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

1.1 Motivation

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

1.2 Related Work

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

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

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

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

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

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

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

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

1.3 Structure

1.4 Contributions

SNIPPET:

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

SNIPPET:

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

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

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

Intuitively, Earley's recognizer works in principle like a top-down parser carrying along all possible parses simultaneously in an efficient manner. In detail, the algorithm works as follows: it 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 ω where we started parsing, and an end j indicating the position in ω we are currently considering next for the remaining right-hand side of the production rule. Note that there will be only one set of earley items or only one bin B and we say an item is conceptually part of bin B_j if 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 (the first index being inclusive the second one exclusive) of ω by $\alpha = S+$, and are trying to parse $\beta = S$ from position 4 in ω.

The algorithm initializes B by applying the *Init* operation. It then proceeds to execute the *Scan*, *Predict* and *Complete* operations listed in Figure 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 ω matches the terminal symbol following the bullet. We add one new item $A \rightarrow\ \alpha a \bullet \beta, i, j + 1$

to B moving the bullet over the parsed terminal symbol.

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

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

   E.g. for the item $S \rightarrow\ S + \bullet S, 0, 2$ in B_2 we add the two items $S \rightarrow\ \bullet x, 2, 2$ and $S \rightarrow\ \bullet S + S, 2, 2$ corresponding to the two alternates of S. The bullet is set to the beginning of the right-hand side of the production rule, the origin and end are set to $j = 2$ to indicate that we are starting to 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 $\omega[j..k\rangle$, as mentioned before 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 on 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 $S \rightarrow\ S + S\bullet, 0, 5$ for two different reasons corresponding to the two different ways we can derive ω. When processing $S \rightarrow\ x\bullet, 4, 5$ we find $S \rightarrow\ S + \bullet S, 0, 4$ in the origin bin B_4 which corresponds to recognizing $(x + x) + x$. We would add the same item again while applying the *Complete* operation to $S \rightarrow\ S + S\bullet, 2, 5$ and $S \rightarrow\ S + \bullet S, 0, 2$ which corresponds to recognizing the input as $x + (x + x)$.

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

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

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

It follows from the following three lemmas:

1. Soundness: for every generated item there exists an according derivation:
   $A \to \alpha \bullet \beta, i, j \in B$ 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 $A \to \alpha \bullet \beta, i, j \in B$
3. Finiteness: there only exist a finite number of Earley items

Init

$$\frac{}{\mathrm{S} \to \bullet\alpha,\ 0,\ 0}$$

Scan

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

Predict

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

Complete

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

Figure 2.1: Earley inference rules

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

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

3 Earley's 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 τ. Basic types include *bool*, *nat*; type variables are written $'a$, $'b$, etc. Pairs are written (a, b); triples are written (a, b, c) and so forth but are internally represented as nested pairs; the nesting is on the first component of a pair. Functions *fst* and *snd* return the first and second component of a pair; the operator $(\times)$ represents pairs at the type level. Most type constructors are written postfix, e.g. $'a\ set$ and $'a\ list$; the function space arrow is $\Rightarrow$; function *set* converts a list into a set. Type synonyms are introduced via the *type_synonym* command. Algebraic data types are defined with the keyword *datatype*. Non-recursive definitions are introduced with the *definition* keyword.

It is standard to define a language as a set of strings over a finite set of symbols. We deviate slightly by introducing a type variable $'a$ for the type of symbols. Thus a string corresponds to a list of symbols and a language is formalized as a set of lists of symbols, 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**
$\quad wf\text{-}rules\ \mathcal{G} \equiv \forall\,(N, \alpha) \in set\ (\mathfrak{R}\ \mathcal{G}).\ N \in set\ (\mathfrak{N}\ \mathcal{G}) \wedge (\forall\, s \in set\ \alpha.\ s \in set\ (\mathfrak{N}\ \mathcal{G}) \cup set\ (\mathfrak{T}\ \mathcal{G}))$

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**
$\quad wf\text{-}\mathcal{G}\ \mathcal{G} \equiv disjunct\text{-}symbols\ \mathcal{G} \wedge wf\text{-}startsymbol\ \mathcal{G} \wedge wf\text{-}rules\ \mathcal{G} \wedge distinct\text{-}rules\ \mathcal{G}$

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 $\{x \mid P\ x\}$. Sets can also be defined inductively using the keyword *inductive_set*.

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

definition is-nonterminal :: 'a cfg ⇒ 'a ⇒ bool where
 is-nonterminal 𝒢 s ≡ s ∈ set (𝔑 𝒢)

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

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

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

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

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

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

definition derives :: 'a cfg ⇒ 'a sentential ⇒ 'a sentential ⇒ bool where
 derives 𝒢 u v ≡ (u, v) ∈ derivations 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*⟩ 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*, *corollary* respectively, followed by their names. They consist of zero or more assumptions marked by *assumes* keywords and one conclusion indicated by *shows*. E.g. we can proof a simple lemma about the interaction between the *slice* function and the append operator (@), stating the conditions under which we can split one slice into two.

```
lemma slice-append:
  assumes i ≤ j
  assumes j ≤ k
  shows xs[i..j⟩ @ xs[j..k⟩ = xs[i..k⟩
```

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 $S \to S + \bullet S, 2, 4$ consists of four parts: a production rule (*item-rule*), a natural number (*item-bullet*) indicating the position of the bullet in the production rule, and two natural numbers (*item-origin* inclusive, *item-end* exclusive) representing the portion of the input string ω that has been 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-*α and *item-*β split the production rule body at the bullet, e.g. $S \to \alpha \bullet \beta$. 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 ω.

```
datatype 'a item =
  Item (item-rule: 'a rule) (item-bullet : nat) (item-origin : nat) (item-end : nat)

type-synonym 'a items = 'a item set

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

definition item-rule-body :: 'a item ⇒ 'a sentential where
  item-rule-body x = rule-body (item-rule x)

definition item-α :: 'a item ⇒ 'a sentential where
  item-α x = take (item-bullet x) (item-rule-body x)

definition item-β :: 'a item ⇒ 'a sentential where
  item-β x = drop (item-bullet x) (item-rule-body x)

definition is-complete :: 'a item ⇒ bool where
  is-complete x ≡ item-bullet x ≥ |item-rule-body x|

definition next-symbol :: 'a item ⇒ 'a option where
  next-symbol x ≡ if is-complete x then None else Some (item-rule-body x ! item-bullet x)

definition is-finished :: 'a cfg ⇒ 'a sentential ⇒ 'a item ⇒ bool where
  is-finished 𝒢 ω x ≡
    item-rule-head x = 𝔖 𝒢 ∧
    item-origin x = 0 ∧
    item-end x = |ω| ∧
    is-complete x

definition recognizing :: 'a items ⇒ 'a cfg ⇒ 'a sentential ⇒ bool where
  recognizing I 𝒢 ω ≡ ∃ x ∈ I. is-finished 𝒢 ω x
```

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

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

definition *inc-item* :: *'a item* ⇒ *nat* ⇒ *'a item* **where**
$\quad$ *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 end the end of this section.

Note that, as mentioned previously, even though we are only constructing one set of Earley items, conceptually all items with the same item end form one Earley bin. Our operational approach is then the following: we generate Earley items bin by bin in ascending order, starting from the 0-th bin 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 the $k + 1$-st bin, 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 the $k + 1$-th bin 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 the length of the input.

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

definition *Init* :: *'a cfg* ⇒ *'a items* **where**
$\quad$ *Init* $\mathcal{G} = \{$ *init-item r 0* $\mid r.\ r \in$ *set* $(\mathfrak{R}\ \mathcal{G}) \wedge$ *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* $x\ (k{+}1) \mid x\ a.$
$\quad\quad\quad x \in$ *bin* $I\ k \wedge$
$\quad\quad\quad \omega!k = a \wedge$
$\quad\quad\quad k < |\omega| \wedge$
$\quad\quad\quad$ *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* $r\ k \mid r\ x.$
$\quad\quad\quad r \in$ *set* $(\mathfrak{R}\ \mathcal{G}) \wedge$
$\quad\quad\quad x \in$ *bin* $I\ k \wedge$
$\quad\quad\quad$ *next-symbol* $x =$ *Some* $($*rule-head* $r)\ \}$

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

definition Earley-step :: nat ⇒ 'a cfg ⇒ 'a sentential ⇒ 'a items ⇒ 'a items where
 Earley-step k 𝒢 ω I = I ∪ Scan k ω I ∪ Complete k I ∪ Predict k 𝒢 I

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

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

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

definition Earley-bin :: nat ⇒ 'a cfg ⇒ 'a sentential ⇒ 'a items ⇒ 'a items where
 Earley-bin k 𝒢 ω I = limit (Earley-step k 𝒢 ω) I

fun Earley :: nat ⇒ 'a cfg ⇒ 'a sentential ⇒ 'a items where
 Earley 0 𝒢 ω = Earley-bin 0 𝒢 ω (Init 𝒢)
| Earley (Suc n) 𝒢 ω = Earley-bin (Suc n) 𝒢 ω (Earley n 𝒢 ω)

definition ℰarley :: 'a cfg ⇒ 'a sentential ⇒ 'a items where
 ℰarley 𝒢 ω = Earley |ω| 𝒢 ω
```

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 proof a property about the basic building blocks, the *Init*, *Scan*, *Predict*, and *Complete* operations. Then we proof that this property is maintained iterating the function *Earley-step*, and thus holds for the *Earley-bin* operation. Finally, we show that the function *Earley* maintains this property for all bins and thus for the *𝓔arley* definition, or the set of Earley items.

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

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

```
definition wf-item :: 'a cfg ⇒ 'a sentential => 'a item ⇒ bool where
 wf-item 𝒢 ω x ≡
   item-rule x ∈ set (𝔑 𝒢) ∧
   item-bullet x ≤ |item-rule-body x| ∧
   item-origin x ≤ item-end x ∧
   item-end x ≤ |ω|
```

```
definition wf-items :: 'a cfg ⇒ 'a sentential ⇒ 'a items ⇒ bool where
 wf-items 𝒢 ω I ≡ ∀ x ∈ I. wf-item 𝒢 ω x
```

```
lemma wf-Init:
 shows wf-items 𝒢 ω (Init 𝒢)
```

```
lemma wf-Scan-Predict-Complete:
 assumes wf-items 𝒢 ω I
 shows wf-items 𝒢 ω (Scan k ω I ∪ Predict k 𝒢 I ∪ Complete k I)
```

```
lemma wf-Earley-step:
 assumes wf-items 𝒢 ω I
 shows wf-items 𝒢 ω (Earley-step k 𝒢 ω 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 𝒢 ω I
```

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

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

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

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

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

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

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

3.4 Soundness

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

definition *sound-item* :: *'a cfg* $\Rightarrow$ *'a sentential* $\Rightarrow$ *'a item* $\Rightarrow$ *bool* **where**
sound-item $\mathcal{G}$ ω $x = \mathcal{G} \vdash [\mathit{item\text{-}rule\text{-}head}\ x] \Rightarrow^* \omega[\mathit{item\text{-}origin}\ x..\mathit{item\text{-}end}\ x\rangle\ @\ \mathit{item\text{-}}\beta\ x$

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

Obua **[Obua:2017]** **[LocalLexing-AFP]** defines derivations at two different abstraction levels. The first representation is as the reflexive-transitive closure of the set of one-step derivations as introduced earlier in this chapter. The second representation is again more operational. He defines a predicate *Derives1* $\mathcal{G}$ *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 ⇒ 'a sentential ⇒ nat ⇒ 'a rule ⇒ 'a sentential ⇒ bool where
  Derives1 𝒢 u i r v ≡
    ∃ α β N γ.
        u = α @ [N] @ β
      ∧ v = α @ γ @ β
      ∧ (N, γ) ∈ set (𝔑 𝒢)
      ∧ r = (N, γ) ∧ i = |α|
```

He then defines the type of a *derivation* as a list of pairs representing precisely the positions and rules used to apply each rewrite step. The predicate *Derivation* is defined recursively as follows: *Derivation* α [] β holds only if $\alpha = \beta$. If the derivation consists of at least one rewrite pair (i, r), or *Derivation* $\mathcal{G}$ α $((i, r)$ # $D)$ β, then there must exist a γ such that *Derives1* $\mathcal{G}$ α i r γ and *Derivation* $\mathcal{G}$ γ D β. Note that we introduce once again a more convenient notation: e.g. *Derivation* α D β 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 × 'a rule) list

fun Derivation :: 'a cfg ⇒ 'a sentential ⇒ 'a derivation ⇒ 'a sentential ⇒ bool where
  Derivation - α [] β = (α = β)
| Derivation 𝒢 α (d#D) β = (∃ γ. Derives1 𝒢 α (fst d) (snd d) γ ∧ Derivation 𝒢 γ D β)

lemma derives-equiv-Derivation:
  shows 𝒢 ⊢ α ⇒^* β ≡ ∃ D. 𝒢 ⊢ α ⇒^D β
```

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

```
lemma Derivation-append-rewrite:
  assumes 𝒢 ⊢ α ⇒^D β @ γ @ δ
  assumes 𝒢 ⊢ γ ⇒^E γ'
  shows ∃ F. 𝒢 ⊢ α ⇒^F β @ γ' @ δ
```

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

```
lemma sound-Complete:
  assumes wf: wf-items 𝒢 ω I
  assumes sound: sound-items 𝒢 ω I
  shows sound-items 𝒢 ω (Complete k I)
```

Proof. Let z denote an arbitrary but fixed item of *Complete k I*. By the definition of the *Complete* operation there exist items x and y such that:

$$
\begin{array}{rlrl}
x \in \mathit{bin}\ I\ (\mathit{item\text{-}origin}\ y) & (1) & \mathit{next\text{-}symbol}\ x = \mathit{Some}\ (\mathit{item\text{-}rule\text{-}head}\ y) & (2) \\
y \in \mathit{bin}\ I\ k & (3) & \mathit{is\text{-}complete}\ y & (4) \\
z = \mathit{inc\text{-}item}\ x\ k & (5) & &
\end{array}
$$

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

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

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

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

Note that $\mathit{item\text{-}\beta}\ x = \mathit{item\text{-}rule\text{-}head}\ y\ \#\ \mathit{tl}\ (\mathit{item\text{-}\beta}\ x)$ 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

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

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* ($\mathit{item\text{-}origin}\ x \leq \mathit{item\text{-}origin}\ y$ and $\mathit{item\text{-}origin}\ y \leq \mathit{item\text{-}end}\ y$) and have

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

Moreover, since $z = \mathit{inc\text{-}item}\ x\ k$ (5) and the next symbol of x is the item rule head of y (2), it follows that $\mathit{tl}\ (\mathit{item\text{-}\beta}\ x) = \mathit{item\text{-}\beta}\ z$, 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* $\mathcal{G}\ \omega\ (\mathit{Init}\ \mathcal{G})$

lemma *sound-Scan*:
 assumes *wf-items* $\mathcal{G}\ \omega\ I$
 assumes *sound-items* $\mathcal{G}\ \omega\ I$

shows *sound-items* $\mathcal{G}$ ω (*Scan k* ω *I*)

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

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

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

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

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

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

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

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

theorem *soundness*:
assumes *recognizing* (*$\mathcal{E}$arley* $\mathcal{G}$ ω) $\mathcal{G}$ ω
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 $\mathit{bin}\ (\mathit{Earley\text{-}bin}\ k\ \mathcal{G}\ \omega\ I)\ i = \mathit{bin}\ I\ i$

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

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

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

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

Proof. The proof is by induction in k for arbitrary i, x, and a:
The base case $k = 0$ is trivial, since we have the assumption $i + 1 \leq 0$.
For the induction step we can assume

$$\begin{array}{rlll} i + 1 \leq k + 1 & (1) & k + 1 \leq |\omega| & (2) \\ x \in \mathit{bin}\ (\mathit{Earley}\ (k + 1)\ \mathcal{G}\ \omega)\ i & (3) & \mathit{next\text{-}symbol}\ x = \mathit{Some}\ a & (4) \\ \omega\ !\ i = a & (5) & & \end{array}$$

Assumptions (1) and (3) imply that $x \in \mathit{bin}\ (\mathit{Earley}\ k\ \mathcal{G}\ \omega)\ i$ 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 $x \in \mathit{bin}\ (\mathit{Earley}\ k\ \mathcal{G}\ \omega)\ i$ and have $\mathit{inc\text{-}item}\ x\ (i + 1) \in \mathit{Earley}\ k\ \mathcal{G}\ \omega$. 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 ω (*Earley* k $\mathcal{G}$ ω) using assumptions (2), (4), (5), and fact $x \in$ *bin* (*Earley* k $\mathcal{G}$ ω) i by the definition of the *Scan* operation. This in turn implies *inc-item* x $(i + 1) \in$ *Earley-step* k $\mathcal{G}$ ω (*Earley* k $\mathcal{G}$ ω) by lemma *Earley-step-sub-Earley-bin* and the definition of *Earley-step*. Since the function *Earley-bin* is idempotent (lemma *Earley-bin-idem*), we have *inc-item* x $(i + 1) \in$ *Earley* k $\mathcal{G}$ ω by definition of *Earley*. And again, the final statement follows by lemma *Earley-step-sub-Earley-bin* and the definition of *Earley-step*.

□

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

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

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

Next we give some intuition about the core idea of the completeness proof. Assume there exists an item $N \rightarrow \bullet A_0 A_1 \ldots A_n$ 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 $i_0 \leq i_1 \leq \cdots \leq i_n \leq i_{n+1}$:

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

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

We formalize this idea as follows: a set I is *partially-completed* if for each non-complete item x in I, the existence of a derivation D from the next symbol of x implies, that the item that can be obtained by moving the bullet over the next symbol of x, is also present in I. The full definition of *partially-completed* below is slightly more involved since we need to keep track of the validity of the indices. Note that the definition also requires that an arbitrary predicate P holds for the derivation D. This predicate is necessary since the completeness proof requires a proof on the length of the derivation D, and thus we sometimes need to limit the *partially-completed* property to derivations that don't 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 $\beta \overset{D}{\Longrightarrow} \omega[j..k)$, then I also contains the complete item $N \rightarrow \alpha\beta\bullet, i, k$. Note that this holds only if $j \leq k$, $k \leq |\omega|$, all items of I are well-formed and most importantly I must be *partially-completed* up to the length of the derivation D.

```
definition partially-completed :: nat ⇒ 'a cfg ⇒ 'a sentential ⇒ 'a items ⇒ ('a derivation ⇒ bool) ⇒
bool where
 partially-completed k 𝒢 ω I P ≡
  ∀ i j x a D.
   i ≤ j ∧ j ≤ k ∧ k ≤ |ω| ∧
   x ∈ bin I i ∧
   next-symbol x = Some a ∧
   𝒢 ⊢ [a] ⇒^D ω[i..j⟩ ∧ P D ⟶
   inc-item x j ∈ 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 $\alpha\beta \overset{D}{\Longrightarrow} \gamma$, can be split into two derivations E and F whose length is bounded by the length of D, and there exist α' and β' such that $\alpha \overset{E}{\Rightarrow} \alpha'$, $\beta \overset{F}{\Rightarrow} \beta'$ and $\gamma = \alpha' @ \beta'$. The proof is by induction on D for arbitrary α and β and quite technical since we need to manipulate the exact indices where each rewriting rule is applied in α and β, and thus we omit it.

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

lemma *Derivation-append-split*:
 assumes $\mathcal{G} \vdash (\alpha @ \beta) \Rightarrow^{D} \gamma$
 shows $\exists\, E\ F\ \alpha'\ \beta'.\ \mathcal{G} \vdash \alpha \Rightarrow^{E} \alpha' \wedge \mathcal{G} \vdash \beta \Rightarrow^{F} \beta' \wedge \gamma = \alpha' @ \beta' \wedge |E| \leq |D| \wedge |F| \leq |D|$

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

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

For the base case we have $\mathit{item}\text{-}\beta\ x = []$ and need to show that $\mathit{Item}\ (N,\ \alpha)\ |\alpha|\ i\ k \in I$:

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

We also know that $j = k$: we have $\mathcal{G} \vdash \mathit{item}\text{-}\beta\ x \Rightarrow^{D} \omega[j..k\rangle$ and $\mathit{item}\text{-}\beta\ x = []$ 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 assumption $x \in I$ and the fact that $x = \mathit{Item}$ $(N,\ \alpha)\ |\alpha|\ i\ j$.

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

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

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

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

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* $k\ \mathcal{G}\ \omega$) (λ-. *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 proof this by complete induction on $|D|$ for arbitrary x, i, a, j, and D, and split the proof into two different cases:

- $D = []$: Since $\mathcal{G} \vdash [a] \Rightarrow^{D} \omega[i..j\rangle$, 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 α such that *Derives1* $\mathcal{G}\ [a]$ (*fst* d) (*snd* d) α 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 $wf\text{-}items\ \mathcal{G}\ \omega\ (Earley\ k\ \mathcal{G}\ \omega)$ by lemma *wf-Earley*. Additionally, we know $\mathcal{G} \vdash item\text{-}\beta\ y \Rightarrow^{\mathcal{D}} \omega[i..j\rangle$ 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 $partially\text{-}completed\ k\ \mathcal{G}\ \omega\ (Earley\ k\ \mathcal{G}\ \omega)\ (\lambda E.\ |E| \leq |\mathcal{D}|)$, since $|\mathcal{D}| \leq |D|$ by definition of *partially-completed*, using once again all of our assumptions. This in turn implies $partially\text{-}completed\ j\ \mathcal{G}\ \omega\ (Earley\ k\ \mathcal{G}\ \omega)\ (\lambda E.\ |E| \leq |\mathcal{D}|)$ 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 $inc\text{-}item\ x\ j \in Earley\ k\ \mathcal{G}\ \omega$ by lemma *Complete-Earley*: Once again we use assumptions (1,2,4), we also know that $next\text{-}symbol\ x = Some\ N$, 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 $wf\text{-}\mathcal{G}\ \mathcal{G}$
shows $partially\text{-}completed\ |\omega|\ \mathcal{G}\ \omega\ (\mathcal{E}arley\ \mathcal{G}\ \omega)\ (\lambda\text{-}.\ True)$

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

theorem *completeness*:
assumes $wf\text{-}\mathcal{G}\ \mathcal{G}$
assumes $is\text{-}sentence\ \mathcal{G}\ \omega$
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 α 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 *finiteness-UNIV-wf-item*:
  shows *finite* $\{\ x \mid x.\ wf\text{-}item\ \mathcal{G}\ \omega\ x\ \}$

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

4 Earley Recognizer Implementation

4.1 The Executable Algorithm

In Chapter 3 we proved correctness of an abstract set-based implementation of Earley's simplified recognizer algorithm. In this chapter we implement an executable version. 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.

First 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 $S \rightarrow\ \bullet x, 0, 0; \bot$ consisting of the item $S \rightarrow\ \bullet x, 0, 0$ and a *Null* pointer denoted by $\bot$.

- 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 the item $S \rightarrow\ x\bullet, 2, 2$ 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 $S \to S + S\bullet, 0, 5; (4, 1, 0), (2, 0, 1)$ 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 $S \to x\bullet, 4, 5$ at index 0 in bin B_5 and the predecessor item $S \to S + \bullet S, 0, 4$ at index 1 in bin B_4, but we can also create it by the complete item $S \to S + S\bullet, 2, 5$ at index 1 in bin B_5 and the predecessor item $S \to S + \bullet S, 0, 2$ 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 $\mathcal{G}$: $S \to x$, $S \to S + S$

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

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

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

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

```
type-synonym 'a bin = 'a entry list

type-synonym 'a bins = 'a bin list

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

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

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 for all the production rules that have the start symbol on their left-hand sides, and finally overwrites the 0-th bin with this initial bin.

```
definition Init-list :: 'a cfg ⇒ 'a sentential ⇒ 'a bins where
  Init-list 𝒢 ω ≡
    let bs = replicate ( |ω| + 1) ([]) in
    let rs = filter (λr. rule-head r = 𝔖 𝒢) (ℜ 𝒢) in
    let b0 = map (λr. (Entry (init-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 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 ω, 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, 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), 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-list :: nat ⇒ 'a sentential ⇒ 'a  ⇒ 'a item ⇒ nat ⇒ 'a entry list where
  Scan-list k ω a x pre ≡
    if ω!k = a then
      let x' = inc-item x (k+1) in
      [Entry x' (Pre pre)]
    else []

  definition Predict-list :: nat ⇒ 'a cfg ⇒ 'a ⇒ 'a entry list where
  Predict-list k 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 item. Function *bin-upds* inserts multiple entries into a specific bin. 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. Finally, function *bins-upd* updates the *k*-th bin by inserting the given list of entries using function *bin-upds*.

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

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

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

The central piece for the list-based implementation is the function *Earley-bin-list'* completes the *k*-th bin starting from index *i*. It updates the bins *bs* using function *bins-upd* and the appropriate operation depending on the next symbol of the current item under consideration which can either be some terminal or non-terminal symbol or *None*. 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 operates on. We prove termination and highlight the relevant Isabelle specific details in Section 4.4. The function *Earley-bin-list* fully completes the *k*-th bin and thus corresponds to the function *Earley-bin*.

```
partial-function (tailrec) Earley-bin-list' :: nat ⇒ 'a cfg ⇒ 'a sentential ⇒ 'a bins ⇒ nat ⇒ 'a bins
where
  Earley-bin-list' k 𝒢 ω bs i = (
    if i ≥ |items (bs!k)| then bs
    else
      let x = items (bs!k) ! i in
      let bs' =
        case next-symbol x of
          Some a ⇒
            if is-terminal 𝒢 a then
              if k < |ω| then bins-upd bs (k+1) (Scan-list k ω a x i)
              else bs
            else bins-upd bs k (Predict-list k 𝒢 a)
        | None ⇒ bins-upd bs k (Complete-list k x bs i)
      in Earley-bin-list' k 𝒢 ω bs' (i+1))
```

```
definition Earley-bin-list :: nat ⇒ 'a cfg ⇒ 'a sentential ⇒ 'a bins ⇒ 'a bins where
  Earley-bin-list k 𝒢 ω bs = Earley-bin-list' k 𝒢 ω bs 0
```

Finally, functions *Earley-list* and *ℰarley-list* are structurally identical to functions *Earley* respectively *ℰarley*, differing only in the type of the used operations and the return type.

```
fun Earley-list :: nat ⇒ 'a cfg ⇒ 'a sentential ⇒ 'a bins where
  Earley-list 0 𝒢 ω = Earley-bin-list 0 𝒢 ω (Init-list 𝒢 ω)
| Earley-list (Suc n) 𝒢 ω = Earley-bin-list (Suc n) 𝒢 ω (Earley-list n 𝒢 ω)
```

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

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 which is $\mathcal{O}(n^3)$ where $n = |\omega|$. 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 which achieves a theoretical running time of $\mathcal{O}(n^3 \log n)$, and highlight possible further performance improvements. Finally, we discuss the choice for our particular implementation.

We state the running time of our implementation of the algorithm in terms of the length n of the input ω, 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 by n. Overall the number of items in each bin B_j is in $\mathcal{O}(n)$.

We have *Init-list* $\in \mathcal{O}(n)$ since the function *replicate* takes time linear in the length of ω, and functions *filter* and *map* operate at most on the size of the grammar $\mathcal{G}$. We also know *Scan-list* $\in \mathcal{O}(n)$. The dominating term is surprisingly (ω ! k), since $0 \leq k \leq n$, and it computes at most one new entry. Function *Predict-list* take time in the the size of the grammar $\mathcal{G}$, due to the *filter* and *map* functions, 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 (*bs* ! *item-origin x*) takes linear time, and functions *items*, *filter-with-index*, and *map* operate on the origin bin which is of at most linear size. 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 in $\mathcal{O}(n)$. The time for one function execution is dominated by the time it takes to update the bins with the newly created items whose number depends on in turn on the operation we applied but is bounded in the worst case by n (*Complete-list*). 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 single linked list, and pointers are no longer represented by the type *pointer* but by actual pointers between Earley items. 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 for a specific item a pointer to its position in the list. Insertion of a new item into a bin works then as follows: if the item is already present in the map, we follow the pointer and update the pointers in the list accordingly depending on the kind of item, otherwise we just append the item and its corresponding pointers to the list. Finally, we 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 which we can follow 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 contains the actual items. E.g. a pointer is either *Null*, or *Pre* x', or *PreRed* (x', y) *xys*. 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 items for each bin. Finally, as mentioned earlier, we omit implementing a lookahead terminal. Note that, although these performance improvements might speed the algorithm quite considerably, 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 is significant but 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. The current approach, appending items to the list and using natural numbers as pointers, maps more easily to the imperative approach. Our original idea was to refine the algorithm once more to an imperative version. But this exceeded the scope of this thesis.

4.3 Sets or Bins as Lists

Draft: Explain abstraction function and how they will be used.

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

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

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

Draft: Explain sets as lists and note upto similar lemmas but omitted.

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 *distinct* (*items b*)
shows *distinct* (*items* (*bin-upd e b*))

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

lemma *distinct-bin-upds*:
assumes *distinct* (*items b*)
shows *distinct* (*items* (*bin-upds es b*))

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

lemma *distinct-bins-upd*:
assumes *distinct* (*items* (*bs*!*k*))
shows *distinct* (*items* (*bins-upd bs k es* ! *k*))

4.4 Well-formedness

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

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

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

lemma *wf-bin-bin-upd*:
assumes *wf-bin* $\mathcal{G}$ ω *k b*
assumes *wf-item* $\mathcal{G}$ ω (*item e*) $\wedge$ *item-end* (*item e*) $= k$
shows *wf-bin* $\mathcal{G}$ ω *k* (*bin-upd e b*)

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

lemma *wf-bins-bins-upd*:
assumes *wf-bins* $\mathcal{G}$ ω *bs*
assumes *distinct* (*items es*)

assumes $\forall x \in \mathit{set}\ (\mathit{items\ es}).\ \mathit{wf\text{-}item}\ \mathcal{G}\ \omega\ \ x \wedge \mathit{item\text{-}end}\ x = k$
shows $\mathit{wf\text{-}bins}\ \mathcal{G}\ \omega\ (\mathit{bins\text{-}upd\ bs\ k\ es})$

Explain termination, how it is proved in Isabelle and custom induction schema.

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

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

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

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

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

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

4.5 Soundness

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

lemma *Scan-list-sub-Scan*:
assumes $\mathit{wf\text{-}bins}\ \mathcal{G}\ \omega\ \mathit{bs}$
assumes $\mathit{bins\text{-}items}\ \mathit{bs} \subseteq I$
assumes $k < |\mathit{bs}|$

assumes $k < |\omega|$
assumes $x \in$ *set* (*items* (*bs*!*k*))
assumes *next-symbol* $x =$ *Some a*
shows *set* (*items* (*Scan-list k* ω *a x pre*)) $\subseteq$ *Scan k* ω *I*

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

lemma *Complete-list-sub-Complete*:
assumes *wf-bins* $\mathcal{G}$ ω *bs*
assumes *bins-items bs* $\subseteq I$
assumes $k < |bs|$
assumes $x \in$ *set* (*items* (*bs*!*k*))
assumes *next-symbol* $x =$ *None*
shows *set* (*items* (*Complete-list k x bs red*)) $\subseteq$ *Complete k I*

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

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

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

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

4.6 Completeness

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

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

lemma *Complete-Un-eq-nonterminal*: — Detailed
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$
assumes *wf-items* $\mathcal{G}$ ω I
assumes *sound-items* $\mathcal{G}$ ω I
assumes *nonempty-derives* $\mathcal{G}$
assumes *wf-item* $\mathcal{G}$ ω x
assumes *item-end* $x = k$
assumes *next-symbol* $z =$ *Some* a
assumes *is-nonterminal* $\mathcal{G}$ a
shows *Complete* k $(I \cup \{x\}) =$ *Complete* k I

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

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

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

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

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

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

lemma *Earley-sub-Earley-list*:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$
assumes *is-sentence* $\mathcal{G}$ ω
assumes *nonempty-derives* $\mathcal{G}$
assumes $k \leq |\omega|$
shows *Earley k* $\mathcal{G}$ $\omega \subseteq$ *bins-items* (*Earley-list k* $\mathcal{G}$ ω)

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

4.7 Main Theorems

definition *recognizing-list* :: *'a bins* ⇒ *'a cfg* ⇒ *'a sentential* ⇒ *bool* **where**
recognizing-list I $\mathcal{G}$ $\omega \equiv \exists x \in$ *set* (*items* (*I* ! $|\omega|$)). *is-finished* $\mathcal{G}$ ω *x*

theorem *recognizing-list-iff-recognizing*:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$
assumes *is-sentence* $\mathcal{G}$ ω

assumes *nonempty-derives* $\mathcal{G}$
shows *recognizing-list* ($\mathcal{E}$*arley-list* $\mathcal{G}$ ω) $\mathcal{G}$ ω $\longleftrightarrow$ *recognizing* ($\mathcal{E}$*arley* $\mathcal{G}$ ω) $\mathcal{G}$ ω

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

SNIPPET:

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

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

5 Earley Parser Implementation

5.1 Pointer lemmas

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

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

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

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

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 *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

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

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

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

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

lemma *sound-mono-ptrs-Earley-bin-list'*: — Detailed
assumes $(k, \mathcal{G}, \omega, \mathit{bs}) \in \mathit{wf\text{-}earley\text{-}input}$
assumes $\mathit{nonempty\text{-}derives}\ \mathcal{G}$
assumes $\mathit{sound\text{-}items}\ \mathcal{G}\ \omega\ (\mathit{bins\text{-}items}\ \mathit{bs})$
assumes $\mathit{sound\text{-}ptrs}\ \omega\ \mathit{bs}$
assumes $\mathit{mono\text{-}red\text{-}ptr}\ \mathit{bs}$
shows $\mathit{sound\text{-}ptrs}\ \omega\ (\mathit{Earley\text{-}bin\text{-}list}'\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ i) \wedge \mathit{mono\text{-}red\text{-}ptr}\ (\mathit{Earley\text{-}bin\text{-}list}'\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ i)$

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

```
  shows sound-ptrs ω (Earley-bin-list k 𝒢 ω bs) ∧ mono-red-ptr (Earley-bin-list k 𝒢 ω bs)

lemma sound-mono-ptrs-Init-list:
  shows sound-ptrs ω (Init-list 𝒢 ω) ∧ mono-red-ptr (Init-list 𝒢 ω)

lemma sound-mono-ptrs-Earley-list:
  assumes wf-𝒢 𝒢
  assumes nonempty-derives 𝒢
  assumes k ≤ |ω|
  shows sound-ptrs ω (Earley-list k 𝒢 ω) ∧ mono-red-ptr (Earley-list k 𝒢 ω)

lemma sound-mono-ptrs-ℰarley-list:
  assumes wf-𝒢 𝒢
  assumes nonempty-derives 𝒢
  shows sound-ptrs ω (ℰarley-list 𝒢 ω) ∧ mono-red-ptr (ℰarley-list 𝒢 ω)
```

5.2 Trees and Forests

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

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

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

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

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

definition wf-yield-tree :: 'a sentential ⇒ 'a item ⇒ 'a tree ⇒ bool where
```

```
 wf-yield-tree ω x t ≡ yield-tree t = ω[item-origin x..item-end x⟩
```

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

```
fun combinations :: 'a list list ⇒ 'a list list where
 combinations [] = [[]]
| combinations (xs#xss) = [ x#cs . x <− xs, cs <− combinations xss ]
```

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

5.3 A Single Parse Tree

```
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
       })
 ))
```

definition *build-tree* :: *'a cfg* $\Rightarrow$ *'a sentential* $\Rightarrow$ *'a bins* $\Rightarrow$ *'a tree option* **where**
  build-tree $\mathcal{G}$ ω *bs* $\equiv$
    let $k = |bs| - 1$ *in* (
    case filter-with-index (λx. *is-finished* $\mathcal{G}$ ω x) (*items* ($bs!k$)) *of*
      [] $\Rightarrow$ *None*
    | (-, i)#- $\Rightarrow$ *build-tree'* *bs* ω k i)

fun *build-tree'-measure* :: (*'a bins* $\times$ *'a sentential* $\times$ *nat* $\times$ *nat*) $\Rightarrow$ *nat* **where**
  build-tree'-measure (bs, ω, k, i) = *foldl* (+) 0 (*map length* (*take k bs*)) + i

definition *wf-tree-input* :: (*'a bins* $\times$ *'a sentential* $\times$ *nat* $\times$ *nat*) *set* **where**
  wf-tree-input = {
    (bs, ω, k, i) | *bs* ω k i.
      sound-ptrs ω *bs* $\wedge$
      mono-red-ptr *bs* $\wedge$
      $k < |bs|$ $\wedge$
      $i < |bs!k|$
  }

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

lemma *wf-item-tree-build-tree'*:
  assumes (bs, ω, k, i) $\in$ *wf-tree-input*
  assumes *wf-bins* $\mathcal{G}$ ω *bs*
  assumes $k < |bs|$
  assumes $i < |bs!k|$
  assumes *build-tree'* *bs* ω k i = *Some* t
  shows *wf-item-tree* $\mathcal{G}$ (*item* ($bs!k!i$)) t

lemma *wf-yield-tree-build-tree'*:
  assumes (bs, ω, k, i) $\in$ *wf-tree-input*
  assumes *wf-bins* $\mathcal{G}$ ω *bs*
  assumes $k < |bs|$
  assumes $k \leq |\omega|$
  assumes $i < |bs!k|$
  assumes *build-tree'* *bs* ω k i = *Some* t
  shows *wf-yield-tree* ω (*item* ($bs!k!i$)) t

theorem *wf-rule-root-yield-tree-build-tree*:
  assumes *wf-bins* $\mathcal{G}$ ω *bs*
  assumes *sound-ptrs* ω *bs*

```
  assumes mono-red-ptr bs
  assumes |bs| = |ω| + 1
  assumes build-tree 𝒢 ω bs = Some t
  shows wf-rule-tree 𝒢 t ∧ root-tree t = 𝔖 𝒢 ∧ yield-tree t = ω
```

```
corollary wf-rule-root-yield-tree-build-tree-ℰarley-list:
  assumes wf-𝒢 𝒢
  assumes nonempty-derives 𝒢
  assumes build-tree 𝒢 ω (ℰarley-list 𝒢 ω) = Some t
  shows wf-rule-tree 𝒢 t ∧ root-tree t = 𝔖 𝒢 ∧ yield-tree t = ω
```

```
theorem correctness-build-tree-ℰarley-list:
  assumes wf-𝒢 𝒢
  assumes is-sentence 𝒢 ω
  assumes nonempty-derives 𝒢
  shows (∃ t. build-tree 𝒢 ω (ℰarley-list 𝒢 ω) = Some t) ⟷ 𝒢 ⊢ [𝔖 𝒢] ⇒* ω
```

5.4 All Parse Trees

```
fun insert-group :: ('a ⇒ 'k) ⇒ ('a ⇒ 'v) ⇒ 'a ⇒ ('k × 'v list) list ⇒ ('k × 'v list) list where
  insert-group K V a [] = [(K a, [V a])]
| insert-group K V a ((k, vs)#xs) = (
    if K a = k then (k, V a # vs) # xs
    else (k, vs) # insert-group K V a xs
  )
```

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

```
partial-function (option) build-trees' :: 'a bins ⇒ 'a sentential ⇒ nat ⇒ nat ⇒ nat set ⇒ 'a forest
list option where
  build-trees' bs 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-trees' 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-trees' bs ω k' pre {pre};
            rss ← those (map (λred. build-trees' 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-trees :: 'a cfg ⇒ 'a sentential ⇒ 'a bins ⇒ 'a forest list option where
  build-trees 𝒢 ω bs ≡
    let k = |bs| − 1 in
    let finished = filter-with-index (λx. is-finished 𝒢 ω x) (items (bs!k)) in
    map-option concat (those (map (λ(-, i). build-trees' bs ω k i {i}) finished))
```

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

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

```
lemma build-trees'-termination:
  assumes (bs, ω, k, i, I) ∈ wf-trees-input
  shows ∃fs. build-trees' bs ω k i I = Some fs ∧ (∀f ∈ set fs. ∃N fss. f = FBranch N fss)
```

```
lemma wf-item-tree-build-trees':
  assumes (bs, ω, k, i, I) ∈ wf-trees-input
```

assumes *wf-bins* $\mathcal{G}$ ω *bs*
assumes $k < |bs|$
assumes $i < |bs!k|$
assumes *build-trees'* *bs* ω *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*

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

theorem *wf-rule-root-yield-tree-build-trees*:
assumes *wf-bins* $\mathcal{G}$ ω *bs*
assumes *sound-ptrs* ω *bs*
assumes $|bs| = |\omega| + 1$
assumes *build-trees* $\mathcal{G}$ ω *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* = ω

corollary *wf-rule-root-yield-tree-build-trees-*$\mathcal{E}$*arley-list*:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$
assumes *nonempty-derives* $\mathcal{G}$
assumes *build-trees* $\mathcal{G}$ ω ($\mathcal{E}$*arley-list* $\mathcal{G}$ ω) = *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* = ω

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

theorem *termination-build-tree-$\mathcal{E}$arley-list*:
  assumes *wf-$\mathcal{G}$ $\mathcal{G}$*
  assumes *nonempty-derives $\mathcal{G}$*
  assumes $\mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$
  shows $\exists fs.$ *build-trees* $\mathcal{G}\ \omega$ ($\mathcal{E}$*arley-list* $\mathcal{G}\ \omega$) = *Some fs*

5.5 A Word on Completeness

SNIPPET:

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

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

6 Usage

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

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

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

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

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

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

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

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

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

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

lemma *nonempty-derives*:

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

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

7 Conclusion

7.1 Summary

7.2 Future Work

Different approaches:

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

Performance improvements:

(1) Look-ahead of k or at least 1 like in the original Earley paper. (2) Optimize the representation of the grammar instead of single list, group by production, ... (3) Complete faster by keeping a map from nonterminal which are next in the items to the actual items (4) Predict faster by organizing the grammar in an efficient manner by nonterminal (5) Refine the algorithm to an imperative version using a single linked list and actual pointers instead of natural numbers.

Parse tree disambiguation:

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

Grammar rewriting:

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

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

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

Parse tree filtering:

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