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

TODO: Acknowledgments

Abstract

TODO: Abstract

Contents

1 QUESTIONS

- How much explain the proofs?
- How reference thm names?
- How to get rid of where?
- How to tum blau assumes shows fun ... keywords?

2 Snippets

2.1 Earley

2.2 Jones

2.3 Scott

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

3.5 Isabelle/HOL

take closest pair paper section and add additional notation as needed, also discuss the different representations of lemmas and definitions (separate or within the text)

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 -> x | S + S for input x + x + x
- Illustrate the complete bins generated by the example
- Illustrate Initial S -> .alpha,0,0, Scan A -> alpha.abeta,i,j | A -> alpha.beta,i,j+1, Predict A -> alpha.Bbeta,i,j and B -> gamma | B -> .gamma,j,j, Complete A -> alpha.Bbeta,i,j and B -> gamma.,j,k | A -> alphaB.beta,i,k
- Define goal: A -> alpha.beta,i,j iff A =>* s[i..j)beta which implies S -> alpha.,0,n+1 iff S =>* s

TODO: Add nicer syntax for derives

4.2 Background Theory

use snippets

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

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

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

datatype *'a cfg* =
 CFG
  (𝔑 : *'a list*)
  (𝔗 : *'a list*)
  (ℜ : *'a rules*)
  (𝔖 : *'a*)

definition *disjunct-symbols* :: *'a cfg* ⇒ *bool* **where**
 disjunct-symbols cfg ⟷ *set* (𝔑 *cfg*) ∩ *set* (𝔗 *cfg*) = {}

definition *valid-startsymbol* :: *'a cfg* ⇒ *bool* **where**
 valid-startsymbol cfg ⟷ 𝔖 *cfg* ∈ *set* (𝔑 *cfg*)

definition *valid-rules* :: *'a cfg* ⇒ *bool* **where**
 valid-rules cfg ⟷ (∀ (*N*, *α*) ∈ *set* (ℜ *cfg*). *N* ∈ *set* (𝔑 *cfg*) ∧ (∀ *s* ∈ *set α*. *s* ∈ *set* (𝔑 *cfg*) ∪ *set* (𝔗 *cfg*)))

definition *distinct-rules* :: *'a cfg* ⇒ *bool* **where**
 distinct-rules cfg = *distinct* (ℜ *cfg*)

definition *wf-cfg* :: *'a cfg* ⇒ *bool* **where**
 wf-cfg cfg ⟷ *disjunct-symbols cfg* ∧ *valid-startsymbol cfg* ∧ *valid-rules cfg* ∧ *distinct-rules cfg*

definition *is-terminal* :: *'a cfg* ⇒ *'a* ⇒ *bool* **where**
 is-terminal cfg s = (*s* ∈ *set* (𝔗 *cfg*))

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

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

definition *wf-sentence* :: *'a cfg* ⇒ *'a sentence* ⇒ *bool* **where**
 wf-sentence cfg s = (∀ *x* ∈ *set s*. *is-symbol cfg x*)

definition *is-word* :: *'a cfg* ⇒ *'a sentence* ⇒ *bool* **where**
 is-word cfg s = (∀ *x* ∈ *set s*. *is-terminal cfg x*)

definition *derives1* :: *'a cfg* ⇒ *'a sentence* ⇒ *'a sentence* ⇒ *bool* **where**
 derives1 cfg u v =
  (∃ *x y N α*.
   u = *x* @ [*N*] @ *y*
  ∧ *v* = *x* @ *α* @ *y*
  ∧ (*N*, *α*) ∈ *set* (ℜ *cfg*))

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

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

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

4.3 Earley Recognizer

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

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

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

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

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

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

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

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

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

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

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

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

definition *is-complete* :: *'a item ⇒ bool* **where**
is-complete x = (item-dot x ≥ length (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-dot x)))

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

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

definition *wf-item* :: *'a cfg ⇒ 'a sentence => 'a item ⇒ bool* **where**
wf-item cfg inp x = (
item-rule x ∈ set (ℜ cfg) ∧
item-dot x ≤ length (item-rule-body x) ∧
item-origin x ≤ item-end x ∧
item-end x ≤ length inp)

definition *wf-items* :: *'a cfg ⇒ 'a sentence ⇒ 'a items ⇒ bool* **where**
wf-items cfg inp I = (∀ x ∈ I. wf-item cfg inp x)

definition *is-finished* :: *'a cfg ⇒ 'a sentence ⇒ 'a item ⇒ bool* **where**
is-finished cfg inp x ⟷ (
item-rule-head x = 𝔖 cfg ∧
item-origin x = 0 ∧
item-end x = length inp ∧
is-complete x)

definition *earley-recognized* :: *'a items ⇒ 'a cfg ⇒ 'a sentence ⇒ bool* **where**
earley-recognized I cfg inp = (∃ x ∈ I. is-finished cfg inp x)

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

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

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

definition *Complete* :: *nat ⇒ 'a items ⇒ 'a items* **where**
Complete k I =
{ inc-item x k | x y.
x ∈ bin I (item-origin y) ∧
y ∈ bin I k ∧

is-complete y ∧
next-symbol x = *Some* (*item-rule-head y*) }

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

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

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

definition *π-step* :: *nat* ⇒ '*a cfg* ⇒ '*a sentence* ⇒ '*a items* ⇒ '*a items* **where**
π-step k cfg inp I = *I* ∪ *Scan k inp I* ∪ *Complete k I* ∪ *Predict k cfg I*

definition *π* :: *nat* ⇒ '*a cfg* ⇒ '*a sentence* ⇒ '*a items* ⇒ '*a items* **where**
π k cfg inp I = *limit* (*π-step k cfg inp*) *I*

fun *I* :: *nat* ⇒ '*a cfg* ⇒ '*a sentence* ⇒ '*a items* **where**
I 0 *cfg inp* = *π* 0 *cfg inp* (*Init cfg*)
| *I* (*Suc n*) *cfg inp* = *π* (*Suc n*) *cfg inp* (*I n cfg inp*)

definition *J* :: '*a cfg* ⇒ '*a sentence* ⇒ '*a items* **where**
J cfg inp = *I* (*length inp*) *cfg inp*

5.3 Wellformedness

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

by definition

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

by definition

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

wf-Scan-Predict-Complete by definition

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

wf-π-step, by induction on n

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

wf-funpower by definition

lemma *wf-π0*:
wf-items cfg inp (*π 0 cfg inp* (*Init cfg*))

wf-Init wf-π by definition

lemma *wf-ℐ*:
wf-items cfg inp (*ℐ n cfg inp*)

wf-π0 wf-π by induction on n

lemma *wf-ℑ*:
wf-items cfg inp (*ℑ cfg inp*)

wf-ℐ by definition

5.4 Soundness

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

definition *sound-items* :: *'a cfg* ⇒ *'a sentence* ⇒ *'a items* ⇒ *bool* **where**
sound-items cfg inp I = (∀ *x* ∈ *I*. *sound-item cfg inp x*)

lemma *sound-Init*:
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-π0*:
sound-items cfg inp (*π 0 cfg inp* (*Init cfg*))
lemma *sound-ℐ*:
sound-items cfg inp (*ℐ k cfg inp*)
lemma *sound-ℑ*:
sound-items cfg inp (*ℑ cfg inp*)
theorem *soundness*:
earley-recognized (*ℑ cfg inp*) *cfg inp* ⟹ *derives cfg* [*𝔖 cfg*] *inp*

5.5 Monotonicity and Absorption

lemma *π-idem*:
π k cfg inp (*π k cfg inp I*) = *π k cfg inp I*
lemma *Scan-bin-absorb*:
Scan k inp (*bin I k*) = *Scan k inp I*
lemma *Predict-bin-absorb*:
Predict k cfg (*bin I k*) = *Predict k cfg I*
lemma *Complete-bin-absorb*:
Complete k (*bin I k*) ⊆ *Complete k I*
lemma *Scan-Predict-Complete-sub-mono*:
assumes *I* ⊆ *J*
shows *Scan k inp I* ⊆ *Scan k inp J Predict k cfg I* ⊆ *Predict k cfg J Complete k I* ⊆ *Complete k J*
lemma *π-step-sub-mono*:
assumes *I* ⊆ *J*
shows *π-step k cfg inp I* ⊆ *π-step k cfg inp J*
lemma *funpower-sub-mono*:
assumes *I* ⊆ *J*
shows *funpower* (*π-step k cfg inp*) *n I* ⊆ *funpower* (*π-step k cfg inp*) *n J*
lemma *π-sub-mono*:
assumes *I* ⊆ *J*

shows $\pi\ k\ cfg\ inp\ I \subseteq \pi\ k\ cfg\ inp\ J$
lemma *Scan-Predict-Complete-π-step-mono*:
$Scan\ k\ inp\ I \cup Predict\ k\ cfg\ I \cup Complete\ k\ I \subseteq \pi\text{-}step\ k\ cfg\ inp\ I$
lemma *π-step-π-mono*:
$\pi\text{-}step\ k\ cfg\ inp\ I \subseteq \pi\ k\ cfg\ inp\ I$
lemma *Scan-Predict-Complete-π-mono*:
$Scan\ k\ inp\ I \cup Predict\ k\ cfg\ I \cup Complete\ k\ I \subseteq \pi\ k\ cfg\ inp\ I$
lemma *π-mono*:
$I \subseteq \pi\ k\ cfg\ inp\ I$
lemma *Scan-bin-empty*:
assumes $i \neq k\ i \neq k{+}1$
shows $bin\ (Scan\ k\ inp\ I)\ i = \{\}$
lemma *Predict-bin-empty*:
assumes $i \neq k$
shows $bin\ (Predict\ k\ cfg\ I)\ i = \{\}$
lemma *Complete-bin-empty*:
assumes $i \neq k$
shows $bin\ (Complete\ k\ I)\ i = \{\}$
lemma *π-step-bin-absorb*:
assumes $i \neq k\ i \neq k + 1$
shows $bin\ (\pi\text{-}step\ k\ cfg\ inp\ I)\ i = bin\ I\ i$
lemma *funpower-bin-absorb*:
assumes $i \neq k\ i \neq k{+}1$
shows $bin\ (funpower\ (\pi\text{-}step\ k\ cfg\ inp)\ n\ I)\ i = bin\ I\ i$
lemma *π-bin-absorb*:
assumes $i \neq k\ i \neq k{+}1$
shows $bin\ (\pi\ k\ cfg\ inp\ I)\ i = bin\ I\ i$

5.6 Completeness

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

definition *Derives1* :: *'a cfg ⇒ 'a sentence ⇒ nat ⇒ 'a rule ⇒ 'a sentence ⇒ bool* **where**
Derives1 cfg u i r v =
(∃ x y N α.
u = x @ [N] @ y
∧ v = x @ α @ y
∧ (N, α) ∈ set (ℜ cfg)
∧ r = (N, α) ∧ i = length x)

fun *Derivation* :: *'a cfg ⇒ 'a sentence ⇒ 'a derivation ⇒ 'a sentence ⇒ bool* **where**
Derivation - a [] b = (a = b)
| Derivation cfg a (d#D) b = (∃ x. Derives1 cfg a (fst d) (snd d) x ∧ Derivation cfg x D b)

definition *partially-completed* :: *nat ⇒ 'a cfg ⇒ 'a sentence ⇒ 'a items ⇒ ('a derivation ⇒ bool) ⇒ bool* **where**
partially-completed k cfg inp I P = (
∀ i j x a D.
i ≤ j ∧ j ≤ k ∧ k ≤ length inp ∧
x ∈ bin I i ∧ next-symbol x = Some a ∧
Derivation cfg [a] D (slice i j inp) ∧ P D ⟶
inc-item x j ∈ I
)

lemma *fully-completed*:
assumes *j ≤ k k ≤ length inp*
assumes *x = Item (N,α) d i j x ∈ I wf-items cfg inp I*
assumes *Derivation cfg (item-β x) D (slice j k inp)*
assumes *partially-completed k cfg inp I (λD'. length D' ≤ length D)*
shows *Item (N,α) (length α) i k ∈ I*
lemma *partially-completed-ℐ*:
assumes *wf-cfg cfg*
shows *partially-completed k cfg inp (ℐ k cfg inp) (λ-. True)*
lemma *partially-completed-ℑ*:
assumes *wf-cfg cfg*
shows *partially-completed (length inp) cfg inp (ℑ cfg inp) (λ-. True)*
theorem *completeness*:
assumes *derives cfg [𝔖 cfg] inp is-word cfg inp wf-cfg cfg*
shows *earley-recognized (ℑ cfg inp) cfg inp*
corollary
assumes *wf-cfg cfg is-word cfg inp*
shows *earley-recognized (ℑ cfg inp) cfg inp ⟷ derives cfg [𝔖 cfg] inp*

5.7 Finiteness

lemma *finiteness-UNIV-wf-item*:
finite { *x* | *x*. *wf-item cfg inp x* }
theorem *finiteness*:
finite (ℑ *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?!
- discus bin(s)_upd(s) functions. Why bin_upds like this -> easier than fold for proofs!
- discuss pi_it and why it is a partial function -> only terminates for valid input and foreshadow how this is done in isabelle
- introduce remaining definitions (analog to sets)
- discuss wf proofs quickly and go into detail about isabelle specifics about termination and the custom induction scheme using finiteness
- outline the approach to proof correctness aka subsumption in both directions
- discuss list to set proofs
- discuss soundness proofs (maybe omit since obvious)

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

- highlight main theorems

7 Earley Recognizer Implementation

7.1 Definitions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

fun *bin-upd* :: *'a entry ⇒ 'a bin ⇒ 'a bin* **where**

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

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

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

partial-function (tailrec) π-it' :: nat ⇒ 'a cfg ⇒ 'a sentence ⇒ 'a bins ⇒ nat ⇒ 'a bins where
  π-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 π-it' k cfg inp bs' (i+1))

definition π-it :: nat ⇒ 'a cfg ⇒ 'a sentence ⇒ 'a bins ⇒ 'a bins where
  π-it k cfg inp bs = π-it' k cfg inp bs 0

fun ℐ-it :: nat ⇒ 'a cfg ⇒ 'a sentence ⇒ 'a bins where
  ℐ-it 0 cfg inp = π-it 0 cfg inp (Init-it cfg inp)
| ℐ-it (Suc n) cfg inp = π-it (Suc n) cfg inp (ℐ-it n cfg inp)

definition ℑ-it :: 'a cfg ⇒ 'a sentence ⇒ 'a bins where
  ℑ-it cfg inp = ℐ-it (length inp) cfg inp
```

7.2 Wellformedness

lemma *distinct-bin-upd*:
distinct (*items b*) ⟹ *distinct* (*items* (*bin-upd e b*))
lemma *distinct-bin-upds*:
distinct (*items b*) ⟹ *distinct* (*items* (*bin-upds es b*))
lemma *distinct-bins-upd*:
distinct (*items* (*bs* ! *k*)) ⟹ *distinct* (*items* (*bins-upd bs k ips* ! *k*))
lemma *distinct-Scan-it*:
distinct (*items* (*Scan-it k inp a x pre*))
sorry

lemma *distinct-Predict-it*:
wf-cfg cfg ⟹ *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 ∀ *x* ∈ *set* (*items es*). *wf-item cfg inp x* ∧ *item-end x* = *k*
shows *wf-bin cfg inp k* (*bin-upds es b*)
lemma *wf-bins-bins-upd*:
assumes *wf-bins cfg inp bs distinct* (*items es*)
assumes ∀ *x* ∈ *set* (*items es*). *wf-item cfg inp x* ∧ *item-end x* = *k*
shows *wf-bins cfg inp* (*bins-upd bs k es*)
lemma *wf-bins-Init-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 ∀ *y* ∈ *set* (*items* (*Scan-it k inp a x pre*)). *wf-item cfg inp y* ∧ *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 ∀ *y* ∈ *set* (*items* (*Predict-it k cfg X*)). *wf-item cfg inp y* ∧ *item-end y* = *k*
lemma *wf-bins-Complete-it*:
assumes *wf-bins cfg inp bs k* < *length bs y* ∈ *set* (*items* (*bs* ! *k*))
shows ∀ *x* ∈ *set* (*items* (*Complete-it k y bs red*)). *wf-item cfg inp x* ∧ *item-end x* = *k*

definition *wellformed-bins* :: (*nat* × *'a cfg* × *'a sentence* × *'a bins*) *set* **where**
wellformed-bins = {

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

typedef *'a wf-bins = wellformed-bins*::(*nat* × *'a cfg* × *'a sentence* × *'a bins*) *set*

lemma *wellformed-bins-Init-it*:
assumes $k \leq length\ inp\ wf\text{-}cfg\ cfg$
shows $(k, cfg, inp, Init\text{-}it\ cfg\ inp) \in wellformed\text{-}bins$
lemma *wellformed-bins-Complete-it*:
assumes $(k, cfg, inp, bs) \in wellformed\text{-}bins \neg length\ (items\ (bs\ !\ k)) \leq i$
assumes $x = items\ (bs\ !\ k)\ !\ i\ next\text{-}symbol\ x = None$
shows $(k, cfg, inp, bins\text{-}upd\ bs\ k\ (Complete\text{-}it\ k\ x\ bs\ red)) \in wellformed\text{-}bins$
lemma *wellformed-bins-Scan-it*:
assumes $(k, cfg, inp, bs) \in wellformed\text{-}bins \neg length\ (items\ (bs\ !\ k)) \leq i$
assumes $x = items\ (bs\ !\ k)\ !\ i\ next\text{-}symbol\ x = Some\ a$
assumes $is\text{-}terminal\ cfg\ a\ k < length\ inp$
shows $(k, cfg, inp, bins\text{-}upd\ bs\ (k{+}1)\ (Scan\text{-}it\ k\ inp\ a\ x\ pre)) \in wellformed\text{-}bins$
lemma *wellformed-bins-Predict-it*:
assumes $(k, cfg, inp, bs) \in wellformed\text{-}bins \neg length\ (items\ (bs\ !\ k)) \leq i$
assumes $x = items\ (bs\ !\ k)\ !\ i\ next\text{-}symbol\ x = Some\ a \neg is\text{-}terminal\ cfg\ a$
shows $(k, cfg, inp, bins\text{-}upd\ bs\ k\ (Predict\text{-}it\ k\ cfg\ a)) \in wellformed\text{-}bins$
fun *earley-measure* :: *nat* × *'a cfg* × *'a sentence* × *'a bins* ⇒ *nat* ⇒ *nat* **where**
$earley\text{-}measure\ (k, cfg, inp, bs)\ i = card\ \{\ x \mid x.\ wf\text{-}item\ cfg\ inp\ x \wedge item\text{-}end\ x = k\ \} - i$

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

lemma *wellformed-bins-π-it'*:
assumes $(k, cfg, inp, bs) \in$ *wellformed-bins*
shows $(k, cfg, inp, \pi\text{-}it'\ k\ cfg\ inp\ bs\ i) \in$ *wellformed-bins*
lemma *wellformed-bins-π-it*:
assumes $(k, cfg, inp, bs) \in$ *wellformed-bins*
shows $(k, cfg, inp, \pi\text{-}it\ k\ cfg\ inp\ bs) \in$ *wellformed-bins*
lemma *wellformed-bins-ℐ-it*:
assumes $k \leq$ *length inp wf-cfg cfg*
shows $(k, cfg, inp, \mathcal{I}\text{-}it\ k\ cfg\ inp) \in$ *wellformed-bins*
lemma *wellformed-bins-ℑ-it*:
$k \leq$ *length inp* $\Longrightarrow$ *wf-cfg cfg* $\Longrightarrow (k, cfg, inp, \mathfrak{I}\text{-}it\ cfg\ inp) \in$ *wellformed-bins*
lemma *wf-bins-π-it'*:
assumes $(k, cfg, inp, bs) \in$ *wellformed-bins*
shows *wf-bins cfg inp* $(\pi\text{-}it'\ k\ cfg\ inp\ bs\ i)$
lemma *wf-bins-π-it*:
assumes $(k, cfg, inp, bs) \in$ *wellformed-bins*
shows *wf-bins cfg inp* $(\pi\text{-}it\ k\ cfg\ inp\ bs)$
lemma *wf-bins-ℐ-it*:
assumes $k \leq$ *length inp wf-cfg cfg*
shows *wf-bins cfg inp* $(\mathcal{I}\text{-}it\ k\ cfg\ inp)$
lemma *wf-bins-ℑ-it*:
wf-cfg cfg $\Longrightarrow$ *wf-bins cfg inp* $(\mathfrak{I}\text{-}it\ cfg\ inp)$

7.3 List to set

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

shows *bins-items* (π-*it'* *k cfg inp bs i*) $\subseteq$ π *k cfg inp I*
lemma π-*it-sub-*π:
assumes (*k*, *cfg*, *inp*, *bs*) $\in$ *wellformed-bins*
assumes *bins-items bs* $\subseteq$ *I*
shows *bins-items* (π-*it k cfg inp bs*) $\subseteq$ π *k cfg inp I*
lemma $\mathcal{I}$-*it-sub-*$\mathcal{I}$:
assumes *k* $\leq$ *length inp wf-cfg cfg*
shows *bins-items* ($\mathcal{I}$-*it k cfg inp*) $\subseteq$ $\mathcal{I}$ *k cfg inp*
lemma $\mathfrak{I}$-*it-sub-*$\mathfrak{I}$:
wf-cfg cfg $\Longrightarrow$ *bins-items* ($\mathfrak{I}$-*it cfg inp*) $\subseteq$ $\mathfrak{I}$ *cfg inp*

7.4 Soundness

lemma *sound-Scan-it*:
assumes *wf-bins cfg inp bs bins-items bs* $\subseteq$ *I x* $\in$ *set* (*items* (*bs* ! *k*)) *k* $<$ *length bs k* $<$ *length inp*
assumes *next-symbol x* $=$ *Some a wf-items cfg inp I sound-items cfg inp I*
shows *sound-items cfg inp* (*set* (*items* (*Scan-it k inp a x i*)))
lemma *sound-Predict-it*:
assumes *wf-bins cfg inp bs bins-items bs* $\subseteq$ *I x* $\in$ *set* (*items* (*bs* ! *k*)) *k* $<$ *length bs*
assumes *next-symbol x* $=$ *Some X sound-items cfg inp I*
shows *sound-items cfg inp* (*set* (*items* (*Predict-it k cfg X*)))
lemma *sound-Complete-it*:
assumes *wf-bins cfg inp bs bins-items bs* $\subseteq$ *I y* $\in$ *set* (*items* (*bs* ! *k*)) *k* $<$ *length bs*
assumes *next-symbol y* $=$ *None wf-items cfg inp I sound-items cfg inp I*
shows *sound-items cfg inp* (*set* (*items* (*Complete-it k y bs i*)))
lemma *sound-*π-*it'*:
assumes (*k*, *cfg*, *inp*, *bs*) $\in$ *wellformed-bins*
assumes *sound-items cfg inp* (*bins-items bs*)
shows *sound-items cfg inp* (*bins-items* (π-*it'* *k cfg inp bs i*))
lemma *sound-*π-*it*:
assumes (*k*, *cfg*, *inp*, *bs*) $\in$ *wellformed-bins*
assumes *sound-items cfg inp* (*bins-items bs*)
shows *sound-items cfg inp* (*bins-items* (π-*it k cfg 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* ⊆ *bins-items bs I* ⊆ *bins-items bs is-complete z wf-bins cfg inp bs wf-item cfg inp z*
shows *Complete k* $(I \cup \{z\})$ ⊆ *bins-items bs* ∪ *set* (*items* (*Complete-it k z bs red*))
lemma *π-it′-mono*:
assumes (*k*, *cfg*, *inp*, *bs*) ∈ *wellformed-bins*
shows *bins-items bs* ⊆ *bins-items* (*π-it′ k cfg inp bs i*)
lemma *π-step-sub-π-it′*:
assumes (*k*, *cfg*, *inp*, *bs*) ∈ *wellformed-bins*
assumes *π-step k cfg inp* (*bins-items-upto bs k i*) ⊆ *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*) ⊆ *bins-items* (*π-it′ k cfg inp bs i*)
lemma *π-step-sub-π-it*:
assumes (*k*, *cfg*, *inp*, *bs*) ∈ *wellformed-bins*
assumes *π-step k cfg inp* (*bins-items-upto bs k 0*) ⊆ *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*) ⊆ *bins-items* (*π-it k cfg inp bs*)
lemma *π-it′-bins-items-eq*:
assumes (*k*, *cfg*, *inp*, *as*) ∈ *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*) ∈ *wellformed-bins*
assumes *i* ≤ *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*) ∈ *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*) ∈ *wellformed-bins*
assumes *π-step k cfg inp* (*bins-items-upto bs k 0*) ⊆ *bins-items bs sound-items cfg inp* (*bins-items bs*)
assumes *is-word cfg inp nonempty-derives cfg*
shows *funpower* (*π-step k cfg inp*) *n* (*bins-items bs*) ⊆ *bins-items* (*π-it k cfg inp bs*)
lemma *π-sub-π-it*:
assumes (*k*, *cfg*, *inp*, *bs*) ∈ *wellformed-bins*
assumes *π-step k cfg inp* (*bins-items-upto bs k 0*) ⊆ *bins-items bs sound-items cfg inp* (*bins-items bs*)
assumes *is-word cfg inp nonempty-derives cfg*
shows *π k cfg inp* (*bins-items bs*) ⊆ *bins-items* (*π-it k cfg inp bs*)

lemma *$\mathcal{I}$-sub-$\mathcal{I}$-it*:
assumes *k* $\leq$ *length inp wf-cfg cfg*
assumes *is-word cfg inp nonempty-derives cfg*
shows *$\mathcal{I}$ k cfg inp* $\subseteq$ *bins-items* (*$\mathcal{I}$-it k cfg inp*)
lemma *$\mathfrak{I}$-sub-$\mathfrak{I}$-it*:
assumes *wf-cfg cfg is-word cfg inp nonempty-derives cfg*
shows *$\mathfrak{I}$ cfg inp* $\subseteq$ *bins-items* (*$\mathfrak{I}$-it cfg inp*)

7.6 Main Theorem

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

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

8 Earley Parser Implementation

8.1 Draft

8.2 Pointer lemmas

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

definition *scans* :: *'a sentence* ⇒ *nat* ⇒ *'a item* ⇒ *'a item* ⇒ *bool* **where**
scans inp k x y ⟷ *y = inc-item x k* ∧ (∃*a*. *next-symbol x = Some a* ∧ *inp*!(*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 sentence* ⇒ *'a bins* ⇒ *nat* ⇒ *'a entry* ⇒ *bool* **where**
sound-pre-ptr inp bs k e = (∀ *pre*. *pointer e = Pre pre* ⟶
k > 0 ∧ *pre* < *length* (*bs*!(*k*−1)) ∧ *scans inp 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* < *length* (*bs*!*k'*) ∧ *red* < *length* (*bs*!*k*) ∧ *completes k* (*item* (*bs*!*k'*!*pre*)) (*item e*) (*item* (*bs*!*k*!*red*)))

definition *sound-ptrs* :: *'a sentence* ⇒ *'a bins* ⇒ *bool* **where**
sound-ptrs inp bs = (∀ *k* < *length bs*. ∀ *e* ∈ *set* (*bs*!*k*).
sound-null-ptr e ∧
sound-pre-ptr inp bs k e ∧
sound-prered-ptr bs k e)

definition *mono-red-ptr* :: *'a bins* ⇒ *bool* **where**
mono-red-ptr bs = (∀ *k* < *length bs*. ∀ *i* < *length* (*bs*!*k*).
∀ *k' pre red ps*. *pointer* (*bs*!*k*!*i*) = *PreRed* (*k'*, *pre*, *red*) *ps* ⟶ *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 ∀ *k*′ *pre red ps. pointer e* = *PreRed* (*k*′, *pre*, *red*) *ps* ⟶ *red* < *length es*
shows *mono-red-ptr* (*bs*[*k* := *bin-upd e es*])
lemma *sound-mono-ptrs-bin-upds*:
assumes *sound-ptrs inp bs mono-red-ptr bs k* < *length bs b* = *bs*!*k distinct* (*items b*) *distinct* (*items es*)
assumes ∀ *e* ∈ *set es. sound-null-ptr e* ∧ *sound-pre-ptr inp bs k e* ∧ *sound-prered-ptr bs k e*
assumes ∀ *e* ∈ *set es.* ∀ *k*′ *pre red ps. pointer e* = *PreRed* (*k*′, *pre*, *red*) *ps* ⟶ *red* < *length b*
shows *sound-ptrs inp* (*bs*[*k* := *bin-upds es b*]) ∧ *mono-red-ptr* (*bs*[*k* := *bin-upds es b*])
lemma *sound-mono-ptrs-π-it*′:
assumes (*k*, *cfg*, *inp*, *bs*) ∈ *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*) ∧ *mono-red-ptr* (*π-it*′ *k cfg inp bs i*)
lemma *sound-mono-ptrs-π-it*:
assumes (*k*, *cfg*, *inp*, *bs*) ∈ *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*) ∧ *mono-red-ptr* (*π-it k cfg inp bs*)
lemma *sound-ptrs-Init-it*:
sound-ptrs inp (*Init-it cfg inp*)
lemma *mono-red-ptr-Init-it*:
mono-red-ptr (*Init-it cfg inp*)
lemma *sound-mono-ptrs-ℐ-it*:
assumes *k* ≤ *length inp wf-cfg cfg nonempty-derives cfg wf-cfg cfg*
shows *sound-ptrs inp* (*ℐ-it k cfg inp*) ∧ *mono-red-ptr* (*ℐ-it k cfg inp*)
lemma *sound-mono-ptrs-ℑ-it*:
assumes *wf-cfg cfg nonempty-derives cfg*
shows *sound-ptrs inp* (*ℑ-it cfg inp*) ∧ *mono-red-ptr* (*ℑ-it cfg inp*)

8.3 Trees and Forests

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

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

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

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

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

definition wf-yield-tree :: 'a sentence ⇒ 'a item ⇒ 'a tree ⇒ bool where
  wf-yield-tree inp x t ⟷ yield-tree t = slice (item-origin x) (item-end x) inp

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

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

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

8.4 A single parse tree

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

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

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


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

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

lemma wellformed-tree-ptrs-pre:
```

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

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

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

```
      ) gs))
    )
  ))

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

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

fun build-forest'-measure :: ('a bins × 'a sentence × nat × nat × nat set) ⇒ 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) ∈ wellformed-forest-ptrs
  assumes e = bs!k!i pointer e = Pre pre
  shows (bs, inp, (k−1), pre, {pre}) ∈ wellformed-forest-ptrs
lemma wellformed-forest-ptrs-prered-pre:
  assumes (bs, inp, k, i, I) ∈ wellformed-forest-ptrs
  assumes e = bs!k!i pointer e = PreRed p ps
  assumes ps' = filter (λ(k', pre, red). red ∉ I) (p#ps)
  assumes gs = group-by (λ(k', pre, red). (k', pre)) (λ(k', pre, red). red) ps'
  assumes ((k', pre), reds) ∈ set gs
  shows (bs, inp, k', pre, {pre}) ∈ wellformed-forest-ptrs
lemma wellformed-forest-ptrs-prered-red:
  assumes (bs, inp, k, i, I) ∈ wellformed-forest-ptrs
  assumes e = bs!k!i pointer e = PreRed p ps
  assumes ps' = filter (λ(k', pre, red). red ∉ I) (p#ps)
  assumes gs = group-by (λ(k', pre, red). (k', pre)) (λ(k', pre, red). red) ps'
  assumes ((k', pre), reds) ∈ set gs red ∈ set reds
  shows (bs, inp, k, red, I ∪ {red}) ∈ wellformed-forest-ptrs
lemma build-trees'-induct:
  assumes (bs, inp, k, i, I) ∈ wellformed-forest-ptrs
```

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

lemma *build-trees′-termination*:
assumes $(bs, inp, k, i, I) \in wellformed\text{-}forest\text{-}ptrs$
shows $\exists fs.\ build\text{-}trees'\ bs\ inp\ k\ i\ I = Some\ fs \wedge (\forall f \in set\ fs.\ \exists N\ fss.\ f = FBranch\ N\ fss)$

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

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

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

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

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

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

assumes *wf-cfg cfg nonempty-derives cfg derives cfg* [$\mathfrak{S}$ *cfg*] *inp*
shows $\exists$*fs*. *build-trees cfg inp* ($\mathfrak{I}$*-it cfg inp*) = *Some fs*

8.6 A word on completeness

9 Examples

9.1 epsilon free CFG

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

lemma *ε-free-impl-non-empty-deriv*:
ε-free cfg ⟹ *N* ∈ *set* (𝔑 *cfg*) ⟹ ¬ *derives cfg* [*N*] []

9.2 Example 1: Addition

datatype *t1* = *x* | *plus*
datatype *n1* = *S*
datatype *s1* = *Terminal t1* | *Nonterminal n1*

definition *nonterminals1* :: *s1 list* **where**
nonterminals1 = [*Nonterminal S*]

definition *terminals1* :: *s1 list* **where**
terminals1 = [*Terminal x*, *Terminal plus*]

definition *rules1* :: *s1 rule list* **where**
rules1 = [
(*Nonterminal S*, [*Terminal x*]),
(*Nonterminal S*, [*Nonterminal S*, *Terminal plus*, *Nonterminal S*])
]

definition *start-symbol1* :: *s1* **where**
start-symbol1 = *Nonterminal S*

definition *cfg1* :: *s1 cfg* **where**
cfg1 = *CFG nonterminals1 terminals1 rules1 start-symbol1*

definition *inp1* :: *s1 list* **where**
inp1 = [*Terminal x*, *Terminal plus*, *Terminal x*, *Terminal plus*, *Terminal x*]

lemma *wf-cfg1*:

wf-cfg cfg1
lemma *is-word-inp1*:
is-word cfg1 inp1
lemma *nonempty-derives1*:
nonempty-derives cfg1
lemma *correctness1*:
earley-recognized-it (*𝔍-it cfg1 inp1*) *cfg1 inp1* ⟷ *derives cfg1* [*𝔖 cfg1*] *inp1*
fun *size-bins* :: *'a bins* ⇒ *nat* **where**
size-bins bs = *fold* (+) (*map length bs*) *0*

value *𝔍-it cfg1 inp1*
value *size-bins* (*𝔍-it cfg1 inp1*)
value *earley-recognized-it* (*𝔍-it cfg1 inp1*) *cfg1 inp1*
value *build-trees cfg1 inp1* (*𝔍-it cfg1 inp1*)
value *map-option* (*map trees*) (*build-trees cfg1 inp1* (*𝔍-it cfg1 inp1*))
value *map-option* (*foldl* (+) *0* ∘ *map length*) (*map-option* (*map trees*) (*build-trees cfg1 inp1* (*𝔍-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*:
wf-cfg cfg2
lemma *is-word-inp2*:
is-word cfg2 inp2
lemma *nonempty-derives2*:
nonempty-derives cfg2
lemma *correctness2*:
earley-recognized-it (*𝔍-it cfg2 inp2*) *cfg2 inp2* ⟷ *derives cfg2* [*𝔖 cfg2*] *inp2*
value *𝔍-it cfg2 inp2*
value *earley-recognized-it* (*𝔍-it cfg2 inp2*) *cfg2 inp2*
value *build-trees cfg2 inp2* (*𝔍-it cfg2 inp2*)
value *map-option* (*map trees*) (*build-trees cfg2 inp2* (*𝔍-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

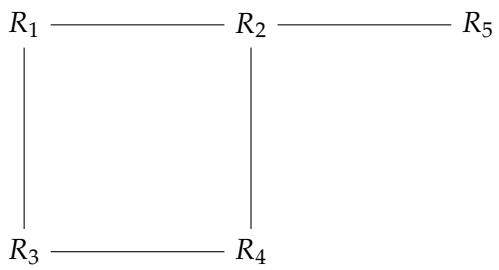

Figure 11.1: An example for a simple drawing.

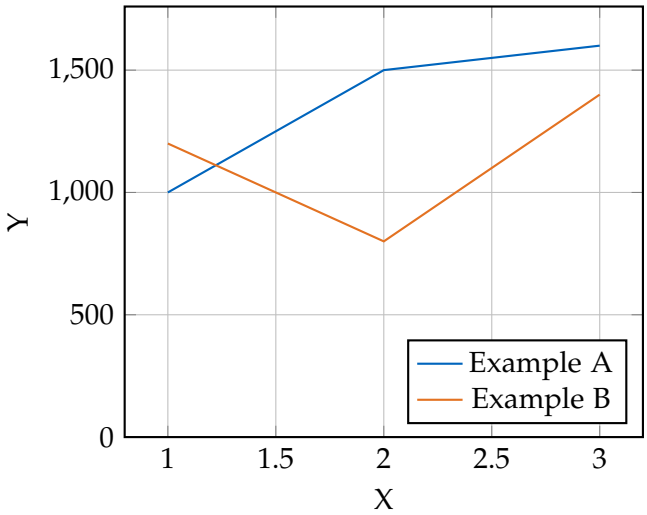

Figure 11.2: An example for a simple plot.

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

Figure 11.3: An example for a source code listing.

List of Figures

List of Tables