DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Formal Verification of an Earley Parser

Martin Rau

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Formal Verification of an Earley Parser

Formale Verifikation eines Earley Parsers

Author:	Martin Rau
Supervisor:	Tobias Nipkow
Advisor:	Tobias Nipkow
Submission Date:	15.06.2023

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.06.2023 Martin Rau

Acknowledgments

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

Abstract

TODO: Abstract

Contents

1 QUESTIONS

- How much explain the proofs?
- How reference thm names?

2 Snippets

2.1 Earley

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

A language is a set of strings over a finite set of symbols. We call these terminal symbols and represent them by lowercase letters: a, b, c. We use a context-free grammar as a formal device for specifying which strings are in the set. This grammar uses another set of symbols, the nonterminals, which we can think of as syntactic classes. We use capitals for nonterminals: A, B, C. String of either terminals or non-terminals are represented by greek letters: alpha, beta, gamma. The empty string is epsilon. There is a finite set of productions or rewriting rules of the form A -> alpha. The nonterminal which stands for sentence is called the root R of the grammar. The productions with a particular nonterminal A on their left sides are called the alternatives of A. We write alpha => beta if exists gamma, delta, ny, A such taht a = gamma A delta and beta = gamma ny delta and A -> ny is a production. We write alpha =>* beta if exists alpha0, alpha1, ... alpham (m > =0) such that alpha = alpha0 => alpha1 => ... => alpham = beta The sequence alphai is called a derivation of beta from alpha. A sentential form is a string alpha such the the root R =>* alpha. A sentence is a sentential form consisting entirely of terminal symbols. The language defined by a grammar L(G) is the set of its sentences. We may represent any sentential form in at least one way as a derivation tree or parse tree reflecting the steps made in deriving it. The degree of ambiguity of a sentence is the number of its distinct derivation trees. A sentence is unambiguous if it has degree 1 of ambiguity. A grammar is unambiguous if each of its sentences is unambiguous. A grammar is reduced if every nonterminal appears in some derivation of some sentence. A recognizer is an algorithm which takes a input a string and either accepts or rejects it depending on whether or not the string is a sentence of the grammer. A parser is a recogizer which also outputs the set of all legal derivation trees for the string.

The algorithm scans an input string X1, ..., Xn from left to right. As eachsymbol Xi is scanned, a set of states Si is constructed which represents the condition of the recognition process at that point in the scan. Each state in the set represents (1) a production such that we are currently scanning a portion of the input string which is derived from its right side, (2) a point in that production which shows how much of the production's right side we have recognized so far, (3) a pointer back to the position in the input string at which we began to look for that instance of the production. In general, we operate on a state set Si as follows: we process the states in the set in order, performing one of three operatins on each one depending on the form of the state. These operations may add more states to Si and may also put states in a new state set Si+1. We describe the operations by example: ... The predictor operation is applicable to a state when there is a nonterminal to the right of the dot. It causes us to add one new state to Si for each alternative of that nonterminal. We put the dot at the beginning of the production in each new state, since we have not scanned any of its symbols yet. The pointer is set to i, since the state was created in Si. Thus the predictor adds to Si all the productions which might generate substrings beginning at Xi+1. The scanner is applicable in case there is a terminal to the right of the dot. The scanner compares that symbol with Xi+1 and if they match, it adds the state to Si+1 with the dot moved over one in the state to indicate that that terminal symbol has been scanned. If we finish processing Si and Si+1 remains empty an error has occurred in the input string. Otherwise, we start to process Si+1. The completer is applicable to a state if its dot is at the end of its production. It goes back to the state set indicated by its pointer and adds all states from this state set which have the dot in front of its nonterminal. It then moves over the dot. Intuitively, the origin state set is the state set we were in when we went looking for that nonterminal. We have now found it, so we go back to all the states which caused us to look for it, and move the dot over in these states to show that it has been successfully scanned. If the algorithm ever produces an Si+1 consisting of the single state S -> alpha dot, 0, n, then the sentence is part of the grammar. Note that the algorithm is in effect a top-down parser in which we carry along all possible parses simultaneously in such a way that we can often combine like subparses.

2.2 Scott

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

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

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

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

2.3 Aycock

Earley's parsing algorithm is a general algorithm, capable of parsing according to any context-free grammar. General parsing algorithms like Earley parsing allow unfettered expression of ambiguous grammar contructs which come up often in practice (REFERENCE).

Earley parsers operate by constructing a sequence of sets, sometime called Earley sets. Given an input $x_1 x_2 \ldots x_n$ the parser builds $n + 1$ sets: an initial set S_0 and one set S_i for each input symbol x_i. Elements of these sets are referred to as Earley items, which consist of three parts: a grammar rule, a position in the right-hand side of the rule indicating how much of that rule has been seen and a pointer to an earlier Earley set. Typically Earley items are written as ... where the position in the rule's right-hand side is denoted by a dot and j is a pointer to set S_j. An Earley set S_i is computed from an initial set of Earley items in S_i and S_{i+1} is initialized, by applying the followingn three steps to the items in S_i until no more can be added. ... An item is added to a set only if it is not in the set already. The initial set S_0 contains the items ... to begin with. If the final set contains the item ... then the input is accepted.

We have not used a lookahead in this description of Earley parsing since it's primary purpose is to increase the efficieny of the Earley parser on a large class of grammars (REFERENCE).

In terms of implementation, the Earley sets are built in increasing order as the input is read. Also, each set is typically represented as a list of items. This list representation of a set is particularly convenient, because the list of items acts as a work queue when building the sets: items are examined in order, applying the transformations as necessary: items added to the set are appended onto the end of the list.

At any given point i in the parse, we have two partially constructed sets. Scanner may add items to S_{i+1} and S_i may have items added to it by Predictor and Completer. It is this latter possibility, adding items to S_i while representing sets as lists, which causes grief with epsilon-rules. When Completer processes an item A -> dot, j which corresponds to the epsilon-rule A -> epsiolon, it must look through S_j for items with the dot before an A. Unfortunately, for epsilon-rule items, j is always equal to i. Completer is thus looking through the partially constructed set S_i. Since implementations process items in S_i in order, if an item B -> alpha dot A beta, k is added to S_i after Completer has processed A -> dot, j, Completer will never add B -> αA dot β, k to S_i. In turn, items resulting directly and indirectly from B -> αA dot β, k will be omitted too. This effectively prunes protential derivation paths which might cause correct input to be rejected. (EXAMPLE) Aho *et al* **[Aho:1972]** propose the stay clam and keep running the Predictor and Completer in turn until neither has anything more to add. Earley himself suggest to have the Completer note that the dot needed to be moved over A, then looking for this whenever future items were added to S_i. For efficiency's sake the collection of on-terminals to watch for should be stored in a data structure which allows fast access. Neither approach is very satisfactory. A third solution [**Aycoack:2002**] is a simple modification of the Predictor based on the idea of nullability. A non-terminal A is said to be nullable if A derives star epsilon. Terminal symbols of course can never be nullable. The nullability of non-terminals in a grammar may be precomputed using well-known techniques [**Appel:2003**] [**Fischer:2009**] Using this notion the Predictor can be stated as follows: if A -> αdot B β, j is in S_i, add B -> dot γ, i to S_i for all rules B -> γ. If B is nullable, also add A -> αB dot β, j to S_i. Explanation why I decided against it. Involves every grammar can be rewritten to not contain epsilon productions. In other words we eagerly move the dot over a nonterminal if that non-terminal can derive epsilon and effectivley disappear. The source implements this precomputation by constructing a variant of a LR(0) deterministic finite automata (DFA). But for an earley parser we must keep track of which parent pointers and LR(0) items belong together which leads to complex and inelegant implementations [**McLean:1996**]. The source resolves this problem by constructing split epsilon DFAs, but still need to adjust the classical earley algorithm by adding not only predecessor links but also causal links, and to construct the split epsilon DFAs not the original grammar but a slightly adjusted equivalent grammar is used that encodes explicitly information that is crucial to reconstructing derivations, called a grammar in nihilist normal form (NNF) which might increase the size of the grammar whereas the authors note empirical results that the increase is quite modest (a factor of 2 at most).

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

2.4 Related Work

2.4.1 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.

2.4.2 Related Verification Work

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

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

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

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

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

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

2.5 Future Work

Different approaches:

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

Performance improvements:

(1) Look-ahead of k or at least 1 like in the original Earley paper. (2) Optimize the representation of the grammar instead of single list, group by production, ... (3) Keep a set of already inserted items to not double check item insertion. (4) Use a queue instead of a list for bins. (5) Refine the algorithm to an imperative version using a single linked list and actual pointers instead of natural numbers.

Parse tree disambiguation:

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

Grammar rewriting:

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

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

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

Parse tree filtering:

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

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 -> 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

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

datatype $'a\ cfg =$
  CFG
    $(\mathfrak{N} : 'a\ list)$

```
  (𝔗 : 'a list)
  (ℜ : 'a rules)
  (𝔖 : 'a)

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)

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

lemma slice-induct:
 assumes ⋀a b. P a b []
 assumes ⋀a x xs. P a 0 (x#xs)
 assumes ⋀b x xs. P 0 b xs ⟹ P 0 (Suc b) (x#xs)
 assumes ⋀a b x xs. P a b xs ⟹ P (Suc a) (Suc b) (x#xs)
 shows P a b xs


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* ($\mathfrak{R}$ *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* ($\mathfrak{T}$ *cfg*))

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

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

definition *wf-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*)

4.3 Earley Recognizer

$$\textsc{Init}\quad \frac{}{\mathrm{S} \to\ \bullet\alpha,\ 0,\ 0} \qquad \textsc{Scan}\quad \frac{\mathrm{A} \to\ \alpha\bullet \mathrm{a}\ \beta,\ \mathrm{i},\ \mathrm{j}}{\mathrm{A} \to\ \alpha \mathrm{a}\ \bullet\beta,\ \mathrm{i},\ \mathrm{j+1}} \qquad \textsc{Predict}\quad \frac{\mathrm{A} \to\ \alpha\bullet \mathrm{B}\ \beta,\ \mathrm{i},\ \mathrm{j} \qquad \mathrm{B} \to\ \gamma\ \in\ set\ (\mathfrak{R}\ cfg)}{\mathrm{B} \to\ \bullet\gamma,\ \mathrm{j},\ \mathrm{j}}$$

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

Figure 4.1: Earley inference rules

$$A \to\ \alpha \bullet \beta, i, j \ \text{ iff }\ A \overset{*}{\Rightarrow} \mathit{slice}\ i\ j\ \mathit{inp}$$

$$\mathfrak{S}\ \mathit{cfg} \to\ \alpha\bullet, 0, |\mathit{inp}| + 1 \ \text{ iff }\ \mathfrak{S}\ \mathit{cfg} \overset{*}{\Rightarrow} \mathit{inp}$$

$S \to x\ S \to S + S$

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

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

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* ⇒ *'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* ($\mathfrak{R}$ *cfg*) ∧
item-dot x ≤ *length* (*item-rule-body x*) ∧
item-origin x ≤ *item-end x* ∧

$item\text{-}end\ x \leq length\ inp)$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.3 Wellformedness

lemma *wf-Init*:
assumes $x \in Init\ cfg$
shows *wf-item cfg inp x*

by definition

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

by definition

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

wf-Scan-Predict-Complete by definition

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

$wf\text{-}\pi\text{-}step$, by induction on n

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

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-*$\mathfrak{I}$:
shows *wf-items cfg inp* ($\mathfrak{I}$ *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-*π*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-*$\mathfrak{I}$:
shows *sound-items cfg inp* ($\mathfrak{I}$ *cfg inp*)
theorem *soundness*:
shows *earley-recognized* ($\mathfrak{I}$ *cfg inp*) *cfg inp* $\Longrightarrow$ *derives cfg* [$\mathfrak{S}$ *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 $Scan\ k\ inp\ I \cup Predict\ k\ cfg\ I \cup Complete\ k\ I \subseteq \pi\ k\ cfg\ inp\ I$
lemma π*-mono*:
shows $I \subseteq \pi\ k\ cfg\ inp\ I$
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 $Derives1 :: 'a\ cfg \Rightarrow 'a\ sentence \Rightarrow nat \Rightarrow 'a\ rule \Rightarrow 'a\ sentence \Rightarrow bool$ **where**
$Derives1\ cfg\ u\ i\ r\ v =$
$(\exists\ x\ y\ N\ \alpha.$
$u = x\ @\ [N]\ @\ y$
$\wedge\ v = x\ @\ \alpha\ @\ y$
$\wedge\ (N,\ \alpha) \in set\ (\mathfrak{R}\ cfg)$

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

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-$\mathcal{I}$*:
assumes *wf-cfg cfg*
shows *partially-completed k cfg inp ($\mathcal{I}$ k cfg inp) (λ-. True)*
lemma *partially-completed-$\mathfrak{I}$*:
assumes *wf-cfg cfg*
shows *partially-completed (length inp) cfg inp ($\mathfrak{I}$ cfg inp) (λ-. True)*
theorem *completeness*:
assumes *derives cfg [$\mathfrak{S}$ cfg] inp is-word cfg inp wf-cfg cfg*
shows *earley-recognized ($\mathfrak{I}$ cfg inp) cfg inp*
corollary
assumes *wf-cfg cfg is-word cfg inp*
shows *earley-recognized ($\mathfrak{I}$ cfg inp) cfg inp ⟷ derives cfg [$\mathfrak{S}$ cfg] inp*

5.7 Finiteness

lemma *finiteness-UNIV-wf-item*:
shows *finite { x | x. wf-item cfg inp x }*
theorem *finiteness*:
shows *finite ($\mathfrak{I}$ 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'* :: $nat \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow ('a \times nat)\ list$ **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 = $\bigcup$ { *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*) (ℜ *cfg*) *in*
  let b0 = *map* (λ*r*. (*Entry* (*init-item r 0*) *Null*)) *rs in*
  let bs = *replicate* (*length inp* + 1) ([]) *in*
  bs[*0* := *b0*])

definition *Scan-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*) (ℜ *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*:
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*) $\wedge$ *item-end* (*item e*) $=$ *k*
shows *wf-bin cfg inp k* (*bin-upd e b*)
lemma *wf-bin-bin-upds*:
assumes *wf-bin cfg inp k b distinct* (*items es*)
assumes $\forall$ *x* $\in$ *set* (*items es*). *wf-item cfg inp x* $\wedge$ *item-end x* $=$ *k*
shows *wf-bin cfg inp k* (*bin-upds es b*)
lemma *wf-bins-bins-upd*:
assumes *wf-bins cfg inp bs distinct* (*items es*)
assumes $\forall$ *x* $\in$ *set* (*items es*). *wf-item cfg inp x* $\wedge$ *item-end x* $=$ *k*
shows *wf-bins cfg inp* (*bins-upd bs k es*)
lemma *wf-bins-Init-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* $\in$ *set* (*items* (*bs* ! *k*)) *k* $<$ *length inp next-symbol x* $\neq$ *None*
shows $\forall$ *y* $\in$ *set* (*items* (*Scan-it k inp a x pre*)). *wf-item cfg inp y* $\wedge$ *item-end y* $=$ (*k*+1)
lemma *wf-bins-Predict-it*:
assumes *wf-bins cfg inp bs k* $<$ *length bs k* $\leq$ *length inp wf-cfg cfg*
shows $\forall$ *y* $\in$ *set* (*items* (*Predict-it k cfg X*)). *wf-item cfg inp y* $\wedge$ *item-end y* $=$ *k*
lemma *wf-bins-Complete-it*:
assumes *wf-bins cfg inp bs k* $<$ *length bs y* $\in$ *set* (*items* (*bs* ! *k*))

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

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

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

7.3 List to set

lemma *Init-it-eq-Init*:
shows $bins\text{-}items\ (Init\text{-}it\ cfg\ inp) = Init\ cfg$
lemma *Scan-it-sub-Scan*:
assumes $wf\text{-}bins\ cfg\ inp\ bs\ bins\text{-}items\ bs \subseteq I\ x \in set\ (items\ (bs\ !\ k))$
assumes $k < length\ bs\ k < length\ inp$
assumes $next\text{-}symbol\ x = Some\ a$
shows $set\ (items\ (Scan\text{-}it\ k\ inp\ a\ x\ pre)) \subseteq Scan\ k\ inp\ I$
lemma *Predict-it-sub-Predict*:
assumes $wf\text{-}bins\ cfg\ inp\ bs\ bins\text{-}items\ bs \subseteq I\ x \in set\ (items\ (bs\ !\ k))\ k < length\ bs$
assumes $next\text{-}symbol\ x = Some\ X$
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-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-*$\mathfrak{I}$:
assumes *wf-cfg cfg*
shows *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* $\subseteq$ *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 $i \le j$ *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 *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* ≤ *length inp wf-cfg cfg*
assumes *is-word cfg inp nonempty-derives cfg*
shows *$\mathcal{I}$ k cfg inp* ⊆ *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* ⊆ *bins-items* (*$\mathfrak{I}$-it cfg inp*)

7.6 Main Theorem

definition *earley-recognized-it* :: *'a bins* ⇒ *'a cfg* ⇒ *'a sentence* ⇒ *bool* **where**
earley-recognized-it I cfg inp = (∃ *x* ∈ *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* ⟷ *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* ⟷ *derives cfg* [*$\mathfrak{S}$ cfg*] *inp*

8 Earley Parser Implementation

8.1 Draft

8.2 Pointer lemmas

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

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

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

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

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

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

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

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

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

8.3 Trees and Forests

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

fun $\textit{yield-tree} :: {'a}\ \textit{tree} \Rightarrow {'a}\ \textit{sentence}$ **where**
$\quad \textit{yield-tree}\ (\textit{Leaf}\ a) = [a]$
$\mid \textit{yield-tree}\ (\textit{Branch}\ \text{-}\ ts) = \textit{concat}\ (\textit{map yield-tree}\ ts)$

fun $\textit{root-tree} :: {'a}\ \textit{tree} \Rightarrow {'a}$ **where**
$\quad \textit{root-tree}\ (\textit{Leaf}\ a) = a$
$\mid \textit{root-tree}\ (\textit{Branch}\ N\ \text{-}) = N$

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

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

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

datatype ${'a}\ \textit{forest} =$
$\quad \textit{FLeaf}\ {'a}$
$\quad \mid \textit{FBranch}\ {'a}\ {'a}\ \textit{forest list list}$

fun $\textit{combinations} :: {'a}\ \textit{list list} \Rightarrow {'a}\ \textit{list list}$ **where**
$\quad \textit{combinations}\ [] = [[]]$
$\mid \textit{combinations}\ (xs\#xss) = [\ x\#cs\ .\ x <\!-\ xs,\ cs <\!-\ \textit{combinations}\ xss\ ]$

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

8.4 A single parse tree

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

```
  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-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-tree′* $bs\ inp\ k\ i = Some\ t$
shows *wf-item-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* $(bs!k)$ $k \le$ *length inp*
assumes *build-tree′* $bs\ inp\ k\ i = Some\ t$
shows *wf-yield-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-rule-tree cfg t* $\wedge$ *root-tree* $t = \mathfrak{S}\ cfg \wedge$ *yield-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* $(\mathfrak{I}$*-it cfg inp*$) = Some\ t$
shows *wf-rule-tree cfg t* $\wedge$ *root-tree* $t = \mathfrak{S}\ cfg \wedge$ *yield-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 $k < length\ bs\ i < length\ (bs!k)\ k \le length\ inp$
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 $\textit{wf-cfg cfg nonempty-derives cfg derives cfg}$ $[\mathfrak{S}\ \textit{cfg}]$ $\textit{inp}$
shows $\exists \textit{fs}.\ \textit{build-trees cfg inp}\ (\mathfrak{I}\textit{-it cfg inp}) = \textit{Some fs}$

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*:

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* ($\mathfrak{I}$*-it cfg1 inp1*) *cfg1 inp1* $\longleftrightarrow$ *derives cfg1* [$\mathfrak{S}$ *cfg1*] *inp1*
fun *size-bins* :: *'a bins* $\Rightarrow$ *nat* **where**
size-bins bs = *fold* (+) (*map length bs*) *0*

value $\mathfrak{I}$*-it cfg1 inp1*
value *size-bins* ($\mathfrak{I}$*-it cfg1 inp1*)
value *earley-recognized-it* ($\mathfrak{I}$*-it cfg1 inp1*) *cfg1 inp1*
value *build-trees cfg1 inp1* ($\mathfrak{I}$*-it cfg1 inp1*)
value *map-option* (*map trees*) (*build-trees cfg1 inp1* ($\mathfrak{I}$*-it cfg1 inp1*))
value *map-option* (*foldl* (+) *0* $\circ$ *map length*) (*map-option* (*map trees*) (*build-trees cfg1 inp1* ($\mathfrak{I}$*-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* ($\mathfrak{I}$-*it cfg2 inp2*) *cfg2 inp2* $\longleftrightarrow$ *derives cfg2* [$\mathfrak{S}$ *cfg2*] *inp2*
value $\mathfrak{I}$-*it cfg2 inp2*
value *earley-recognized-it* ($\mathfrak{I}$-*it cfg2 inp2*) *cfg2 inp2*
value *build-trees cfg2 inp2* ($\mathfrak{I}$-*it cfg2 inp2*)
value *map-option* (*map trees*) (*build-trees cfg2 inp2* ($\mathfrak{I}$-*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

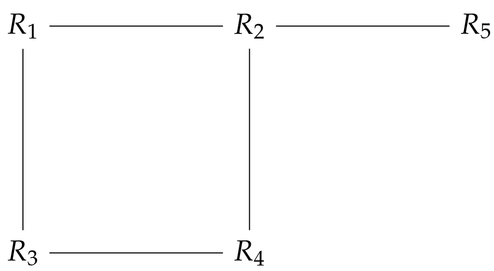

Figure 11.1: An example for a simple drawing.

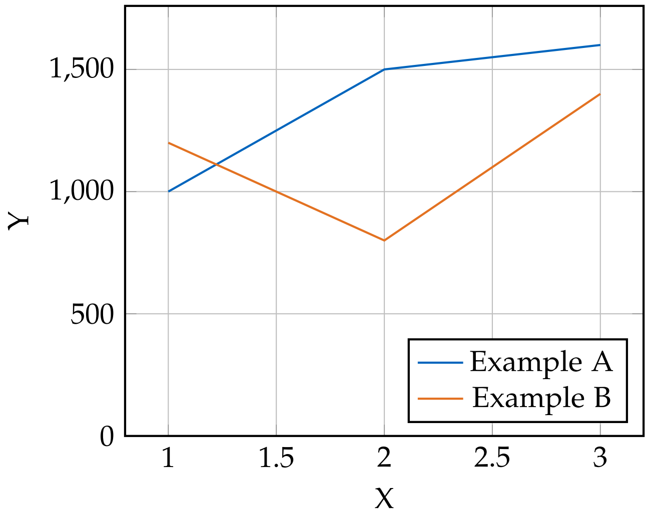

Figure 11.2: An example for a simple plot.

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

Figure 11.3: An example for a source code listing.

List of Figures

List of Tables