfrak{R}\ \mathcal{G})$

**definition** *derivations1* :: *'a cfg ⇒ ('a sentential × 'a sentential) set* **where**
  *derivations1* $\mathcal{G} = \{ (u,v) \mid u\ v.\ \mathcal{G} \vdash u \Rightarrow v \}$

**definition** *derivations* :: *'a cfg ⇒ ('a sentential × 'a sentential) set* **where**
  *derivations* $\mathcal{G} = ($*derivations1* $\mathcal{G})\hat{}*$

**definition** *derives* :: *'a cfg ⇒ 'a sentential ⇒ 'a sentential ⇒ bool* **where**
  *derives* $\mathcal{G}$ $u$ $v \equiv (u, v) \in$ *derivations* $\mathcal{G}$

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 xs i j* computes the slice of a list *xs* between indices *i* (inclusive) and *j* (exclusive), e.g. *slice* [*a*, *b*, *c*, *d*, *e*] *2 4* evaluates to [*c*, *d*]. We also introduce a shorthand notation: e.g. *slice xs i j* is written $xs[i..j\rangle$ in the following.

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

Lemmas, theorems and corollaries are presented using the keywords *lemma*, *theorem*, and *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 prove a simple lemma about the interaction between the *slice* function and the append operator (@), stating the conditions under which we can combine two slices into a single one.

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

## 3.2 The Formalized Algorithm

Next we formalize the algorithm presented in Chapter 2. 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 $\omega$ that has been parsed 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-*$\alpha$ and *item-*$\beta$ 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 parsed 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 $\omega$.

$$\textbf{datatype}\ {'a}\ \mathit{item} = \\ \quad \mathit{Item}\ (\mathit{item\text{-}rule}\colon {'a}\ \mathit{rule})\ (\mathit{item\text{-}bullet} : \mathit{nat})\ (\mathit{item\text{-}origin} : \mathit{nat})\ (\mathit{item\text{-}end} : \mathit{nat})$$

$$\textbf{type-synonym}\ {'a}\ \mathit{items} = {'a}\ \mathit{item}\ \mathit{set}$$

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

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

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

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

$$\textbf{definition}\ \mathit{is\text{-}complete} :: {'a}\ \mathit{item} \Rightarrow \mathit{bool}\ \textbf{where} \\ \quad \mathit{is\text{-}complete}\ x \equiv \mathit{item\text{-}bullet}\ x \geq |\mathit{item\text{-}rule\text{-}body}\ x|$$

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

$$\textbf{definition}\ \mathit{is\text{-}finished} :: {'a}\ \mathit{cfg} \Rightarrow {'a}\ \mathit{sentential} \Rightarrow {'a}\ \mathit{item} \Rightarrow \mathit{bool}\ \textbf{where} \\ \quad \mathit{is\text{-}finished}\ \mathcal{G}\ \omega\ x \equiv \\ \qquad \mathit{item\text{-}rule\text{-}head}\ x = \mathfrak{S}\ \mathcal{G}\ \wedge \\ \qquad \mathit{item\text{-}origin}\ x = 0\ \wedge \\ \qquad \mathit{item\text{-}end}\ x = |\omega|\ \wedge \\ \qquad \mathit{is\text{-}complete}\ x$$

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

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.

$$\textbf{definition}\ \mathit{init\text{-}item} :: {'a}\ \mathit{rule} \Rightarrow \mathit{nat} \Rightarrow {'a}\ \mathit{item}\ \textbf{where} \\ \quad \mathit{init\text{-}item}\ r\ k = \mathit{Item}\ r\ 0\ k\ k$$

**definition** *inc-item* :: *'a item* ⇒ *nat* ⇒ *'a item* **where**
$\quad$ *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 2.1. 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 at 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 that 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 bin $k+1$, whereas the *Predict* and *Complete* operations only generate items belonging to the $k$-th bin. We then define a function *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 bin $k+1$ by iterating *Earley-step* until the set of items converges, 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 function *Earley* to $|\omega|$.

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

**definition** *Init* :: *'a cfg* ⇒ *'a items* **where**
$\quad$ *Init* $\mathcal{G}$ = { *init-item r 0* | *r*. *r* ∈ *set* ($\mathfrak{R}$ $\mathcal{G}$) ∧ *fst r* = ($\mathfrak{S}$ $\mathcal{G}$) }

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

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

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

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

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

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

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

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

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

**definition** $\mathcal{E}arley$ :: ${}'a\ cfg \Rightarrow {}'a\ sentential \Rightarrow {}'a\ items$ **where**
$\mathcal{E}arley\ \mathcal{G}\ \omega = Earley\ |\omega|\ \mathcal{G}\ \omega$

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**], who 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.

## 3.3 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 prove a property about the basic building blocks, the *Init*, *Scan*, *Predict*, and *Complete* operations. Then we prove 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 bins and thus for the $\mathcal{E}arley$ definition, or the set of Earley items.

Before we start to prove 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-item* $\mathcal{G}$ $\omega$ $x \equiv$
    *item-rule* $x \in$ *set* $(\mathfrak{R}\ \mathcal{G})$ $\wedge$
    *item-bullet* $x \leq |$*item-rule-body* $x|$ $\wedge$
    *item-origin* $x \leq$ *item-end* $x$ $\wedge$
    *item-end* $x \leq |\omega|$

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

**lemma** *wf-Init*:
  **shows** *wf-items* $\mathcal{G}$ $\omega$ (*Init* $\mathcal{G}$)

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

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

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* $\mathcal{G}$ $\omega$ $I$
  **shows** *wf-items* $\mathcal{G}$ $\omega$ (*funpower* (*Earley-step* $k$ $\mathcal{G}$ $\omega$) $n$ $I$)

**lemma** *wf-Earley-bin*:
**assumes** *wf-items* $\mathcal{G}$ $\omega$ $I$
**shows** *wf-items* $\mathcal{G}$ $\omega$ (*Earley-bin* $k$ $\mathcal{G}$ $\omega$ $I$)

**lemma** *wf-Earley-bin0*:
**shows** *wf-items* $\mathcal{G}$ $\omega$ (*Earley-bin 0* $\mathcal{G}$ $\omega$ (*Init* $\mathcal{G}$))

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

**lemma** *wf-Earley*:
**shows** *wf-items* $\mathcal{G}$ $\omega$ (*Earley* $n$ $\mathcal{G}$ $\omega$)

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

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*.

## 3.4 Soundness

Next we prove the soundness of the generated items, or: $A \to \alpha \bullet \beta, i, j \in B$ implies $A \overset{*}{\Rightarrow} \omega[i..j\rangle\beta$ 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 presenting 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* $\mathcal{G}$ $\omega$ $x$ = $\mathcal{G} \vdash$ [*item-rule-head* $x$] $\Rightarrow^*$ $\omega$[*item-origin* $x$..*item-end* $x\rangle$ @ *item-$\beta$* $x$

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

Obua [**Obua:2017**] [**LocalLexing-AFP**] defines derivations at two different layers of abstraction. 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* $\mathcal{G}$ $u$ $i$ $r$ $v$ that is conceptually analogous to the predicate $\mathcal{G} \vdash u \Rightarrow v$ but also captures the rule $r$ used for a single rewriting step and the position $i$ in $u$ where the rewriting occurs.

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

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* $\alpha$ [] $\beta$ holds only if $\alpha = \beta$. If the derivation consists of at least one rewrite pair $(i, r)$, or *Derivation* $\mathcal{G}\ \alpha\ ((i,\ r)\ \#\ D)\ \beta$ holds, then there must exist a $\gamma$ such that *Derives1* $\mathcal{G}\ \alpha\ i\ r\ \gamma$ and *Derivation* $\mathcal{G}\ \gamma\ D\ \beta$ hold. Note that we introduce once again a more convenient notation: e.g. *Derivation* $\alpha\ D\ \beta$ is written $\mathcal{G} \vdash \alpha \Rightarrow^D \beta$ in the following. Obua then proves that both notions of a derivation are equivalent (lemma *derives-equiv-Derivation*)

**type-synonym** $'a\ derivation = (nat \times 'a\ rule)\ list$

**fun** *Derivation* :: $'a\ cfg \Rightarrow 'a\ sentential \Rightarrow 'a\ derivation \Rightarrow 'a\ sentential \Rightarrow bool$ **where**
$Derivation\ \text{-}\ \alpha\ []\ \beta = (\alpha = \beta)$
$|\ Derivation\ \mathcal{G}\ \alpha\ (d\#D)\ \beta = (\exists\,\gamma.\ Derives1\ \mathcal{G}\ \alpha\ (fst\ d)\ (snd\ d)\ \gamma \wedge Derivation\ \mathcal{G}\ \gamma\ D\ \beta)$

**lemma** *derives-equiv-Derivation*:
**shows** $\mathcal{G} \vdash \alpha \Rightarrow^* \beta \equiv \exists\,D.\ \mathcal{G} \vdash \alpha \Rightarrow^D \beta$

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** $\mathcal{G} \vdash \alpha \Rightarrow^D \beta\ @\ \gamma\ @\ \delta$
**assumes** $\mathcal{G} \vdash \gamma \Rightarrow^E \gamma'$
**shows** $\exists\,F.\ \mathcal{G} \vdash \alpha \Rightarrow^F \beta\ @\ \gamma'\ @\ \delta$

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

**lemma** *sound-Complete*:
**assumes** *wf-items* $\mathcal{G}\ \omega\ I$
**assumes** *sound-items* $\mathcal{G}\ \omega\ I$
**shows** *sound-items* $\mathcal{G}\ \omega\ (Complete\ k\ I)$

*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:

$$x \in \textit{bin } I \ (\textit{item-origin } y) \quad (1) \qquad \textit{next-symbol } x = \textit{Some } (\textit{item-rule-head } y) \quad (2)$$
$$y \in \textit{bin } I\ k \quad (3) \qquad \textit{is-complete } y \quad (4)$$
$$z = \textit{inc-item } x\ k \quad (5)$$

Since $y$ is in bin $k$ (3), it is complete (4) and the set $I$ is sound by assumption, there exists a derivation $E$ such that

$$\mathcal{G} \vdash [\textit{item-rule-head } y] \Rightarrow^{E} \omega[\textit{item-origin } y..\textit{item-end } y\rangle \qquad (6)$$

by lemma *derives-equiv-Derivation*. Similarly, since $x$ is in bin *item-origin* $y$ (1) and due to assumption of soundness, there exists a derivation $D$ such that

$$\mathcal{G} \vdash [\textit{item-rule-head } x] \Rightarrow^{D} \omega[\textit{item-origin } x..\textit{item-origin } y\rangle \ @ \ \textit{item-}\beta\ x \qquad (7)$$

Note that $\textit{item-}\beta\ x = \textit{item-rule-head } y\ \#\ tl\ (\textit{item-}\beta\ x)$ since the next symbol of $x$ is equal to the item rule head of $y$ (2). Thus, by lemma *Derivation-append-rewrite*, and the definition of $D$ (7) and $E$ (6), there exists a derivation $F$ such that

$$\begin{aligned}\mathcal{G} \vdash [\textit{item-rule-head } x] \Rightarrow^{F} \omega[\textit{item-origin } x..\textit{item-origin } y\rangle\ @ \\ \omega[\textit{item-origin } y..\textit{item-end } y\rangle\ @\ tl\ (\textit{item-}\beta\ x)\end{aligned}$$

Additionally, we know that $x$ and $y$ are well-formed items due to the facts that $x$ is in bin *item-origin* $y$ (1), $y$ is in bin $k$ (3), and the assumption *wf-items* $\mathcal{G}\ \omega\ I$. Thus, we can discharge the assumptions of lemma *slice-append* ($\textit{item-origin } x \leq \textit{item-origin } y$ and $\textit{item-origin } y \leq \textit{item-end } y$) and have

$$\mathcal{G} \vdash [\textit{item-rule-head } x] \Rightarrow^{F} \omega[\textit{item-origin } x..\textit{item-end } y\rangle\ @\ tl\ (\textit{item-}\beta\ x)$$

Moreover, since $z = \textit{inc-item } x\ k$ (5) and the next symbol of x is the item rule head of y (2), it follows that $tl\ (\textit{item-}\beta\ x) = \textit{item-}\beta\ z$, and ultimately *sound-item* $\mathcal{G}\ \omega\ z$, again by the definition of $z$ (5) and lemma *derives-equiv-Derivation*.

□

**lemma** *sound-Init*:
**shows** *sound-items* $\mathcal{G}\ \omega\ (\textit{Init } \mathcal{G})$

**lemma** *sound-Scan*:
**assumes** *wf-items* $\mathcal{G}\ \omega\ I \land$ *sound-items* $\mathcal{G}\ \omega\ I$
**shows** *sound-items* $\mathcal{G}\ \omega\ (\textit{Scan } k\ \omega\ I)$

**lemma** *sound-Predict*:
**assumes** *sound-items* $\mathcal{G}$ $\omega$ $I$
**shows** *sound-items* $\mathcal{G}$ $\omega$ $(Predict\ k\ \mathcal{G}\ I)$

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

**lemma** *sound-funpower*:
**assumes** *wf-items* $\mathcal{G}$ $\omega$ $I$ $\wedge$ *sound-items* $\mathcal{G}$ $\omega$ $I$
**shows** *sound-items* $\mathcal{G}$ $\omega$ $(funpower\ (Earley\text{-}step\ k\ \mathcal{G}\ \omega)\ n\ I)$

**lemma** *sound-Earley-bin*:
**assumes** *wf-items* $\mathcal{G}$ $\omega$ $I$ $\wedge$ *sound-items* $\mathcal{G}$ $\omega$ $I$
**shows** *sound-items* $\mathcal{G}$ $\omega$ $(Earley\text{-}bin\ k\ \mathcal{G}\ \omega\ I)$

**lemma** *sound-Earley-bin0*:
**shows** *sound-items* $\mathcal{G}$ $\omega$ $(Earley\text{-}bin\ 0\ \mathcal{G}\ \omega\ (Init\ \mathcal{G}))$

**lemma** *sound-Earley*:
**shows** *sound-items* $\mathcal{G}$ $\omega$ $(Earley\ k\ \mathcal{G}\ \omega)$

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

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 $\omega$ from the start symbol of the grammar.

**theorem** *soundness*:
**assumes** *recognizing* $(\mathcal{E}arley\ \mathcal{G}\ \omega)\ \mathcal{G}\ \omega$
**shows** $\mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$

## 3.5 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* $k$ $\mathcal{G}$ $\omega$ $I$ 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** $bin\ (Earley\text{-}bin\ k\ \mathcal{G}\ \omega\ I)\ i = bin\ I\ i$

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

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

Next, we prove 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** $k \leq |\omega|$
  **assumes** $x \in bin\ (Earley\ k\ \mathcal{G}\ \omega)\ i$
  **assumes** $next\text{-}symbol\ x = Some\ a$
  **assumes** $\omega!i = a$
  **shows** $inc\text{-}item\ x\ (i{+}1) \in Earley\ k\ \mathcal{G}\ \omega$

*Proof.* The proof is by induction on $k$ 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

$$i + 1 \leq k + 1 \quad (1) \qquad k + 1 \leq |\omega| \quad (2)$$
$$x \in bin\ (Earley\ (k + 1)\ \mathcal{G}\ \omega)\ i \quad (3) \qquad next\text{-}symbol\ x = Some\ a \quad (4)$$
$$\omega\ !\ i = a \quad (5)$$

Assumptions (1) and (3) imply that $x \in bin\ (Earley\ k\ \mathcal{G}\ \omega)\ i$ by lemma *Earley-bin-bin-idem*.
We then consider two cases:

- $i + 1 \leq k$: We can apply the induction hypothesis using assumptions (2), (4), (5), and fact $x \in bin\ (Earley\ k\ \mathcal{G}\ \omega)\ i$ and have $inc\text{-}item\ x\ (i + 1) \in Earley\ k\ \mathcal{G}\ \omega$. The statement to proof follows by lemma *Earley-step-sub-Earley-bin* and the definition of *Earley-step*.

- $k < i + 1$: hence we have $i = k$ by assumption (1). Thus, we have *inc-item* $x$ $(i + 1)$ $\in$ *Scan* $k$ $\omega$ (*Earley* $k$ $\mathcal{G}$ $\omega$) using assumptions (2), (4), (5), and the fact $x \in$ *bin* (*Earley* $k$ $\mathcal{G}$ $\omega$) $i$ by the definition of the *Scan* operation. This in turn implies *inc-item* $x$ $(i + 1)$ $\in$ *Earley-step* $k$ $\mathcal{G}$ $\omega$ (*Earley* $k$ $\mathcal{G}$ $\omega$) 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* $k$ $\mathcal{G}$ $\omega$ 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* $k$ $\mathcal{G}$ $\omega$) $i$
  **assumes** *next-symbol* $x$ = *Some* $N$
  **assumes** $(N,\alpha) \in$ *set* $(\mathfrak{R}\ \mathcal{G})$
  **shows** *init-item* $(N,\alpha)$ $i \in$ *Earley* $k$ $\mathcal{G}$ $\omega$

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

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 $N \rightarrow \bullet A_0 A_1 \ldots A_n$ 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 $i_0 \leq i_1 \leq \dots \leq i_n \leq i_{n+1}$:

$$\begin{aligned}
&\mathcal{G} \vdash A_0 \Rightarrow^* \omega[i_0..i_1\rangle \\
&\mathcal{G} \vdash A_1 \Rightarrow^* \omega[i_1..i_2\rangle \\
&\dots \\
&\mathcal{G} \vdash A_n \Rightarrow^* \omega[i_n..i_{n+1}\rangle
\end{aligned}$$

We have one derivation to move the bullet over each terminal or non-terminal $A_i$ and consequently the item $N \rightarrow A_0 A_1 \dots A_n \bullet$ 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 sometimes need to limit the *partially-completed* property to derivations that do not exceed a certain length.

Lemma *partially-completed-upto* then formalizes the core idea: if the item $N \rightarrow \alpha \bullet \beta, i, j$ exists in a set of items $I$ and there exists a derivation $\beta \overset{D}{\Rightarrow} \omega[j..k\rangle$, 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* ⇒ *'a cfg* ⇒ *'a sentential* ⇒ *'a items* ⇒ (*'a derivation* ⇒ *bool*) ⇒ *bool* **where**
  *partially-completed k* $\mathcal{G}$ $\omega$ *I P* ≡ ∀ *i j x a D*.
    $i \leq j \wedge j \leq k \wedge k \leq |\omega| \wedge$
    $x \in$ *bin I i* $\wedge$
    *next-symbol x* = *Some a* $\wedge$
    $\mathcal{G} \vdash [a] \Rightarrow^D \omega[i..j\rangle \wedge P\ D \longrightarrow$
      *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 $\alpha'$ and $\beta'$ 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 $\alpha$ and $\beta$ and quite technical since we need to manipulate the exact indices where each rewriting rule is applied in $\alpha$ and $\beta$, and thus we omit it.

The second one is a, in spirit similar, lemma about splitting slices. 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** $\mathcal{G} \vdash (\alpha @ \beta) \Rightarrow^{D} \gamma$
  **shows** $\exists\, E\ F\ \alpha'\ \beta'.\ \mathcal{G} \vdash \alpha \Rightarrow^{E} \alpha' \wedge \mathcal{G} \vdash \beta \Rightarrow^{F} \beta' \wedge \gamma = \alpha' \mathbin{@} \beta' \wedge |E| \leq |D| \wedge |F| \leq |D|$

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

**lemma** *partially-completed-upto*:
  **assumes** $\textit{wf-items}\ \mathcal{G}\ \omega\ I$
  **assumes** $j \leq k$
  **assumes** $k \leq |\omega|$
  **assumes** $x = \textit{Item}\ (N,\alpha)\ b\ i\ j$
  **assumes** $x \in I$
  **assumes** $\mathcal{G} \vdash (\textit{item-}\beta\ x) \Rightarrow^{D} \omega[j..k\rangle$
  **assumes** $\textit{partially-completed}\ k\ \mathcal{G}\ \omega\ I\ (\lambda D'.\ |D'| \leq |D|\ )$
  **shows** $\textit{Item}\ (N,\alpha)\ |\alpha|\ i\ k \in I$

*Proof.* The proof is by induction on $(\textit{item-}\beta\ x)$ for arbitrary $b$, $i$, $j$, $k$, $N$, $\alpha$, $x$, and $D$:

For the base case we have $\textit{item-}\beta\ x = []$ and need to show that $\textit{Item}\ (N,\ \alpha)\ |\alpha|\ i\ k \in I$: The bullet of $x$ is right before $\textit{item-}\beta\ x$, or $\textit{item-}\alpha\ x = \alpha$. Thus, the value of the bullet must be equal to the length of $\alpha$, which implies $x = \textit{Item}\ (N,\ \alpha)\ |\alpha|\ i\ j$, since $x$ is a well-formed item and $\textit{item-}\beta\ x = []$.

We also know that $j = k$: we have $\mathcal{G} \vdash \textit{item-}\beta\ x \Rightarrow^{D} \omega[j..k\rangle$ and $\textit{item-}\beta\ x = []$ which in turn implies that $\omega[j..k\rangle = []$, and thus $j = k$ as trivial fact about the function *slice* follows.

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

For the induction step we need to show that $\textit{Item}\ (N,\ \alpha)\ |\alpha|\ i\ k \in I$ using assumptions:

$$\begin{array}{rlrl}
a \mathbin{\#} as = \textit{item-}\beta\ x & (1) & \textit{wf-items}\ \mathcal{G}\ \omega\ I & (2) \\
j \leq k & (3) & k \leq |\omega| & (4) \\
x = \textit{Item}\ (N,\ \alpha)\ b\ i\ j & (5) & x \in I & (6) \\
\mathcal{G} \vdash \textit{item-}\beta\ x \Rightarrow^{D} \omega[j..k\rangle & (7) & & \\
\textit{partially-completed}\ k\ \mathcal{G}\ \omega\ I\ (\lambda D'.\ |D'| \leq |D|) & (8) & &
\end{array}$$

Using assumptions (1), (3), and (7) there exists an index $j'$ and derivations $E$ and $F$ by lemmas *Derivation-append-split* and *slice-append-split* such that:

$$\begin{array}{rlrl}
\mathcal{G} \vdash [a] \Rightarrow^{E} \omega[j..j'\rangle & (9) & |E| \leq |D| & (10) \\
\mathcal{G} \vdash as \Rightarrow^{F} \omega[j'..k\rangle & (11) & |F| \leq |D| & (12) \\
j \leq j' & (13) & j' \leq k & (14)
\end{array}$$

We have *next-symbol* $x$ = *Some* $a$ due to assumption (1), consequently we have *inc-item* $x\ j' \in I$ using additionally the facts about derivation $E$ (9-10), the bounds on $j'$ (13-14) and the assumptions (4-7) by the definition of *partially-completed*. Note that *inc-item* $x\ j'$ = *Item* $(N, \alpha)\ (b + 1)\ i\ j'$, which we will from now on refer to as item $x'$.

From assumption (8) and fact (12) follows *partially-completed* $k\ \mathcal{G}\ \omega\ I\ (\lambda D'.\ |D'| \leq |F|)$. We also have $as$ = *item-β* $x'$ and $x' \in I$ by the definition of $x'$ and $x$ and the assumptions (1,5,6). Hence, we can apply the induction hypothesis for $x'$ using additionally the assumptions (2,4), and the facts about derivation $F$ (11-12) from above, and have *Item* $(N, \alpha)\ |\alpha|\ i\ k \in I$, what we intended to show.

□

**lemma** *partially-completed-Earley*:
**assumes** *wf-*$\mathcal{G}$ $\mathcal{G}$
**shows** *partially-completed* $k\ \mathcal{G}\ \omega$ (*Earley* $k\ \mathcal{G}\ \omega$) ($\lambda$-. *True*)

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

By definition of *partially-completed* we need to show *inc-item* $x\ j \in$ *Earley* $k\ \mathcal{G}\ \omega$ and can assume

$$\begin{array}{rlll}
i \leq j & (1) & j \leq k & (2) \\
k \leq |\omega| & (3) & x \in bin\ (Earley\ k\ \mathcal{G}\ \omega)\ i & (4) \\
next\text{-}symbol\ x = Some\ a & (5) & \mathcal{G} \vdash [a] \Rightarrow^{D} \omega[i..j\rangle & (6)
\end{array}$$

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

- $D$ = []: Since $\mathcal{G} \vdash [a] \Rightarrow^{D} \omega[i..j\rangle$, we have $[a] = \omega[i..j\rangle$, and consequently $\omega\ !\ i = a$ and $j = i + 1$. Now we discharge the assumptions of lemma *Scan-Earley*, by assumptions (4,5) and the fact $j \leq |\omega|$ (that follows from assumptions (2,3)), and have *inc-item* $x\ (i + 1) \in$ *Earley* $k\ \mathcal{G}\ \omega$ which finishes the proof since $j = i + 1$.

- $D = d\ \#\ \mathcal{D}$: Due to assumption $\mathcal{G} \vdash [a] \Rightarrow^{D} \omega[i..j\rangle$, there exists an $\alpha$ such that *Derives1* $\mathcal{G}$ $[a]$ (*fst* $d$) (*snd* $d$) $\alpha$ and $\mathcal{G} \vdash \alpha \Rightarrow^{\mathcal{D}} \omega[i..j\rangle$ by the definition of *Derivation*. From the definition of *Derives1* we see that there exists a non-terminal $N$ such that $a = N$, $(N, \alpha) \in$ *set* $(\mathfrak{R}\ \mathcal{G})$, *fst* $d$ = *0*, and *snd* $d = (N, \alpha)$.

Let $y$ denote *Item* $(N, \alpha)$ *0 i i*. Since we have $i \leq k$ (assumptions (1,2)), and assumptions (4,5), and we showed that $a = N$ and $(N, \alpha) \in set\ (\mathfrak{R}\ \mathcal{G})$, and $y$ is an initial item, we have $y \in Earley\ k\ \mathcal{G}\ \omega$ 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* $k\ \mathcal{G}\ \omega$ since we have a derivation $\mathcal{G} \vdash \alpha \Rightarrow^{\mathcal{D}} \omega[i..j\rangle$, or *Item* $(N, \alpha)\ |\alpha|\ i\ j \in bin\ (Earley\ k\ \mathcal{G}\ \omega)\ j$: we use assumptions (1-3); have proven $y \in Earley\ k\ \mathcal{G}\ \omega$; and have *wf-items* $\mathcal{G}\ \omega\ (Earley\ k\ \mathcal{G}\ \omega)$ by lemma *wf-Earley*. Additionally, we know $\mathcal{G} \vdash$ *item-β* $y \Rightarrow^{\mathcal{D}} \omega[i..j\rangle$ since $\mathcal{G} \vdash [a] \Rightarrow^{\mathcal{D}} \omega[i..j\rangle$ and $a = N$, by the definition of item $y$. Finally, we use the induction hypothesis to show *partially-completed* $k\ \mathcal{G}\ \omega\ (Earley\ k\ \mathcal{G}\ \omega)$ $(\lambda E.\ |E| \leq |\mathcal{D}|)$, since $|\mathcal{D}| \leq |D|$ by definition of *partially-completed*, using once again all of our assumptions. This in turn implies *partially-completed* $j\ \mathcal{G}\ \omega\ (Earley\ k\ \mathcal{G}\ \omega)\ (\lambda E.\ |E| \leq |\mathcal{D}|)$ 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* $x\ j \in Earley\ k\ \mathcal{G}\ \omega$ by lemma *Complete-Earley*: Once again we use assumptions (1,2,4), we also know that *next-symbol* $x =$ *Some* $N$, due to assumption (5) and the fact $a = N$. Moreover, we have $(N, \alpha) \in set\ (\mathfrak{R}\ \mathcal{G})$ and most importantly *Item* $(N, \alpha)\ |\alpha|\ i\ j \in bin\ (Earley\ k\ \mathcal{G}\ \omega)\ j$, 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-$\mathcal{G}$* $\mathcal{G}$
**shows** *partially-completed* $|\omega|\ \mathcal{G}\ \omega\ (\mathcal{E}arley\ \mathcal{G}\ \omega)\ (\lambda\text{-}.\ True)$

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

**theorem** *completeness*:
**assumes** *wf-$\mathcal{G}$* $\mathcal{G}$
**assumes** *is-sentence* $\mathcal{G}\ \omega$
**assumes** $\mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$
**shows** *recognizing* $(\mathcal{E}arley\ \mathcal{G}\ \omega)\ \mathcal{G}\ \omega$

*Proof.* We know that there exists an $\alpha$ and a derivation $D$ such that $(\mathfrak{S}\ \mathcal{G}, \alpha) \in set\ (\mathfrak{R}\ \mathcal{G})$ and $\mathcal{G} \vdash \alpha \Rightarrow^{D} \omega$, since $\mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$. Let $x$ denote the item *Item* $(\mathfrak{S}\ \mathcal{G}, \alpha)$ *0 0 0*.

By definition of $x$ and the *Init* operation and $\mathcal{E}arley$ function, and the fact that *Init* $\mathcal{G} \subseteq$ *Earley* $k\ \mathcal{G}\ \omega$, we have $x \in \mathcal{E}arley\ \mathcal{G}\ \omega$, moreover we have *partially-completed* $|\omega|\ \mathcal{G}\ \omega\ (\mathcal{E}arley\ \mathcal{G}\ \omega)\ (\lambda\text{-}.\ True)$ using lemma *partially-completed-$\mathcal{E}$arley* and assumption *wf-$\mathcal{G}$* $\mathcal{G}$, and thus have *Item* $(\mathfrak{S}\ \mathcal{G},\ \alpha)\ |\alpha|\ 0\ |\omega| \in \mathcal{E}arley\ \mathcal{G}\ \omega$ by lemmas *partially-completed-upto* and *wf-$\mathcal{E}$arley* and the definition of *partially-completed*. The statement *recognizing* $(\mathcal{E}arley\ \mathcal{G}\ \omega)\ \mathcal{G}\ \omega$ follows immediately by the definition of *recognizing*, *is-finished*, and *is-complete*.

□

**corollary** *correctness-$\mathcal{E}$arley*:
**assumes** *wf-$\mathcal{G}$* $\mathcal{G}$
**assumes** *is-sentence* $\mathcal{G}\ \omega$
**shows** *recognizing* $(\mathcal{E}arley\ \mathcal{G}\ \omega)\ \mathcal{G}\ \omega \longleftrightarrow \mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$

## 3.6 Finiteness

At last, we prove that the set of Earley items is finite. In Chapter 4 we are using this result to prove the termination of an executable version of the algorithm.

Since $\mathcal{E}arley\ \mathcal{G}\ \omega$ 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

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

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

**lemma** *finite-wf-item*:
**shows** *finite* $\{\ x \mid x.\ wf\text{-}item\ \mathcal{G}\ \omega\ x\ \}$

**theorem** *finite-$\mathcal{E}$arley*:
**shows** *finite* $(\mathcal{E}arley\ \mathcal{G}\ \omega)$

# 4 Earley Recognizer Implementation

In Chapter 3 we proved correctness of a set-based, non-executable version of Earley's simplified recognizer algorithm. In this chapter we implement an executable algorithm. But instead of re-proving soundness and completeness for the executable algorithm, we follow the approach of Jones [**Jones:1972**]. We refine our set-based approach from Chapter 3 to a *functional* list-based implementation and prove subsumption in both directions, or each item generated by the list-based approach is also generated by the set-based approach which implies soundness of the executable algorithm, and vice versa which implies in turn completeness. We extend the algorithm of Chapter 3 in a second orthogonal way by already adding the necessary information to construct parse trees. We only introduce and explain the needed data structures but refrain from presenting any proofs in this chapter since constructing parse trees is the primary subject of Chapter 5.

## 4.1 The Executable Algorithm

We introduce a new data representation: instead of a set of Earley items we work with the data structure *bins*: a list of static length ($|\omega|$ + *1*) containing in turn bins implemented as variable length lists of Earley *entries*. An entry consists of an Earley item and a new data type *pointer* representing conceptually an imperative pointer describing the origin of its accompanying item. Table 4.1 illustrates the bins for our running example. There are three possible reasons, corresponding to the three basic operations, for the existence of an entry with Earley item $x$ in a specific bin $k$:

- It was predicted. In that case we consider it created from thin air and do not need to track any additional information, thus the pointer is *Null*. For our example, bin $B_0$ contains the entry $S \rightarrow \bullet x, 0, 0; \perp$ consisting of the item $S \rightarrow \bullet x, 0, 0$ and a *Null* pointer denoted by $\perp$.

- It was scanned. Then there exists another Earley item $x'$ in the previous bin $k-1$ from which this item was computed. Hence, we keep a predecessor pointer *Pre pre* where *pre* is a natural number indicating the index of item $x'$ in bin $k-1$. Table 4.1 contains the entry $S \rightarrow x\bullet, 2, 3; 1$ in bin $B_3$, the predecessor pointer is 1 (we omit the *Pre* constructor for readability) since this item was created by the item $S \rightarrow \bullet x, 2, 2$ of the entry at index 1 in $B_2$.

- It was completed. Note that an item might be completed in more than one way. In each case the item $x$ has a complete reduction item $y$ in the current bin and a predecessor item $x'$ in the origin bin of $y$. We track this information by at least one reduction pointer (*PreRed* ($k'$, *pre*, *red*) *reds*) where $k'$, *pre*, and *red* are respectively the origin index of the complete item $y$ or the bin of item $x'$, *pre* is the index of $x'$ in bin $k'$, and *red* is the index of $y$ in the current bin $k$. The list *reds* contains other valid reduction triples for this item. This is illustrated by the entry $S \rightarrow S + S\bullet, 0, 5; (4, 1, 0), (2, 0, 1)$ in bin $B_5$ of Table 4.1. We omit the *PreRed* and list constructors again for readability. This entry (without the second reduction triple) was first created due to the complete item $S \rightarrow x\bullet, 4, 5$ at index 0 in bin $B_5$ and the predecessor item $S \rightarrow S + \bullet S, 0, 4$ at index 1 in bin $B_4$, but we can also create it by the complete item $S \rightarrow S + S\bullet, 2, 5$ at index 1 in bin $B_5$ and the predecessor item $S \rightarrow S + \bullet S, 0, 2$ at index 0 in bin $B_2$, or the two possible ways to derive the input $\omega = (x + x) + x$ and $\omega = x + (x + x)$.

Additionally, we define two useful abbreviations *items* and *pointers* that map a given bin to the list of items respectively pointers it consists of.

Table 4.1: Earley items with pointers for the grammar $\mathcal{G}$: $S \rightarrow x$, $S \rightarrow S + S$

| | $B_0$ | $B_1$ | $B_2$ |
|---|---|---|---|
| 0 | $S \rightarrow \bullet x, 0, 0; \perp$ | $S \rightarrow x\bullet, 0, 1; 0$ | $S \rightarrow S + \bullet S, 0, 2; 1$ |
| 1 | $S \rightarrow \bullet S + S, 0, 0; \perp$ | $S \rightarrow S \bullet + S, 0, 1; (0, 1, 0)$ | $S \rightarrow \bullet x, 2, 2; \perp$ |
| 2 | | | $S \rightarrow \bullet S + S, 2, 2; \perp$ |

| | $B_3$ | $B_4$ | $B_5$ |
|---|---|---|---|
| 0 | $S \rightarrow x\bullet, 2, 3; 1$ | $S \rightarrow S + \bullet S, 2, 4; 2$ | $S \rightarrow x\bullet, 4, 5; 2$ |
| 1 | $S \rightarrow S + S\bullet, 0, 3; (2, 0, 0)$ | $S \rightarrow S + \bullet S, 0, 4; 3$ | $S \rightarrow S + S\bullet, 2, 5; (4, 0, 0)$ |
| 2 | $S \rightarrow S \bullet + S, 2, 3; (2, 2, 0)$ | $S \rightarrow \bullet x, 4, 4; \perp$ | $S \rightarrow S + S\bullet, 0, 5; (4, 1, 0), (2, 0, 1)$ |
| 3 | $S \rightarrow S \bullet + S, 0, 3; (0, 1, 1)$ | $S \rightarrow \bullet S + S, 4, 4; \perp$ | $S \rightarrow S \bullet + S, 4, 5; (4, 3, 0)$ |
| 4 | | | $S \rightarrow S \bullet + S, 2, 5; (2, 2, 1)$ |
| 5 | | | $S \rightarrow S \bullet + S, 0, 5; (0, 1, 2)$ |

**datatype** *pointer* =
*Null*
| *Pre nat* — pre
| *PreRed nat* × *nat* × *nat* (*nat* × *nat* × *nat*) *list* — (k', pre, red) reds

**datatype** *'a entry* =
*Entry* (*item* : *'a item*) (*pointer* : *pointer*)

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

**definition** $items :: 'a\ bin \Rightarrow 'a\ item\ list$ **where**
$items\ b = map\ item\ b$

**definition** $pointers :: 'a\ bin \Rightarrow pointer\ list$ **where**
$pointers\ b = map\ pointer\ b$

Next we implement list-based versions of the *Init*, *Scan*, *Predict*, and *Complete* operations. Function *Init-list* creates a list of ($|\omega| + 1$) empty lists or bins. Subsequently, it constructs an initial bin containing entries consisting of initial items with *Null* pointers for all the production rules that have the start symbol on their left-hand sides, and finally it overwrites the 0-th bin with this initial bin.

**definition** $Init\text{-}list :: 'a\ cfg \Rightarrow 'a\ sentential \Rightarrow 'a\ bins$ **where**
$Init\text{-}list\ \mathcal{G}\ \omega \equiv$
$let\ bs = replicate\ (\ |\omega| + 1)\ ([])\ in$
$let\ rs = filter\ (\lambda r.\ rule\text{-}head\ r = \mathfrak{S}\ \mathcal{G})\ (\mathfrak{R}\ \mathcal{G})\ in$
$let\ b0 = map\ (\lambda r.\ (Entry\ (init\text{-}item\ r\ 0)\ Null))\ rs\ in$
$bs[0 := b0]$

Functions *Scan-list*, *Predict-list*, and *Complete-list* are defined analogously to the definitions of *Scan*, *Predict*, and *Complete* and thus we only highlight noteworthy differences. The set-based implementations take accumulated as arguments the index $k$ of the current bin, the grammar $\mathcal{G}$, the input $\omega$, and the current set of Earley items $I$. The list-based definitions are more specific. The $k$-th bin is no longer only conceptional and we replace the argument $I$ in the following ways: function *Scan-list* takes as arguments the currently considered item $x$, its next *terminal* symbol $a$ (as plain value and not wrapped in an option) and the index *pre* of $x$ in the current bin $k$ (the item $x$ will be the predecessor of any item the function *Scan-list* computes, and sets the predecessor pointer accordingly. Function *Predict-list* only needs access to the next non-terminal symbol $N$ of $x$, and returns only entries with *Null* pointers. The implementation of *Complete-list* is slightly more involved. It takes as arguments again $x$ and the index *red* of $x$ in the current bin $k$ (since $x$ is a complete reduction item this time around), but also the complete bins *bs*, since it needs to find all potential predecessor items as well as their indices in the origin bin of $x$ (see *find-with-index*), and sets the reduction triples accordingly.

**definition** $Scan\text{-}list :: nat \Rightarrow 'a\ sentential \Rightarrow 'a\ \Rightarrow 'a\ item \Rightarrow nat \Rightarrow 'a\ entry\ list$ **where**
$Scan\text{-}list\ k\ \omega\ a\ x\ pre \equiv$
$if\ \omega!k = a\ then$
$let\ x' = inc\text{-}item\ x\ (k+1)\ in$
$[Entry\ x'\ (Pre\ pre)]$

```
  else []

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

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

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

In our data representation a bin is just a simple list but it implements a set. Hence, we need to make sure that updating a bin (*bin-upd*) or inserting an additional entry into a bin maintains its set properties. Additionally, since it is possible to generate multiple reduction pointers for the same item, we have to take care to update the pointer information accordingly, in particular merge reduction triples, if the item of the entry to be inserted matches the item of an already present entry. Function *bin-upds* inserts multiple entries into a specific bin. Finally, function *bins-upd* updates the $k$-th bin by inserting the given list of entries using function *bin-upds*. Note that an alternative but equivalent implementation of *bin-upds* is *fold bin-upd es b*. We primarily choose the explicit definition since it simplified some of the proofs, but overall the choice is stylistic in nature.

```
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)
```

$$
\begin{array}{l}
\textbf{fun } \textit{bin-upds} :: {}'a\ \textit{entry list} \Rightarrow {}'a\ \textit{bin} \Rightarrow {}'a\ \textit{bin}\ \textbf{where} \\
\quad \textit{bin-upds}\ []\ b = b \\
|\ \textit{bin-upds}\ (e\#es)\ b = \textit{bin-upds}\ es\ (\textit{bin-upd}\ e\ b)
\end{array}
$$

$$
\begin{array}{l}
\textbf{definition } \textit{bins-upd} :: {}'a\ \textit{bins} \Rightarrow \textit{nat} \Rightarrow {}'a\ \textit{entry list} \Rightarrow {}'a\ \textit{bins}\ \textbf{where} \\
\quad \textit{bins-upd}\ bs\ k\ es = bs[k := \textit{bin-upds}\ es\ (bs!k)]
\end{array}
$$

The central piece for the list-based implementation is the function *Earley-bin-list'*. A function call of the form *Earley-bin-list'* $k\ \mathcal{G}\ \omega\ bs\ i$ completes the $k$-th bin starting from index $i$. For the current item $x$ under consideration the function first computes the possible new entries depending on the next symbol of $x$ which can either be some terminal symbol - we scan -, or non-terminal symbol - we predict -, or *None* - we complete. It then updates the bins *bs* appropriately using the function *bins-upd*. Note that we have to define the function as a *partial-function*, since it might never terminate if it keeps appending newly generated items to the $k$-th bin it currently operates on. We prove termination and highlight the relevant Isabelle specific details in Section 4.4. The function *Earley-bin-list* then fully completes the $k$-th bin, or starts its computation at index 0, and thus corresponds in functionality to the function *Earley-bin* of Chapter 3.

$$
\begin{array}{l}
\textbf{partial-function } (\textit{tailrec})\ \textit{Earley-bin-list}' :: \textit{nat} \Rightarrow {}'a\ \textit{cfg} \Rightarrow {}'a\ \textit{sentential} \Rightarrow {}'a\ \textit{bins} \Rightarrow \textit{nat} \Rightarrow \\
{}'a\ \textit{bins}\ \textbf{where} \\
\quad \textit{Earley-bin-list}'\ k\ \mathcal{G}\ \omega\ bs\ i = ( \\
\quad\quad \textit{if}\ i \geq |\textit{items}\ (bs!k)|\ \textit{then}\ bs \\
\quad\quad \textit{else} \\
\quad\quad\quad \textit{let}\ x = \textit{items}\ (bs!k)\ !\ i\ \textit{in} \\
\quad\quad\quad \textit{let}\ bs' = \\
\quad\quad\quad\quad \textit{case next-symbol}\ x\ \textit{of} \\
\quad\quad\quad\quad\quad \textit{Some}\ a \Rightarrow \\
\quad\quad\quad\quad\quad\quad \textit{if is-terminal}\ \mathcal{G}\ a\ \textit{then} \\
\quad\quad\quad\quad\quad\quad\quad \textit{if}\ k < |\omega|\ \textit{then bins-upd}\ bs\ (k{+}1)\ (\textit{Scan-list}\ k\ \omega\ a\ x\ i) \\
\quad\quad\quad\quad\quad\quad\quad \textit{else}\ bs \\
\quad\quad\quad\quad\quad\quad \textit{else bins-upd}\ bs\ k\ (\textit{Predict-list}\ k\ \mathcal{G}\ a) \\
\quad\quad\quad\quad |\ \textit{None} \Rightarrow \textit{bins-upd}\ bs\ k\ (\textit{Complete-list}\ k\ x\ bs\ i) \\
\quad\quad\quad \textit{in Earley-bin-list}'\ k\ \mathcal{G}\ \omega\ bs'\ (i{+}1))
\end{array}
$$

$$
\begin{array}{l}
\textbf{definition } \textit{Earley-bin-list} :: \textit{nat} \Rightarrow {}'a\ \textit{cfg} \Rightarrow {}'a\ \textit{sentential} \Rightarrow {}'a\ \textit{bins} \Rightarrow {}'a\ \textit{bins}\ \textbf{where} \\
\quad \textit{Earley-bin-list}\ k\ \mathcal{G}\ \omega\ bs = \textit{Earley-bin-list}'\ k\ \mathcal{G}\ \omega\ bs\ 0
\end{array}
$$

Finally, functions *Earley-list* and $\mathcal{E}$*arley-list* are structurally identical to functions *Earley* respectively $\mathcal{E}$*arley* of Chapter 3, differing only in the type of the used operations and the return type: bins or lists of items instead of sets of items.

$$
\textbf{fun } \textit{Earley-list} :: \textit{nat} \Rightarrow {}'a\ \textit{cfg} \Rightarrow {}'a\ \textit{sentential} \Rightarrow {}'a\ \textit{bins}\ \textbf{where}
$$

$\quad$ *Earley-list 0* $\mathcal{G}$ $\omega$ = *Earley-bin-list 0* $\mathcal{G}$ $\omega$ (*Init-list* $\mathcal{G}$ $\omega$)
| *Earley-list* (*Suc n*) $\mathcal{G}$ $\omega$ = *Earley-bin-list* (*Suc n*) $\mathcal{G}$ $\omega$ (*Earley-list n* $\mathcal{G}$ $\omega$)

**definition** $\mathcal{E}$*arley-list* :: *'a cfg* $\Rightarrow$ *'a sentential* $\Rightarrow$ *'a bins* **where**
$\quad$ $\mathcal{E}$*arley-list* $\mathcal{G}$ $\omega$ = *Earley-list* $|\omega|$ $\mathcal{G}$ $\omega$

## 4.2 A Word on Performance

Earley [**Earley:1970**] implements his recognizer algorithm in the imperative programming paradigm and provides an informal argument for the running time $\mathcal{O}(n^3)$ where $n = |\omega|$. In contrast, our implementation is purely functional, and one might expect a quite significant decrease in performance. In this section we provide an informal argument showing that, although we cannot quite achieve the time complexity of an imperative implementation, we are 'only' one order of magnitude slower or the running time of our implementation is $\mathcal{O}(n^4)$. Then we summarize Earley's imperative implementation approach and the additional steps that are needed to achieve the desired running time. Additionally, we sketch a slightly different and more complicated functional implementation that achieves a theoretical running time of $\mathcal{O}(n^3 \log n)$, and highlight possible further performance improvements. Finally, we discuss why we choose our particular implementation.

We state the running time of our implementation of the algorithm in terms of the length $n$ of the input $\omega$, and provide an informal argument that its running time is $\mathcal{O}(n^4)$. Each bin $B_j$ $(0 \leq j \leq n)$ contains only items of the form *Item r b i j*. The number of possible production rules $r$, and possible bullet positions $b$ are both independent of $n$ and can thus be considered (possible large) constants. The origin $i$ is bounded by $0 \leq i \leq j$ and thus depends on $j$ which is in turn dependent on $n$. Thus, the number of items in each bin $B_j$ is overall $\mathcal{O}(n)$.

We also have *Init-list* $\in \mathcal{O}(n)$ since the function *replicate* takes time linear in the length of $\omega$, and functions *filter* and *map* operate at most on the size of the grammar $\mathcal{G}$ or constant time. We also know *Scan-list* $\in \mathcal{O}(n)$. The dominating term is surprisingly $\omega$ ! $k$, since $0 \leq k \leq n$, and it computes at most one new entry. Function *Predict-list* takes time in the the size of the grammar $\mathcal{G}$, due to the *filter* and *map* functions, or constant time, and computes at most $|\mathcal{G}|$ new items. Function *Complete-list* again takes linear time, since finding the origin bin of the given item $x$ takes linear time, and functions *items*, *filter-with-index*, and *map* operate on the origin bin which is - in the worst case - of linear size as argued in the previous paragraph. Consequently, the function also computes at most $\mathcal{O}(n)$ new items.

Updating a bin (*bin-upd*) with a single entry takes at most linear time, inserting $e$ new entries (*bin-upds*) thus takes time $e \cdot \mathcal{O}(n)$, and hence function *bins-upd* also runs in time

$e \cdot \mathcal{O}(n)$. The analysis of function *Earley-bin-list′* is slightly more involved. It computes the contents of a bin $B_j$, or it calls itself recursively at most $n$ times, since the number of items in any bin is $\mathcal{O}(n)$. The time for one function execution is dominated by the time it takes to update the bins with the newly created items whose number in turn depends on the operation we applied but is bounded in the worst case by $n$ during the *Complete-list* operation. All the other operations of the function body run in at most linear time. Overall we have for the body of *Earley-bin-list′*: $\mathcal{O}(n) + e \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$. And thus *Earley-bin-list′* $\in \mathcal{O}(n^3)$. The same bound holds trivially for *Earley-bin-list*. Since functions $\mathcal{E}$*arley-list* or *Earley-list* call *Earley-bin-list* once for each bin $B_j$ and $0 \leq j \leq n$, the overall running time is $\mathcal{O}(n^4)$.

One might be tempted to think that the decrease in performance compared to an imperative implementation is due to the fact that we are representing bins as functional lists and appending to and indexing into bins which takes linear time and not constant time. This is not the case. Earley implements the algorithm as follows. On the top-level bins are no longer a list but an array. Each bin is a singly-linked list, and pointers are no longer represented by the type *pointer* but by actual pointers. The worst case running time of the algorithm is still $\mathcal{O}(n^4)$. The algorithm still iterates over $n$ bins, traverses in the worst case $\mathcal{O}(n)$ items in each bin and for each item, the worst case operation, completion, still generates $\mathcal{O}(n)$ new items that all have to be inserted into the current bin which takes linear time for *each* new item. To achieve the running time of $\mathcal{O}(n^3)$ we need to find a way to add a new item into a bin in constant time. In an imperative setting one obvious way is to not only keep a singly-linked list of items and pointers but additionally a map. The keys are the items of the list and the map stores as value for a specific item a pointer to itself or its position in the list. Insertion of a new item into a bin then works as follows: if the item is already present in the map, we follow the pointer to the item and update the parse tree pointers of the item in the list accordingly depending on the kind of item. Otherwise we just append the item and its corresponding parse tree pointers to the list and insert the item and a pointer to its position in the linked list into the map.

Sadly, this approach does not work in a functional setting. Appending an item to a list takes linear and not constant time. But even if we preprend the new item onto the list there is another problem. We cannot simply store pointers in the map that we can chase in constant time to the location of the item in the list, but still have to store the index of the corresponding item. And consequently updating the pointer information takes again linear time due to the indexing. One possible solution is to change one's point of view. In the imperative approach the list serves two purposes: it represents the bin and is at the same time a worklist for the algorithm. The map only optimizes performance. We can obtain a $\mathcal{O}(n^3 \log n)$ functional implementation if we consider the list only a worklist and the map (or its keys) the bin. We also need to adapt the pointer datatype. Instead of wrapping indices representing predecessor or reduction items in the list, a pointer should contain the actual items. E.g. a pointer is either *Null*, or *Pre* $x'$, or *PreRed* $(x', y)$ *xys* where $x'$ is respectively the predecessor item and $y$ is the complete reduction item. Overall the running time for inserting a new item into a bin consists of prepending the item onto the worklist, or constant time, and inserting the item into the map which can be done in logarithmic time. Thus, the overall running time of this approach is $\mathcal{O}(n^3 \log n)$.

Since we are already talking about performance, we highlight some of the more common performance improvements. We can predict faster if we organize the grammar in a more efficient manner. Currently, the *Predict* operation needs to pass through the whole grammar to find the alternatives for a specific non-terminal. The first performance improvement is to group the production rules by their left-hand side non-terminals. We can also complete more efficiently. The *Complete* operation scans through the origin bin of an complete item, searching for items where the next symbol matches the rule head of the production rule of the complete item. We can optimize this search by keeping an additional map from 'next symbol' non-terminals to their corresponding items for each bin. Finally, as mentioned earlier, we omitted implementing a lookahead terminal which would reduce the number of Earley items and thus improve performance by pruning dead end derivations. Note that, although these performance improvements might speed up the algorithm quite considerably, particularly the lookahead terminal, none of them improve the worst case running time.

We decided against implementing the map-based functional approach with a running time of $\mathcal{O}(n^3 \log n)$ and 'settle' for the current approach with a running time of $\mathcal{O}(n^4)$ due to two reasons. The map-based functional approach is more complicated and the improvement of the running time, although significant, still does not reach the optimum. If we optimize our approach only to achieve better performance, we would like to achieve optimal performance, at least asymptotically. Furthermore, the current approach, appending items to the list and using natural numbers as pointers, maps more easily to the imperative approach. And our original intention was to refine the algorithm once more to an imperative version. Unfortunately, this exceeded the scope of this thesis but is worthwhile future work.

## 4.3 Sets or Bins as Lists

In this section we prove that the list representation of bins, in particular updating a bin or bins with the functions *bin-upd*, *bin-upds*, and *bins-upd*, fulfills the required set semantics. We define a function *bins* that accumulates all bins into one set of Earley items. Note that a call of the form *Earley-bin-list'* $k\ \mathcal{G}\ \omega\ bs\ i$ iterates through the entries of the $k$-th bin or the current worklist in ascending order starting at index $i$. All items at indices $j < i$ are untouched and thus should already have been processed accordingly. We make two further definitions capturing the set of items which should already be 'done'. The term *bin-upto b i* represents the items of a bin $b$ up to but not including the $i$-th index. Similarly, function *bins-upto* computes the set of items consisting of the $k$-th bin up to but not including the $i$-th index and the items of all previous bins.

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

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

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

The next six lemmas then prove the set semantics of updating one bin with one item (*bin-upd*), multiple items (*bin-upds*), or updating a particular bin with multiple items (*bins-upd*). The proofs are straightforward and respectively by induction on the bin $b$ for an arbitrary item $e$, by induction on the items $es$ to be inserted for an arbitrary bin $b$, or by definition of *bin-upds* and *bins*, each time using previously proven lemmas in the appropriate proofs.

**lemma** *set-items-bin-upd*:
$\mathit{set}\ (\mathit{items}\ (\mathit{bin\text{-}upd}\ e\ b)) = \mathit{set}\ (\mathit{items}\ b) \cup \{\ \mathit{item}\ e\ \}$

**lemma** *distinct-bin-upd*:
**assumes** $\mathit{distinct}\ (\mathit{items}\ b)$
**shows** $\mathit{distinct}\ (\mathit{items}\ (\mathit{bin\text{-}upd}\ e\ b))$

**lemma** *set-items-bin-upds*:
$\mathit{set}\ (\mathit{items}\ (\mathit{bin\text{-}upds}\ \mathit{es}\ b)) = \mathit{set}\ (\mathit{items}\ b) \cup \mathit{set}\ (\mathit{items}\ \mathit{es})$

**lemma** *distinct-bin-upds*:
**assumes** $\mathit{distinct}\ (\mathit{items}\ b)$
**shows** $\mathit{distinct}\ (\mathit{items}\ (\mathit{bin\text{-}upds}\ \mathit{es}\ b))$

**lemma** *bins-bins-upd*:
**assumes** $k < |\mathit{bs}|$
**shows** $\mathit{bins}\ (\mathit{bins\text{-}upd}\ \mathit{bs}\ k\ \mathit{es}) = \mathit{bins}\ \mathit{bs} \cup \mathit{set}\ (\mathit{items}\ \mathit{es})$

**lemma** *distinct-bins-upd*:
**assumes** $\mathit{distinct}\ (\mathit{items}\ (\mathit{bs}!k))$
**shows** $\mathit{distinct}\ (\mathit{items}\ (\mathit{bins\text{-}upd}\ \mathit{bs}\ k\ \mathit{es}\ !\ k))$

In our formalization we prove further basic lemmas about functions *bin-upd*, *bin-upds*, and *bins-upd*. In particular how updating bins changes the length of a bin, interacts with indexing into a bin or does not change the ordering of the items in a bin. Furthermore, we prove similar lemmas about functions *bin-upto* and *bins-upto* and their interplay with bin(s) updates. We omit them for brevity.

## 4.4 Well-formedness and Termination

We also need to refine the notion of well-formed items to well-formed *bin* items. An item is a well-formed bin item for the $k$-th bin if it is a well-formed item and its end index coincides with $k$. We call a bin well-formed if it only contains well-formed bin items and its items are distinct, and lift this notion of well-formedness to the toplevel list of bins.

**definition** *wf-bin-item* :: $'a\ cfg \Rightarrow\ 'a\ sentential \Rightarrow nat \Rightarrow\ 'a\ item \Rightarrow bool$ **where**
  $\mathit{wf\text{-}bin\text{-}item}\ \mathcal{G}\ \omega\ k\ x \equiv \mathit{wf\text{-}item}\ \mathcal{G}\ \omega\ x \wedge \mathit{item\text{-}end}\ x = k$

**definition** *wf-bin-items* :: $'a\ cfg \Rightarrow\ 'a\ sentential \Rightarrow nat \Rightarrow\ 'a\ item\ list \Rightarrow bool$ **where**
  $\mathit{wf\text{-}bin\text{-}items}\ \mathcal{G}\ \omega\ k\ xs \equiv \forall\, x \in set\ xs.\ \mathit{wf\text{-}bin\text{-}item}\ \mathcal{G}\ \omega\ k\ x$

**definition** *wf-bin* :: $'a\ cfg \Rightarrow\ 'a\ sentential \Rightarrow nat \Rightarrow\ 'a\ bin \Rightarrow bool$ **where**
  $\mathit{wf\text{-}bin}\ \mathcal{G}\ \omega\ k\ b \equiv distinct\ (items\ b) \wedge \mathit{wf\text{-}bin\text{-}items}\ \mathcal{G}\ \omega\ k\ (items\ b)$

**definition** *wf-bins* :: $'a\ cfg \Rightarrow\ 'a\ list \Rightarrow\ 'a\ bins \Rightarrow bool$ **where**
  $\mathit{wf\text{-}bins}\ \mathcal{G}\ \omega\ bs \equiv \forall\, k < |bs|.\ \mathit{wf\text{-}bin}\ \mathcal{G}\ \omega\ k\ (bs!k)$

Next we prove that inserting well-formed bin items maintains the well-formedness of a bin or bins. The proofs are structurally analogous to those of Section 4.3.

**lemma** *wf-bin-bin-upd*:
  **assumes** $\mathit{wf\text{-}bin}\ \mathcal{G}\ \omega\ k\ b$
  **assumes** $\mathit{wf\text{-}bin\text{-}item}\ \mathcal{G}\ \omega\ k\ (item\ e)$
  **shows** $\mathit{wf\text{-}bin}\ \mathcal{G}\ \omega\ k\ (\mathit{bin\text{-}upd}\ e\ b)$

**lemma** *wf-bin-bin-upds*:
  **assumes** $\mathit{wf\text{-}bin}\ \mathcal{G}\ \omega\ k\ b$
  **assumes** $\forall\, x \in set\ (items\ es).\ \mathit{wf\text{-}bin\text{-}item}\ \mathcal{G}\ \omega\ k\ x$
  **assumes** $distinct\ (items\ es)$
  **shows** $\mathit{wf\text{-}bin}\ \mathcal{G}\ \omega\ k\ (\mathit{bin\text{-}upds}\ es\ b)$

**lemma** *wf-bins-bins-upd*:
  **assumes** $\mathit{wf\text{-}bins}\ \mathcal{G}\ \omega\ bs$
  **assumes** $\forall\, x \in set\ (items\ es).\ \mathit{wf\text{-}bin\text{-}item}\ \mathcal{G}\ \omega\ k\ x$
  **assumes** $distinct\ (items\ es)$

**shows** *wf-bins* $\mathcal{G}$ $\omega$ (*bins-upd bs k es*)

At this point we would like to prove that function *Earley-bin-list'* also maintains the well-formedness of the bins. But since it is a partial function we first need to take a short excursion into function definitions in Isabelle. Intuitively, a recursive function terminates if for every recursive call the size of its input strictly decreases. And normally all functions defined in Isabelle must be total. But there are different ways to define a recursive function depending on the complexity of its termination: (1) with the *fun* keyword. Isabelle then tries to find a measure of the input that proves termination. If successful we obtain an induction schema corresponding to the function definition. (2) via the *function* keyword. We then need to define and prove a suitable measure by hand. (3) if the function is a partial function we need to define it with the keyword *partial-function*. For tail-recursive functions the definition is straightforward, otherwise we have to wrap the return type in an option to signal possible non-termination. But contrary to total functions we do *not* obtain the usual induction schema. To prove anything useful about a partial function we have to define the set of inputs and a corresponding measure for which the function terminates and subsequently prove an appropriate induction schema by hand.

As previously explained, in Section 4.1 we defined the function *Earley-bin-list'* as a partial function since a call of the form *Earley-bin-list'* $k$ $\mathcal{G}$ $\omega$ *bs* $i$ might never terminate if the function keeps appending arbitrary new items to the $k$-th bin it currently operates on. But we know that the newly generated items are not arbitrary but well-formed bin items. From lemma *finite-*$\mathcal{E}$*arley* of Chapter 3 we also know that the set of well-formed items is finite. Since we made sure to only add each item once to a bin, the function *Earley-bin-list'* will eventually run out of new items to insert into the bin it currently operates on and terminate.

In Isabelle we define the set of well-formed earley inputs as a set of quadruples consisting of the index $k$ of the current bin, the grammar $\mathcal{G}$, the input $\omega$, and the bins *bs*. Note that we not only require the bins to be well-formed but also suitable bounds on $k$ and the length of the bins to make sure that we are not indexing outside the input or the bins as well as a well-formed grammar to ensure we only generate well-formed bin items. We then define a suitable measure for the termination of *Earley-bin-list'* $k$ $\mathcal{G}$ $\omega$ *bs* $i$ that intuitively corresponds to the number of well-formed bin items that are still possible to generate from index $i$ onwards. Finally, we prove an induction schema (*earley induction*) for the function by complete induction on the measure of the input. We omit showing the schema explicitly since it is rather verbose. But intuitively it partitions the function into five cases: the base (*Base*) case where we have run out of items to operate on and terminate; one case for completion (*Complete*) and prediction (*Predict*) each; and two cases for scanning, covering the normal (*Scan*) and the special case (*Pass*) where $k$ exceeds the length of the input.

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

**fun** *earley-measure* :: $nat \times {}'a\ cfg \times {}'a\ sentential \times {}'a\ bins \Rightarrow nat \Rightarrow nat$ **where**
$earley\text{-}measure\ (k, \mathcal{G}, \omega, bs)\ i = card\ \{\ x \mid x.\ wf\text{-}bin\text{-}item\ \mathcal{G}\ \omega\ k\ x\ \} - i$

Concluding this section, we prove that we maintain the well-formedness of the input for the function *Earley-bin-list'*. The proof is by *earley induction*, lemma *wf-bins-bins-upd* and - straightforward and thus omitted - auxiliary lemmas stating that the scanning, predicting and completing only generates well-formed bin items. The proofs for functions *Earley-bin-list*, *Earley-list*, and *$\mathcal{E}$arley-list* are respectively by definition, by induction on $k$ using additionally the fact that the initial bins are well-formed, and once more by definition, each time using previously proven lemmas appropriately.

**lemma** *wf-earley-input-Earley-bin-list'*:
**assumes** $(k, \mathcal{G}, \omega, bs) \in wf\text{-}earley\text{-}input$
**shows** $(k, \mathcal{G}, \omega, Earley\text{-}bin\text{-}list'\ k\ \mathcal{G}\ \omega\ bs\ i) \in wf\text{-}earley\text{-}input$

**lemma** *wf-earley-input-Earley-bin-list*:
**assumes** $(k, \mathcal{G}, \omega, bs) \in wf\text{-}earley\text{-}input$
**shows** $(k, \mathcal{G}, \omega, Earley\text{-}bin\text{-}list\ k\ \mathcal{G}\ \omega\ bs) \in wf\text{-}earley\text{-}input$

**lemma** *wf-earley-input-Earley-list*:
**assumes** $wf\text{-}\mathcal{G}\ \mathcal{G}$
**assumes** $k \leq |\omega|$
**shows** $(k, \mathcal{G}, \omega, Earley\text{-}list\ k\ \mathcal{G}\ \omega) \in wf\text{-}earley\text{-}input$

**lemma** *wf-earley-input-$\mathcal{E}$arley-list*:
**assumes** $wf\text{-}\mathcal{G}\ \mathcal{G}$
**assumes** $k \leq |\omega|$
**shows** $(k, \mathcal{G}, \omega, \mathcal{E}arley\text{-}list\ \mathcal{G}\ \omega) \in wf\text{-}earley\text{-}input$

## 4.5 Soundness

Now we are ready to prove subsumption in both directions. Since functions *Earley-list* and *$\mathcal{E}$arley-list* are structurally identical to *Earley* respectively *$\mathcal{E}$arley*, the main task for the next two sections is to show that function *Earley-bin-list* or *Earley-bin-list'* computes the same items as the function *Earley-bin* that computes in turn the fixpoint of *Earley-step*. We start with the easier direction: every item generated by the list-based approach is also present in the set-based approach which implies soundness of the list-based algorithm. This is the 'easier' direction due to the fact that during execution of the body of *Earley-bin-list'* we only consider a single item $x$ in bin $k$ at position $i$ and apply the appropriate operation. In contrast, one execution of function *Earley-step* applies the scan, predict and complete operations for all previously computed items.

We start the soundness proof with three auxiliary lemmas proving subsumption of the three basic operations. The proofs of lemmas *Scan-list-sub-Scan*, *Predict-list-sub-Predict*, and *Complete-list-sub-Complete* are each straightforward by definition of the corresponding functions.

**lemma** *Scan-list-sub-Scan*:
**assumes** $\textit{wf-bins}\ \mathcal{G}\ \omega\ bs$
**assumes** $\textit{bins}\ bs \subseteq I$
**assumes** $k < |bs|\ k < |\omega|$
**assumes** $x \in \textit{set}\ (\textit{items}\ (bs!k))$
**assumes** $\textit{next-symbol}\ x = \textit{Some}\ a$
**shows** $\textit{set}\ (\textit{items}\ (\textit{Scan-list}\ k\ \omega\ a\ x\ pre)) \subseteq \textit{Scan}\ k\ \omega\ I$

**lemma** *Predict-list-sub-Predict*:
**assumes** $\textit{wf-bins}\ \mathcal{G}\ \omega\ bs$
**assumes** $\textit{bins}\ bs \subseteq I$
**assumes** $k < |bs|$
**assumes** $x \in \textit{set}\ (\textit{items}\ (bs!k))$
**assumes** $\textit{next-symbol}\ x = \textit{Some}\ N$
**shows** $\textit{set}\ (\textit{items}\ (\textit{Predict-list}\ k\ \mathcal{G}\ N)) \subseteq \textit{Predict}\ k\ \mathcal{G}\ I$

**lemma** *Complete-list-sub-Complete*:
**assumes** $\textit{wf-bins}\ \mathcal{G}\ \omega\ bs$
**assumes** $\textit{bins}\ bs \subseteq I$
**assumes** $k < |bs|$
**assumes** $x \in \textit{set}\ (\textit{items}\ (bs!k))$
**assumes** $\textit{is-complete}\ x$
**shows** $\textit{set}\ (\textit{items}\ (\textit{Complete-list}\ k\ x\ bs\ red)) \subseteq \textit{Complete}\ k\ I$

We then prove that all items generated by the function *Earley-bin-list'* are also present in the set produced by the function *Earley-bin*. The proof is by *earley induction* for an arbitrary set of items $I$. The cases *Base* and *Pass* are trivial. The other three cases follow the same structure and we only highlight the *Complete* case. Lemma *Earley-bin-list-sub-Earley-bin* follows from *Earley-bin-list'-sub-Earley-bin* by definition.

**lemma** *Earley-bin-list'-sub-Earley-bin*:
**assumes** $(k, \mathcal{G}, \omega, bs) \in \textit{wf-earley-input}$
**assumes** $\textit{bins}\ bs \subseteq I$

**shows** $bins\ (Earley\text{-}bin\text{-}list'\ k\ \mathcal{G}\ \omega\ bs\ i) \subseteq Earley\text{-}bin\ k\ \mathcal{G}\ \omega\ I$

*Proof.* We are in the case *Complete*. Hence, the item $x$ in the $k$-th bin at index $i$ is complete and the new bins $bs'$ are $bins\text{-}upd\ bs\ k\ (Complete\text{-}list\ k\ x\ bs\ i)$. We can discharge the assumptions of lemma *Complete-list-sub-Complete* by our assumptions of well-formed earley input and $bins\ bs \subseteq I$ and the additional assumption that we are in the *Complete* case, and have $bins\ bs' \subseteq I \cup Complete\ k\ I$. Since updating the bins maintains well-formedness of the input we can use the induction hypothesis and obtain the fact

$$bins\ (Earley\text{-}bin\text{-}list'\ k\ \mathcal{G}\ \omega\ bs\ i) \subseteq Earley\text{-}bin\ k\ \mathcal{G}\ \omega\ (I \cup Complete\ k\ I) \tag{1}$$

We also know that $I \cup Complete\ k\ I \subseteq Earley\text{-}bin\ k\ \mathcal{G}\ \omega\ I$ since *Earley-bin* is the fixpoint iteration of *Earley-step* that is in turn defined as $I \cup Scan\ k\ \omega\ I \cup Complete\ k\ I \cup Predict\ k\ \mathcal{G}\ I$. Moreover we know that function *Earley-bin* is monotonic in the set it operates on. And thus we have

$$Earley\text{-}bin\ k\ \mathcal{G}\ \omega\ (I \cup Complete\ k\ I) \subseteq Earley\text{-}bin\ k\ \mathcal{G}\ \omega\ (Earley\text{-}bin\ k\ \mathcal{G}\ \omega\ I) \tag{2}$$

The statement to proof follows from (1), (2) and the fact that the function *Earley-bin* is idempotent.

□

**lemma** *Earley-bin-list-sub-Earley-bin*:
 **assumes** $(k, \mathcal{G}, \omega, bs) \in wf\text{-}earley\text{-}input$
 **assumes** $bins\ bs \subseteq I$
 **shows** $bins\ (Earley\text{-}bin\text{-}list\ k\ \mathcal{G}\ \omega\ bs) \subseteq Earley\text{-}bin\ k\ \mathcal{G}\ \omega\ I$

We prove lemma *Earley-list-sub-Earley* by induction on $k$ using the additional lemma *Init-list-eq-Init*, that shows that the set of items created by the *Init-list* and *Init* functions are identical, and lemma *Earley-bin-list-sub-Earley-bin*. Lemma *$\mathcal{E}$arley-list-sub-$\mathcal{E}$arley* follows by definition and concludes the first half of the subsumption proof that implies soundness of the list-based implementation due to the soundness proof of the set of Earley items of Chapter 3 (lemma *sound-$\mathcal{E}$arley*).

**lemma** *Init-list-eq-Init*:
 **shows** $bins\ (Init\text{-}list\ \mathcal{G}\ \omega) = Init\ \mathcal{G}$

**lemma** *Earley-list-sub-Earley*:
 **assumes** $wf\text{-}\mathcal{G}\ \mathcal{G}\ k \leq |\omega|$
 **shows** $bins\ (Earley\text{-}list\ k\ \mathcal{G}\ \omega) \subseteq Earley\ k\ \mathcal{G}\ \omega$

**lemma** *$\mathcal{E}$arley-list-sub-$\mathcal{E}$arley*:
 **assumes** $wf\text{-}\mathcal{G}\ \mathcal{G}$
 **shows** $bins\ (\mathcal{E}arley\text{-}list\ \mathcal{G}\ \omega) \subseteq \mathcal{E}arley\ \mathcal{G}\ \omega$

## 4.6 Completeness

In this section we prove completeness of the list-based algorithm. The two main complications are the following. The function *Earley-bin-list′* starts it computation at a specific index $i$ in the $k$-th bin. In contrast, while completing the $k$-th bin, the set-based approach of Chapter 3 applies the function *Earley-step* in each iteration of the fixpoint computation to all items. Hence, we have to generalize the proofs such that all items at indices $j < i$ are already 'done'. The second problem is more severe: as stated the algorithm is incorrect, at least for some classes of grammars. In contrast to the fixpoint computation of the set-based approach the list-based implementation imposes an order on the creation of items, and sometimes order matters. Consider for example an item $A \to \bullet, i, j$, or an epsilon-rule $A \to \epsilon$, that the list-based implementation encounters during creation of bin $B_j$. Since the item is complete we apply the *Complete* operation. The algorithm first determines the origin bin $i$ of the item which always coincides with $j$ for epsilon rules. Consequently, we search the current bin $B_j$ for any items of the form $B \to \alpha \bullet A\beta, i', j$. But bin $B_j$ is only partially constructed at this point in time. Hence, we might be missing some of these items, either since they have not been predicted, or completed up to this point. Thus, if we apply the complete operation to item $A \to \bullet, i, j$ immediately we might not generate all items of the form $B \to \alpha A \bullet \beta, i', j$ and in turn not all items depending on those items. In essence, we might be missing potential derivation paths.

There exist various approaches to deal with this problem. Aho *et al* [**Aho:1972**] take a rather relaxed point of view and propose to keep interleaving the *Predict* and *Complete* operations until no more new items are being generated. Earley [**Earley:1970**] suggests to have the *Complete* operation note that we actually need to move the bullet over the non-terminal $A$ when encountering the item $A \to \bullet, i, j$, and taking this information into account in the subsequent execution of the algorithm. Or, in essence, delaying the *Complete* operation for item $A \to \bullet, i, j$ until we are sure that we have encountered all items of the form $B \to \alpha \bullet A\beta, i', j$. Earley suggests that the algorithm should keep an additional collection of non-terminals to look out for stored in an appropriate data structure. Aycock *et al* [**Aycock:2002**] propose yet another approach based on a slight modification of the *Predict* operation. Note that the problem during completion only arises if the non-terminal $A$ is nullable, or there exists a derivation such that $\mathcal{G} \vdash A \Rightarrow^* \varepsilon$. The authors suggest the following approach. Pre-compute nullable non-terminals using well-know approaches [**Appel:2003**][**Fischer:2009**]. If the algorithm encounters an item of the form $A \to \alpha \bullet B\beta, i, j$, predict items $B \to \bullet\gamma, j, j$ for each rule $B \to \gamma$ of the grammar $\mathcal{G}$. But additionally add the item $A \to \alpha B \bullet \beta, j, j$ if the non-terminal $B$ is nullable.

Interleaving prediction and completion until we generate no new items seems rather impractical in our opinion. Thus, we only considered the approaches of Earley and Aycock *et al.* Both ideas are straightforward to implement in the context of a pure recognizer. But complications arise when we need to annotate the items with the needed information to construct parse trees. For the approach of Earley it is no longer sufficient to keep solely a list of nullable non-terminals to look out for but we need to maintain additional information of the origin of these non-terminals to update the reduction and predecessor pointers accordingly. The approach of Aycock *et al* implies even more complications. For a pure recognizer they construct an LR(0) automaton for the modified *Predict* operation, but for an Earley parser they introduce a new type of automaton, a split-epsilon DFA, and also slightly rewrite the grammar into *nihilist normal form* to encode the necessary information to reconstruct derivations.

In the end, we decided against implementing any of the approaches above and follow the approach of Jones [**Jones:1972**]. We restrict the grammar. If we disallow any non-terminal to derive $\epsilon$, the problem does not arise in the first place. Our justification for this approach is that it is by far the simplest solution while still being practical and allowing a wide enough range of grammars to be supported.

Overall, our obligation for the remainder of the section is to prove that restricting the grammar to not contain empty derivations ensures that the order of constructing items does not matter in the end, and that the list-based approach covers the fixpoint computation of Chapter 3.

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

The core lemma is the following: if the grammar is well-formed and does not allow empty derivations, and a given item is well-formed, sound and complete, then its item origin and item end cannot coincide, which implies that the origin of the item is strictly smaller than the item end due to the well-formedness of the item. And consequently there do not exist any items of the form $A \to \epsilon, i, j$ in any bin $B_j$.

**lemma** *impossible-complete-item*:
  **assumes** *wf-*$\mathcal{G}$ $\mathcal{G}$
  **assumes** *nonempty-derives* $\mathcal{G}$
  **assumes** *wf-item* $\mathcal{G}\ \omega\ x$
  **assumes** *sound-item* $\mathcal{G}\ \omega\ x$
  **assumes** *is-complete* $x$
  **assumes** *item-origin* $x = k$
  **assumes** *item-end* $x = k$
  **shows** *False*

*Proof.* From assumptions *sound-item* $\mathcal{G}\ \omega\ x$, *is-complete* $x$, *item-origin* $x = k$, and *item-end* $x = k$ we have by definition of a sound and complete item that

$$\mathcal{G} \vdash \textit{item-rule-head } x \Rightarrow^* []$$

Since the grammar $\mathcal{G}$ and the item $x$ are well-formed, we also know that the item rule head of $x$ is indeed a non-terminal. The proof concludes by assumption *nonempty-derives* $\mathcal{G}$ by definition.

□

Lemma *Complete-Un-absorb* then captures the idea that it does not matter for the *Complete* operation if we add an additional item $z$ of the form $B \to \alpha \bullet A\beta, i, k$ to bin $B_k$ while constructing the $k$-th bin under the assumption of well-formedness and non-empty derivations.

**lemma** *Complete-Un-absorb*:
**assumes** *wf-*$\mathcal{G}$ $\mathcal{G}$
**assumes** *wf-items* $\mathcal{G}$ $\omega$ $I$
**assumes** *sound-items* $\mathcal{G}$ $\omega$ $I$
**assumes** *nonempty-derives* $\mathcal{G}$
**assumes** *wf-item* $\mathcal{G}$ $\omega$ $z$
**assumes** *item-end* $z = k$
**assumes** *next-symbol* $z =$ *Some* $A$
**shows** *Complete* $k$ $(I \cup \{z\}) =$ *Complete* $k$ $I$

*Proof.* Assume for the sake of contradiction that *Complete* $k$ $(I \cup \{z\}) \neq$ *Complete* $k$ $I$. Then we know that *Complete* $k$ $I \subset$ *Complete* $k$ $(I \cup \{z\})$ since the *Complete* operation is monotonic in $I$. Hence, there exist by definition of *Complete* items $x$, $x'$, and $y$ such that

$$\begin{aligned} x &\in \textit{Complete } k\ (I \cup z) \quad (1) & x &\notin \textit{Complete } k\ I \quad (2)\\ x' &\in \textit{bin } (I \cup \{z\})\ (\textit{item-origin } y) \quad (3) & \textit{next-symbol } x' &= \textit{Some } (\textit{item-rule-head } y) \quad (4)\\ y &\in \textit{bin } (I \cup \{z\})\ k \quad (5) & &\textit{is-complete } y \quad (6)\\ x &= \textit{inc-item } x'\ k \quad (7) \end{aligned}$$

From assumptions (2-7) and the definition of *Complete* we need to consider two cases:

- $z = x'$: Due to assumption *item-end* $z = k$ and (3,5) we know that the item origin and end of item $y$ is $k$. Additionally, the item is sound and well-formed due to assumptions (5,6) and *wf-items* $\mathcal{G}$ $\omega$ $I$, *wf-item* $\mathcal{G}$ $\omega$ $z$, and *sound-items* $\mathcal{G}$ $\omega$ $I$, *next-symbol* $z =$ *Some* $A$. Moreover, using assumptions *wf-*$\mathcal{G}$ $\mathcal{G}$ and *nonempty-derives* $\mathcal{G}$ and the fact that $y$ is complete (6), we can discharge the assumptions of lemma *impossible-complete-item* and arrive at a contradiction.

- $z = y$: Thus we know that $z$ must be complete since $y$ is complete by (6). But we also know that *next-symbol* $z$ = *Some* $A$, a contradiction.

□

Next we prove that the items generated by function *Earley-bin-list'* cover the items generated by a single *Earley-step*. Note the assumption *Earley-step* $k$ $\mathcal{G}$ $\omega$ (*bins-upto bs k i*) $\subseteq$ *bins bs* stating that all items up to index $i$ can already considered to be 'done' or applying the function *Earley-step* to any of those items does not change the bins *bs*. This assumption is necessary since a call of the form *Earley-bin-list'* $k$ $\mathcal{G}$ $\omega$ *bs* $i$ intuitively skips the first $i - 1$ items, as mentioned previously. The proof is by *earley induction* and we only highlight the *Predict* case where we need lemma *Complete-Un-absorb*. The other cases are similar in overall structure. Lemma *Earley-step-sub-Earley-bin-list* then follows once more by definition.

**lemma** *Earley-step-sub-Earley-bin-list'*:
  **assumes** ($k$, $\mathcal{G}$, $\omega$, *bs*) $\in$ *wf-earley-input*
  **assumes** *sound-items* $\mathcal{G}$ $\omega$ (*bins bs*)
  **assumes** *is-sentence* $\mathcal{G}$ $\omega$
  **assumes** *nonempty-derives* $\mathcal{G}$
  **assumes** *Earley-step* $k$ $\mathcal{G}$ $\omega$ (*bins-upto bs k i*) $\subseteq$ *bins bs*
  **shows** *Earley-step* $k$ $\mathcal{G}$ $\omega$ (*bins bs*) $\subseteq$ *bins* (*Earley-bin-list'* $k$ $\mathcal{G}$ $\omega$ *bs* $i$)

*Proof.* We are only highlighting the *Predict* case. Hence, we are currently considering an item $x$ in the $k$-th bin at index $i$ whose next symbol is some non-terminal $N$. Let $bs'$ denote the updated bins or *bins-upd bs k* (*Predict-list* $k$ $\mathcal{G}$ $N$). We know that the function *bins-upd* maintains well-formedness and soundness of the items, but to apply our induction hypothesis we need to proof one additional statement:

$$\textit{Earley-step}\ k\ \mathcal{G}\ \omega\ (\textit{bins-upto}\ bs'\ k\ (i + 1)) \subseteq \textit{bins}\ bs'$$

Since *Earley-step* is defined as the union of the basic three operations we split this proof into these three cases:

- *Scan* $k$ $\omega$ (*bins-upto* $bs'$ $k$ ($i$ + 1)) $\subseteq$ *bins* $bs'$:

$$\begin{aligned}
&\textit{Scan}\ k\ \omega\ (\textit{bins-upto}\ bs'\ k\ (i + 1)) \\
&\qquad = \textit{Scan}\ k\ \omega\ (\textit{bins-upto}\ bs'\ k\ i \cup \{\textit{items}\ (bs'\ !\ k)\ !\ i\}) && (1)\\
&\qquad = \textit{Scan}\ k\ \omega\ (\textit{bins-upto}\ bs\ k\ i \cup \{x\}) && (2)\\
&\qquad \subseteq \textit{bins}\ bs \cup \textit{Scan}\ k\ \omega\ \{x\} && (3)\\
&\qquad = \textit{bins}\ bs && (4)\\
&\qquad \subseteq \textit{bins}\ bs' && (5)
\end{aligned}$$

(1) by definition of *bins-upto*. (2) function *bins-upd* does not change the order of the items of bin $k$ upto and including index $i$. (3) function *Scan* distributes over set union, assumption *Earley-step* $k\ \mathcal{G}\ \omega\ (\textit{bins-upto bs k i}) \subseteq \textit{bins bs}$ and the definition of *Earley-step*. (4) the next symbol of $x$ is the non-terminal $N$ and thus the *Scan* operation yield an empty set. (5) the set semantics of function *bins-upd*.

- $\textit{Predict}\ k\ \mathcal{G}\ (\textit{bins-upto}\ bs'\ k\ (i + 1)) \subseteq \textit{bins}\ bs'$

$$
\begin{aligned}
&\textit{Predict}\ k\ \mathcal{G}\ (\textit{bins-upto}\ bs'\ k\ (i + 1)) \\
&\quad = \textit{Predict}\ k\ \mathcal{G}\ (\textit{bins-upto}\ bs'\ k\ i \cup \{\textit{items}\ (bs'\ !\ k)\ !\ i\}) && (1)\\
&\quad = \textit{Predict}\ k\ \mathcal{G}\ (\textit{bins-upto}\ bs\ k\ i \cup \{x\}) && (2)\\
&\quad \subseteq \textit{bins}\ bs \cup \textit{Predict}\ k\ \mathcal{G}\ \{x\} && (3)\\
&\quad = \textit{bins}\ bs \cup \textit{set}\ (\textit{items}\ (\textit{Predict-list}\ k\ \mathcal{G}\ N)) && (4)\\
&\quad \subseteq \textit{bins}\ bs' && (5)
\end{aligned}
$$

(1-3,5) are identical to the first case. (4) the next symbol of $x$ is the non-terminal $N$ and thus the list-based implementation yields the same items as the set-based implementation.

- $\textit{Complete}\ k\ (\textit{bins-upto}\ bs'\ k\ (i + 1)) \subseteq \textit{bins}\ bs'$

$$
\begin{aligned}
&\textit{Complete}\ k\ (\textit{bins-upto}\ bs'\ k\ (i + 1)) \\
&\quad = \textit{Complete}\ k\ (\textit{bins-upto}\ bs'\ k\ i \cup \{\textit{items}\ (bs'\ !\ k)\ !\ i\}) && (1)\\
&\quad = \textit{Complete}\ k\ (\textit{bins-upto}\ bs\ k\ i \cup \{x\}) && (2)\\
&\quad = \textit{Complete}\ k\ (\textit{bins-upto}\ bs\ k\ i) && (3)\\
&\quad \subseteq \textit{bins}\ bs && (4)\\
&\quad \subseteq \textit{bins}\ bs' && (5)
\end{aligned}
$$

(1-2,5) are identical to the first case. (3) by lemma *Complete-Un-absorb* using the well-formedness, soundness, non-empty derivation assumptions and the fact that the item $x$ is in the $k$-th bin and its next symbol is the non-terminal $N$. (4) by assumption *Earley-step* $k\ \mathcal{G}\ \omega\ (\textit{bins-upto bs k i}) \subseteq \textit{bins bs}$ and the definition of *Earley-step*.

□

**lemma** *Earley-step-sub-Earley-bin-list*:
**assumes** $(k, \mathcal{G}, \omega, bs) \in \textit{wf-earley-input}$
**assumes** *sound-items* $\mathcal{G}\ \omega\ (\textit{bins bs})$

**assumes** *is-sentence* $\mathcal{G}$ $\omega$
**assumes** *nonempty-derives* $\mathcal{G}$
**assumes** *Earley-step* $k$ $\mathcal{G}$ $\omega$ (*bins-upto* *bs* $k$ *0*) $\subseteq$ *bins bs*
**shows** *Earley-step* $k$ $\mathcal{G}$ $\omega$ (*bins bs*) $\subseteq$ *bins* (*Earley-bin-list* $k$ $\mathcal{G}$ $\omega$ *bs*)

We have proven that the items generated by the execution of the list-based approach cover *one* single step of the set-based approach. Our next objective is to generalize this statement to the whole fixpoint computation, or an arbitrary number of steps. We need two, albeit small, quite technical lemmas, proving that the function *Earley-bin-list* is idempotent. This follows from the next lemma which states that when we execute the function *Earley-bin-list'* two times, passing as the argument for the bins of the second round the result of the first round, and are starting the execution from possibly different initial indices the result of the smaller index prevails. The intuition is clear: if we run through the worklist starting from index $i \leq j$, starting a second time from index $j$ does not yield any new items, since we already covered all items of the second execution in the first turn and repeated computations are void due to the assumption of non-empty derivations. The proof is by *earley induction* for arbitrary $j$ and once more utilizes lemma *impossible-complete-item*, we omit showing any details.

**lemma** *Earley-bin-list'-idem*:
**assumes** ($k$, $\mathcal{G}$, $\omega$, *bs*) $\in$ *wf-earley-input*
**assumes** *sound-items* $\mathcal{G}$ $\omega$ (*bins bs*)
**assumes** *nonempty-derives* $\mathcal{G}$
**assumes** $i \leq j$
**shows** *bins* (*Earley-bin-list'* $k$ $\mathcal{G}$ $\omega$ (*Earley-bin-list'* $k$ $\mathcal{G}$ $\omega$ *bs* $i$) $j$) =
*bins* (*Earley-bin-list'* $k$ $\mathcal{G}$ $\omega$ *bs* $i$)

**lemma** *Earley-bin-list-idem*:
**assumes** ($k$, $\mathcal{G}$, $\omega$, *bs*) $\in$ *wf-earley-input*
**assumes** *sound-items* $\mathcal{G}$ $\omega$ (*bins bs*)
**assumes** *nonempty-derives* $\mathcal{G}$
**shows** *bins* (*Earley-bin-list* $k$ $\mathcal{G}$ $\omega$ (*Earley-bin-list* $k$ $\mathcal{G}$ $\omega$ *bs*)) = *bins* (*Earley-bin-list* $k$ $\mathcal{G}$ $\omega$ *bs*)

Lemma *Earley-bin-sub-Earley-bin-list* concludes the subsumption proof for a single bin. Since the function *Earley-bin* is defined as the fixpoint of the function *Earley-step* and the fact that $x \in$ *limit* $f$ $X \equiv \exists\, n.\ x \in$ *funpower* $f$ $n$ $X$ the core proof is by induction on the computation of *funpower*.

**lemma** *Earley-bin-sub-Earley-bin-list*:
**assumes** ($k$, $\mathcal{G}$, $\omega$, *bs*) $\in$ *wf-earley-input*
**assumes** *sound-items* $\mathcal{G}$ $\omega$ (*bins bs*)
**assumes** *is-sentence* $\mathcal{G}$ $\omega$
**assumes** *nonempty-derives* $\mathcal{G}$
**assumes** *Earley-step* $k$ $\mathcal{G}$ $\omega$ (*bins-upto* *bs* $k$ *0*) $\subseteq$ *bins bs*

**shows** $Earley\text{-}bin\ k\ \mathcal{G}\ \omega\ (bins\ bs) \subseteq bins\ (Earley\text{-}bin\text{-}list\ k\ \mathcal{G}\ \omega\ bs)$

*Proof.* The goal is $funpower\ (Earley\text{-}step\ k\ \mathcal{G}\ \omega)\ (Suc\ n)\ (bins\ bs) \subseteq bins\ (Earley\text{-}bin\text{-}list\ k\ \mathcal{G}\ \omega\ bs)$.

For the base case we have $funpower\ (Earley\text{-}step\ k\ \mathcal{G}\ \omega)\ 0\ (bins\ bs) = bins\ bs$. And we conclude the proof due to the fact that the function *Earley-bin-list* is monotonic in the bins.

For the induction step we first need to proof a necessary precondition for our induction hypothesis:

$$\begin{aligned}
&Earley\text{-}step\ k\ \mathcal{G}\ \omega\ (bins\text{-}upto\ (Earley\text{-}bin\text{-}list\ k\ \mathcal{G}\ \omega\ bs)\ k\ 0) \\
&\quad = Earley\text{-}step\ k\ \mathcal{G}\ \omega\ (bins\text{-}upto\ bs\ k\ 0) && (1)\\
&\quad \subseteq bins\ bs && (2)\\
&\quad \subseteq bins\ (Earley\text{-}bin\text{-}list\ k\ \mathcal{G}\ \omega\ bs) && (3)
\end{aligned}$$

(1) $Earley\text{-}bin\text{-}list\ k\ \mathcal{G}\ \omega\ bs$ does not change the contents of any bins $B_j$ where $j < k$ by definition of *bins-upto*. (2) by assumption. (3) function *Earley-bin-list* only adds to the bins.

$$\begin{aligned}
&funpower\ (Earley\text{-}step\ k\ \mathcal{G}\ \omega)\ (Suc\ n)\ (bins\ bs) \\
&\quad = Earley\text{-}step\ k\ \mathcal{G}\ \omega\ (funpower\ (Earley\text{-}step\ k\ \mathcal{G}\ \omega)\ n\ (bins\ bs)) && (1)\\
&\quad \subseteq Earley\text{-}step\ k\ \mathcal{G}\ \omega\ (bins\ (Earley\text{-}bin\text{-}list\ k\ \mathcal{G}\ \omega\ bs)) && (2)\\
&\quad \subseteq bins\ (Earley\text{-}bin\text{-}list\ k\ \mathcal{G}\ \omega\ (Earley\text{-}bin\text{-}list\ k\ \mathcal{G}\ \omega\ bs)) && (3)\\
&\quad \subseteq bins\ (Earley\text{-}bin\text{-}list\ k\ \mathcal{G}\ \omega\ bs) && (4)
\end{aligned}$$

(1) by definition of *funpower*. (2) by induction hypothesis and fact that the function *Earley-step* is monotonic in the set of items. (3) by lemma *Earley-step-sub-Earley-bin-list* using well-formedness, soundness, non-empty derivations assumptions. (4) by lemma *Earley-bin-list-idem* using once more the soundness and non-empty derivation assumptions. □

We finish the subsumption proof with lemmas *Earley-sub-Earley-list* and *$\mathcal{E}$arley-sub-$\mathcal{E}$arley-list*. The proofs are respectively by induction on $k$ using lemmas *Init-list-eq-Init* and *Earley-bin-sub-Earley-bin-list*, and once more by definition using the previous lemma.

**lemma** *Earley-sub-Earley-list*:
  **assumes** $wf\text{-}\mathcal{G}\ \mathcal{G}$
  **assumes** $is\text{-}sentence\ \mathcal{G}\ \omega$
  **assumes** $nonempty\text{-}derives\ \mathcal{G}$

**assumes** $k \leq |\omega|$
**shows** *Earley* $k\ \mathcal{G}\ \omega \subseteq$ *bins* (*Earley-list* $k\ \mathcal{G}\ \omega$)

**lemma** $\mathcal{E}$*arley-sub-*$\mathcal{E}$*arley-list*:
**assumes** *wf-*$\mathcal{G}\ \mathcal{G}$
**assumes** *is-sentence* $\mathcal{G}\ \omega$
**assumes** *nonempty-derives* $\mathcal{G}$
**shows** $\mathcal{E}$*arley* $\mathcal{G}\ \omega \subseteq$ *bins* ($\mathcal{E}$*arley-list* $\mathcal{G}\ \omega$)

## 4.7 Correctness

We conclude the chapter presenting the final correctness theorem stating that there exists a finished item in the bins generated by the list-based implementation if and only if there exists a derivation of the input from the start symbol of the grammar. The proof is by lemmas *correctness-*$\mathcal{E}$*arley*, $\mathcal{E}$*arley-list-sub-*$\mathcal{E}$*arley*, and $\mathcal{E}$*arley-sub-*$\mathcal{E}$*arley-list*.

**theorem** *correctness-*$\mathcal{E}$*arley-list*:
**assumes** *wf-*$\mathcal{G}\ \mathcal{G}$
**assumes** *is-sentence* $\mathcal{G}\ \omega$
**assumes** *nonempty-derives* $\mathcal{G}$
**shows** *recognizing* (*bins* ($\mathcal{E}$*arley-list* $\mathcal{G}\ \omega$)) $\mathcal{G}\ \omega \longleftrightarrow \mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$

# 5 Earley Parser Implementation

Although a recognizer is a useful tool, for most practical applications we would like to not only know if the language specified by the grammar accepts the input, but we also want to obtain additional information of how the input can be derived in the form of parse trees. In particular, for our running example, the grammar $S ::= S + S \mid x$ and the input $\omega = x + x + x$, we want to obtain the two possible parse trees illustrated in Figures 5.1 and 5.2. But constructing all possible parse trees at once is no trivial task.

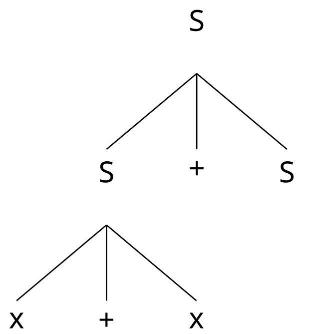


Figure 5.1: Parse Tree: $\omega = (x + x) + x$

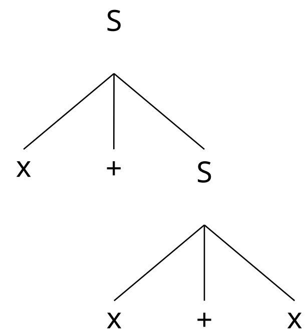


Figure 5.2: Parse Tree: $\omega = x + (x + x)$

Earley [**Earley:1970**] extends his recognizer to a parser by adding the following pointers. If the algorithm performs a completion and constructs an item $B \to \alpha A \bullet \beta, i, k$, it adds a pointer from the *instance of the non-terminal* $A$ to the complete item $A \to \gamma \bullet, j, k$. If there exists more than one possible way to complete the non-terminal $A$ and obtain the item $B \to \alpha A \bullet \beta, i, k$, then multiple pointers originate from the instance of the non-terminal $A$. Annotating every non-terminal of the right-hand side of the item $A \to \gamma \bullet, j, k$ recursively with pointers thus represents the derivation trees for the non-terminal $A$. Finally, after termination of the algorithm, the non-terminal that represents the start symbol contains pointers representing all possible derivation trees.

Note that Earley's pointers connect instances of non-terminals, but Tomita [**Tomita:1985**] showed that this approach is incorrect and may lead to spurious derivations in certain cases. Scott [**Scott:2008**] presents an example for the grammar $S ::= SS \mid x$ and the input $\omega = xxx$. Earley's parser correctly constructs the parse trees for the input but additionally returns erroneous parse trees representing derivations of $xx$ and $xxxx$. The problem lies in the fact that left- and rightmost derivations are intertwined when they should not be, since pointers originate from instances of non-terminals and don't connect Earley items.

In this chapter we develop an efficient functional algorithm constructing a single parse tree in Section 5.1 and prove its correctness. In Section 5.2 we generalize this approach, introducing a data structure representing all possible parse trees as a parse forest, adjusting the parse tree algorithm to compute such a forest, prove termination and soundness of the algorithm, and discuss the missing completeness proof. Finally, in Section 5.3 we summarize lessons learned from our approach and highlight a different data representation and implementation approaches for parse forests, comparing our approach to the algorithms of Scott [**Scott:2008**].

## 5.1 A Single Parse Tree

The data structure *tree* represents parse trees as shown in Figures 5.1 and 5.2. A *Leaf* always contains a single symbol (either terminal or non-terminal for partial derivation trees), a *Branch* consists of one non-terminal symbol and a list of subtrees. The function *root-tree* returns the symbol of the root of the parse tree. The yield of a leaf is its single symbol; to compute the yield for a branch with subtrees *ts* we apply the function *yield-tree* recursively and concatenate the results.

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

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

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

We introduce three notions of well-formedness for parse trees:

- *wf-rule-tree*: A parse tree must represent a valid derivation tree according to the grammar $\mathcal{G}$. A leaf of a parse tree is always well-formed by construction. For each branch there has to exist a production rule of the grammar $\mathcal{G}$ such that the root of this branch matches the left-hand side of the production rule, each subtree is in turn well-formed, and the roots of the subtrees coincide with the right-hand side of the production rule.

- *wf-item-tree*: Each branch corresponds to an Earley item $N \rightarrow \alpha \bullet \beta, i, j$ such that the roots of the subtrees match the prefix $\alpha$ up to the bullet of the right-hand side of the item's production rule. Note that a branch is only well-formed according to the grammar $\mathcal{G}$ if the roots of the subtrees form a *complete* right-hand side of a production rule. In contrast, a branch is well-formed according to an item if the roots of the subtrees are equal to $\alpha$, or, since we assume that Earley items are themselves well-formed, a *prefix* of a right-hand side of a production rule.
- *wf-yield-tree*: For an item $N \rightarrow \alpha \bullet \beta, i, j$ the yield of a parse tree has to match the substring $\omega[i..j\rangle$ of the input.

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

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

**definition** *wf-yield-tree* :: $'a\ sentential \Rightarrow\ 'a\ item \Rightarrow\ 'a\ tree \Rightarrow bool$ **where**
  *wf-yield-tree* $\omega\ x\ t \equiv yield\text{-}tree\ t = \omega[item\text{-}origin\ x..item\text{-}end\ x\rangle$

### 5.1.1 Pointer Lemmas

In Chapter 4 we extended the algorithm of Chapter 3 in two orthogonal ways: implementing sets as lists and adding the additional information to construct parse trees in the form of null, predecessor, and predecessor/reduction pointers. But we did not formally define the semantics of these pointers nor prove anything about their construction. In the following we define and proof soundness of the pointers.

- A null pointer *Null* of an entry is sound if it *predicts* the item $x$ of the entry, or the bullet of $x$ is at the beginning of the right-hand side of its production rule and we have not yet scanned any subsequence of the input, or item end and origin are identical.
- A predecessor pointer *Pre pre* of an entry $e$ is sound for the input $\omega$, bins $bs$, and the index of the current bin $k$ if $k > 0$, the predecessor index does not exceed the length of the predecessor bin at index $k-1$, and the predecessor item in bin $k-1$

at index *pre* *scans* the item of the entry $e$. An item $x'$ *scans* item $x$ for index $k$ if the next symbol of $x'$ coincides with the terminal symbol at index $k-1$ in the input $\omega$ and the item $x$ can be obtained the function call by *inc-item* $x'$ $k$.

- Finally, we define the soundness of a pointer *PreRed p ps* of an entry $e$ for each predecessor/reduction triple $(k', \mathit{pre}, \mathit{red}) \in \mathit{set}\ (p\ \#\ ps)$. The index $k'$ of the predecessor bin must be strictly smaller than $k$, and both the predecessor and the reduction index must be within the bounds of their respective bins, or bin $k'$ and $k$. Additionally, predicate *completes* holds for $k$, the predecessor item $x'$, the item $x$ of entry $e$ and the reduction item $y$, capturing the semantics of the *Complete* operation: the next symbol of $x'$ is the non-terminal $N$ which coincides with the item rule head of $y$. Furthermore, the item $y$ is complete and the origin index of $y$ aligns with the end index of $x'$. Finally, item $x$ can be obtained once more by the function call *inc-item* $x'$ $k$.

**definition** *predicts* :: *'a item* ⇒ *bool* **where**
*predicts x* ≡ *item-bullet x* = *0* ∧ *item-origin x* = *item-end x*

**definition** *sound-null-ptr* :: *'a entry* ⇒ *bool* **where**
*sound-null-ptr e* ≡ *pointer e* = *Null* ⟶ *predicts* (*item e*)

**definition** *scans* :: *'a sentential* ⇒ *nat* ⇒ *'a item* ⇒ *'a item* ⇒ *bool* **where**
*scans ω k x′ x* ≡ *x* = *inc-item x′ k* ∧ (∃ *a*. *next-symbol x′* = *Some a* ∧ *ω*!(*k*−*1*) = *a*)

**definition** *sound-pre-ptr* :: *'a sentential* ⇒ *'a bins* ⇒ *nat* ⇒ *'a entry* ⇒ *bool* **where**
*sound-pre-ptr ω bs k e* ≡ ∀ *pre*. *pointer e* = *Pre pre* ⟶
*k* > *0* ∧ *pre* < |*bs*!(*k*−*1*)| ∧
*scans ω k* (*item* (*bs*!(*k*−*1*)!*pre*)) (*item e*)

**definition** *completes* :: *nat* ⇒ *'a item* ⇒ *'a item* ⇒ *'a item* ⇒ *bool* **where**
*completes k x′ x y* ≡ *x* = *inc-item x′ k* ∧ *is-complete y* ∧ *item-origin y* = *item-end x′* ∧
(∃ *N*. *next-symbol x′* = *Some N* ∧ *N* = *item-rule-head y*)

**definition** *sound-prered-ptr* :: *'a bins* ⇒ *nat* ⇒ *'a entry* ⇒ *bool* **where**
*sound-prered-ptr bs k e* ≡ ∀ *p ps k′ pre red*. *pointer e* = *PreRed p ps* ∧
(*k′*, *pre*, *red*) ∈ *set* (*p*#*ps*) ⟶ *k′* < *k* ∧ *pre* < |*bs*!*k′*| ∧ *red* < |*bs*!*k*| ∧
*completes k* (*item* (*bs*!*k′*!*pre*)) (*item e*) (*item* (*bs*!*k*!*red*))

**definition** *sound-ptrs* :: *'a sentential* ⇒ *'a bins* ⇒ *bool* **where**
*sound-ptrs ω bs* ≡ ∀ *k* < |*bs*|. ∀ *e* ∈ *set* (*bs*!*k*).
*sound-null-ptr e* ∧ *sound-pre-ptr ω bs k e* ∧ *sound-prered-ptr bs k e*

We then prove the semantics of the pointers. The structure of the proofs is as usual:

we first prove pointer soundness for the most basic operations *bin-upd*, *bin-upds*, and *bins-upd*. Followed by the corresponding proofs for the computation of a single bin or functions *Earley-bin-list′* and *Earley-bin-list*. Finally, we prove that the initial bins only contain sound pointers, and functions *Earley-list* and $\mathcal{E}$*arley-list* maintain this property. Although it should be intuitively clear that the semantics of pointers hold, since the predicates *predicts*, *scans*, and *completes* basically restate the conditions of the operations *Predict*, *Scan*, and *Complete* or their list variations, and the bounds of the indices are sound by construction. But the proofs are surprisingly not trivial at all, especially the soundness proofs for functions *bin-upd* and *Earley-bin-list′*. The complexity mostly stems from the predecessor/reduction case that requires a quite significant amount of case splitting due to the indexing and dependence on the type of the pointers of the newly inserted items. Nonetheless, since the proofs do not reveal anything new in structure but are very technical, we only state them and omit going into any detail.

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

**lemma** *sound-ptrs-bin-upds*:
  **assumes** $k < |bs|$
  **assumes** *distinct* $(items\ (bs!k))$
  **assumes** *distinct* $(items\ es)$
  **assumes** *sound-ptrs* $\omega\ bs$
  **assumes** $\forall\, e \in set\ es.\ \textit{sound-null-ptr}\ e \wedge \textit{sound-pre-ptr}\ \omega\ bs\ k\ e \wedge \textit{sound-prered-ptr}\ bs\ k\ e$
  **shows** *sound-ptrs* $\omega\ (bs[k := \textit{bin-upds}\ es\ (bs!k)])$

**lemma** *sound-ptrs-Earley-bin-list′*:
  **assumes** $(k, \mathcal{G}, \omega, bs) \in \textit{wf-earley-input}$
  **assumes** *nonempty-derives* $\mathcal{G}$
  **assumes** *sound-items* $\mathcal{G}\ \omega\ (bins\ bs)$
  **assumes** *sound-ptrs* $\omega\ bs$
  **shows** *sound-ptrs* $\omega\ (\textit{Earley-bin-list}'\ k\ \mathcal{G}\ \omega\ bs\ i)$

**lemma** *sound-ptrs-Earley-bin-list*:
  **assumes** $(k, \mathcal{G}, \omega, bs) \in \textit{wf-earley-input}$
  **assumes** *nonempty-derives* $\mathcal{G}$
  **assumes** *sound-items* $\mathcal{G}\ \omega\ (bins\ bs)$
  **assumes** *sound-ptrs* $\omega\ bs$

**shows** *sound-ptrs* $\omega$ (*Earley-bin-list* $k$ $\mathcal{G}$ $\omega$ *bs*)

**lemma** *sound-ptrs-Init-list*:
**shows** *sound-ptrs* $\omega$ (*Init-list* $\mathcal{G}$ $\omega$)

**lemma** *sound-ptrs-Earley-list*:
**assumes** *wf-*$\mathcal{G}$ $\mathcal{G}$
**assumes** *nonempty-derives* $\mathcal{G}$
**assumes** $k \leq |\omega|$
**shows** *sound-ptrs* $\omega$ (*Earley-list* $k$ $\mathcal{G}$ $\omega$)

**lemma** *sound-ptrs-*$\mathcal{E}$*arley-list*:
**assumes** *wf-*$\mathcal{G}$ $\mathcal{G}$
**assumes** *nonempty-derives* $\mathcal{G}$
**shows** *sound-ptrs* $\omega$ ($\mathcal{E}$*arley-list* $\mathcal{G}$ $\omega$)

### 5.1.2 A Parse Tree Algorithm

After execution of the $\mathcal{E}$*arley-list* algorithm we obtain bins representing the complete set of Earley items. The null, predecessor, and predecessor/reduction pointers provide a way to navigate between items or through these bins, and, since they are sound, a way to construct derivation trees. The function *build-tree'* constructs a *single* parse tree corresponding to the item $x$ of entry $e$ at index $i$ of the $k$-th bin according to the well-formedness definitions from the beginning of this section.

If the pointer of entry $e$ is a null pointer, the algorithm starts building the tree rooted at the left-hand side non-terminal $N$ of the production rule of the item $x$ by constructing an initially empty branch containing the non-terminal $N$ and an empty list of subtrees. If the algorithm encounters a predecessor pointer *Pre pre*, it first recursively calls itself, for the previous bin $k-1$ and the predecessor index *pre*, obtaining a partial parse branch. Since the predecessor pointer is sound, in particular the *scans* predicate holds, we append a Leaf containing the terminal symbol at index $k-1$ of the input $\omega$ to the list of substrees of the branch. In the case that the pointer contains predecessor/reduction triples the algorithm only considers the first triple ($k'$, *pre*, *red*) due to the fact that we are only constructing a single derivation tree. As for the predecessor case, it recursively calls itself obtaining a partial derivation tree *Branch N ts* for the predecessor index *pre* and bin $k'$, followed by yet another recursive call for the reduction item at the reduction index *red* in the current bin $k$, constructing a complete derivation tree $t$. This time the *completes* predicate holds, thus the next symbol of the predecessor item coincides with the item rule head of the reduction item, or we are allowed to append the complete tree $t$ to the list of substrees *ts*.

Some minor implementation details to note are: the function *build-tree'* is a partial function, and not tail recursive, hence it has to return an optional value, as explained in Section 4.4. Furthermore, we are using the monadic do-notation commonly found in functional programming languages for the option monad. An alternative but equivalent implementation would use explicit case distinctions. Finally, if the function computes some value it is always a branch, never a single leaf.

**partial-function** (*option*) *build-tree'* :: *'a bins* ⇒ *'a sentential* ⇒ *nat* ⇒ *nat* ⇒ *'a tree option* **where**

```
build-tree' bs ω 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 ω (k−1) pre;
        case t of
          Branch N ts ⇒ Some (Branch N (ts @ [Leaf (ω!(k−1))]))
        | - ⇒ None })
  | PreRed (k', pre, red) - ⇒ (
      do {
        t ← build-tree' bs ω k' pre;
        case t of
          Branch N ts ⇒
            do {
              t ← build-tree' bs ω k red;
              Some (Branch N (ts @ [t]))
            }
        | - ⇒ None })))
```

The function *build-tree* computes a complete derivation tree if there exists one. It searches the last bin for any finished items or items of the form $S \to \gamma\bullet, 0, n$ where $S$ is the start symbol of the grammar $\mathcal{G}$ and $n$ denotes the length of the input $\omega$. If there exists such an item, it calls function *build-tree'* obtaining some parse tree representing the derivation $\mathcal{G} \vdash S \Rightarrow^* \omega$ (we will have to prove that this call never returns *None*), otherwise it returns *None* since there cannot exist a valid parse tree due to the correctness proof for the Earley bins of Chapter 3 if the argument *bs* was constructed by the $\mathcal{E}$*arley-list* function.

**definition** *build-tree* :: *'a cfg* ⇒ *'a sentential* ⇒ *'a bins* ⇒ *'a tree option* **where**

```
build-tree 𝒢 ω bs ≡
  let k = |bs| − 1 in (
  case filter-with-index (λx. is-finished 𝒢 ω x) (items (bs!k)) of
    [] ⇒ None
  | (-, i)#- ⇒ build-tree' bs ω k i)
```

### 5.1.3 Termination

The function *build-tree'* uses the null, predecessor and predecessor/reduction pointers to navigate through the given bins by calling itself recursively. Sound pointers ensure that we are not indexing outside of the bins, but this does not imply that the algorithm terminates. In the following we outline the cases for which it always terminates with some parse tree. Let's assume the function starts its computation at index $i$ of the $k$-th bin. If it encounters a null pointer, it terminates immediately. If the pointer is a simple predecessor pointer, it calls itself recursively for the previous bin. Due to the soundness of the predecessor pointer the index $k-1$ of this bin is strictly smaller than $k$. A similar argument holds for the first recursive call if the pointer is a predecessor/reduction pointer for the predecessor case, or for the index of the predecessor bin $k'$ we have $k' < k$. Consequently, we are following the pointers *strictly* back to the origin bin $B_0$ and thus must terminate at some point. But for the reduction pointer *red* we run into a problem: the recursive call for the item at index $i$ is in the same bin $k$. The entry at this position might again contain a reduction pointer *red'* leading to yet another recursive call for the same bin $k$, and so on. Hence, it is possible that we end up in a cycle of reductions and never terminate. Take for example the grammer $A ::= x \mid B$, $B ::= A$ and the input $\omega = x$. Table 5.1 illustrates the bins computed by the algorithm of Chapter 3. Bin $B_1$ contains the entry $B \to A\bullet, 0, 1; (0,2,0), (0,2,2)$ at index 1 and its second reduction triple $(0,2,2)$ a reduction pointer to index 2 of the same bin. There we find the entry $A \to B\bullet, 0, 1; (0,0,1)$ with a reduction pointer to index 1 completing the cycle. This is indeed valid since the grammar itself is cyclic, allowing for derivations of the form $A \to B \to A \to \cdots \to A \to x$.

Table 5.1: Cyclic reduction pointers

| | $B_0$ | $B_1$ |
|---|---|---|
| 0 | $A \to \bullet B, 0, 0; \perp$ | $A \to x\bullet, 0, 1; 1$ |
| 1 | $A \to \bullet x, 0, 0; \perp$ | $B \to A\bullet, 0, 1; (0,2,0), (0,2,2)$ |
| 2 | $B \to \bullet A, 0, 0; \perp$ | $A \to B\bullet, 0, 1; (0,0,1)$ |

We need to address this problem when constructing all possible parse trees in Section 5.2, but for now we are lucky. While constructing a single parse tree the algorithm always follows the first reduction triple that is created when the entry is constructed initially. Since we only append new entries to bins, the complete reduction item that lead to the creation of the new entry with the reduction triple necessarily appears strictly before this entry. Furthermore, the implementation of the function *bin-upd* also makes sure to not change this first triple. Thus, we know for any item at index $i$ in the $k$-th bin that its first reduction pointer *red*, that we follow while constructing a single parse tree, is strictly smaller than $i$.

To summarize: if the algorithm encounters a null pointer it terminates immediately, for predecessor pointers it calls itself recursively in a bin with a strictly smaller index, and for reduction pointers it calls itself in the same bin but for a strictly smaller index.

**definition** *mono-red-ptrs* :: *'a bins* ⇒ *bool* **where**
*mono-red-ptrs bs* ≡ ∀ *k* < |*bs*|. ∀ *i* < |*bs*!*k*|.
∀ *k' pre red ps. pointer* (*bs*!*k*!*i*) = *PreRed* (*k'*, *pre*, *red*) *ps* ⟶ *red* < *i*

The proofs for the monotonicity of the first reduction pointer for functions *bin-upd*, *bin-upds*, *bins-upd*, *Earley-bin-list'*, *Earley-bin-list*, *Earley-list*, and *ℰarley-list* are completely analogous to the soundness proof of the pointers. We omit them.

Similarly to Chapter 3 we define a suitable measure and a notion of well-formedness for the input of the function *build-tree'* and prove a suitable induction schema, in the following referred to as *tree induction*, by complete induction on the measure. For the input quadruple (*bs*, *ω*, *k*, *i*) the measure corresponds to the number of entries in the first $k-1$ bins plus $i$. If the algorithm calls itself recursively for predecessor bins $k-1$ or $k'$, that are respectively strictly less than $k$, the measure decreases by at least $i+1$, and for any reduction index *red* it decreases by $i - red > 0$. We call the input well-formed if it satisfies the following conditions: sound and monotonic pointers, the bin index $k$ does not exceed the length of the bins, and the item index $i$ is within the bounds of the $k$-th bin.

**fun** *build-tree'-measure* :: (*'a bins* × *'a sentential* × *nat* × *nat*) ⇒ *nat* **where**
*build-tree'-measure* (*bs*, *ω*, *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*, *ω*, *k*, *i*) | *bs ω k i*.
*sound-ptrs ω bs* ∧ *mono-red-ptrs bs* ∧ *k* < |*bs*| ∧ *i* < |*bs*!*k*| }

To conclude this subsection, we prove termination of the function *build-tree'*, or for well-formed input it always terminates with some branch, by *tree induction*.

**lemma** *build-tree'-termination*:
**assumes** (*bs*, *ω*, *k*, *i*) ∈ *wf-tree-input*
**shows** ∃ *N ts. build-tree' bs ω k i* = *Some* (*Branch N ts*)

### 5.1.4 Correctness

We know that for well-formed input a call of the form *build-tree' bs ω k i* always terminates and yields some parse tree $t$ from the previous lemma. The following lemma proves that for well-formed bins $t$ represents a parse tree according to the semantics of the Earley item $N \rightarrow \alpha \bullet \beta, j, k$ at index $i$ in the $k$-th bin. The parse tree is rooted at the item rule head $N$, each of its subtrees is a complete derivation tree following the rules of the grammar, and the list of roots of the subtrees themselves coincide with $\alpha$. Moreover, the yield of $t$ matches the subsequence from $j$ to $k$ of the input $\omega$.

**lemma** *wf-item-yield-build-tree′*:
**assumes** $(bs, \omega, k, i) \in$ *wf-tree-input*
**assumes** *wf-bins* $\mathcal{G}\ \omega\ bs$
**assumes** *build-tree′* $bs\ \omega\ k\ i = $ *Some* $t$
**shows** *wf-item-tree* $\mathcal{G}$ (*item* $(bs!k!i)$) $t \wedge$ *wf-yield-tree* $\omega$ (*item* $(bs!k!i)$) $t$

*Proof.* The proof is by *tree induction* and we split it into three cases according to the kind of pointer the algorithm encounters. Let $e$ denote the entry at index $i$ in bin $k$, and $x$ be the item of $e$, or $x = N \to \alpha \bullet \beta, j, k$.

- *pointer* $e =$ *Null*: We have $t =$ *Branch* (*item-rule-head* $x$) $[]$. The root of $t$ coincides with the item rule head of $x$ by construction. Since the list of subtrees is empty, each of the subtrees is trivially well-formed according to the grammar. Moreover, we know *predicts* $x$, due to the null pointer, or the bullet of $x$ is at position 0. Thus, we have $\alpha = []$ and the list of subtrees $[]$ matches. In summary, we have *wf-item-tree* $\mathcal{G}\ x\ t$. From *predicts* $x$, we also know that $j = k$, or $\omega[j..k\rangle = []$ by definition of the *slice* function. Since the yield of $t$ is empty, we have *wf-yield-tree* $\omega\ x\ t$ and conclude the proof for the null pointer.

- *pointer* $e =$ *Pre pre*: Let $x'$ denote the predecessor *item* $(bs\ !\ (k - 1)\ !\ pre)$ of the recursive function call for bin $k-1$ and index *pre*. The function always terminates with some branch for well-formed input. Hence, there exists a tree *Branch N ts* corresponding to the predecessor item $x'$, and we have:

$$t = Branch\ N\ (ts\ @\ [Leaf\ (\omega\ !\ (k - 1))])$$

 We also have $(bs, \omega, k - 1, pre) \in$ *wf-tree-input* by assumption since the predecessor pointer is sound and the algorithm does not change the bins. Thus we can use the induction hypothesis and obtain:

$$\begin{aligned} &wf\text{-}item\text{-}tree\ \mathcal{G}\ x'\ (Branch\ N\ ts) && (IH_1)\\ &wf\text{-}yield\text{-}tree\ \omega\ x'\ (Branch\ N\ ts) && (IH_2)\end{aligned}$$

 Since the pointer is a simple predecessor pointer, the predicate *scans* $\omega\ k\ x'\ x$ holds and we also know that $x$ as well as $x'$ are well-formed bin items. Consequently, we obtain the following facts:

$$\begin{aligned}
&\textit{item-rule-head } x' = \textit{item-rule-head } x && (a)\\
&\textit{item-rule-body } x' = \textit{item-rule-body } x && (b)\\
&\textit{item-bullet } x' + \mathit{1} = \textit{item-bullet } x && (c)\\
&\textit{next-symbol } x' = \textit{Some } (\omega \ ! \ (k - \mathit{1})) && (d)\\
&\textit{item-origin } x' = \textit{item-origin } x && (e)\\
&\textit{item-end } x = k && (f)\\
&\textit{item-end } x' = k - \mathit{1} && (g)
\end{aligned}$$

We first prove *wf-item-tree* $\mathcal{G}$ $x$ $t$:

$$\begin{aligned}
&\textit{map root-tree } (ts \ @ \ [\textit{Leaf } (\omega \ ! \ (k - \mathit{1}))])\\
&\quad = \textit{map root-tree } ts \ @ \ [\omega \ ! \ (k - \mathit{1})] && (1)\\
&\quad = \textit{take } (\textit{item-bullet } x') \ (\textit{item-rule-body } x') \ @ \ [\omega \ ! \ (k - \mathit{1})] && (2)\\
&\quad = \textit{take } (\textit{item-bullet } x') \ (\textit{item-rule-body } x) \ @ \ [\omega \ ! \ (k - \mathit{1})] && (3)\\
&\quad = \textit{take } (\textit{item-bullet } x) \ (\textit{item-rule-body } x) && (4)
\end{aligned}$$

(1) by definition. (2) by $(IH_1)$. (3) by (b). (4) by (b,c,d). The statement *wf-item-tree* $\mathcal{G}$ $x$ $t$ follows by (a), using once more $(IH_1)$ to prove that all subtrees are complete according to the grammar by definition of *wf-item-tree*.

To conclude the proof for the simple predecessor pointer, we prove the statement *wf-yield-tree* $\omega$ $x$ $t$:

$$\begin{aligned}
&\textit{yield-tree } (\textit{Branch } N \ (ts \ @ \ [\textit{Leaf } (\omega \ ! \ (k - \mathit{1}))]))\\
&\quad = \textit{concat } (\textit{map yield-tree } ts) \ @ \ [\omega \ ! \ (k - \mathit{1})] && (1)\\
&\quad = \omega[\textit{item-origin } x'..\textit{item-end } x'\rangle \ @ \ [\omega \ ! \ (k - \mathit{1})] && (2)\\
&\quad = \omega[\textit{item-origin } x'..\textit{item-end } x' + \mathit{1}\rangle && (3)\\
&\quad = \omega[\textit{item-origin } x..\textit{item-end } x' + \mathit{1}\rangle && (4)\\
&\quad = \omega[\textit{item-origin } x..\textit{item-end } x\rangle && (5)
\end{aligned}$$

(1) by definition. (2) by $(IH_2)$. (3) by (g) and the definition of *slice*. (4) by (e). (5) by (f,g).

- *pointer* $e$ = *PreRed* $(k'$, *pre*, *red*$)$ *ps*: The proof is similar in structure to the proof of the simple predecessor case. We only highlight the main differences. In contrast to only one recursive call for the predecessor item $x'$, we have another recursive call for the complete reduction item $y$. But we have also have an additional induction hypothesis. The proofs of $\textit{wf-item-tree}\ \mathcal{G}\ x\ t$ and $\textit{wf-yield-tree}\ \omega\ x\ t$ are analogous to the case above replacing $\textit{Leaf}\ (\omega\ !\ (k\ -\ \mathit{1}))$ with the branch obtained from the second recursive call. Statements similar to (a-g) hold since all items are well-formed and the predicate $\textit{completes}\ k\ x'\ x\ y$ is true.

□

Next we prove that, if the function *build-tree* returns a parse tree, it is a complete and well-formed tree according to the grammar, the root of the tree is the start symbol of the grammar, and the yield of the tree corresponds to the input. The subsequent corollary then proves that the theorem in particular holds if we generate the bins using the algorithm of Chapter 4 if we adjust the assumptions accordingly.

**theorem** *wf-rule-root-yield-build-tree*:
**assumes** $\textit{wf-bins}\ \mathcal{G}\ \omega\ \textit{bs}$
**assumes** $\textit{sound-ptrs}\ \omega\ \textit{bs}$
**assumes** $\textit{mono-red-ptrs}\ \textit{bs}$
**assumes** $|\textit{bs}| = |\omega| + \mathit{1}$
**assumes** $\textit{build-tree}\ \mathcal{G}\ \omega\ \textit{bs} = \textit{Some}\ t$
**shows** $\textit{wf-rule-tree}\ \mathcal{G}\ t \wedge \textit{root-tree}\ t = \mathfrak{S}\ \mathcal{G} \wedge \textit{yield-tree}\ t = \omega$

*Proof.* The function *build-tree* searches the last bin for any finished items. Since it returns a tree by assumption, it is successful, or finds a finished item $x$ at index $i$, and calls the function $\textit{build-tree}'\ \textit{bs}\ \omega\ (|\textit{bs}|\ -\ \mathit{1})\ i$. By assumption the input and the bins are well-formed, we can discharge the assumptions of the previous two lemmas, obtain a tree $t = \textit{Branch}\ N\ \textit{ts}$ and have:

$$\textit{wf-item-tree}\ \mathcal{G}\ x\ t \wedge \textit{wf-yield-tree}\ \omega\ x\ t$$

Furthermore, the item $x$ is finished or its rule head is the start symbol of the grammar, it is complete, and its origin and end respectively are 0 and $|\omega|$. Due to the completeness and well-formedness of the item $\textit{wf-item-tree}\ \mathcal{G}\ x\ t$ implies $\textit{wf-rule-tree}\ \mathcal{G}\ t$ and $\textit{root-tree}\ t = \mathfrak{S}\ \mathcal{G}$. From $\textit{wf-yield-tree}\ \omega\ x\ t$ we have $\textit{yield-tree}\ t = \omega[\textit{item-origin}\ x..\textit{item-end}\ x\rangle$ by definition, and consequently $\textit{yield-tree}\ t = \omega$.

□

**corollary** *wf-rule-root-yield-build-tree-ℰarley-list*:
**assumes** $\textit{wf-}\mathcal{G}\ \mathcal{G}$
**assumes** $\textit{nonempty-derives}\ \mathcal{G}$
**assumes** $\textit{build-tree}\ \mathcal{G}\ \omega\ (\mathcal{E}\textit{arley-list}\ \mathcal{G}\ \omega) = \textit{Some}\ t$
**shows** $\textit{wf-rule-tree}\ \mathcal{G}\ t \wedge \textit{root-tree}\ t = \mathfrak{S}\ \mathcal{G} \wedge \textit{yield-tree}\ t = \omega$

We conclude this section with the final theorem stating that the function *build-tree* returns some parse tree if and only if there exists a derivation of the input from the start symbol of the grammar, provided we generated the bins with the algorithm of Chapter 4, and grammar and input are well-formed.

**theorem** *correctness-build-tree-$\mathcal{E}$arley-list*:
  **assumes** *wf-$\mathcal{G}$* $\mathcal{G}$
  **assumes** *is-sentence* $\mathcal{G}$ $\omega$
  **assumes** *nonempty-derives* $\mathcal{G}$
  **shows** $(\exists\, t.\ \textit{build-tree}\ \mathcal{G}\ \omega\ (\mathcal{E}\textit{arley-list}\ \mathcal{G}\ \omega) = \textit{Some}\ t) \longleftrightarrow \mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$

*Proof.* The function *build-tree* searches the last bin for a finished item $x$. It finds such an item and returns a parse tree if and only if the bins generated by $\mathcal{E}$*arley-list* $\mathcal{G}$ $\omega$ are *recognizing* which in turn holds if and only if there exists a derivation of the input from the start symbol of the grammar by lemma *correctness-$\mathcal{E}$arley-list* using our assumptions. □

## 5.2 All Parse Trees

While computing a single parse tree is sufficient for unambiguous grammars, an Earley parser - in its most general form - can handle all context-free grammars. For ambiguous grammars there might exist multiple parse trees for a specific input, there might even be exponentially many. One example of a highly ambiguous grammar that produces exponentially many parse trees is our running example. To be precise, the number of parse trees for an input $\omega = x + \cdots + x$ is the Catalan number $C_n$ where $n - 1$ is the number of times the terminal $x$ occurs in $\omega$. It is well known that the $n$-th Catalan number can be expressed as $C_n = \frac{1}{n+1}\binom{2n}{n}$ and thus grows asympotically at least exponentially. For example, the number of parse trees for an input $\omega$ containing 12 times the terminal $x$ is already $C_{11} = 58786$. Thus, it is infeasible to compute all possible parse trees in a naive fashion.

In the following we generalize the algorithm for a single parse tree to compute a representation of all parse trees, or a parse forest. The key idea is to find a data structure that allows as much structural sharing as possible between different parse trees. As an initial step, we make the following observation: for two reduction triples $(k'_A,\ pre_A,\ red_A)$ and $(k'_B,\ pre_B,\ red_B)$ of an Earley item we know that $red_A \neq red_B$, but it might be the case that $k'_A = k'_B$ (which implies $pre_A = pre_B$ due to the set semantics of the bins). In other words, for different reduction items, we might have the same predecessor item and thus can share the subtree representing the predecessor.

We define a data type *forest* capturing this idea and representing parse forests. Consider an arbitrary production rule $S \to AxB$ for non-terminals $S, A, B$ and terminal $x$. A branch of a single tree contains a list of length 3 containing the three subtrees $t_A$, $t_x$, and $t_B$ corresponding to the three symbols $A$, $x$, and $B$. For a parse forest we still have a list of length 3, but each element is now again a list of forests sharing subforests derived from the same non-terminal. For example, the list of subforests of a branch might look like $[[f_{A1}, f_{A2}], [f_x], [f_B]]$ if there are two possible parse forests derived from the non-terminal $A$, and one parse forest derived from the symbols $x$ and $B$ each. Note that if the subforest is a forest leaf than the list contains just this single leaf, or there never occurs a situation like $[[f_A], [f_{x1}, f_{x2}], [f_B]]$.

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

We introduce an abstraction function *trees* computing all possible parse trees for a given parse forest. For a forest leaf this is trivial, for a forest branch we first apply the function *trees* recursively for all subforests, then concatenate the results for each subforest, and finally create a tree branch for each of the possible combinations of subtrees. E.g. for the subforests $[[f_{A1}, f_{A2}], [f_x], [f_B]]$ we might obtain the possible subtrees $[[t_{A11}, t_{A12}, t_{A2}], [t_x], [t_B]]$ if the forest $f_{A1}$ yields two parse trees $t_{A11}$ and $t_{A12}$ and every other forest yields only a single tree. The three possible combinations of subtrees for the non-terminal $N$ are then: $[t_{A11}, t_x, t_B]$, $[t_{A12}, t_x, t_B]$, and $[t_{A2}, t_x, t_B]$.

```
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)
  )
```

### 5.2.1 A Parse Forest Algorithm

We define two auxiliary functions *group-by* and *insert-group* grouping a list of values *xs* according to a key-mapping $K$ and a value-mapping $V$ by key. E.g. for the list of tuples *xs* = [(*1*, *a*), (*2*, *b*), (*1*, *c*)] and mappings $K$ = *fst* and $V$ = *snd* we obtain the association list [(*1*, [*a*, *c*]), (*2*, [*b*])].

```
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* :: $('a \Rightarrow 'k) \Rightarrow ('a \Rightarrow 'v) \Rightarrow 'a\ list \Rightarrow ('k \times 'v\ list)\ list$ **where**
 *group-by* $K\ V\ [] = []$
$|$ *group-by* $K\ V\ (x\#xs) =$ *insert-group* $K\ V\ x$ (*group-by* $K\ V\ xs$)

Next we implement the function *build-forests'*. It takes as arguments the bins *bs*, the indices of the bin and item, $k$ respectively $i$, and a set of natural numbers $I$, and returns an optional list of parse forests. There are two things to note here: the return type and the argument $I$. One might expect that we can return a single parse forest and not a list of forests. This is not the case. Although we are sharing subforests for two distinct reduction triples $(k'_A,\ pre_A,\ red_A)$ and $(k'_B,\ pre_B,\ red_B)$ if $(k'_A,\ pre_A) = (k'_B,\ pre_B)$, we can not share the subforests if the predecessor items are distinct, and hence need to return two distinct forests in this case. Furthermore, the algorithm returns an optional value since the function might not terminate if the pointers are not sound as it was the case for function *build-tree'*. But unfortunately, this time around the situation is even worse: even for sound pointers the function *build-forests'* might not terminate. As explained in Subsection 5.1.3, the bins computed by the algorithm $\mathcal{E}$*arley-list* contain cyclic reduction pointers for cyclic grammars, and thus naively following all reduction pointers might lead to non-termination. To ensure the termination of the algorithm we keep track of the items the algorithm already visited in a single bin by means of the additional argument $I$ representing the indices of the previous function calls in the same bin. The algorithm proceeds as follows:

Let $e$ denote the $i$-th item in the $k$-th bin.. If the pointer of $e$ is a null pointer the forest algorithm proceeds analogously to the tree algorithm, constructing an initially empty forest branch.

For the simple predecessor case it calls itself recursively for the previous bin $k - 1$, predecessor index *pre*, and initializes the set of visited indices for bin $k - 1$ with the single index *pre* to indicate that the computation for this index has already been started, obtaining a list of optional predecessor forests. It then appends to the list of subforests of each of these predecessor forests a new forest leaf containing the terminal symbol at index $k - 1$ of the input. Note the monadic do-notation for the option monad, and the use of the function *those* that converts a list of optional values into an optional list of values if and only if each of the optional values is present, or not none.

In the case that the algorithm encounters a predecessor/reduction pointer it first makes sure to not enter a cycle of reductions by discarding any reduction indices that are contained in $I$ and thus the algorithm has already started to process in earlier recursive calls. It then groups the reduction triples by predecessor. Subsequently, for each tuple of predecessor ($k'$ and *pre*) and reduction (*reds*) indices it proceeds as follows. It first calls itself once recursively for the predecessor, initializing the set $I$ as $\{pre\}$, and obtaining a list of predecessor forests. Then it executes one recursive call for each reduction index $red \in set\ reds$ in the current bin $k$ making sure to mark the index *red* as already visited by adding it to $I$. Finally, it appends to the list of subforests of each predecessor forests the list of reduction forests computed in the previous step.

The function *build-forests* is then defined analogously to the function *build-tree*.

```
partial-function (option) build-forests' :: 'a bins ⇒ 'a sentential ⇒ nat ⇒ nat ⇒ nat set ⇒
'a forest list option where
  build-forests' bs ω 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-forests' bs ω (k−1) pre {pre};
          those (map (λf.
            case f of
              FBranch N fss ⇒ Some (FBranch N (fss @ [[FLeaf (ω!(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-forests' bs ω k' pre {pre};
            rss ← those (map (λred. build-forests' bs ω 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-forests* :: $'a$ *cfg* $\Rightarrow$ $'a$ *sentential* $\Rightarrow$ $'a$ *bins* $\Rightarrow$ $'a$ *forest list option* **where**

$$
\begin{array}{l}
\textit{build-forests}\ \mathcal{G}\ \omega\ \textit{bs} \equiv \\
\quad \textit{let}\ k = |\textit{bs}| - \textit{1 in} \\
\quad \textit{let finished} = \textit{filter-with-index}\ (\lambda x.\ \textit{is-finished}\ \mathcal{G}\ \omega\ x)\ (\textit{items}\ (\textit{bs}!k))\ \textit{in} \\
\quad \textit{map-option concat}\ (\textit{those}\ (\textit{map}\ (\lambda(\text{-},\ i).\ \textit{build-forests}'\ \textit{bs}\ \omega\ k\ i\ \{i\})\ \textit{finished}))
\end{array}
$$

### 5.2.2 Termination

Similarly to the single tree algorithm we need to define the well-formedess of the input and a suitable measure for the forest algorithm to prove an applicable induction schema (*forest induction*) by complete induction on the measure. An input quintuplet ($bs$, $\omega$, $k$, $i$, $I$) is well-formed if the pointers are sound, the indices $k$ and $i$ are within their respective bounds, and the set of already visited indices $I$ contains the current index $i$ and only consists of indices bounded by the size of the current bin $k$. As termination measure we count the number of items in the first $k$ bins minus the indices the algorithm already visited in the $k$-th bin, or the contents of $I$. Thus, if the algorithm calls itself recursively for a predecessor, either in bin $k - 1$ or $k'$ where $k' < k$, the measure reduces by at least the difference between the size of the current bin $k$ and the cardinality of $I$ (plus one, since the new set $I$ contains the index *pre*). In the case that the algorithm calls itself recursively for a reduction pointer *red*, the measure decreases by exactly one, since the algorithm adds *red* to the set of already visited indices $I$ and it has not encountered this index beforehand.

To summarize: if the algorithm encounters a null pointer it terminates immediately. For predecessor pointers it calls itself recursively in a bin with a strictly smaller index, and for chains of reduction pointers it visits each index of the current bin at most once.

We then prove by *forest induction* that the function *build-forests'* always terminates with some list of forests containing only forest branches for well-formed input.

$$
\begin{array}{l}
\textbf{definition}\ \textit{wf-forest-input} :: ('a\ \textit{bins} \times 'a\ \textit{sentential} \times \textit{nat} \times \textit{nat} \times \textit{nat set})\ \textit{set}\ \textbf{where} \\
\quad \textit{wf-forest-input} = \{\ (\textit{bs},\ \omega,\ k,\ i,\ I) \mid \textit{bs}\ \omega\ k\ i\ I. \\
\qquad \textit{sound-ptrs}\ \omega\ \textit{bs} \land k < |\textit{bs}| \land i < |\textit{bs}!k| \land i \in I \land I \subseteq \{\textit{0}..<|\textit{bs}!k|\}\ \}
\end{array}
$$

$$
\begin{array}{l}
\textbf{fun}\ \textit{build-forest}'\textit{-measure} :: ('a\ \textit{bins} \times 'a\ \textit{sentential} \times \textit{nat} \times \textit{nat} \times \textit{nat set}) \Rightarrow \textit{nat}\ \textbf{where} \\
\quad \textit{build-forest}'\textit{-measure}\ (\textit{bs},\ \omega,\ k,\ i,\ I) = \textit{foldl}\ (+)\ \textit{0}\ (\textit{map length}\ (\textit{take}\ (k{+}\textit{1})\ \textit{bs})) - \textit{card}\ I
\end{array}
$$

$$
\begin{array}{l}
\textbf{lemma}\ \textit{build-forests}'\textit{-termination}: \\
\quad \textbf{assumes}\ (\textit{bs},\ \omega,\ k,\ i,\ I) \in \textit{wf-forest-input} \\
\quad \textbf{shows}\ \exists\,\textit{fs}.\ \textit{build-forests}'\ \textit{bs}\ \omega\ k\ i\ I = \textit{Some fs} \land (\forall f \in \textit{set fs}.\ \exists\,N\ \textit{fss}.\ f = \textit{FBranch}\ N\ \textit{fss})
\end{array}
$$

At this point, one might wonder if the argument $I$ is really needed. The problem regarding non-termination are the cyclic reduction pointers. In theory we could modify the algorithm of Chapter 4 to not add any cyclic pointers at all to the bins, prove an according lemma, and require non-cyclic pointers for the well-formedness of the input of the forest algorithm. Subsequently, we could remove the - no longer needed - argument $I$ from the function *build-forests'* and adjust the implementation accordingly.

But a problem of technical nature arises while trying to prove the *forest induction* schema. We need to define a suitable measure capturing the termination argument in terms of the input, or a function of the form $('a\ bins \times 'a\ sentential \times nat \times nat) \Rightarrow nat$. But we cannot express the termination argument just in terms of the current input $(bs, \omega, k, i)$, but need access to the history of recursive calls to argue that - for non-cyclic pointers - the algorithm calls itself at most once for each index in the current bin $k$ during chains of reductions. Hence, we need to reintroduce the argument $I$ of already visited indices or an equivalent argument. Note that this still simplifies the function *build-forests'* slightly due to the fact that we no longer need to filter the list of reduction pointers, but comes at the cost of computing cycles of reduction pointers in the algorithm of Chapter 4. Additionally, the bins only contain cyclic pointers if the grammar itself is cyclic and hence not adding any cyclic pointers in the first place could be considered incorrect.

In summary, the - already visited indices - argument $I$ serves two purposes: checking for pointer cycles while constructing parse forests and expressing the termination argument of the algorithm.

### 5.2.3 Soundness

The following four lemmas prove soundness of the functions *build-forests'* and *build-forests*. The proof are analogous to the corresponding proofs for the functions *build-tree'* and *build-tree*, replacing *tree induction* with *forest induction*. We might add that although the forest algorithm is only a slight generalization of the tree algorithm and hence one might suspect that the proof should generalize easily, this is unfortunately not the case. The proofs are rather unpleasant and cumbersome due to the complexity that occurs from the interplay of the option monad (and actually the list monad we are working with but not using explicit do notation for), functions *those*, *map-option*, *concat*, and the quite involved definition of the abstraction function *trees*. We refrain from presenting any proofs in detail.

**lemma** *wf-item-yield-build-forests'*:
  **assumes** $(bs, \omega, k, i, I) \in$ *wf-forest-input*
  **assumes** *wf-bins* $\mathcal{G}$ $\omega$ *bs*
  **assumes** *build-forests'* $bs\ \omega\ k\ i\ I = Some\ fs$
  **assumes** $f \in set\ fs$
  **assumes** $t \in set\ (trees\ f)$
  **shows** *wf-item-tree* $\mathcal{G}$ $(item\ (bs!k!i))\ t \wedge$ *wf-yield-tree* $\omega\ (item\ (bs!k!i))\ t$

**theorem** *wf-rule-root-yield-build-forests*:

**assumes** *wf-bins* $\mathcal{G}$ $\omega$ *bs*
**assumes** *sound-ptrs* $\omega$ *bs*
**assumes** $|bs| = |\omega| + 1$
**assumes** *build-forests* $\mathcal{G}$ $\omega$ *bs* = *Some fs*
**assumes** $f \in$ *set fs*
**assumes** $t \in$ *set* (*trees f*)
**shows** *wf-rule-tree* $\mathcal{G}$ $t$ $\wedge$ *root-tree* $t = \mathfrak{S}$ $\mathcal{G}$ $\wedge$ *yield-tree* $t = \omega$

**corollary** *wf-rule-root-yield-build-forests-$\mathcal{E}$arley-list*:
**assumes** *wf-$\mathcal{G}$* $\mathcal{G}$
**assumes** *nonempty-derives* $\mathcal{G}$
**assumes** *build-forests* $\mathcal{G}$ $\omega$ ($\mathcal{E}$*arley-list* $\mathcal{G}$ $\omega$) = *Some fs*
**assumes** $f \in$ *set fs*
**assumes** $t \in$ *set* (*trees f*)
**shows** *wf-rule-tree* $\mathcal{G}$ $t$ $\wedge$ *root-tree* $t = \mathfrak{S}$ $\mathcal{G}$ $\wedge$ *yield-tree* $t = \omega$

**theorem** *soundness-build-forests-$\mathcal{E}$arley-list*:
**assumes** *wf-$\mathcal{G}$* $\mathcal{G}$
**assumes** *is-sentence* $\mathcal{G}$ $\omega$
**assumes** *nonempty-derives* $\mathcal{G}$
**assumes** *build-forests* $\mathcal{G}$ $\omega$ ($\mathcal{E}$*arley-list* $\mathcal{G}$ $\omega$) = *Some fs*
**assumes** $f \in$ *set fs*
**assumes** $t \in$ *set* (*trees f*)
**shows** $\mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$

### 5.2.4 Completeness

At this point we would like to prove that the forest algorithm indeed computes all possible parse trees. But before we can attempt such a proof we first need to define completeness. Recall the cyclic grammer $A ::= x \mid B$, $B ::= A$. There exist an infinite amount of parse trees for the input $\omega = x$. Although there certainly exist data structures modeling parse forests that enable an representation of an infinite amount of parse trees, our data type *forest* is not expressive enough. Note that, since we assume a finite grammar, there necessarily has to exist a cycle in the grammar if there exist an infinite amount of parse trees. The algorithm *build-forests'* does not complete any cycles and thus returns only those parse trees up to the depth of the cycle in the grammar, or the parse trees $A - x$ and $A - B - A - x$ for the given cyclic grammar. In conclusion, we can only prove the completeness of the algorithm for non-cyclic grammars.

But, as already mentioned in the introduction for this chapter, we decided against formally proving completeness for the parse forest algorithm. The reasoning is twofold. The completeness proof is far from trivial and exceeded the scope of this thesis. Secondly, the algorithm is only of theoretical interest and far from being practical due to its poor performance. The simple sharing of subforests for identical predecessor items is one optimization over the naive approach, but unfortunately not enough to make the algorithm practical, as some experimentation suggests. We would need to introduce further performance improvements. One obvious improvement is to use more structural sharing of subtrees. At the moment the algorithm always appends new lists of subforests. We can avoid copying the current list of subtrees if we preprend instead of append, and finally reverse the subtrees for complete items. Another concern is the number of executed recursive calls. As implemented, the algorithm might call itself recursively more than once for the same Earley item or identical bins and item indices. This occurs for example if we have two different predecessor items but the same reduction item, e.g. items of the form $A \rightarrow \alpha_A C \bullet \beta_A, i_A, j$, $B \rightarrow \alpha_B C \bullet \beta_B, i_B, j$, and $C \rightarrow \gamma \bullet, i_C, j$. The algorithm calls itself at least twice for the complete reduction item and thus also more than once for any items that can be reached from that item recursively. We could avoid repeated recursive calls using common memoization techniques. But some experiments with both performance improvements lead us to the following sentiment: the result is a highly complex algorithm with still subpar performance.

We can conclude: the straightforward generalization from the single parse tree algorithm to a parse forest algorithm is probably correct (at least sound). Nonetheless, we abandon this approach due to the awkwardness of the termination proof which requires and additional argument that cannot be fully eliminated due to not only but also including technical reasons. The second reason is the poor performance of the algorithm that we observed during some experimentation. This can be remedied to some degree by the improvements sketched above, but has other downsides like a convoluted and unnecessarily complex implementation.

## 5.3 A Word on Parse Forests

We have two main decisions to make while choosing an appropriate algorithm and data structure for implementing an Earley parser. (1) should the construction of a parse forest be intertwined with the generation of the Earley items or not. In other words: do we want a single or two phase algorithm. (2) and most importantly, we need to choose an appropriate data structure for a parse forest that can represent all parse trees and be constructed in optimally cubic space and time.

One of the main lessons of Section 5.2 is that we should prefer a single phase over a two phase algorithm. Any two phase algorithm must store some sort of data structure to indicate the origin of each Earley item generated during the first phase. In the second phase it then walks this data structure while constructing a parse forest and encounters the same complications regarding termination as the algorithm of the previous section. In contrast, a single phase algorithm that constructs a parse forest while generating the bins can be embedded into the algorithm of Chapter 3. Consequently, it is tail-recursive, and thus can return a plain and not an optional value. Moreover, it can reuse the termination argument that the number of Earley items is finite, and hence avoid introducing any superfluous arguments that are only needed to prove the termination of the algorithm.

Regarding the choice of an appropriate data structure for parse forests: the most well-known data structure for representing all possible derivations, a shared packed parse forest (SPPF), was introduced by Tomita [**Tomita:1985**]. The nodes of a SPPF are labelled by triples $(N, i, j)$ where $\omega[i..j\rangle$ is the subsequence matched by the non-terminal $N$. A SPPF utilizes two types of sharing. Nodes that have the same tree below them are shared. Additionally, the SPPF might contain so-called packed nodes representing a family of children. Each child stands for a different derivation of the same subsequence $\omega[i..j\rangle$ from the same non-terminal but following an alternate production rule. Scott [**Scott:2008**] adjusts the SPPF data structure of Tomita slightly and presents two algorithms - one single and one two phase - based on Earley's recognizer that are of worst case cubic space and time. Both approaches can be implemented on top of our implementation of the Earley recognizer of Chapter 3, although we strongly advise for the single phase algorithm due to the argument stated above. We did not attempt to formalize the algorithm of Scott since the implementation is rather complex - we already glossed over some important details of the SPPF data structure that are necessary to achieve the optimal cubic running time - and hence out of scope for this thesis.

# 6 The Running Example in Isabelle

This chapter illustrates how the running example is implemented in Isabelle and highlights the corresponding correctness theorems for functions *$\mathcal{E}$arley-list*, *build-tree*, and *build-forests*. But first we make a small addition to easily compute if a grammar allows empty derivations, or if $\mathcal{G} \vdash [N] \Rightarrow^* []$ holds for any non-terminal $N$ of grammar $\mathcal{G}$. We call a grammar *ε-free* if there does not exists any production rule of the form $N \rightarrow \epsilon$. For a well-formed grammar, strictly speaking we only require the left-hand side of any production rule to be a non-terminal, we then prove a lemma stating that a grammar does only allow non-empty derivations for any non-terminal if and only if it is epsilon-free.

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

**lemma** *nonempty-derives-iff-ε-free*:
  **assumes** *wf-*$\mathcal{G}$ $\mathcal{G}$
  **shows** *nonempty-derives* $\mathcal{G} \longleftrightarrow$ *ε-free* $\mathcal{G}$

Next we define the grammar $S ::= S + S \mid x$ in Isabelle. We introduce data types *T*, *N*, and *symbol* respectively representing terminal symbols $\{x, +\}$, the non-terminal $S$, and the type of symbols. Subsequently, we define the grammar as its four constituents: a list of non-terminal symbols, a list of terminal symbols, the production rules, and the start symbol. Finally, we specify the input $\omega = x + x + x$.

**datatype** *T* = *x* | *plus*
**datatype** *N* = *S*
**datatype** *symbol* = *Terminal T* | *Nonterminal N*

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

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

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

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

**definition** $\mathcal{G}$ :: *symbol cfg* **where**
$\mathcal{G}$ = *CFG nonterminals terminals rules start-symbol*

**definition** $\omega$ :: *symbol list* **where**
$\omega$ = [*Terminal x*, *Terminal plus*, *Terminal x*, *Terminal plus*, *Terminal x*]

The following three lemmas state the well-formedness of the grammar and input. The proofs are automatic by definition with addition of lemma *nonempty-derives-iff-ε-free*.

**lemma** *wf-*$\mathcal{G}$:
**shows** *wf-*$\mathcal{G}$ $\mathcal{G}$

**lemma** *nonempty-derives-*$\mathcal{G}$:
**shows** *nonempty-derives* $\mathcal{G}$

**lemma** *is-sentence-*$\omega$:
**shows** *is-sentence* $\mathcal{G}$ $\omega$

This section concludes by illustrating the following main theorems for functions $\mathcal{E}$*arley-list*, *build-tree*, and *build-forests* for the well-formed grammar $\mathcal{G}$ and input $\omega$ introduced above.

**lemma** *correctness-bins*:
**shows** $\mathit{recognizing}\ (\mathit{bins}\ (\mathcal{E}\mathit{arley\text{-}list}\ \mathcal{G}\ \omega))\ \mathcal{G}\ \omega \longleftrightarrow \mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$

**lemma** *wf-tree*:
**assumes** $\mathit{build\text{-}tree}\ \mathcal{G}\ \omega\ (\mathcal{E}\mathit{arley\text{-}list}\ \mathcal{G}\ \omega) = \mathit{Some}\ t$
**shows** $\mathit{wf\text{-}rule\text{-}tree}\ \mathcal{G}\ t \wedge \mathit{root\text{-}tree}\ t = \mathfrak{S}\ \mathcal{G} \wedge \mathit{yield\text{-}tree}\ t = \omega$

**lemma** *correctness-tree*:
**shows** $(\exists\, t.\ \mathit{build\text{-}tree}\ \mathcal{G}\ \omega\ (\mathcal{E}\mathit{arley\text{-}list}\ \mathcal{G}\ \omega) = \mathit{Some}\ t) \longleftrightarrow \mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$

**lemma** *wf-trees*:
**assumes** $\mathit{build\text{-}forests}\ \mathcal{G}\ \omega\ (\mathcal{E}\mathit{arley\text{-}list}\ \mathcal{G}\ \omega) = \mathit{Some}\ \mathit{fs}$
**assumes** $f \in \mathit{set}\ \mathit{fs}$
**assumes** $t \in \mathit{set}\ (\mathit{trees}\ f)$
**shows** $\mathit{wf\text{-}rule\text{-}tree}\ \mathcal{G}\ t \wedge \mathit{root\text{-}tree}\ t = \mathfrak{S}\ \mathcal{G} \wedge \mathit{yield\text{-}tree}\ t = \omega$

**lemma** *soundness-trees*:
**assumes** $\mathit{build\text{-}forests}\ \mathcal{G}\ \omega\ (\mathcal{E}\mathit{arley\text{-}list}\ \mathcal{G}\ \omega) = \mathit{Some}\ \mathit{fs}$
**assumes** $f \in \mathit{set}\ \mathit{fs}$
**assumes** $t \in \mathit{set}\ (\mathit{trees}\ f)$
**shows** $\mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$

# 7 Conclusion

In this chapter we shortly summarize the contributions of this thesis and point the reader towards - at least in our opinion - relevant and worthwhile future work.

## 7.1 Summary

We formalized and verified an Earley Parser. Our approach and proof development is incremental and tries to separate the concerns of recognizing and parsing as much as possible.

In an initial step, we introduced an abstract set-based implementation of an Earley recognizer. Our contributions build upon the foundation of the work on Local Lexing of Obua [**Obua:2017**] [**LocalLexing-AFP**] who formalized the basic building blocks about context-free grammars and derivations. We then proved the core theorems soundness, completeness and finiteness putting the main focus on the completeness proof that is mostly inspired by the work of Jones [**Jones:1972**].

Subsequently, we refined the Earley recognizer algorithm to a functional executable implementation modeled after the original imperative implementation of Earley [**Earley:1970**]. Then we followed once more Jones footsteps, by not explicitly reproving correctness of the algorithm, but instead replacing set-based specifications by list-based implementations and proving subsumption in both directions. In the process, we encountered the hurdles that concern Earley recognizers and grammars containing non-empty derivations, and ultimately followed Jones one final time and restrained the grammar. Finally, we provided an informal argument for the running time of our functional implementation and discussed alternative implementation approaches.

We moved on to extend the recognizer into a fully-fledged parser. First, we enriched the recognizer with the necessary information to construct parse trees in the form of pointers that we modeled after the work of Scott [**Scott:2008**] and thus avoided erroneous derivations that occurred in Earley's original version. We then developed a functional algorithm that utilizes the pointers to construct a single parse tree and proved its correctness. In a last step, we attempted to generalize our algorithm from a parse tree to a parse forest. We succeeded in proving soundness, but ultimately abandoned the approach. Nonetheless, we learned valuable lessons that lay the foundation for future work.

We rounded off the development and highlighted the main theorems by implementing the running example in Isabelle.

## 7.2 Future Work

Unfortunately, or fortunately, the work on the formalization of Earley parsing is by no means completed. In the following we sketch - in our opinion - worthwhile future work, roughly presented in decreasing order of importance, although this might by subject to personal preference.

As mentioned in Chapter 3, we followed an operational approach to specify an abstract set-based implementation of an Earley recognizer. In hindsight, it would have been more elegant to not follow the approach of Obua and instead define the set of Earley items inductively in Isabelle. This would probably also simplify some of the subsumption proofs of Chapter 4 that resolve around function iteration.

From a practical point of view, the last missing piece of the formalization is the construction of a complete parse forest that represent all possible derivation trees compactly. As discussed in Chapter 5, the most promising approaches are the algorithms presented by Scott [**Scott:2008**], in particular the algorithm that intertwines the construction of the shared-packed parse forest with the generation of the Earley bins and thus does not need to concern itself with any technicalities whilst proving termination. Note that Scott describes her algorithm at high-level of abstraction and consequently any attempted formalization entails a significant amount of work.

A next step are the possibly endless opportunities to improve the performance of the algorithm(s). One non-trivial optimization is another refinement step towards an imperative implementation that incorporates the necessary performance optimization to achieve the optimal cubic time and space bounds described in Chapter 3. A formal proof of the complexity bounds should be attempted at this point. Other performance improvements include incorporating a look-head symbol to prune dead-end derivation trees, and optimizing the representation of the grammar and the bins to enable faster prediction and completion operations, but this list is by no means conclusive.

In a last step, one might want to investigate how to incorporate parse tree disambiguation into an Earley parser. One of strengths of an Earley parser is its ability to work with arbitrary context-free grammars (at least grammars that contain non-empty derivations in our case). Nonetheless, one might want to resolve some ambiguities by allowing the user the specify precedence and associativity declarations restricting the set of allowed parser that are commonly found in parser generators. During our initial research we investigated possible approaches and found the following non-conclusive approaches: Adams *et al* [**Adams:2017**] describe a grammar rewriting approach that reinterprets context-free grammars as tree automata, intersects them with automata encoding the desired restrictions and reinterpret the results into a context-free grammar. Afroozeh *et al* [**Afroozeh:2013**] present an approach of 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. Last but not least, and possibly the most interesting approach since it avoids grammar rewriting and is solely based on parse tree filtering: Klint *et al* [**Klint:1997**] propose a framework of filters, that select from a set of parse trees the intended trees, to describe and compare a wide range of disambiguation problems in a parser-independent way.