



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

Abstract

TODO: Abstract

Contents

Acknowledgments	iii
Abstract	iv
1 QUESTIONS	1
2 Snippets	2
2.1 Earley	2
2.2 Jones	2
2.3 Scott	2
2.4 Aycock	2
2.5 Related Work	2
2.5.1 Related Parsing Algorithms	2
2.5.2 Related Verification Work	2
2.6 Future Work	3
3 Introduction	4
3.1 Motivation	4
3.2 Structure	4
3.3 Related Work	4
3.4 Contributions	4
4 Earley’s Algorithm	5
4.1 Draft	5
4.2 Background Theory	5
4.3 Earley Recognizer	7
5 Earley Formalization	9
5.1 Draft	9
5.2 Definitions	9
5.3 Wellformedness	12
5.4 Soundness	13
5.5 Monotonicity and Absorption	14

5.6	Completeness	15
5.7	Finiteness	16
6	Draft	17
7	Earley Recognizer Implementation	19
7.1	Definitions	19
7.2	Wellformedness	22
7.3	List to set	24
7.4	Soundness	25
7.5	Set to list	26
7.6	Main Theorem	27
8	Earley Parser Implementation	28
8.1	Draft	28
8.2	Pointer lemmas	28
8.3	Trees and Forests	29
8.4	A single parse tree	30
8.5	Parse trees	33
8.6	A word on completeness	36
9	Examples	37
9.1	epsilon free CFG	37
9.2	Example 1: Addition	37
9.2.1	Example 2: Cyclic reduction pointers	38
10	Conclusion	40
10.1	Summary	40
10.2	Future Work	40
11	Templates	41
11.1	Section	41
11.1.1	Subsection	41
	List of Figures	43
	List of Tables	44

1 QUESTIONS

- How much explain the proofs?
- How reference thm names?
- How to turn blau assumes shows fun ... keywords?

2 Snippets

2.1 Earley

2.2 Jones

2.3 Scott

2.4 Aycock

2.5 Related Work

2.5.1 Related Parsing Algorithms

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.

2.5.2 Related Verification Work

TODO: LocalLexing!!!

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. Definitions and lemmas which were only slightly adapted from this work are marked with *LocalLexing* throughout this thesis.

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 prove soundness and completeness for all non-left-recursive grammars using the Coq proof assistant.

2.6 Future Work

3 Introduction

3.1 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.

work with the snippets, reformulate!

3.2 Structure

standard blabla

3.3 Related Work

see folder and bibliography

3.4 Contributions

what did I do, what is new

4 Earley's Algorithm

4.1 Draft

- Introduce background theory about CFG
- Introduce the Earley recognizer in the abstract set form with pointer, note the original error in Earley's algorithm
- Introduce the running example $S \rightarrow x \mid S + S$ for input $x + x + x$
- Illustrate the complete bins generated by the example
- Illustrate Initial $S \rightarrow .\text{alpha},0,0$, Scan $A \rightarrow \text{alpha}.\text{abeta},i,j \mid A \rightarrow \text{alpha}.\text{beta},i,j+1$, Predict $A \rightarrow \text{alpha}.\text{Bbeta},i,j$ and $B \rightarrow \text{gamma} \mid B \rightarrow .\text{gamma},j,j$, Complete $A \rightarrow \text{alpha}.\text{Bbeta},i,j$ and $B \rightarrow \text{gamma}.,j,k \mid A \rightarrow \text{alphaB}.\text{beta},i,k$
- Define goal: $A \rightarrow \text{alpha}.\text{beta},i,j$ iff $A \Rightarrow^* s[i..j)\text{beta}$ which implies $S \rightarrow \text{alpha}.,0,n+1$ iff $S \Rightarrow^* s$

TODO: Add nicer syntax for derives

4.2 Background Theory

type-synonym $'a \text{ rule} = 'a \times 'a \text{ list}$

type-synonym $'a \text{ rules} = 'a \text{ rule list}$

type-synonym $'a \text{ sentence} = 'a \text{ list}$

datatype $'a \text{ cfg} =$

CFG

$(\mathcal{N} : 'a \text{ list})$

(\mathcal{T} : 'a list)
 (\mathcal{R} : 'a rules)
 (\mathcal{S} : 'a)

definition *derives1* :: 'a cfg \Rightarrow 'a sentence \Rightarrow 'a sentence \Rightarrow bool **where**

derives1 cfg u v =
 (\exists x y N α .
 u = x @ [N] @ y
 \wedge v = x @ α @ y
 \wedge (N, α) \in set (\mathcal{R} cfg))

definition *derivations1* :: 'a cfg \Rightarrow ('a sentence \times 'a sentence) set **where**

derivations1 cfg = { (u,v) | u v. *derives1* cfg u v }

definition *derivations* :: 'a cfg \Rightarrow ('a sentence \times 'a sentence) set **where**

derivations cfg = (*derivations1* cfg)^{*}

definition *derives* :: 'a cfg \Rightarrow 'a sentence \Rightarrow 'a sentence \Rightarrow bool **where**

derives cfg u v = ((u, v) \in *derivations* cfg)

fun *slice* :: nat \Rightarrow nat \Rightarrow 'a list \Rightarrow 'a list **where**

slice - - [] = []
 | *slice* 0 (x#xs) = []
 | *slice* 0 (Suc b) (x#xs) = x # *slice* 0 b xs
 | *slice* (Suc a) (Suc b) (x#xs) = *slice* a b xs

lemma *slice-induct*:

assumes $\bigwedge a b. P a b []$
assumes $\bigwedge a x xs. P a 0 (x\#xs)$
assumes $\bigwedge b x xs. P 0 b xs \implies P 0 (Suc b) (x\#xs)$
assumes $\bigwedge a b x xs. P a b xs \implies P (Suc a) (Suc b) (x\#xs)$
shows $P a b xs$

definition *disjunct-symbols* :: 'a cfg \Rightarrow bool **where**

disjunct-symbols cfg \longleftrightarrow set (\mathcal{R} cfg) \cap set (\mathcal{T} cfg) = {}

definition *valid-startsymbol* :: 'a cfg \Rightarrow bool **where**

valid-startsymbol cfg \longleftrightarrow \mathcal{S} cfg \in set (\mathcal{R} cfg)

definition *valid-rules* :: 'a cfg \Rightarrow bool **where**

valid-rules cfg \longleftrightarrow (\forall (N, α) \in set (\mathcal{R} cfg). N \in set (\mathcal{R} cfg) \wedge (\forall s \in set α . s \in set (\mathcal{R} cfg) \cup set (\mathcal{T} cfg)))

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

definition *wf-cfg* :: 'a cfg \Rightarrow bool **where**
wf-cfg cfg \longleftrightarrow disjunct-symbols cfg \wedge valid-startsymbol cfg \wedge valid-rules cfg \wedge distinct-rules cfg

definition *is-terminal* :: 'a cfg \Rightarrow 'a \Rightarrow bool **where**
is-terminal cfg s = (s \in set (\mathfrak{T} cfg))

definition *is-nonterminal* :: 'a cfg \Rightarrow 'a \Rightarrow bool **where**
is-nonterminal cfg s = (s \in set (\mathfrak{N} cfg))

definition *is-symbol* :: 'a cfg \Rightarrow 'a \Rightarrow bool **where**
is-symbol cfg s \longleftrightarrow is-terminal cfg s \vee is-nonterminal cfg s

definition *wf-sentence* :: 'a cfg \Rightarrow 'a sentence \Rightarrow bool **where**
wf-sentence cfg s = ($\forall x \in$ set s. is-symbol cfg x)

definition *is-word* :: 'a cfg \Rightarrow 'a sentence \Rightarrow bool **where**
is-word cfg s = ($\forall x \in$ set s. is-terminal cfg x)

4.3 Earley Recognizer

INIT	SCAN	PREDICT
$\frac{}{S \rightarrow \bullet\alpha, 0, 0}$	$\frac{A \rightarrow \alpha \bullet a \beta, i, j}{A \rightarrow \alpha a \bullet \beta, i, j+1}$	$\frac{A \rightarrow \alpha \bullet B \beta, i, j \quad B \rightarrow \gamma \in \text{set}(\mathfrak{R} \text{cfg})}{B \rightarrow \bullet \gamma, j, j}$
	COMPLETE	
	$\frac{A \rightarrow \alpha \bullet B \beta, i, j \quad B \rightarrow \gamma \bullet, j, k}{A \rightarrow \alpha B \bullet \beta, i, k}$	

Figure 4.1: Earley inference rules

$$A \rightarrow \alpha \bullet \beta, i, j \text{ iff } A \xRightarrow{*} \text{slice } i \ j \ \text{inp}$$

$$\mathfrak{S} \text{cfg} \rightarrow \alpha \bullet, 0, |\text{inp}| + 1 \text{ iff } \mathfrak{S} \text{cfg} \xRightarrow{*} \text{inp}$$

$S \rightarrow x \ S \rightarrow S + S$

Table 4.1: Earley items for the CFG $S \rightarrow x, S \rightarrow S + S$

0	1	2
$S \rightarrow \bullet x, 0, 0$ $S \rightarrow \bullet S + S, 0, 0$	$S \rightarrow x \bullet, 0, 1$ $S \rightarrow S \bullet + S, 0, 1$	$S \rightarrow S + \bullet S, 0, 2$ $S \rightarrow \bullet x, 2, 2$ $S \rightarrow \bullet S + S, 2, 2$
3	4	5
$S \rightarrow x \bullet, 2, 3$ $S \rightarrow S + S \bullet, 0, 3$ $S \rightarrow S \bullet + S, 2, 3$ $S \rightarrow S \bullet + S, 0, 3$	$S \rightarrow S + \bullet S, 2, 4$ $S \rightarrow S + \bullet S, 0, 4$ $S \rightarrow \bullet x, 4, 4$ $S \rightarrow \bullet S + S, 4, 4$	$S \rightarrow x \bullet, 4, 5$ $S \rightarrow S + S \bullet, 2, 5$ $S \rightarrow S + S \bullet, 0, 5$ $S \rightarrow S \bullet + S, 4, 5$ $S \rightarrow S \bullet + S, 2, 5$ $S \rightarrow S \bullet + S, 0, 5$

5 Earley Formalization

5.1 Draft

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

5.2 Definitions

definition *rule-head* :: 'a rule \Rightarrow 'a **where**
rule-head = *fst*

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

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

type-synonym 'a items = 'a item set

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

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

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

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

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

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

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

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

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

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

$item-end\ x \leq length\ inp$)

definition $wf-items :: 'a\ cfg \Rightarrow 'a\ sentence \Rightarrow 'a\ items \Rightarrow bool$ **where**
 $wf-items\ cfg\ inp\ I = (\forall x \in I. wf-item\ cfg\ inp\ x)$

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

definition $earley-recognized :: 'a\ items \Rightarrow 'a\ cfg \Rightarrow 'a\ sentence \Rightarrow bool$ **where**
 $earley-recognized\ I\ cfg\ inp = (\exists x \in I. is-finished\ cfg\ inp\ x)$

definition $Init :: 'a\ cfg \Rightarrow 'a\ items$ **where**
 $Init\ cfg = \{ init-item\ r\ 0 \mid r. r \in set\ (\mathfrak{R}\ cfg) \wedge fst\ r = (\mathfrak{S}\ cfg) \}$

definition $Scan :: nat \Rightarrow 'a\ sentence \Rightarrow 'a\ items \Rightarrow 'a\ items$ **where**
 $Scan\ k\ inp\ I =$
 $\{ inc-item\ x\ (k+1) \mid x\ a.$
 $x \in bin\ I\ k \wedge$
 $inp!k = a \wedge$
 $k < length\ inp \wedge$
 $next-symbol\ x = Some\ a \}$

definition $Predict :: nat \Rightarrow 'a\ cfg \Rightarrow 'a\ items \Rightarrow 'a\ items$ **where**
 $Predict\ k\ cfg\ I =$
 $\{ init-item\ r\ k \mid r\ x.$
 $r \in set\ (\mathfrak{R}\ cfg) \wedge$
 $x \in bin\ I\ k \wedge$
 $next-symbol\ x = Some\ (rule-head\ r) \}$

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

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

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

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

definition $\pi\text{-step} :: \text{nat} \Rightarrow 'a \text{ cfg} \Rightarrow 'a \text{ sentence} \Rightarrow 'a \text{ items} \Rightarrow 'a \text{ items}$ **where**
 $\pi\text{-step } k \ \text{cfg} \ \text{inp} \ I = I \cup \text{Scan } k \ \text{inp} \ I \cup \text{Complete } k \ I \cup \text{Predict } k \ \text{cfg} \ I$

definition $\pi :: \text{nat} \Rightarrow 'a \text{ cfg} \Rightarrow 'a \text{ sentence} \Rightarrow 'a \text{ items} \Rightarrow 'a \text{ items}$ **where**
 $\pi \ k \ \text{cfg} \ \text{inp} \ I = \text{limit } (\pi\text{-step } k \ \text{cfg} \ \text{inp}) \ I$

fun $\mathcal{I} :: \text{nat} \Rightarrow 'a \text{ cfg} \Rightarrow 'a \text{ sentence} \Rightarrow 'a \text{ items}$ **where**
 $\mathcal{I} \ 0 \ \text{cfg} \ \text{inp} = \pi \ 0 \ \text{cfg} \ \text{inp} \ (\text{Init } \text{cfg})$
 $\mid \mathcal{I} \ (\text{Suc } n) \ \text{cfg} \ \text{inp} = \pi \ (\text{Suc } n) \ \text{cfg} \ \text{inp} \ (\mathcal{I} \ n \ \text{cfg} \ \text{inp})$

definition $\mathcal{J} :: 'a \text{ cfg} \Rightarrow 'a \text{ sentence} \Rightarrow 'a \text{ items}$ **where**
 $\mathcal{J} \ \text{cfg} \ \text{inp} = \mathcal{I} \ (\text{length } \text{inp}) \ \text{cfg} \ \text{inp}$

5.3 Wellformedness

lemma wf-Init :
assumes $x \in \text{Init } \text{cfg}$
shows $\text{wf-item } \text{cfg} \ \text{inp} \ x$
 by definition

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

lemma $\text{wf-}\pi\text{-step}$:
assumes $\text{wf-items } \text{cfg} \ \text{inp} \ I$
shows $\text{wf-items } \text{cfg} \ \text{inp} \ (\pi\text{-step } k \ \text{cfg} \ \text{inp} \ I)$
 $\text{wf-Scan-Predict-Complete}$ by definition

lemma wf-funpower :
assumes $\text{wf-items } \text{cfg} \ \text{inp} \ I$
shows $\text{wf-items } \text{cfg} \ \text{inp} \ (\text{funpower } (\pi\text{-step } k \ \text{cfg} \ \text{inp}) \ n \ I)$
 $\text{wf-}\pi\text{-step}$, by induction on n

lemma $\text{wf-}\pi$:

assumes *wf-items cfg inp I*
shows *wf-items cfg inp (π k cfg inp I)*

wf-funpower by definition

lemma *wf- π 0:*

shows *wf-items cfg inp (π 0 cfg inp (Init cfg))*

wf-Init wf- π by definition

lemma *wf- \mathcal{I} :*

shows *wf-items cfg inp (\mathcal{I} n cfg inp)*

wf- π 0 wf- π by induction on n

lemma *wf- \mathcal{J} :*

shows *wf-items cfg inp (\mathcal{J} cfg inp)*

wf- \mathcal{I} by definition

5.4 Soundness

definition *sound-item :: 'a cfg \Rightarrow 'a sentence \Rightarrow 'a item \Rightarrow bool* **where**

sound-item cfg inp x = derives cfg [item-rule-head x] (slice (item-origin x) (item-end x) inp @ item- β x)

definition *sound-items :: 'a cfg \Rightarrow 'a sentence \Rightarrow 'a items \Rightarrow bool* **where**

sound-items cfg inp I = ($\forall x \in I$. sound-item cfg inp x)

lemma *sound-Init:*

shows *sound-items cfg inp (Init cfg)*

lemma *sound-item-inc-item:*

assumes *wf-item cfg inp x sound-item cfg inp x*

assumes *next-symbol x = Some a k < length inp inp!k = a item-end x = k*

shows *sound-item cfg inp (inc-item x (k+1))*

lemma *sound-Scan:*

assumes *wf-items cfg inp I sound-items cfg inp I*

shows *sound-items cfg inp (Scan k inp I)*

lemma *sound-Predict:*

assumes *sound-items cfg inp I*

shows *sound-items cfg inp (Predict k cfg I)*

lemma *sound-Complete:*

assumes *wf-items cfg inp I sound-items cfg inp I*

shows *sound-items cfg inp (Complete k I)*

lemma *sound- π -step:*

assumes *wf-items cfg inp I sound-items cfg inp I*

shows *sound-items cfg inp* (π -step *k cfg inp I*)

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

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

lemma *sound- $\pi 0$* :
shows *sound-items cfg inp* (π 0 *cfg inp* (*Init cfg*))

lemma *sound- \mathcal{I}* :
shows *sound-items cfg inp* (\mathcal{I} *k cfg inp*)

lemma *sound- \mathcal{J}* :
shows *sound-items cfg inp* (\mathcal{J} *cfg inp*)

theorem *soundness*:
shows *earley-recognized* (\mathcal{J} *cfg inp*) *cfg inp* \implies *derives cfg* [\mathcal{S} *cfg*] *inp*

5.5 Monotonicity and Absorption

lemma *π -idem*:
shows π *k cfg inp* (π *k cfg inp I*) = π *k cfg inp I*

lemma *Scan-bin-absorb*:
shows *Scan k inp* (*bin I k*) = *Scan k inp I*

lemma *Predict-bin-absorb*:
shows *Predict k cfg* (*bin I k*) = *Predict k cfg I*

lemma *Complete-bin-absorb*:
shows *Complete k* (*bin I k*) \subseteq *Complete k I*

lemma *Scan-Predict-Complete-sub-mono*:
assumes $I \subseteq J$
shows *Scan k inp I* \subseteq *Scan k inp J* *Predict k cfg I* \subseteq *Predict k cfg J* *Complete k I* \subseteq *Complete k J*

lemma *π -step-sub-mono*:
assumes $I \subseteq J$
shows π -step *k cfg inp I* \subseteq π -step *k cfg inp J*

lemma *funpower-sub-mono*:
assumes $I \subseteq J$
shows *funpower* (π -step *k cfg inp*) *n I* \subseteq *funpower* (π -step *k cfg inp*) *n J*

lemma *π -sub-mono*:
assumes $I \subseteq J$
shows π *k cfg inp I* \subseteq π *k cfg inp J*

lemma *Scan-Predict-Complete- π -step-mono*:
shows *Scan k inp I* \cup *Predict k cfg I* \cup *Complete k I* \subseteq π -step *k cfg inp I*

lemma *π -step- π -mono*:
shows π -step *k cfg inp I* \subseteq π *k cfg inp I*

lemma *Scan-Predict-Complete- π -mono*:

shows $\text{Scan } k \text{ inp } I \cup \text{Predict } k \text{ cfg } I \cup \text{Complete } k I \subseteq \pi k \text{ cfg inp } I$

lemma $\pi\text{-mono}$:
shows $I \subseteq \pi k \text{ cfg inp } I$

lemma Scan-bin-empty :
assumes $i \neq k \ i \neq k+1$
shows $\text{bin } (\text{Scan } k \text{ inp } I) i = \{\}$

lemma Predict-bin-empty :
assumes $i \neq k$
shows $\text{bin } (\text{Predict } k \text{ cfg } I) i = \{\}$

lemma $\text{Complete-bin-empty}$:
assumes $i \neq k$
shows $\text{bin } (\text{Complete } k I) i = \{\}$

lemma $\pi\text{-step-bin-absorb}$:
assumes $i \neq k \ i \neq k+1$
shows $\text{bin } (\pi\text{-step } k \text{ cfg inp } I) i = \text{bin } I i$

lemma $\text{funpower-bin-absorb}$:
assumes $i \neq k \ i \neq k+1$
shows $\text{bin } (\text{funpower } (\pi\text{-step } k \text{ cfg inp}) n I) i = \text{bin } I i$

lemma $\pi\text{-bin-absorb}$:
assumes $i \neq k \ i \neq k+1$
shows $\text{bin } (\pi k \text{ cfg inp } I) i = \text{bin } I i$

5.6 Completeness

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

lemma $\text{Predict-}\mathcal{I}$:
assumes $i \leq k \ x \in \text{bin } (\mathcal{I} k \text{ cfg inp}) i \text{ next-symbol } x = \text{Some } N \ (N, \alpha) \in \text{set } (\mathfrak{R} \text{ cfg})$
shows $\text{init-item } (N, \alpha) i \in \mathcal{I} k \text{ cfg inp}$

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

type-synonym $'a \text{ derivation} = (\text{nat} \times 'a \text{ rule}) \text{ list}$

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

$$\wedge r = (N, \alpha) \wedge i = \text{length } x)$$

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

definition *partially-completed* :: nat \Rightarrow 'a cfg \Rightarrow 'a sentence \Rightarrow 'a items \Rightarrow ('a derivation \Rightarrow bool) \Rightarrow bool **where**

partially-completed k cfg inp I P = (
 \forall i j x a D.
 $i \leq j \wedge j \leq k \wedge k \leq \text{length } \text{inp} \wedge$
 $x \in \text{bin } I \ i \wedge \text{next-symbol } x = \text{Some } a \wedge$
Derivation cfg [a] D (slice i j inp) \wedge P D \longrightarrow
inc-item x j \in I
)

lemma *fully-completed*:

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

lemma *partially-completed-I*:

assumes wf-cfg cfg
shows *partially-completed* k cfg inp (I k cfg inp) (λ -. True)

lemma *partially-completed-J*:

assumes wf-cfg cfg
shows *partially-completed* (length inp) cfg inp (J cfg inp) (λ -. True)

theorem *completeness*:

assumes *derives* cfg [S cfg] inp is-word cfg inp wf-cfg cfg
shows *earley-recognized* (J cfg inp) cfg inp

corollary

assumes wf-cfg cfg is-word cfg inp
shows *earley-recognized* (J cfg inp) cfg inp \longleftrightarrow *derives* cfg [S cfg] inp

5.7 Finiteness

lemma *finiteness-UNIV-wf-item*:

shows finite { x | x. wf-item cfg inp x }

theorem *finiteness*:

shows finite (J cfg inp)

6 Draft

- introduce auxiliary definitions: `filter_with_index`, `pointer`, `entry` in more detail most everything else in text
- overview over earley implementation with linked list and pointers and the mapping into a functional setting
- introduce `Init_it`, `Scan_it`, `Predict_it` and `Complete_it`, compare them with the set notation and discuss performance improvements (Grammar in more specific form) Why do they all return a list?!
- discuss `bin(s)_upd(s)` functions. Why `bin_upds` like this -> easier than fold for proofs!
- discuss `pi_it` and why it is a partial function -> only terminates for valid input and foreshadow how this is done in isabelle
- introduce remaining definitions (analog to sets)
- discuss wf proofs quickly and go into detail about isabelle specifics about termination and the custom induction scheme using finiteness
- outline the approach to proof correctness aka subsumption in both directions
- discuss list to set proofs
- discuss soundness proofs (maybe omit since obvious)

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

7 Earley Recognizer Implementation

7.1 Definitions

fun *filter-with-index'* :: *nat* \Rightarrow (*'a* \Rightarrow *bool*) \Rightarrow *'a list* \Rightarrow (*'a* \times *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* \Rightarrow *bool*) \Rightarrow *'a list* \Rightarrow (*'a* \times *nat*) *list* **where**
 filter-with-index *P* *xs* = *filter-with-index'* 0 *P* *xs*

datatype *pointer* =
 Null
 | *Pre* *nat*
 | *PreRed* *nat* \times *nat* \times *nat* (*nat* \times *nat* \times *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* \Rightarrow *'a item list* **where**
 items *b* = *map item b*

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

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

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

definition *bin-items-upto* :: 'a bin \Rightarrow nat \Rightarrow 'a items **where**

bin-items-upto b i = { items b ! j | j. j < i \wedge j < length (items b) }

definition *bins-items-upto* :: 'a bins \Rightarrow nat \Rightarrow nat \Rightarrow 'a items **where**

bins-items-upto bs k i = \bigcup { set (items (bs ! l)) | l. l < k } \cup *bin-items-upto* (bs ! k) i

definition *wf-bin-items* :: 'a cfg \Rightarrow 'a sentence \Rightarrow nat \Rightarrow 'a item list \Rightarrow bool **where**

wf-bin-items cfg inp k xs = ($\forall x \in$ set xs. *wf-item* cfg inp x \wedge item-end x = k)

definition *wf-bin* :: 'a cfg \Rightarrow 'a sentence \Rightarrow nat \Rightarrow 'a bin \Rightarrow bool **where**

wf-bin cfg inp k b \longleftrightarrow distinct (items b) \wedge *wf-bin-items* cfg inp k (items b)

definition *wf-bins* :: 'a cfg \Rightarrow 'a list \Rightarrow 'a bins \Rightarrow bool **where**

wf-bins cfg inp bs \longleftrightarrow ($\forall k <$ length bs. *wf-bin* cfg inp k (bs ! k))

definition *nonempty-derives* :: 'a cfg \Rightarrow bool **where**

nonempty-derives cfg = ($\forall N. N \in$ set (\mathfrak{N} cfg) $\longrightarrow \neg$ derives cfg [N] [])

definition *Init-it* :: 'a cfg \Rightarrow 'a sentence \Rightarrow 'a bins **where**

Init-it cfg inp = (
 let rs = filter ($\lambda r. \text{rule-head } r = \mathfrak{S} \text{ cfg}$) ($\mathfrak{R} \text{ cfg}$) in
 let b0 = map ($\lambda r. (\text{Entry } (\text{init-item } r \ 0) \ \text{Null})$) rs in
 let bs = replicate (length inp + 1) ([]) in
 bs[0 := b0])

definition *Scan-it* :: nat \Rightarrow 'a sentence \Rightarrow 'a \Rightarrow 'a item \Rightarrow nat \Rightarrow 'a entry list **where**

Scan-it k inp a x pre = (
 if inp!k = a then
 let x' = inc-item x (k+1) in
 [Entry x' (Pre pre)]
 else [])

definition *Predict-it* :: nat \Rightarrow 'a cfg \Rightarrow 'a \Rightarrow 'a entry list **where**

Predict-it k cfg X = (
 let rs = filter ($\lambda r. \text{rule-head } r = X$) ($\mathfrak{R} \text{ cfg}$) in
 map ($\lambda r. (\text{Entry } (\text{init-item } r \ k) \ \text{Null})$) rs)

definition *Complete-it* :: nat \Rightarrow 'a item \Rightarrow 'a bins \Rightarrow nat \Rightarrow 'a entry list **where**

Complete-it k y bs red = (
 let orig = bs ! (item-origin y) in
 let is = filter-with-index ($\lambda x. \text{next-symbol } x = \text{Some } (\text{item-rule-head } y)$) (items orig) in
 map ($\lambda(x, \text{pre}). (\text{Entry } (\text{inc-item } x \ k) (\text{PreRed } (\text{item-origin } y, \text{pre}, \text{red}) []))$) is)

fun *bin-upd* :: 'a entry \Rightarrow 'a bin \Rightarrow 'a bin **where**

```

bin-upd e' [] = [e']
| bin-upd e' (e#es) = (
  case (e', e) of
    (Entry x (PreRed px xs), Entry y (PreRed py ys)) =>
      if x = y then Entry x (PreRed py (px#xs@ys)) # es
      else e # bin-upd e' es
  | - =>
    if item e' = item e then e # es
    else e # bin-upd e' es)

```

```

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

```

```

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

```

```

partial-function (tailrec)  $\pi$ -it' :: nat => 'a cfg => 'a sentence => 'a bins => nat => 'a bins where
   $\pi$ -it' k cfg inp bs i = (
    if i ≥ length (items (bs ! k)) then bs
  else
    let x = items (bs!k) ! i in
    let bs' =
      case next-symbol x of
        Some a =>
          if is-terminal cfg a then
            if k < length inp then bins-upd bs (k+1) (Scan-it k inp a x i)
            else bs
          else bins-upd bs k (Predict-it k cfg a)
        | None => bins-upd bs k (Complete-it k x bs i)
    in  $\pi$ -it' k cfg inp bs' (i+1))

```

```

definition  $\pi$ -it :: nat => 'a cfg => 'a sentence => 'a bins => 'a bins where
   $\pi$ -it k cfg inp bs =  $\pi$ -it' k cfg inp bs 0

```

```

fun  $\mathcal{I}$ -it :: nat => 'a cfg => 'a sentence => 'a bins where
   $\mathcal{I}$ -it 0 cfg inp =  $\pi$ -it 0 cfg inp (Init-it cfg inp)
|  $\mathcal{I}$ -it (Suc n) cfg inp =  $\pi$ -it (Suc n) cfg inp ( $\mathcal{I}$ -it n cfg inp)

```

```

definition  $\mathcal{J}$ -it :: 'a cfg => 'a sentence => 'a bins where
   $\mathcal{J}$ -it cfg inp =  $\mathcal{I}$ -it (length inp) cfg inp

```

7.2 Wellformedness

lemma *distinct-bin-upd*:

assumes *distinct* (items *b*)

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

lemma *distinct-bin-upds*:

assumes *distinct* (items *b*)

shows *distinct* (items (bin-upds *es b*))

lemma *distinct-bins-upd*:

assumes *distinct* (items (*bs ! k*))

shows *distinct* (items (bins-upd *bs k ips ! k*))

lemma *distinct-Scan-it*:

shows *distinct* (items (Scan-it *k inp a x pre*))

sorry

lemma *distinct-Predict-it*:

assumes *wf-cfg* *cfg*

shows *distinct* (items (Predict-it *k cfg X*))

lemma *distinct-Complete-it*:

assumes *wf-bins* *cfg inp bs item-origin y < length bs*

shows *distinct* (items (Complete-it *k y bs red*))

lemma *wf-bin-bin-upd*:

assumes *wf-bin* *cfg inp k b wf-item* *cfg inp (item e) ∧ item-end (item e) = k*

shows *wf-bin* *cfg inp k (bin-upd e b)*

lemma *wf-bin-bin-upds*:

assumes *wf-bin* *cfg inp k b distinct* (items *es*)

assumes $\forall x \in \text{set (items es)}. \text{wf-item } \text{cfg } \text{inp } x \wedge \text{item-end } x = k$

shows *wf-bin* *cfg inp k (bin-upds es b)*

lemma *wf-bins-bins-upd*:

assumes *wf-bins* *cfg inp bs distinct* (items *es*)

assumes $\forall x \in \text{set (items es)}. \text{wf-item } \text{cfg } \text{inp } x \wedge \text{item-end } x = k$

shows *wf-bins* *cfg inp (bins-upd bs k es)*

lemma *wf-bins-Init-it*:

assumes *wf-cfg* *cfg*

shows *wf-bins* *cfg inp (Init-it* *cfg inp*)

lemma *wf-bins-Scan-it*:

assumes *wf-bins* *cfg inp bs k < length bs x ∈ set (items (bs ! k)) k < length inp next-symbol x ≠*

None

shows $\forall y \in \text{set (items (Scan-it } k \text{ inp } a \text{ x pre))}. \text{wf-item } \text{cfg } \text{inp } y \wedge \text{item-end } y = (k+1)$

lemma *wf-bins-Predict-it*:

assumes *wf-bins* *cfg inp bs k < length bs k ≤ length inp wf-cfg* *cfg*

shows $\forall y \in \text{set (items (Predict-it } k \text{ cfg } X))}. \text{wf-item } \text{cfg } \text{inp } y \wedge \text{item-end } y = k$

lemma *wf-bins-Complete-it*:

assumes *wf-bins* *cfg inp bs k < length bs y ∈ set (items (bs ! k))*

shows $\forall x \in \text{set } (\text{items } (\text{Complete-it } k \ y \ bs \ red)). \text{wf-item } \text{cfg } \text{inp } x \wedge \text{item-end } x = k$

definition *wellformed-bins* :: $(\text{nat} \times 'a \ \text{cfg} \times 'a \ \text{sentence} \times 'a \ \text{bins}) \ \text{set}$ **where**

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

typedef *'a wf-bins* = *wellformed-bins*:: $(\text{nat} \times 'a \ \text{cfg} \times 'a \ \text{sentence} \times 'a \ \text{bins}) \ \text{set}$

lemma *wellformed-bins-Init-it*:

assumes $k \leq \text{length } \text{inp} \ \text{wf-cfg } \text{cfg}$
shows $(k, \text{cfg}, \text{inp}, \text{Init-it } \text{cfg } \text{inp}) \in \text{wellformed-bins}$

lemma *wellformed-bins-Complete-it*:

assumes $(k, \text{cfg}, \text{inp}, \text{bs}) \in \text{wellformed-bins} \neg \text{length } (\text{items } (\text{bs } ! \ k)) \leq i$
assumes $x = \text{items } (\text{bs } ! \ k) \ ! \ i \ \text{next-symbol } x = \text{None}$
shows $(k, \text{cfg}, \text{inp}, \text{bins-upd } \text{bs } k \ (\text{Complete-it } k \ x \ \text{bs } \text{red})) \in \text{wellformed-bins}$

lemma *wellformed-bins-Scan-it*:

assumes $(k, \text{cfg}, \text{inp}, \text{bs}) \in \text{wellformed-bins} \neg \text{length } (\text{items } (\text{bs } ! \ k)) \leq i$
assumes $x = \text{items } (\text{bs } ! \ k) \ ! \ i \ \text{next-symbol } x = \text{Some } a$
assumes $\text{is-terminal } \text{cfg } a \ k < \text{length } \text{inp}$
shows $(k, \text{cfg}, \text{inp}, \text{bins-upd } \text{bs } (k+1) \ (\text{Scan-it } k \ \text{inp } a \ x \ \text{pre})) \in \text{wellformed-bins}$

lemma *wellformed-bins-Predict-it*:

assumes $(k, \text{cfg}, \text{inp}, \text{bs}) \in \text{wellformed-bins} \neg \text{length } (\text{items } (\text{bs } ! \ k)) \leq i$
assumes $x = \text{items } (\text{bs } ! \ k) \ ! \ i \ \text{next-symbol } x = \text{Some } a \neg \text{is-terminal } \text{cfg } a$
shows $(k, \text{cfg}, \text{inp}, \text{bins-upd } \text{bs } k \ (\text{Predict-it } k \ \text{cfg } a)) \in \text{wellformed-bins}$

fun *earley-measure* :: $\text{nat} \times 'a \ \text{cfg} \times 'a \ \text{sentence} \times 'a \ \text{bins} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**

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

lemma $\pi\text{-it}'\text{-induct}$:

assumes $(k, \text{cfg}, \text{inp}, \text{bs}) \in \text{wellformed-bins}$
assumes *base*: $\bigwedge k \ \text{cfg} \ \text{inp} \ \text{bs} \ i. \ i \geq \text{length } (\text{items } (\text{bs } ! \ k)) \Longrightarrow P \ k \ \text{cfg} \ \text{inp} \ \text{bs} \ i$
assumes *complete*: $\bigwedge k \ \text{cfg} \ \text{inp} \ \text{bs} \ i \ x. \neg i \geq \text{length } (\text{items } (\text{bs } ! \ k)) \Longrightarrow x = \text{items } (\text{bs } ! \ k) \ ! \ i \Longrightarrow$
 $\text{next-symbol } x = \text{None} \Longrightarrow P \ k \ \text{cfg} \ \text{inp} \ (\text{bins-upd } \text{bs } k \ (\text{Complete-it } k \ x \ \text{bs } i)) \ (i+1) \Longrightarrow P \ k$
 $\text{cfg } \text{inp} \ \text{bs} \ i$
assumes *scan*: $\bigwedge k \ \text{cfg} \ \text{inp} \ \text{bs} \ i \ x \ a. \neg i \geq \text{length } (\text{items } (\text{bs } ! \ k)) \Longrightarrow x = \text{items } (\text{bs } ! \ k) \ ! \ i \Longrightarrow$
 $\text{next-symbol } x = \text{Some } a \Longrightarrow \text{is-terminal } \text{cfg } a \Longrightarrow k < \text{length } \text{inp} \Longrightarrow$
 $P \ k \ \text{cfg} \ \text{inp} \ (\text{bins-upd } \text{bs } (k+1) \ (\text{Scan-it } k \ \text{inp } a \ x \ i)) \ (i+1) \Longrightarrow P \ k \ \text{cfg} \ \text{inp} \ \text{bs} \ i$
assumes *pass*: $\bigwedge k \ \text{cfg} \ \text{inp} \ \text{bs} \ i \ x \ a. \neg i \geq \text{length } (\text{items } (\text{bs } ! \ k)) \Longrightarrow x = \text{items } (\text{bs } ! \ k) \ ! \ i \Longrightarrow$
 $\text{next-symbol } x = \text{Some } a \Longrightarrow \text{is-terminal } \text{cfg } a \Longrightarrow \neg k < \text{length } \text{inp} \Longrightarrow$
 $P \ k \ \text{cfg} \ \text{inp} \ \text{bs} \ (i+1) \Longrightarrow P \ k \ \text{cfg} \ \text{inp} \ \text{bs} \ i$

assumes *predict*: $\wedge k \text{ cfg inp bs } i \ x \ a. \neg i \geq \text{length } (\text{items } (bs ! k)) \implies x = \text{items } (bs ! k) ! i \implies$
 $\text{next-symbol } x = \text{Some } a \implies \neg \text{is-terminal cfg } a \implies$
 $P \ k \ \text{cfg inp } (\text{bins-upd } bs \ k \ (\text{Predict-it } k \ \text{cfg } a)) \ (i+1) \implies P \ k \ \text{cfg inp bs } i$
shows $P \ k \ \text{cfg inp bs } i$
lemma *wellformed-bins- π -it'*:
assumes $(k, \text{cfg}, \text{inp}, bs) \in \text{wellformed-bins}$
shows $(k, \text{cfg}, \text{inp}, \pi\text{-it}' \ k \ \text{cfg inp bs } i) \in \text{wellformed-bins}$
lemma *wellformed-bins- π -it*:
assumes $(k, \text{cfg}, \text{inp}, bs) \in \text{wellformed-bins}$
shows $(k, \text{cfg}, \text{inp}, \pi\text{-it } k \ \text{cfg inp bs}) \in \text{wellformed-bins}$
lemma *wellformed-bins- \mathcal{I} -it*:
assumes $k \leq \text{length inp wf-cfg cfg}$
shows $(k, \text{cfg}, \text{inp}, \mathcal{I}\text{-it } k \ \text{cfg inp}) \in \text{wellformed-bins}$
lemma *wellformed-bins- \mathcal{J} -it*:
assumes $k \leq \text{length inp wf-cfg cfg}$
shows $(k, \text{cfg}, \text{inp}, \mathcal{J}\text{-it } k \ \text{cfg inp}) \in \text{wellformed-bins}$
lemma *wf-bins- π -it'*:
assumes $(k, \text{cfg}, \text{inp}, bs) \in \text{wellformed-bins}$
shows $\text{wf-bins cfg inp } (\pi\text{-it}' \ k \ \text{cfg inp bs } i)$
lemma *wf-bins- π -it*:
assumes $(k, \text{cfg}, \text{inp}, bs) \in \text{wellformed-bins}$
shows $\text{wf-bins cfg inp } (\pi\text{-it } k \ \text{cfg inp bs})$
lemma *wf-bins- \mathcal{I} -it*:
assumes $k \leq \text{length inp wf-cfg cfg}$
shows $\text{wf-bins cfg inp } (\mathcal{I}\text{-it } k \ \text{cfg inp})$
lemma *wf-bins- \mathcal{J} -it*:
assumes wf-cfg cfg
shows $\text{wf-bins cfg inp } (\mathcal{J}\text{-it } k \ \text{cfg inp})$

7.3 List to set

lemma *Init-it-eq-Init*:
shows $\text{bins-items } (\text{Init-it } k \ \text{cfg inp}) = \text{Init } k \ \text{cfg}$
lemma *Scan-it-sub-Scan*:
assumes $\text{wf-bins cfg inp bs bins-items bs} \subseteq I \ x \in \text{set } (\text{items } (bs ! k))$
assumes $k < \text{length bs } k < \text{length inp}$
assumes $\text{next-symbol } x = \text{Some } a$
shows $\text{set } (\text{items } (\text{Scan-it } k \ \text{inp } a \ x \ \text{pre})) \subseteq \text{Scan } k \ \text{inp } I$
lemma *Predict-it-sub-Predict*:
assumes $\text{wf-bins cfg inp bs bins-items bs} \subseteq I \ x \in \text{set } (\text{items } (bs ! k)) \ k < \text{length bs}$
assumes $\text{next-symbol } x = \text{Some } X$
shows $\text{set } (\text{items } (\text{Predict-it } k \ \text{cfg } X)) \subseteq \text{Predict } k \ \text{cfg } I$
lemma *Complete-it-sub-Complete*:

assumes $wf\text{-}bins\ cf\ g\ inp\ bs\ bins\text{-}items\ bs \subseteq I\ y \in set\ (items\ (bs\ !\ k))\ k < length\ bs$
assumes $next\text{-}symbol\ y = None$
shows $set\ (items\ (Complete\text{-}it\ k\ y\ bs\ red)) \subseteq Complete\ k\ I$
lemma $\pi\text{-}it'\text{-}sub\text{-}\pi$:
assumes $(k, cf\ g, inp, bs) \in wellformed\text{-}bins$
assumes $bins\text{-}items\ bs \subseteq I$
shows $bins\text{-}items\ (\pi\text{-}it'\ k\ cf\ g\ inp\ bs\ i) \subseteq \pi\ k\ cf\ g\ inp\ I$
lemma $\pi\text{-}it\text{-}sub\text{-}\pi$:
assumes $(k, cf\ g, inp, bs) \in wellformed\text{-}bins$
assumes $bins\text{-}items\ bs \subseteq I$
shows $bins\text{-}items\ (\pi\text{-}it\ k\ cf\ g\ inp\ bs) \subseteq \pi\ k\ cf\ g\ inp\ I$
lemma $\mathcal{I}\text{-}it\text{-}sub\text{-}\mathcal{I}$:
assumes $k \leq length\ inp\ wf\text{-}cf\ g\ cf\ g$
shows $bins\text{-}items\ (\mathcal{I}\text{-}it\ k\ cf\ g\ inp) \subseteq \mathcal{I}\ k\ cf\ g\ inp$
lemma $\mathcal{J}\text{-}it\text{-}sub\text{-}\mathcal{J}$:
assumes $wf\text{-}cf\ g\ cf\ g$
shows $bins\text{-}items\ (\mathcal{J}\text{-}it\ cf\ g\ inp) \subseteq \mathcal{J}\ cf\ g\ inp$

7.4 Soundness

lemma $sound\text{-}Scan\text{-}it$:
assumes $wf\text{-}bins\ cf\ g\ inp\ bs\ bins\text{-}items\ bs \subseteq I\ x \in set\ (items\ (bs\ !\ k))\ k < length\ bs\ k < length\ inp$
assumes $next\text{-}symbol\ x = Some\ a\ wf\text{-}items\ cf\ g\ inp\ I\ sound\text{-}items\ cf\ g\ inp\ I$
shows $sound\text{-}items\ cf\ g\ inp\ (set\ (items\ (Scan\text{-}it\ k\ inp\ a\ x\ i)))$
lemma $sound\text{-}Predict\text{-}it$:
assumes $wf\text{-}bins\ cf\ g\ inp\ bs\ bins\text{-}items\ bs \subseteq I\ x \in set\ (items\ (bs\ !\ k))\ k < length\ bs$
assumes $next\text{-}symbol\ x = Some\ X\ sound\text{-}items\ cf\ g\ inp\ I$
shows $sound\text{-}items\ cf\ g\ inp\ (set\ (items\ (Predict\text{-}it\ k\ cf\ g\ X)))$
lemma $sound\text{-}Complete\text{-}it$:
assumes $wf\text{-}bins\ cf\ g\ inp\ bs\ bins\text{-}items\ bs \subseteq I\ y \in set\ (items\ (bs\ !\ k))\ k < length\ bs$
assumes $next\text{-}symbol\ y = None\ wf\text{-}items\ cf\ g\ inp\ I\ sound\text{-}items\ cf\ g\ inp\ I$
shows $sound\text{-}items\ cf\ g\ inp\ (set\ (items\ (Complete\text{-}it\ k\ y\ bs\ i)))$
lemma $sound\text{-}\pi\text{-}it'$:
assumes $(k, cf\ g, inp, bs) \in wellformed\text{-}bins$
assumes $sound\text{-}items\ cf\ g\ inp\ (bins\text{-}items\ bs)$
shows $sound\text{-}items\ cf\ g\ inp\ (bins\text{-}items\ (\pi\text{-}it'\ k\ cf\ g\ inp\ bs\ i))$
lemma $sound\text{-}\pi\text{-}it$:
assumes $(k, cf\ g, inp, bs) \in wellformed\text{-}bins$
assumes $sound\text{-}items\ cf\ g\ inp\ (bins\text{-}items\ bs)$
shows $sound\text{-}items\ cf\ g\ inp\ (bins\text{-}items\ (\pi\text{-}it\ k\ cf\ g\ inp\ bs))$

7.5 Set to list

lemma *impossible-complete-item*:

assumes *wf-cfg cfg wf-item cfg inp x sound-item cfg inp x*

assumes *is-complete x item-origin x = k item-end x = k nonempty-derives cfg*

shows *False*

lemma *Complete-Un-eq-terminal*:

assumes *next-symbol z = Some a is-terminal cfg a wf-items cfg inp I wf-item cfg inp z wf-cfg cfg*

shows *Complete k (I \cup {z}) = Complete k I*

lemma *Complete-Un-eq-nonterminal*:

assumes *next-symbol z = Some a is-nonterminal cfg a sound-items cfg inp I item-end z = k*

assumes *wf-items cfg inp I wf-item cfg inp z wf-cfg cfg nonempty-derives cfg*

shows *Complete k (I \cup {z}) = Complete k I*

lemma *Complete-sub-bins-Un-Complete-it*:

assumes *Complete k I \subseteq bins-items bs I \subseteq bins-items bs is-complete z wf-bins cfg inp bs wf-item cfg inp z*

shows *Complete k (I \cup {z}) \subseteq bins-items bs \cup set (items (Complete-it k z bs red))*

lemma *π -it'-mono*:

assumes *(k, cfg, inp, bs) \in wellformed-bins*

shows *bins-items bs \subseteq bins-items (π -it' k cfg inp bs i)*

lemma *π -step-sub- π -it'*:

assumes *(k, cfg, inp, bs) \in wellformed-bins*

assumes *π -step k cfg inp (bins-items-upto bs k i) \subseteq bins-items bs*

assumes *sound-items cfg inp (bins-items bs) is-word cfg inp nonempty-derives cfg*

shows *π -step k cfg inp (bins-items bs) \subseteq bins-items (π -it' k cfg inp bs i)*

lemma *π -step-sub- π -it*:

assumes *(k, cfg, inp, bs) \in wellformed-bins*

assumes *π -step k cfg inp (bins-items-upto bs k 0) \subseteq bins-items bs*

assumes *sound-items cfg inp (bins-items bs) is-word cfg inp nonempty-derives cfg*

shows *π -step k cfg inp (bins-items bs) \subseteq bins-items (π -it k cfg inp bs)*

lemma *π -it'-bins-items-eq*:

assumes *(k, cfg, inp, as) \in wellformed-bins*

assumes *bins-eq-items as bs wf-bins cfg inp as*

shows *bins-eq-items (π -it' k cfg inp as i) (π -it' k cfg inp bs i)*

lemma *π -it'-idem*:

assumes *(k, cfg, inp, bs) \in wellformed-bins*

assumes *$i \leq j$ sound-items cfg inp (bins-items bs) nonempty-derives cfg*

shows *bins-items (π -it' k cfg inp (π -it' k cfg inp bs i) j) = bins-items (π -it' k cfg inp bs i)*

lemma *π -it-idem*:

assumes *(k, cfg, inp, bs) \in wellformed-bins*

assumes *sound-items cfg inp (bins-items bs) nonempty-derives cfg*

shows *bins-items (π -it k cfg inp (π -it k cfg inp bs)) = bins-items (π -it k cfg inp bs)*

lemma *funpower- π -step-sub- π -it*:

assumes *(k, cfg, inp, bs) \in wellformed-bins*

assumes $\pi\text{-step } k \text{ cfg inp } (\text{bins-items-upto } bs \ k \ 0) \subseteq \text{bins-items } bs \ \text{sound-items } \text{cfg inp } (\text{bins-items } bs)$

assumes $\text{is-word } \text{cfg inp nonempty-derives } \text{cfg}$

shows $\text{funpower } (\pi\text{-step } k \text{ cfg inp}) \ n \ (\text{bins-items } bs) \subseteq \text{bins-items } (\pi\text{-it } k \text{ cfg inp } bs)$

lemma $\pi\text{-sub-}\pi\text{-it}$:

assumes $(k, \text{cfg}, \text{inp}, bs) \in \text{wellformed-bins}$

assumes $\pi\text{-step } k \text{ cfg inp } (\text{bins-items-upto } bs \ k \ 0) \subseteq \text{bins-items } bs \ \text{sound-items } \text{cfg inp } (\text{bins-items } bs)$

assumes $\text{is-word } \text{cfg inp nonempty-derives } \text{cfg}$

shows $\pi \ k \ \text{cfg inp } (\text{bins-items } bs) \subseteq \text{bins-items } (\pi\text{-it } k \ \text{cfg inp } bs)$

lemma $\mathcal{I}\text{-sub-}\mathcal{I}\text{-it}$:

assumes $k \leq \text{length } \text{inp } \text{wf-cfg } \text{cfg}$

assumes $\text{is-word } \text{cfg inp nonempty-derives } \text{cfg}$

shows $\mathcal{I} \ k \ \text{cfg inp} \subseteq \text{bins-items } (\mathcal{I}\text{-it } k \ \text{cfg inp})$

lemma $\mathcal{J}\text{-sub-}\mathcal{J}\text{-it}$:

assumes $\text{wf-cfg } \text{cfg is-word } \text{cfg inp nonempty-derives } \text{cfg}$

shows $\mathcal{J} \ \text{cfg inp} \subseteq \text{bins-items } (\mathcal{J}\text{-it } \text{cfg inp})$

7.6 Main Theorem

definition $\text{earley-recognized-it} :: 'a \ \text{bins} \Rightarrow 'a \ \text{cfg} \Rightarrow 'a \ \text{sentence} \Rightarrow \text{bool}$ **where**
 $\text{earley-recognized-it } I \ \text{cfg inp} = (\exists x \in \text{set } (\text{items } (I \ ! \ \text{length } \text{inp}))). \text{is-finished } \text{cfg inp } x)$

theorem $\text{earley-recognized-it-iff-earley-recognized}$:

assumes $\text{wf-cfg } \text{cfg is-word } \text{cfg inp nonempty-derives } \text{cfg}$

shows $\text{earley-recognized-it } (\mathcal{J}\text{-it } \text{cfg inp}) \ \text{cfg inp} \longleftrightarrow \text{earley-recognized } (\mathcal{J} \ \text{cfg inp}) \ \text{cfg inp}$

corollary correctness-list :

assumes $\text{wf-cfg } \text{cfg is-word } \text{cfg inp nonempty-derives } \text{cfg}$

shows $\text{earley-recognized-it } (\mathcal{J}\text{-it } \text{cfg inp}) \ \text{cfg inp} \longleftrightarrow \text{derives } \text{cfg } [\mathcal{S} \ \text{cfg}] \ \text{inp}$

8 Earley Parser Implementation

8.1 Draft

8.2 Pointer lemmas

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *sound-ptrs-Init-it*:
shows *sound-ptrs inp (Init-it cfg inp)*

lemma *mono-red-ptr-Init-it*:
shows *mono-red-ptr (Init-it cfg inp)*

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

lemma *sound-mono-ptrs- \mathcal{J} -it*:
assumes *wf-cfg cfg nonempty-derives cfg*
shows *sound-ptrs inp (\mathcal{J} -it cfg inp) \wedge mono-red-ptr (\mathcal{J} -it cfg inp)*

8.3 Trees and Forests

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

```

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

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

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

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

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

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

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

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

```

8.4 A single parse tree

```

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

```

```

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

```

definition build-tree :: 'a cfg \Rightarrow 'a sentence \Rightarrow 'a bins \Rightarrow 'a tree option **where**
 build-tree cfg inp bs = (
 let k = length bs - 1 in (
 case filter-with-index ($\lambda x. \text{is-finished } \text{cfg } \text{inp } x$) (items (bs!k)) of
 [] \Rightarrow None
 | (-, i)#- \Rightarrow build-tree' bs inp k i
))

definition wellformed-tree-ptrs :: ('a bins \times 'a sentence \times nat \times nat) set **where**
 wellformed-tree-ptrs = {
 (bs, inp, k, i) | bs inp k i.
 sound-ptrs inp bs \wedge
 mono-red-ptr bs \wedge
 k < length bs \wedge
 i < length (bs!k)
 }

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

lemma wellformed-tree-ptrs-pre:

assumes $(bs, inp, k, i) \in \text{wellformed-tree-ptrs}$
assumes $e = bs!k!i$ pointer $e = \text{Pre } pre$
shows $(bs, inp, (k-1), pre) \in \text{wellformed-tree-ptrs}$
lemma *wellformed-tree-ptrs-prered-pre*:
assumes $(bs, inp, k, i) \in \text{wellformed-tree-ptrs}$
assumes $e = bs!k!i$ pointer $e = \text{PreRed } (k', pre, red) \text{ } ps$
shows $(bs, inp, k', pre) \in \text{wellformed-tree-ptrs}$
lemma *wellformed-tree-ptrs-prered-red*:
assumes $(bs, inp, k, i) \in \text{wellformed-tree-ptrs}$
assumes $e = bs!k!i$ pointer $e = \text{PreRed } (k', pre, red) \text{ } ps$
shows $(bs, inp, k, red) \in \text{wellformed-tree-ptrs}$
lemma *build-tree'-induct*:
assumes $(bs, inp, k, i) \in \text{wellformed-tree-ptrs}$
assumes $\wedge bs \text{ } inp \text{ } k \text{ } i$.
 $(\wedge e \text{ } pre. e = bs!k!i \implies \text{pointer } e = \text{Pre } pre \implies P \text{ } bs \text{ } inp \text{ } (k-1) \text{ } pre) \implies$
 $(\wedge e \text{ } k' \text{ } pre \text{ } red \text{ } ps. e = bs!k!i \implies \text{pointer } e = \text{PreRed } (k', pre, red) \text{ } ps \implies P \text{ } bs \text{ } inp \text{ } k' \text{ } pre) \implies$
 $(\wedge e \text{ } k' \text{ } pre \text{ } red \text{ } ps. e = bs!k!i \implies \text{pointer } e = \text{PreRed } (k', pre, red) \text{ } ps \implies P \text{ } bs \text{ } inp \text{ } k \text{ } red) \implies$
 $P \text{ } bs \text{ } inp \text{ } k \text{ } i$
shows $P \text{ } bs \text{ } inp \text{ } k \text{ } i$
lemma *build-tree'-termination*:
assumes $(bs, inp, k, i) \in \text{wellformed-tree-ptrs}$
shows $\exists N \text{ } ts. \text{build-tree}' \text{ } bs \text{ } inp \text{ } k \text{ } i = \text{Some } (\text{Branch } N \text{ } ts)$
lemma *wf-item-tree-build-tree'*:
assumes $(bs, inp, k, i) \in \text{wellformed-tree-ptrs}$
assumes $wf\text{-bins } cfg \text{ } inp \text{ } bs$
assumes $k < \text{length } bs \text{ } i < \text{length } (bs!k)$
assumes $\text{build-tree}' \text{ } bs \text{ } inp \text{ } k \text{ } i = \text{Some } t$
shows $wf\text{-item-tree } cfg \text{ } (\text{item } (bs!k!i)) \text{ } t$
lemma *wf-yield-tree-build-tree'*:
assumes $(bs, inp, k, i) \in \text{wellformed-tree-ptrs}$
assumes $wf\text{-bins } cfg \text{ } inp \text{ } bs$
assumes $k < \text{length } bs \text{ } i < \text{length } (bs!k) \text{ } k \leq \text{length } inp$
assumes $\text{build-tree}' \text{ } bs \text{ } inp \text{ } k \text{ } i = \text{Some } t$
shows $wf\text{-yield-tree } inp \text{ } (\text{item } (bs!k!i)) \text{ } t$
theorem *wf-rule-root-yield-tree-build-tree*:
assumes $wf\text{-bins } cfg \text{ } inp \text{ } bs \text{ } \text{sound-ptrs } inp \text{ } bs \text{ } \text{mono-red-ptr } bs \text{ } \text{length } bs = \text{length } inp + 1$
assumes $\text{build-tree } cfg \text{ } inp \text{ } bs = \text{Some } t$
shows $wf\text{-rule-tree } cfg \text{ } t \wedge \text{root-tree } t = \mathfrak{S} \text{ } cfg \wedge \text{yield-tree } t = inp$
corollary *wf-rule-root-yield-tree-build-tree- \mathfrak{I} -it*:
assumes $wf\text{-cfg } cfg \text{ } \text{nonempty-derives } cfg$
assumes $\text{build-tree } cfg \text{ } inp \text{ } (\mathfrak{I}\text{-it } cfg \text{ } inp) = \text{Some } t$
shows $wf\text{-rule-tree } cfg \text{ } t \wedge \text{root-tree } t = \mathfrak{S} \text{ } cfg \wedge \text{yield-tree } t = inp$
theorem *correctness-build-tree- \mathfrak{I} -it*:
assumes $wf\text{-cfg } cfg \text{ } \text{is-word } cfg \text{ } inp \text{ } \text{nonempty-derives } cfg$

shows $(\exists t. \text{build-tree cfg inp } (\mathcal{I}\text{-it cfg inp}) = \text{Some } t) \longleftrightarrow \text{derives cfg } [\mathcal{S} \text{ cfg}] \text{ inp}$

8.5 Parse trees

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

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

partial-function (option) build-trees' :: 'a bins \Rightarrow 'a sentence \Rightarrow nat \Rightarrow nat \Rightarrow nat set \Rightarrow 'a forest list
option **where**

```
build-trees' bs inp k i I = (
  let e = bs!k!i in (
    case pointer e of
      Null  $\Rightarrow$  Some ([FBranch (item-rule-head (item e)) []])
    | Pre pre  $\Rightarrow$  (
      do {
        pres  $\leftarrow$  build-trees' bs inp (k-1) pre {pre};
        those (map ( $\lambda f.$ 
          case f of
            FBranch N fss  $\Rightarrow$  Some (FBranch N (fss @ [[FLeaf (inp!(k-1))]]))
          | -  $\Rightarrow$  None
        ) pres)
      })
    | PreRed p ps  $\Rightarrow$  (
      let ps' = filter ( $\lambda (k', pre, red). \text{red} \notin I$ ) (p#ps) in
      let gs = group-by ( $\lambda (k', pre, red). (k', pre)$ ) ( $\lambda (k', pre, red). \text{red}$ ) ps' in
      map-option concat (those (map ( $\lambda (k', pre, reds).$ 
        do {
          pres  $\leftarrow$  build-trees' bs inp k' pre {pre};
          rss  $\leftarrow$  those (map ( $\lambda \text{red}. \text{build-trees' bs inp k red } (I \cup \{\text{red}\})$ ) reds);
          those (map ( $\lambda f.$ 
            case f of
              FBranch N fss  $\Rightarrow$  Some (FBranch N (fss @ [concat rss]))
            | -  $\Rightarrow$  None
          ) pres)
        })
      ))
    )
  )
}
```



```

    ) gs))
  )
))

```

definition *build-trees* :: 'a cfg \Rightarrow 'a sentence \Rightarrow 'a bins \Rightarrow 'a forest list option **where**

```

build-trees cfg inp bs = (
  let k = length bs - 1 in
  let finished = filter-with-index ( $\lambda x. \text{is-finished } \text{cfg } \text{inp } x$ ) (items (bs!k)) in
  map-option concat (those (map ( $\lambda(-, i). \text{build-trees}' \text{ bs } \text{inp } k \ i \ \{i\}$ ) finished))
)

```

definition *wellformed-forest-ptrs* :: ('a bins \times 'a sentence \times nat \times nat \times nat set) set **where**

```

wellformed-forest-ptrs = {
  (bs, inp, k, i, I) | bs inp k i I.
    sound-ptrs inp bs  $\wedge$ 
    k < length bs  $\wedge$ 
    i < length (bs!k)  $\wedge$ 
    I  $\subseteq$  {0.. $\text{length } (bs!k)$ }  $\wedge$ 
    i  $\in$  I
}

```

fun *build-forest'-measure* :: ('a bins \times 'a sentence \times nat \times nat \times nat set) \Rightarrow nat **where**

```

build-forest'-measure (bs, inp, k, i, I) = foldl (+) 0 (map length (take (k+1) bs)) - card I

```

lemma *wellformed-forest-ptrs-pre*:

assumes (bs, inp, k, i, I) \in *wellformed-forest-ptrs*

assumes $e = \text{bs!k!i}$ pointer $e = \text{Pre } \text{pre}$

shows (bs, inp, (k-1), pre, {pre}) \in *wellformed-forest-ptrs*

lemma *wellformed-forest-ptrs-prered-pre*:

assumes (bs, inp, k, i, I) \in *wellformed-forest-ptrs*

assumes $e = \text{bs!k!i}$ pointer $e = \text{PreRed } p \ \text{ps}$

assumes $\text{ps}' = \text{filter } (\lambda(k', \text{pre}, \text{red}). \text{red} \notin I) (p\#\text{ps})$

assumes $\text{gs} = \text{group-by } (\lambda(k', \text{pre}, \text{red}). (k', \text{pre})) (\lambda(k', \text{pre}, \text{red}). \text{red}) \ \text{ps}'$

assumes ((k', pre), reds) \in set gs

shows (bs, inp, k', pre, {pre}) \in *wellformed-forest-ptrs*

lemma *wellformed-forest-ptrs-prered-red*:

assumes (bs, inp, k, i, I) \in *wellformed-forest-ptrs*

assumes $e = \text{bs!k!i}$ pointer $e = \text{PreRed } p \ \text{ps}$

assumes $\text{ps}' = \text{filter } (\lambda(k', \text{pre}, \text{red}). \text{red} \notin I) (p\#\text{ps})$

assumes $\text{gs} = \text{group-by } (\lambda(k', \text{pre}, \text{red}). (k', \text{pre})) (\lambda(k', \text{pre}, \text{red}). \text{red}) \ \text{ps}'$

assumes ((k', pre), reds) \in set gs $\text{red} \in$ set reds

shows (bs, inp, k, red, I \cup {red}) \in *wellformed-forest-ptrs*

lemma *build-trees'-induct*:

assumes (bs, inp, k, i, I) \in *wellformed-forest-ptrs*

assumes $\wedge bs \text{ inp } k \text{ i } I.$

$(\wedge e \text{ pre. } e = bs!k!i \implies \text{pointer } e = \text{Pre pre} \implies P \text{ bs inp } (k-1) \text{ pre } \{pre\}) \implies$
 $(\wedge e \text{ p ps ps' gs k' pre reds. } e = bs!k!i \implies \text{pointer } e = \text{PreRed p ps} \implies$
 $ps' = \text{filter } (\lambda(k', \text{pre}, \text{red}). \text{red} \notin I) (p\#ps) \implies$
 $gs = \text{group-by } (\lambda(k', \text{pre}, \text{red}). (k', \text{pre})) (\lambda(k', \text{pre}, \text{red}). \text{red}) ps' \implies$
 $((k', \text{pre}), \text{reds}) \in \text{set } gs \implies P \text{ bs inp } k' \text{ pre } \{pre\}) \implies$
 $(\wedge e \text{ p ps ps' gs k' pre red reds reds'. } e = bs!k!i \implies \text{pointer } e = \text{PreRed p ps} \implies$
 $ps' = \text{filter } (\lambda(k', \text{pre}, \text{red}). \text{red} \notin I) (p\#ps) \implies$
 $gs = \text{group-by } (\lambda(k', \text{pre}, \text{red}). (k', \text{pre})) (\lambda(k', \text{pre}, \text{red}). \text{red}) ps' \implies$
 $((k', \text{pre}), \text{reds}) \in \text{set } gs \implies \text{red} \in \text{set } reds \implies P \text{ bs inp } k \text{ red } (I \cup \{\text{red}\})) \implies$
 $P \text{ bs inp } k \text{ i } I$

shows $P \text{ bs inp } k \text{ i } I$

lemma *build-trees'-termination:*

assumes $(bs, \text{inp}, k, i, I) \in \text{wellformed-forest-pters}$

shows $\exists fs. \text{build-trees}' \text{ bs inp } k \text{ i } I = \text{Some } fs \wedge (\forall f \in \text{set } fs. \exists N \text{ fss. } f = \text{FBranch } N \text{ fss})$

lemma *wf-item-tree-build-trees':*

assumes $(bs, \text{inp}, k, i, I) \in \text{wellformed-forest-pters}$

assumes $\text{wf-bins cfg inp bs}$

assumes $k < \text{length } bs \text{ i} < \text{length } (bs!k)$

assumes $\text{build-trees}' \text{ bs inp } k \text{ i } I = \text{Some } fs$

assumes $f \in \text{set } fs$

assumes $t \in \text{set } (\text{trees } f)$

shows $\text{wf-item-tree cfg (item (bs!k!i)) } t$

lemma *wf-yield-tree-build-trees':*

assumes $(bs, \text{inp}, k, i, I) \in \text{wellformed-forest-pters}$

assumes $\text{wf-bins cfg inp bs}$

assumes $k < \text{length } bs \text{ i} < \text{length } (bs!k) \text{ k} \leq \text{length } \text{inp}$

assumes $\text{build-trees}' \text{ bs inp } k \text{ i } I = \text{Some } fs$

assumes $f \in \text{set } fs$

assumes $t \in \text{set } (\text{trees } f)$

shows $\text{wf-yield-tree inp (item (bs!k!i)) } t$

theorem *wf-rule-root-yield-tree-build-trees:*

assumes $\text{wf-bins cfg inp bs sound-pters inp bs length } bs = \text{length } \text{inp} + 1$

assumes $\text{build-trees cfg inp bs} = \text{Some } fs \text{ f} \in \text{set } fs \text{ t} \in \text{set } (\text{trees } f)$

shows $\text{wf-rule-tree cfg t} \wedge \text{root-tree t} = \mathfrak{S} \text{ cfg} \wedge \text{yield-tree t} = \text{inp}$

corollary *wf-rule-root-yield-tree-build-trees- \mathfrak{I} -it:*

assumes $\text{wf-cfg cfg nonempty-derives cfg}$

assumes $\text{build-trees cfg inp } (\mathfrak{I}\text{-it cfg inp}) = \text{Some } fs \text{ f} \in \text{set } fs \text{ t} \in \text{set } (\text{trees } f)$

shows $\text{wf-rule-tree cfg t} \wedge \text{root-tree t} = \mathfrak{S} \text{ cfg} \wedge \text{yield-tree t} = \text{inp}$

theorem *soundness-build-trees- \mathfrak{I} -it:*

assumes $\text{wf-cfg cfg is-word cfg inp nonempty-derives cfg}$

assumes $\text{build-trees cfg inp } (\mathfrak{I}\text{-it cfg inp}) = \text{Some } fs \text{ f} \in \text{set } fs \text{ t} \in \text{set } (\text{trees } f)$

shows $\text{derives cfg } [\mathfrak{S} \text{ cfg}] \text{ inp}$

theorem *termination-build-tree- \mathfrak{I} -it:*

assumes *wf-cfg cfg nonempty-derives cfg derives cfg* $[\S \text{ cfg}] \text{ inp}$

shows $\exists fs. \text{ build-trees } \text{cfg inp} (\mathcal{I}\text{-it } \text{cfg inp}) = \text{Some } fs$

8.6 A word on completeness

9 Examples

9.1 epsilon free CFG

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

lemma $\varepsilon\text{-free-impl-non-empty-deriv}$:
 $\varepsilon\text{-free } \text{cfg} \Longrightarrow N \in \text{set } (\mathfrak{N} \text{ cfg}) \Longrightarrow \neg \text{derives } \text{cfg } [N] []$

9.2 Example 1: Addition

datatype $t1 = x \mid \text{plus}$
datatype $n1 = S$
datatype $s1 = \text{Terminal } t1 \mid \text{Nonterminal } n1$

definition $\text{nonterminals1} :: s1 \text{ list}$ **where**
 $\text{nonterminals1} = [\text{Nonterminal } S]$

definition $\text{terminals1} :: s1 \text{ list}$ **where**
 $\text{terminals1} = [\text{Terminal } x, \text{Terminal } \text{plus}]$

definition $\text{rules1} :: s1 \text{ rule list}$ **where**
 $\text{rules1} = [$
 $(\text{Nonterminal } S, [\text{Terminal } x]),$
 $(\text{Nonterminal } S, [\text{Nonterminal } S, \text{Terminal } \text{plus}, \text{Nonterminal } S])$
]

definition $\text{start-symbol1} :: s1$ **where**
 $\text{start-symbol1} = \text{Nonterminal } S$

definition $\text{cfg1} :: s1 \text{ cfg}$ **where**
 $\text{cfg1} = \text{CFG } \text{nonterminals1 } \text{terminals1 } \text{rules1 } \text{start-symbol1}$

definition $\text{inp1} :: s1 \text{ list}$ **where**
 $\text{inp1} = [\text{Terminal } x, \text{Terminal } \text{plus}, \text{Terminal } x, \text{Terminal } \text{plus}, \text{Terminal } x]$

lemma wf-cfg1 :

```

shows wf-cfg cfg1
lemma is-word-inp1:
  shows is-word cfg1 inp1
lemma nonempty-derives1:
  shows nonempty-derives cfg1
lemma correctness1:
  shows earley-recognized-it ( $\mathcal{I}$ -it cfg1 inp1) cfg1 inp1  $\longleftrightarrow$  derives cfg1 [ $\mathcal{S}$  cfg1] inp1
fun size-bins :: 'a bins  $\Rightarrow$  nat where
  size-bins bs = fold (+) (map length bs) 0

value  $\mathcal{I}$ -it cfg1 inp1
value size-bins ( $\mathcal{I}$ -it cfg1 inp1)
value earley-recognized-it ( $\mathcal{I}$ -it cfg1 inp1) cfg1 inp1
value build-trees cfg1 inp1 ( $\mathcal{I}$ -it cfg1 inp1)
value map-option (map trees) (build-trees cfg1 inp1 ( $\mathcal{I}$ -it cfg1 inp1))
value map-option (foldl (+) 0  $\circ$  map length) (map-option (map trees) (build-trees cfg1 inp1 ( $\mathcal{I}$ -it cfg1 inp1)))

```

9.2.1 Example 2: Cyclic reduction pointers

```

datatype t2 = x
datatype n2 = A | B
datatype s2 = Terminal t2 | Nonterminal n2

definition nonterminals2 :: s2 list where
  nonterminals2 = [Nonterminal A, Nonterminal B]

```

```

definition terminals2 :: s2 list where
  terminals2 = [Terminal x]

```

```

definition rules2 :: s2 rule list where
  rules2 = [
    (Nonterminal B, [Nonterminal A]),
    (Nonterminal A, [Nonterminal B]),
    (Nonterminal A, [Terminal x])
  ]

```

```

definition start-symbol2 :: s2 where
  start-symbol2 = Nonterminal A

```

```

definition cfg2 :: s2 cfg where
  cfg2 = CFG nonterminals2 terminals2 rules2 start-symbol2

```

```

definition inp2 :: s2 list where

```

inp2 = [Terminal *x*]

lemma *wf-cfg2*:

shows *wf-cfg* *cfg2*

lemma *is-word-inp2*:

shows *is-word* *cfg2 inp2*

lemma *nonempty-derives2*:

shows *nonempty-derives* *cfg2*

lemma *correctness2*:

shows *earley-recognized-it* (*ℑ-it* *cfg2 inp2*) *cfg2 inp2* \longleftrightarrow *derives* *cfg2* [*ℑ* *cfg2*] *inp2*

value *ℑ-it* *cfg2 inp2*

value *earley-recognized-it* (*ℑ-it* *cfg2 inp2*) *cfg2 inp2*

value *build-trees* *cfg2 inp2* (*ℑ-it* *cfg2 inp2*)

value *map-option* (*map trees*) (*build-trees* *cfg2 inp2* (*ℑ-it* *cfg2 inp2*))

10 Conclusion

10.1 Summary

10.2 Future Work

11 Templates

11.1 Section

Citation test [latex].

11.1.1 Subsection

See [Table 11.1](#), [Figure 11.1](#), [Figure 11.2](#), [Figure 11.3](#).

Table 11.1: An example for a simple table.

A	B	C	D
1	2	1	2
2	3	2	3

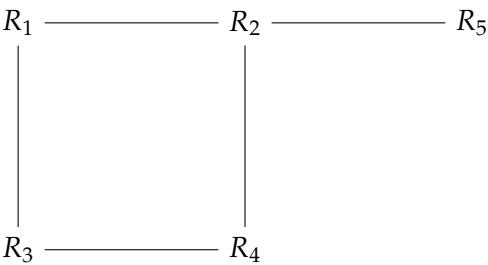


Figure 11.1: An example for a simple drawing.

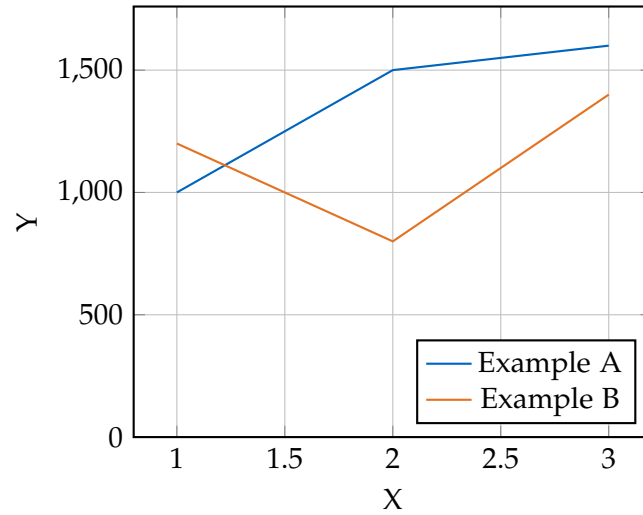


Figure 11.2: An example for a simple plot.

```
SELECT * FROM tbl WHERE tbl.str = "str"
```

Figure 11.3: An example for a source code listing.

List of Figures

4.1 Earley Inference Rules	7
11.1 Example drawing	41
11.2 Example plot	42
11.3 Example listing	42

List of Tables

4.1	Earley Items	8
11.1	Example table	41