



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# **Formal Verification of an Earley Parser**

Martin Rau



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# **Formal Verification of an Earley Parser**

## **Formale Verifikation eines Earley Parsers**

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

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

Munich, 15.06.2023

Martin Rau

## Acknowledgments

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

# Abstract

TODO: Abstract

# Contents

<b>Acknowledgments</b>	<b>iii</b>
<b>Abstract</b>	<b>iv</b>
<b>1 Snippets</b>	<b>1</b>
1.1 Earley . . . . .	1
1.2 Scott . . . . .	3
1.3 Aycock . . . . .	5
1.4 Related Work . . . . .	6
1.4.1 Related Parsing Algorithms . . . . .	6
1.4.2 Related Verification Work . . . . .	7
1.5 Future Work . . . . .	7
<b>2 Introduction</b>	<b>9</b>
2.1 Motivation . . . . .	9
2.2 Structure . . . . .	9
2.3 Related Work . . . . .	9
2.4 Contributions . . . . .	9
<b>3 Earley’s Algorithm</b>	<b>10</b>
3.1 Draft . . . . .	10
3.2 Earley Recognizer . . . . .	10
<b>4 Earley Formalization</b>	<b>12</b>
4.1 Draft . . . . .	12
4.2 Background Theory . . . . .	13
4.3 Definitions . . . . .	14
4.4 Wellformedness . . . . .	17
4.5 Soundness . . . . .	18
4.6 Completeness . . . . .	19
4.7 Finiteness . . . . .	20
<b>5 Draft</b>	<b>21</b>

<b>6</b>	<b>Earley Recognizer Implementation</b>	<b>23</b>
6.1	Definitions . . . . .	23
6.2	Wellformedness . . . . .	26
6.3	List to set . . . . .	27
6.4	Set to list . . . . .	28
6.5	Main Theorem . . . . .	30
<b>7</b>	<b>Earley Parser Implementation</b>	<b>31</b>
7.1	Draft . . . . .	31
7.2	Pointer lemmas . . . . .	31
7.3	Trees and Forests . . . . .	32
7.4	A single parse tree . . . . .	33
7.5	Parse trees . . . . .	35
7.6	A word on completeness . . . . .	38
<b>8</b>	<b>Usage</b>	<b>39</b>
<b>9</b>	<b>Conclusion</b>	<b>41</b>
9.1	Summary . . . . .	41
9.2	Future Work . . . . .	41
	<b>List of Figures</b>	<b>42</b>
	<b>List of Tables</b>	<b>43</b>

# 1 Snippets

## 1.1 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  $A \rightarrow \alpha$ . 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  $\alpha \Rightarrow \beta$  if exists  $\gamma, \delta, \eta, A$  such that  $\alpha = \gamma A \delta$  and  $\beta = \gamma \eta \delta$  and  $A \rightarrow \eta$  is a production. We write  $\alpha \Rightarrow^* \beta$  if exists  $\alpha_0, \alpha_1, \dots, \alpha_m$  ( $m \geq 0$ ) such that  $\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_m = \beta$ . The sequence  $\alpha_i$  is called a derivation of  $\beta$  from  $\alpha$ . A sentential form is a string  $\alpha$  such that the root  $R \Rightarrow^* \alpha$ . 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 an input string and either accepts or rejects it depending on whether or not the string is a sentence of the grammar. A parser is a recognizer which also outputs the set of all legal derivation trees for the string.

The algorithm scans an input string  $X_1, \dots, X_n$  from left to right. As each symbol  $X_i$  is scanned, a set of states  $S_i$  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  $S_i$  as follows: we process the states in the set in order, performing one of three operations on each one depending on the form of the state. These operations may add more states to  $S_i$  and may also put states in a new state set  $S_{i+1}$ . 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  $S_i$  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  $S_i$ . Thus the predictor adds to  $S_i$  all the productions which might generate substrings beginning at  $X_{i+1}$ . The scanner is applicable in case there is a terminal to the right of the dot. The scanner compares that symbol with  $X_{i+1}$  and if they match, it adds the state to  $S_{i+1}$  with the dot moved over one in the state to indicate that that terminal symbol has been scanned. If we finish processing  $S_i$  and  $S_{i+1}$  remains empty an error has occurred in the input string. Otherwise, we start to process  $S_{i+1}$ . 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  $S_{i+1}$  consisting of the single state  $S \rightarrow \alpha \cdot, 0, n$ , 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.

## 1.2 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 parser which 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 grammars and worst case  $n^2$  on non-ambiguous 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 the 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 in the grammar

and report the problem back to the user. Another possibility 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 derivation that is returned. A truly general parser will return all possible derivations in some form. Perhaps the most well known representation is the shared packed parse forest SPPF described and used by Tomita [Tomita:1985]. Tomita's description of the representation does not allow for the infinitely many derivations which arise from grammars which contain cycles, the source adapts 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 also 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 process is often difficult to carry out correctly. Earley gave an algorithm for constructing derivations of a string accepted by his recogniser, 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 recursion and, as remarked above, had no mechanism for constructing complete SPPFs for grammars 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 the pointers in a way which allows a binarised SPPF to be constructed by walking the resulting structure. However, in order to construct a binarised SPPF we also have to introduce additional nodes for grammar rules of length greater than two, complicating the final algorithm.

### 1.3 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 constructs which come up often in practice (REFERENCE).

Earley parsers operate by constructing a sequence of sets, sometime called Earley sets. Given an input  $x_1x_2 \dots x_n$  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  $\dots$  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 following three steps to the items in  $S_i$  until no more can be added.  $\dots$  An item is added to a set only if it is not in the set already. The initial set  $S_0$  contains the items  $\dots$  to begin with. If the final set contains the item  $\dots$  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 efficiency 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  $A \rightarrow \text{dot}, j$  which corresponds to the epsilon-rule  $A \rightarrow \text{epsilon}$ , 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  $B \rightarrow \alpha \text{dot} A \beta, k$  is added to  $S_i$  after Completer has processed  $A \rightarrow \text{dot}, j$ , Completer will never add  $B \rightarrow \alpha A \text{dot} \beta, k$  to  $S_i$ . In turn, items resulting directly and indirectly from  $B \rightarrow \alpha A \text{dot} \beta, k$  will be omitted too. This effectively prunes potential 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  $A \rightarrow \alpha \text{dot} B \beta$ ,  $j$  is in  $S_i$ , add  $B \rightarrow \text{dot} \gamma$ ,  $i$  to  $S_i$  for all rules  $B \rightarrow \gamma$ . If  $B$  is nullable, also add  $A \rightarrow \alpha B \text{dot} \beta$ ,  $j$  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 effectively 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:  $S \rightarrow \text{AAAA}$ ,  $A \rightarrow a$ ,  $A \rightarrow E$ ,  $E \rightarrow \text{epsilon}$ , input a  $S_0$   $S \rightarrow \text{dot AAAA}$ , 0,  $A \rightarrow \text{dot a}$ , 0,  $A \rightarrow \text{dot E}$ , 0,  $E \rightarrow \text{dot}$ , 0,  $A \rightarrow E \text{dot}$ , 0,  $S \rightarrow A \text{dot AAA}$ , 0  $S_1$   $A \rightarrow a \text{dot}$ , 0,  $S \rightarrow A \text{dot AAA}$ , 0,  $S \rightarrow AA \text{dot AA}$ , 0,  $A \rightarrow \text{dot a}$ , 1,  $A \rightarrow \text{dot E}$ , 1,  $E \rightarrow \text{dot}$ , 1,  $A \rightarrow E \text{dot}$ , 1,  $S \rightarrow AAA \text{dot A}$ , 0

## 1.4 Related Work

### 1.4.1 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.

### 1.4.2 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.

## 1.5 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 grammars by allowing the user to 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.

## 2 Introduction

### 2.1 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!

### 2.2 Structure

standard blabla

### 2.3 Related Work

see folder and bibliography

### 2.4 Contributions

what did I do, what is new



## 3 Earley's Algorithm

### 3.1 Draft

- Introduce the Earley recognizer in the abstract set form
- Introduce the running example  $S \rightarrow x \mid S + S$  for input  $x + x + x$
- Illustrate the complete bins generated by the example
- Illustrate Initial  $S \rightarrow .\alpha, 0, 0$ , Scan  $A \rightarrow \alpha.\alpha\beta, i, j \mid A \rightarrow \alpha.\beta, i, j+1$ , Predict  $A \rightarrow \alpha.\beta, i, j$  and  $B \rightarrow \gamma \mid B \rightarrow .\gamma, j, j$ , Complete  $A \rightarrow \alpha.\beta, i, j$  and  $B \rightarrow \gamma, j, k \mid A \rightarrow \alpha\beta, i, k$
- Define goal:  $A \rightarrow \alpha.\beta, i, j$  iff  $A \Rightarrow^* s[i..j)\beta$  which implies  $S \rightarrow \alpha., 0, n+1$  iff  $S \Rightarrow^* s$

### 3.2 Earley Recognizer

$$A \rightarrow \alpha \bullet \beta, i, j \text{ iff } A \xRightarrow{*} \omega[i..j)$$

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

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

$$\begin{array}{c}
 \text{INIT} \\
 \hline
 S \rightarrow \bullet \alpha, 0, 0 \\
 \\
 \text{SCAN} \\
 \hline
 \frac{A \rightarrow \alpha \bullet a \beta, i, j \quad \omega[j] = a}{A \rightarrow \alpha a \bullet \beta, i, j+1} \\
 \\
 \text{PREDICT} \\
 \hline
 \frac{A \rightarrow \alpha \bullet B \beta, i, j \quad B \rightarrow \gamma \in \mathcal{G}}{B \rightarrow \bullet \gamma, j, j} \\
 \\
 \text{COMPLETE} \\
 \hline
 \frac{A \rightarrow \alpha \bullet B \beta, i, j \quad B \rightarrow \gamma \bullet, j, k}{A \rightarrow \alpha B \bullet \beta, i, k}
 \end{array}$$

Figure 3.1: Earley inference rules

Table 3.1: Earley items for the CFG  $S \rightarrow x, S \rightarrow S + S$

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

## 4 Earley Formalization

### 4.1 Draft

- Introduce background theory about CFG and Isabelle syntax
- 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

## 4.2 Background Theory

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

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

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

**datatype**  $'a \text{ cfg} =$

$\text{CFG}$

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

$(\mathfrak{T} : 'a \text{ list})$

$(\mathfrak{R} : 'a \text{ rules})$

$(\mathfrak{S} : 'a)$

**definition**  $\text{derives1} :: 'a \text{ cfg} \Rightarrow 'a \text{ sentential} \Rightarrow 'a \text{ sentential} \Rightarrow \text{bool}$  **where**

$\text{derives1 } \text{cfg } u \ v \equiv$

$\exists x \ y \ N \ \alpha.$

$u = x @ [N] @ y$

$\wedge v = x @ \alpha @ y$

$\wedge (N, \alpha) \in \text{set } (\mathfrak{R} \text{ cfg})$

**definition**  $\text{derivations1} :: 'a \text{ cfg} \Rightarrow ('a \text{ sentential} \times 'a \text{ sentential}) \text{ set}$  **where**

$\text{derivations1 } \text{cfg} = \{ (u, v) \mid u \ v. \text{derives1 } \text{cfg } u \ v \}$

**definition**  $\text{derivations} :: 'a \text{ cfg} \Rightarrow ('a \text{ sentential} \times 'a \text{ sentential}) \text{ set}$  **where**

$\text{derivations } \text{cfg} = (\text{derivations1 } \text{cfg})^*$

**definition**  $\text{derives} :: 'a \text{ cfg} \Rightarrow 'a \text{ sentential} \Rightarrow 'a \text{ sentential} \Rightarrow \text{bool}$  **where**

$\text{derives } \text{cfg } u \ v \equiv (u, v) \in \text{derivations } \text{cfg}$

**fun**  $\text{slice} :: \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$  **where**

$\text{slice } - \ 0 = []$

$| \text{slice } - \ 0 \ (x \# xs) = []$

$| \text{slice } 0 \ (\text{Suc } b) \ (x \# xs) = x \# \text{slice } 0 \ b \ xs$

$| \text{slice } (\text{Suc } a) \ (\text{Suc } b) \ (x \# xs) = \text{slice } a \ b \ xs$

**lemma**  $\text{slice-append}:$

**assumes**  $a \leq b \ b \leq c$

**shows**  $\text{slice } a \ b \ xs @ \text{slice } b \ c \ xs = \text{slice } a \ c \ xs$

**definition**  $\text{disjunct-symbols} :: 'a \text{ cfg} \Rightarrow \text{bool}$  **where**

$\text{disjunct-symbols } \text{cfg} \longleftrightarrow \text{set } (\mathfrak{N} \text{ cfg}) \cap \text{set } (\mathfrak{T} \text{ cfg}) = \{\}$

**definition**  $\text{wf-startsymbol} :: 'a \text{ cfg} \Rightarrow \text{bool}$  **where**

$\text{wf-startsymbol } \text{cfg} \longleftrightarrow \mathfrak{S} \text{ cfg} \in \text{set } (\mathfrak{N} \text{ cfg})$

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

*wf-rules* cfg  $\equiv \forall (N, \alpha) \in \text{set } (\mathfrak{N} \text{ cfg}). N \in \text{set } (\mathfrak{N} \text{ cfg}) \wedge (\forall s \in \text{set } \alpha. s \in \text{set } (\mathfrak{N} \text{ cfg}) \cup \text{set } (\mathfrak{T} \text{ cfg}))$

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

*distinct-rules* cfg = *distinct* ( $\mathfrak{N}$  cfg)

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

*wf-cfg* cfg  $\longleftrightarrow \text{disjunct-symbols} \text{ cfg} \wedge \text{wf-startsymbol} \text{ cfg} \wedge \text{wf-rules} \text{ cfg} \wedge \text{distinct-rules} \text{ cfg}$

**definition** *is-terminal* :: 'a cfg  $\Rightarrow$  'a  $\Rightarrow$  bool **where**

*is-terminal* cfg s  $\equiv s \in \text{set } (\mathfrak{T} \text{ cfg})$

**definition** *is-nonterminal* :: 'a cfg  $\Rightarrow$  'a  $\Rightarrow$  bool **where**

*is-nonterminal* cfg s  $\equiv s \in \text{set } (\mathfrak{N} \text{ cfg})$

**definition** *is-symbol* :: 'a cfg  $\Rightarrow$  'a  $\Rightarrow$  bool **where**

*is-symbol* cfg s  $\longleftrightarrow \text{is-terminal} \text{ cfg } s \vee \text{is-nonterminal} \text{ cfg } s$

**definition** *wf-sentential* :: 'a cfg  $\Rightarrow$  'a sentential  $\Rightarrow$  bool **where**

*wf-sentential* cfg s  $\equiv \forall x \in \text{set } s. \text{is-symbol} \text{ cfg } x$

**definition** *is-sentence* :: 'a cfg  $\Rightarrow$  'a sentential  $\Rightarrow$  bool **where**

*is-sentence* cfg s  $\equiv \forall x \in \text{set } s. \text{is-terminal} \text{ cfg } x$

### 4.3 Definitions

**definition** *rule-head* :: 'a rule  $\Rightarrow$  'a **where**

*rule-head* = fst

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

*rule-body* = snd

**datatype** 'a item =

*Item*

(*item-rule*: 'a rule)

(*item-dot*: nat)

(*item-origin*: nat)

(*item-end*: nat)

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

**definition** *item-rule-head* :: 'a item  $\Rightarrow$  'a **where**

*item-rule-head* x = *rule-head* (*item-rule* x)

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

**definition** *item- $\alpha$*  :: 'a item  $\Rightarrow$  'a sentential **where**  
*item- $\alpha$*   $x = \text{take } (\text{item-dot } x) (\text{item-rule-body } x)$

**definition** *item- $\beta$*  :: 'a item  $\Rightarrow$  'a sentential **where**  
*item- $\beta$*   $x = \text{drop } (\text{item-dot } x) (\text{item-rule-body } x)$

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

**definition** *is-complete* :: 'a item  $\Rightarrow$  bool **where**  
*is-complete*  $x \equiv \text{item-dot } x \geq \text{length } (\text{item-rule-body } x)$

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

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

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

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

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

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

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

**definition**  $\text{Init} :: 'a \text{ cfg} \Rightarrow 'a \text{ items where}$

$\text{Init } \text{cfg} = \{ \text{init-item } r \ 0 \mid r. r \in \text{set } (\mathfrak{R} \text{ cfg}) \wedge \text{fst } r = (\mathfrak{S} \text{ cfg}) \}$

**definition**  $\text{Scan} :: \text{nat} \Rightarrow 'a \text{ sentential} \Rightarrow 'a \text{ items} \Rightarrow 'a \text{ items where}$

$\text{Scan } k \text{ inp } I =$   
 $\{ \text{inc-item } x \ (k+1) \mid x \ a.$   
 $x \in \text{bin } I \ k \wedge$   
 $\text{inp}!k = a \wedge$   
 $k < \text{length } \text{inp} \wedge$   
 $\text{next-symbol } x = \text{Some } a \}$

**definition**  $\text{Predict} :: \text{nat} \Rightarrow 'a \text{ cfg} \Rightarrow 'a \text{ items} \Rightarrow 'a \text{ items where}$

$\text{Predict } k \text{ cfg } I =$   
 $\{ \text{init-item } r \ k \mid r \ x.$   
 $r \in \text{set } (\mathfrak{R} \text{ cfg}) \wedge$   
 $x \in \text{bin } I \ k \wedge$   
 $\text{next-symbol } x = \text{Some } (\text{rule-head } r) \}$

**definition**  $\text{Complete} :: \text{nat} \Rightarrow 'a \text{ items} \Rightarrow 'a \text{ items where}$

$\text{Complete } k \ I =$   
 $\{ \text{inc-item } x \ k \mid x \ y.$   
 $x \in \text{bin } I \ (\text{item-origin } y) \wedge$   
 $y \in \text{bin } I \ k \wedge$   
 $\text{is-complete } y \wedge$   
 $\text{next-symbol } x = \text{Some } (\text{item-rule-head } y) \}$

**fun**  $\text{funpower} :: ('a \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow ('a \Rightarrow 'a) \text{ where}$

$\text{funpower } f \ 0 \ x = x$   
 $\mid \text{funpower } f \ (\text{Suc } n) \ x = f \ (\text{funpower } f \ n \ x)$

**definition**  $\text{natUnion} :: (\text{nat} \Rightarrow 'a \text{ set}) \Rightarrow 'a \text{ set where}$

$\text{natUnion } f = \bigcup \{ f \ n \mid n. \text{True} \}$

**definition**  $\text{limit} :: ('a \text{ set} \Rightarrow 'a \text{ set}) \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set where}$

$\text{limit } f \ x = \text{natUnion } (\lambda n. \text{funpower } f \ n \ x)$

**definition**  $\text{E-step} :: \text{nat} \Rightarrow 'a \text{ cfg} \Rightarrow 'a \text{ sentential} \Rightarrow 'a \text{ items} \Rightarrow 'a \text{ items where}$

$\text{E-step } k \text{ cfg } \text{inp } I = I \cup \text{Scan } k \text{ inp } I \cup \text{Complete } k \ I \cup \text{Predict } k \text{ cfg } I$

**definition**  $\text{E} :: \text{nat} \Rightarrow 'a \text{ cfg} \Rightarrow 'a \text{ sentential} \Rightarrow 'a \text{ items} \Rightarrow 'a \text{ items where}$

$\text{E } k \text{ cfg } \text{inp } I = \text{limit } (\text{E-step } k \text{ cfg } \text{inp}) \ I$

**fun**  $\mathcal{E} :: \text{nat} \Rightarrow 'a \text{ cfg} \Rightarrow 'a \text{ sentential} \Rightarrow 'a \text{ items where}$

$\mathcal{E} \ 0 \ \text{cfg} \ \text{inp} = E \ 0 \ \text{cfg} \ \text{inp} \ (\text{Init} \ \text{cfg})$   
 $| \ \mathcal{E} \ (\text{Suc} \ n) \ \text{cfg} \ \text{inp} = E \ (\text{Suc} \ n) \ \text{cfg} \ \text{inp} \ (\mathcal{E} \ n \ \text{cfg} \ \text{inp})$

**definition** *earley* :: 'a cfg  $\Rightarrow$  'a sentential  $\Rightarrow$  'a items **where**  
*earley* cfg inp =  $\mathcal{E} \ (\text{length} \ \text{inp}) \ \text{cfg} \ \text{inp}$

## 4.4 Wellformedness

**lemma** *wf-Init*:

**assumes**  $x \in \text{Init} \ \text{cfg}$

**shows** *wf-item* cfg inp  $x$

by definition

**lemma** *wf-Scan-Predict-Complete*:

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

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

by definition

**lemma** *wf-E-step*:

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

**shows** *wf-items* cfg inp  $(\text{E-step} \ k \ \text{cfg} \ \text{inp} \ I)$

*wf-Scan-Predict-Complete* by definition

**lemma** *wf-funpower*:

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

**shows** *wf-items* cfg inp  $(\text{funpower} \ (\text{E-step} \ k \ \text{cfg} \ \text{inp}) \ n \ I)$

*wf-E-step*, by induction on  $n$

**lemma** *wf-E*:

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

**shows** *wf-items* cfg inp  $(E \ k \ \text{cfg} \ \text{inp} \ I)$

*wf-funpower* by definition

**lemma** *wf-E0*:

**shows** *wf-items* cfg inp  $(E \ 0 \ \text{cfg} \ \text{inp} \ (\text{Init} \ \text{cfg}))$

*wf-Init wf-E* by definition

**lemma** *wf- $\mathcal{E}$* :

**shows** *wf-items* cfg inp  $(\mathcal{E} \ n \ \text{cfg} \ \text{inp})$

*wf-E0 wf-E* by induction on  $n$

**lemma** *wf-earley*:

**shows** *wf-items* cfg inp  $(\text{earley} \ \text{cfg} \ \text{inp})$

*wf- $\mathcal{E}$*  by definition



## 4.5 Soundness

**definition** *sound-item* :: 'a cfg  $\Rightarrow$  'a sentential  $\Rightarrow$  '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- $\beta$  x)

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

**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-E-step*:

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

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

**lemma** *sound-E*:

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

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

**lemma** *sound-E0*:

**shows** *sound-items* cfg inp (E 0 cfg inp (Init cfg))

**lemma** *sound- $\mathcal{E}$* :

**shows** *sound-items* cfg inp ( $\mathcal{E}$  k cfg inp)

**lemma** *sound-earley*:

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

**theorem** *soundness*:

**assumes** *earley-recognized* (earley cfg inp) cfg inp

**shows** derives cfg [ $\mathcal{S}$  cfg] inp

## 4.6 Completeness

**lemma** *Scan- $\mathcal{E}$* :

**assumes**  $i+1 \leq k \leq \text{length } \text{inp}$   $x \in \text{bin } (\mathcal{E} \ k \ \text{cfg} \ \text{inp}) \ i$

**assumes**  $\text{next-symbol } x = \text{Some } a \ \text{inp}!i = a$

**shows**  $\text{inc-item } x \ (i+1) \in \mathcal{E} \ k \ \text{cfg} \ \text{inp}$

**lemma** *Predict- $\mathcal{E}$* :

**assumes**  $i \leq k$   $x \in \text{bin } (\mathcal{E} \ k \ \text{cfg} \ \text{inp}) \ i$   $\text{next-symbol } x = \text{Some } N \ (N, \alpha) \in \text{set } (\mathfrak{R} \ \text{cfg})$

**shows**  $\text{init-item } (N, \alpha) \ i \in \mathcal{E} \ k \ \text{cfg} \ \text{inp}$

**lemma** *Complete- $\mathcal{E}$* :

**assumes**  $i \leq j \leq k$   $x \in \text{bin } (\mathcal{E} \ k \ \text{cfg} \ \text{inp}) \ i$   $\text{next-symbol } x = \text{Some } N \ (N, \alpha) \in \text{set } (\mathfrak{R} \ \text{cfg})$

**assumes**  $i = \text{item-origin } y$   $y \in \text{bin } (\mathcal{E} \ k \ \text{cfg} \ \text{inp}) \ j$   $\text{item-rule } y = (N, \alpha)$   $\text{is-complete } y$

**shows**  $\text{inc-item } x \ j \in \mathcal{E} \ k \ \text{cfg} \ \text{inp}$

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

**definition** *Derives1* ::  $'a \ \text{cfg} \Rightarrow 'a \ \text{sentential} \Rightarrow \text{nat} \Rightarrow 'a \ \text{rule} \Rightarrow 'a \ \text{sentential} \Rightarrow \text{bool}$  **where**

$\text{Derives1} \ \text{cfg} \ u \ i \ r \ v \equiv$

$\exists \ x \ y \ N \ \alpha.$

$u = x \ @ \ [N] \ @ \ y$

$\wedge \ v = x \ @ \ \alpha \ @ \ y$

$\wedge \ (N, \alpha) \in \text{set } (\mathfrak{R} \ \text{cfg})$

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

**fun** *Derivation* ::  $'a \ \text{cfg} \Rightarrow 'a \ \text{sentential} \Rightarrow 'a \ \text{derivation} \Rightarrow 'a \ \text{sentential} \Rightarrow \text{bool}$  **where**

$\text{Derivation} \ a \ [] \ b = (a = b)$

$| \text{Derivation} \ \text{cfg} \ a \ (d \# D) \ b = (\exists \ x. \text{Derives1} \ \text{cfg} \ a \ (\text{fst } d) \ (\text{snd } d) \ x \wedge \text{Derivation} \ \text{cfg} \ x \ D \ b)$

**definition** *partially-completed* ::  $\text{nat} \Rightarrow 'a \ \text{cfg} \Rightarrow 'a \ \text{sentential} \Rightarrow 'a \ \text{items} \Rightarrow ('a \ \text{derivation} \Rightarrow \text{bool}) \Rightarrow \text{bool}$  **where**

$\text{partially-completed} \ k \ \text{cfg} \ \text{inp} \ I \ P \equiv$

$\forall \ i \ j \ x \ a \ D.$

$i \leq j \wedge j \leq k \wedge k \leq \text{length } \text{inp} \wedge$

$x \in \text{bin } I \ i \wedge \text{next-symbol } x = \text{Some } a \wedge$

$\text{Derivation} \ \text{cfg} \ [a] \ D \ (\text{slice } i \ j \ \text{inp}) \wedge P \ D \longrightarrow$

$\text{inc-item } x \ j \in I$

**lemma** *fully-completed*:

**assumes**  $j \leq k \leq \text{length } \text{inp}$

**assumes**  $x = \text{Item } (N, \alpha) \ d \ i \ j \ x \in I \ \text{wf-items} \ \text{cfg} \ \text{inp} \ I$

**assumes**  $\text{Derivation} \ \text{cfg} \ (\text{item-}\beta \ x) \ D \ (\text{slice } j \ k \ \text{inp})$

**assumes**  $\text{partially-completed} \ k \ \text{cfg} \ \text{inp} \ I \ (\lambda D'. \text{length } D' \leq \text{length } D)$

**shows**  $\text{Item } (N, \alpha) \ (\text{length } \alpha) \ i \ k \in I$

**lemma** *partially-completed- $\mathcal{E}$* :

**assumes**  $\text{wf-cfg} \ \text{cfg}$

**shows** *partially-completed*  $k$  *cfg inp* ( $\mathcal{E}$   $k$  *cfg inp*) ( $\lambda\cdot$ . *True*)  
**lemma** *partially-completed-earley*:  
  **assumes** *wf-cfg* *cfg*  
  **shows** *partially-completed* (*length inp*) *cfg inp* (*earley cfg inp*) ( $\lambda\cdot$ . *True*)  
**theorem** *completeness*:  
  **assumes** *derives* *cfg* [ $\mathcal{S}$  *cfg*] *inp* *is-sentence* *cfg inp* *wf-cfg* *cfg*  
  **shows** *earley-recognized* (*earley cfg inp*) *cfg inp*  
**corollary**  
  **assumes** *wf-cfg* *cfg* *is-sentence* *cfg inp*  
  **shows** *earley-recognized* (*earley cfg inp*) *cfg inp*  $\longleftrightarrow$  *derives* *cfg* [ $\mathcal{S}$  *cfg*] *inp*

### 4.7 Finiteness

**lemma** *finiteness-UNIV-wf-item*:  
  **shows** *finite*  $\{ x \mid x. \text{wf-item } \text{cfg } \text{inp } x \}$   
**theorem** *finiteness*:  
  **shows** *finite* (*earley cfg inp*)

## 5 Draft

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

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

## 6 Earley Recognizer Implementation

### 6.1 Definitions

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

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

**datatype** *pointer* =  
  Null  
  | *Pre* *nat*  
  | *PreRed* *nat*  $\times$  *nat*  $\times$  *nat* (*nat*  $\times$  *nat*  $\times$  *nat*) *list*

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

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

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

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

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

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

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

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

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

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

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

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

*wf-bin-items* cfg inp k xs  $\equiv \forall x \in \text{set } xs. \text{wf-item } \text{cfg } \text{inp } x \wedge \text{item-end } x = k$

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

*wf-bin* cfg inp k b  $\equiv \text{distinct } (\text{items } b) \wedge \text{wf-bin-items } \text{cfg } \text{inp } k (\text{items } b)$

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

*wf-bins* cfg inp bs  $\equiv \forall k < \text{length } bs. \text{wf-bin } \text{cfg } \text{inp } k (bs ! k)$

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

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

**definition** *Init-list* :: 'a cfg  $\Rightarrow$  'a sentential  $\Rightarrow$  'a bins **where**

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

**definition** *Scan-list* :: nat  $\Rightarrow$  'a sentential  $\Rightarrow$  'a  $\Rightarrow$  'a item  $\Rightarrow$  nat  $\Rightarrow$  'a entry list **where**

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

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

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

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

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

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

```

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

```

```

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

```

```

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

```

```

partial-function (tailrec) E-list' :: nat => 'a cfg => 'a sentential => 'a bins => nat => 'a bins where
  E-list' 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-list k inp a x i)
            else bs
          else bins-upd bs k (Predict-list k cfg a)
        | None => bins-upd bs k (Complete-list k x bs i)
    in E-list' k cfg inp bs' (i+1))

```

```

definition E-list :: nat => 'a cfg => 'a sentential => 'a bins => 'a bins where
  E-list k cfg inp bs = E-list' k cfg inp bs 0

```

```

fun E-list :: nat => 'a cfg => 'a sentential => 'a bins where
  E-list 0 cfg inp = E-list 0 cfg inp (Init-list cfg inp)
| E-list (Suc n) cfg inp = E-list (Suc n) cfg inp (E-list n cfg inp)

```

```

definition earley-list :: 'a cfg => 'a sentential => 'a bins where
  earley-list cfg inp = E-list (length inp) cfg inp

```



## 6.2 Wellformedness

**lemma** *wf-bin-bin-upd*:

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

**lemma** *wf-bin-bin-upds*:

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

**lemma** *wf-bins-bins-upd*:

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

**lemma** *wf-bins-Init-list*:

**assumes** *wf-cfg cfg*  
**shows** *wf-bins cfg inp (Init-list cfg inp)*

**lemma** *wf-bins-Scan-list*:

**assumes** *wf-bins cfg inp bs k < length bs x ∈ set (items (bs!k)) k < length inp next-symbol x ≠ None*  
**shows**  $\forall y \in \text{set } (\text{items } (\text{Scan-list } k \text{ inp } a \text{ pre})). \text{wf-item cfg inp } y \wedge \text{item-end } y = k+1$

**lemma** *wf-bins-Predict-list*:

**assumes** *wf-bins cfg inp bs k < length bs k ≤ length inp wf-cfg cfg*  
**shows**  $\forall y \in \text{set } (\text{items } (\text{Predict-list } k \text{ cfg } X)). \text{wf-item cfg inp } y \wedge \text{item-end } y = k$

**lemma** *wf-bins-Complete-list*:

**assumes** *wf-bins cfg inp bs k < length bs y ∈ set (items (bs!k))*  
**shows**  $\forall x \in \text{set } (\text{items } (\text{Complete-list } k \text{ y bs red})). \text{wf-item cfg inp } x \wedge \text{item-end } x = k$

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

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

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

*wellformed-bins* = {  
 (*k, cfg, inp, bs*) | *k* *cfg inp bs*.  
*k* ≤ *length inp* ∧  
*length bs* = *length inp* + 1 ∧  
*wf-cfg cfg* ∧  
*wf-bins cfg inp bs*  
}

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

**lemma** *wellformed-bins-Init-list*:

**assumes** *k ≤ length inp wf-cfg cfg*  
**shows** (*k, cfg, inp, Init-list cfg inp*) ∈ *wellformed-bins*

**lemma** *wellformed-bins-Complete-list*:

**assumes** (*k, cfg, inp, bs*) ∈ *wellformed-bins*  $\neg \text{length } (\text{items } (bs!k)) \leq i$

**assumes**  $x = \text{items } (bs!k)!i \text{ next-symbol } x = \text{None}$   
**shows**  $(k, \text{cfg}, \text{inp}, \text{bins-upd } bs \ k \ (\text{Complete-list } k \ x \ bs \ \text{red})) \in \text{wellformed-bins}$   
**lemma** *wellformed-bins-Scan-list*:  
**assumes**  $(k, \text{cfg}, \text{inp}, bs) \in \text{wellformed-bins} \neg \text{length } (\text{items } (bs!k)) \leq i$   
**assumes**  $x = \text{items } (bs!k)!i \text{ next-symbol } x = \text{Some } a$   
**assumes**  $\text{is-terminal } \text{cfg } a \ k < \text{length } \text{inp}$   
**shows**  $(k, \text{cfg}, \text{inp}, \text{bins-upd } bs \ (k+1) \ (\text{Scan-list } k \ \text{inp } a \ x \ \text{pre})) \in \text{wellformed-bins}$   
**lemma** *wellformed-bins-Predict-list*:  
**assumes**  $(k, \text{cfg}, \text{inp}, bs) \in \text{wellformed-bins} \neg \text{length } (\text{items } (bs!k)) \leq i$   
**assumes**  $x = \text{items } (bs!k)!i \text{ next-symbol } x = \text{Some } a \neg \text{is-terminal } \text{cfg } a$   
**shows**  $(k, \text{cfg}, \text{inp}, \text{bins-upd } bs \ k \ (\text{Predict-list } k \ \text{cfg } a)) \in \text{wellformed-bins}$   
**lemma** *wellformed-bins-E-list'*:  
**assumes**  $(k, \text{cfg}, \text{inp}, bs) \in \text{wellformed-bins}$   
**shows**  $(k, \text{cfg}, \text{inp}, \text{E-list}' \ k \ \text{cfg} \ \text{inp} \ bs \ i) \in \text{wellformed-bins}$   
**lemma** *wellformed-bins-E-list*:  
**assumes**  $(k, \text{cfg}, \text{inp}, bs) \in \text{wellformed-bins}$   
**shows**  $(k, \text{cfg}, \text{inp}, \text{E-list } k \ \text{cfg} \ \text{inp} \ bs) \in \text{wellformed-bins}$   
**lemma** *wellformed-bins-E-list*:  
**assumes**  $k \leq \text{length } \text{inp} \ \text{wf-cfg } \text{cfg}$   
**shows**  $(k, \text{cfg}, \text{inp}, \text{E-list } k \ \text{cfg} \ \text{inp}) \in \text{wellformed-bins}$   
**lemma** *wellformed-bins-earley-list*:  
**assumes**  $k \leq \text{length } \text{inp} \ \text{wf-cfg } \text{cfg}$   
**shows**  $(k, \text{cfg}, \text{inp}, \text{earley-list } \text{cfg} \ \text{inp}) \in \text{wellformed-bins}$   
**lemma** *wf-bins-E-list'*:  
**assumes**  $(k, \text{cfg}, \text{inp}, bs) \in \text{wellformed-bins}$   
**shows**  $\text{wf-bins } \text{cfg} \ \text{inp} \ (\text{E-list}' \ k \ \text{cfg} \ \text{inp} \ bs \ i)$   
**lemma** *wf-bins-E-list*:  
**assumes**  $(k, \text{cfg}, \text{inp}, bs) \in \text{wellformed-bins}$   
**shows**  $\text{wf-bins } \text{cfg} \ \text{inp} \ (\text{E-list } k \ \text{cfg} \ \text{inp} \ bs)$   
**lemma** *wf-bins-E-list*:  
**assumes**  $k \leq \text{length } \text{inp} \ \text{wf-cfg } \text{cfg}$   
**shows**  $\text{wf-bins } \text{cfg} \ \text{inp} \ (\text{E-list } k \ \text{cfg} \ \text{inp})$   
**lemma** *wf-bins-earley-list*:  
**assumes**  $\text{wf-cfg } \text{cfg}$   
**shows**  $\text{wf-bins } \text{cfg} \ \text{inp} \ (\text{earley-list } \text{cfg} \ \text{inp})$

### 6.3 List to set

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

**shows**  $\text{set } (\text{items } (\text{Scan-list } k \text{ inp } a \text{ } x \text{ } \text{pre})) \subseteq \text{Scan } k \text{ inp } I$

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

**lemma** *Complete-list-sub-Complete*:  
**assumes**  $\text{wf-bins } \text{cfg } \text{inp } \text{bs } \text{bins-items } \text{bs} \subseteq I \text{ } y \in \text{set } (\text{items } (\text{bs } ! \text{ } k)) \text{ } k < \text{length } \text{bs}$   
**assumes**  $\text{next-symbol } y = \text{None}$   
**shows**  $\text{set } (\text{items } (\text{Complete-list } k \text{ } y \text{ } \text{bs red})) \subseteq \text{Complete } k \text{ } I$

**lemma** *E-list'-sub-E*:  
**assumes**  $(k, \text{cfg}, \text{inp}, \text{bs}) \in \text{wellformed-bins}$   
**assumes**  $\text{bins-items } \text{bs} \subseteq I$   
**shows**  $\text{bins-items } (\text{E-list}' k \text{ cfg } \text{inp } \text{bs } i) \subseteq E \text{ } k \text{ cfg } \text{inp } I$

**lemma** *E-list-sub-E*:  
**assumes**  $(k, \text{cfg}, \text{inp}, \text{bs}) \in \text{wellformed-bins}$   
**assumes**  $\text{bins-items } \text{bs} \subseteq I$   
**shows**  $\text{bins-items } (\text{E-list } k \text{ cfg } \text{inp } \text{bs}) \subseteq E \text{ } k \text{ cfg } \text{inp } I$

**lemma** *E-list-sub-E*:  
**assumes**  $k \leq \text{length } \text{inp } \text{wf-cfg } \text{cfg}$   
**shows**  $\text{bins-items } (\text{E-list } k \text{ cfg } \text{inp}) \subseteq E \text{ } k \text{ cfg } \text{inp}$

**lemma** *earley-list-sub-earley*:  
**assumes**  $\text{wf-cfg } \text{cfg}$   
**shows**  $\text{bins-items } (\text{earley-list } \text{cfg } \text{inp}) \subseteq \text{earley } \text{cfg } \text{inp}$

## 6.4 Set to list

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

**lemma** *Complete-Un-eq-terminal*:  
**assumes**  $\text{next-symbol } z = \text{Some } a \text{ is-terminal } \text{cfg } a \text{ wf-items } \text{cfg } \text{inp } I \text{ wf-item } \text{cfg } \text{inp } z \text{ wf-cfg } \text{cfg}$   
**shows**  $\text{Complete } k \text{ } (I \cup \{z\}) = \text{Complete } k \text{ } I$

**lemma** *Complete-Un-eq-nonterminal*:  
**assumes**  $\text{next-symbol } z = \text{Some } a \text{ is-nonterminal } \text{cfg } a \text{ sound-items } \text{cfg } \text{inp } I \text{ item-end } z = k$   
**assumes**  $\text{wf-items } \text{cfg } \text{inp } I \text{ wf-item } \text{cfg } \text{inp } z \text{ wf-cfg } \text{cfg } \text{nonempty-derives } \text{cfg}$   
**shows**  $\text{Complete } k \text{ } (I \cup \{z\}) = \text{Complete } k \text{ } I$

**lemma** *Complete-sub-bins-Un-Complete-list*:  
**assumes**  $\text{Complete } k \text{ } I \subseteq \text{bins-items } \text{bs } I \subseteq \text{bins-items } \text{bs } \text{is-complete } z \text{ wf-bins } \text{cfg } \text{inp } \text{bs } \text{wf-item } \text{cfg } \text{inp } z$   
**shows**  $\text{Complete } k \text{ } (I \cup \{z\}) \subseteq \text{bins-items } \text{bs} \cup \text{set } (\text{items } (\text{Complete-list } k \text{ } z \text{ } \text{bs red}))$

**lemma** *E-list'-mono*:  
**assumes**  $(k, \text{cfg}, \text{inp}, \text{bs}) \in \text{wellformed-bins}$

**shows**  $\text{bins-items } bs \subseteq \text{bins-items } (E\text{-list}' k \text{ cfg inp } bs \ i)$

**lemma**  $E\text{-step-sub-}E\text{-list}'$ :

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

**assumes**  $E\text{-step } k \text{ cfg inp } (\text{bins-items-upto } bs \ k \ i) \subseteq \text{bins-items } bs$

**assumes**  $\text{sound-items } \text{cfg inp } (\text{bins-items } bs) \text{ is-sentence } \text{cfg inp nonempty-derives } \text{cfg}$

**shows**  $E\text{-step } k \text{ cfg inp } (\text{bins-items } bs) \subseteq \text{bins-items } (E\text{-list}' k \text{ cfg inp } bs \ i)$

**lemma**  $E\text{-step-sub-}E\text{-list}$ :

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

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

**assumes**  $\text{sound-items } \text{cfg inp } (\text{bins-items } bs) \text{ is-sentence } \text{cfg inp nonempty-derives } \text{cfg}$

**shows**  $E\text{-step } k \text{ cfg inp } (\text{bins-items } bs) \subseteq \text{bins-items } (E\text{-list } k \text{ cfg inp } bs)$

**lemma**  $E\text{-list}'\text{-bins-items-eq}$ :

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

**assumes**  $\text{bins-eq-items } as \ bs \ \text{wf-bins } \text{cfg inp } as$

**shows**  $\text{bins-eq-items } (E\text{-list}' k \text{ cfg inp } as \ i) \ (E\text{-list}' k \text{ cfg inp } bs \ i)$

**lemma**  $E\text{-list}'\text{-idem}$ :

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

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

**shows**  $\text{bins-items } (E\text{-list}' k \text{ cfg inp } (E\text{-list}' k \text{ cfg inp } bs \ i) \ j) = \text{bins-items } (E\text{-list}' k \text{ cfg inp } bs \ i)$

**lemma**  $E\text{-list-idem}$ :

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

**assumes**  $\text{sound-items } \text{cfg inp } (\text{bins-items } bs) \text{ nonempty-derives } \text{cfg}$

**shows**  $\text{bins-items } (E\text{-list } k \text{ cfg inp } (E\text{-list } k \text{ cfg inp } bs)) = \text{bins-items } (E\text{-list } k \text{ cfg inp } bs)$

**lemma**  $\text{funpower-}E\text{-step-sub-}E\text{-list}$ :

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

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

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

**shows**  $\text{funpower } (E\text{-step } k \text{ cfg inp } n \ (\text{bins-items } bs)) \subseteq \text{bins-items } (E\text{-list } k \text{ cfg inp } bs)$

**lemma**  $E\text{-sub-}E\text{-list}$ :

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

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

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

**shows**  $E \ k \ \text{cfg inp } (\text{bins-items } bs) \subseteq \text{bins-items } (E\text{-list } k \text{ cfg inp } bs)$

**lemma**  $\mathcal{E}\text{-sub-}\mathcal{E}\text{-list}$ :

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

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

**shows**  $\mathcal{E} \ k \ \text{cfg inp} \subseteq \text{bins-items } (\mathcal{E}\text{-list } k \text{ cfg inp})$

**lemma**  $\text{earley-sub-earley-list}$ :

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

**shows**  $\text{earley } \text{cfg inp} \subseteq \text{bins-items } (\text{earley-list } \text{cfg inp})$

## 6.5 Main Theorem

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

**theorem** *earley-recognized-list-iff-earley-recognized*:

**assumes** wf-cfg cfg is-sentence cfg inp nonempty-derives cfg

**shows** *earley-recognized-list* (earley-list cfg inp) cfg inp  $\longleftrightarrow$  *earley-recognized* (earley cfg inp) cfg inp

**corollary** *correctness-list*:

**assumes** wf-cfg cfg is-sentence cfg inp nonempty-derives cfg

**shows** *earley-recognized-list* (earley-list cfg inp) cfg inp  $\longleftrightarrow$  *derives* cfg [S cfg] inp

## 7 Earley Parser Implementation

### 7.1 Draft

### 7.2 Pointer lemmas

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**lemma** *sound-mono-ptrs-earley-list*:  
**assumes** *wf-cfg cfg nonempty-derives cfg*  
**shows** *sound-ptrs inp (earley-list cfg inp)  $\wedge$  mono-red-ptr (earley-list cfg inp)*

## 7.3 Trees and Forests

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

```

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

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

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

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

definition wf-yield-tree :: 'a sentential  $\Rightarrow$  'a item  $\Rightarrow$  'a tree  $\Rightarrow$  bool where
  wf-yield-tree inp x t  $\equiv \text{yield-tree } t = \text{slice } (\text{item-origin } x) (\text{item-end } x) \text{ inp}$ 

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

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

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

```

## 7.4 A single parse tree

```

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

```



```

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

```

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

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

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

**lemma** wellformed-tree-ptrs-pre:  
**assumes** (bs, inp, k, i)  $\in$  wellformed-tree-ptrs  
**assumes** e = bs!k!i pointer e = Pre pre

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

## 7.5 Parse trees

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

)

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

**partial-function** (*option*) *build-trees'* :: 'a bins ⇒ 'a sentential ⇒ nat ⇒ nat ⇒ nat set ⇒ 'a forest list *option* **where**

*build-trees'* bs *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 sentential ⇒ '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))

**fun** build-forest'-measure :: ('a bins  $\times$  'a sentential  $\times$  nat  $\times$  nat  $\times$  nat set)  $\Rightarrow$  nat **where**  
 build-forest'-measure (bs, inp, k, i, I) = foldl (+) 0 (map length (take (k+1) bs)) - card I

**definition** wellformed-forest-ptrs :: ('a bins  $\times$  'a sentential  $\times$  nat  $\times$  nat  $\times$  nat set) set **where**  
 wellformed-forest-ptrs = {  
   (bs, inp, k, i, I) | bs inp k i I.  
   sound-ptrs inp bs  $\wedge$   
   k < length bs  $\wedge$   
   i < length (bs!k)  $\wedge$   
   I  $\subseteq$  {0.. $\text{length (bs!k)}$ }  $\wedge$   
   i  $\in$  I  
 }

**lemma** wellformed-forest-ptrs-pre:  
**assumes** (bs, inp, k, i, I)  $\in$  wellformed-forest-ptrs  
**assumes** e = bs!k!i pointer e = Pre pre  
**shows** (bs, inp, (k-1), pre, {pre})  $\in$  wellformed-forest-ptrs

**lemma** wellformed-forest-ptrs-prered-pre:  
**assumes** (bs, inp, k, i, I)  $\in$  wellformed-forest-ptrs  
**assumes** e = bs!k!i pointer e = PreRed p ps  
**assumes** ps' = filter ( $\lambda(k', pre, red). \text{red} \notin I$ ) (p#ps)  
**assumes** gs = group-by ( $\lambda(k', pre, red). (k', pre)$ ) ( $\lambda(k', pre, red). \text{red}$ ) ps'  
**assumes** ((k', pre), reds)  $\in$  set gs  
**shows** (bs, inp, k', pre, {pre})  $\in$  wellformed-forest-ptrs

**lemma** wellformed-forest-ptrs-prered-red:  
**assumes** (bs, inp, k, i, I)  $\in$  wellformed-forest-ptrs  
**assumes** e = bs!k!i pointer e = PreRed p ps  
**assumes** ps' = filter ( $\lambda(k', pre, red). \text{red} \notin I$ ) (p#ps)  
**assumes** gs = group-by ( $\lambda(k', pre, red). (k', pre)$ ) ( $\lambda(k', pre, red). \text{red}$ ) ps'  
**assumes** ((k', pre), reds)  $\in$  set gs red  $\in$  set reds  
**shows** (bs, inp, k, red, I  $\cup$  {red})  $\in$  wellformed-forest-ptrs

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

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

**lemma** wf-forest-build-trees':  
**assumes** (bs, inp, k, i, I)  $\in$  wellformed-forest-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**  $wf\text{-}rule\text{-}root\text{-}yield\text{-}tree\text{-}build\text{-}trees$ :  
**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**  $wf\text{-}rule\text{-}tree\ cfg\ t \wedge root\text{-}tree\ t = \S\ cfg \wedge yield\text{-}tree\ t = inp$   
**corollary**  $wf\text{-}rule\text{-}root\text{-}yield\text{-}tree\text{-}build\text{-}trees\text{-}earley\text{-}list$ :  
**assumes**  $wf\text{-}cfg\ cfg\ nonempty\text{-}derives\ cfg$   
**assumes**  $build\text{-}trees\ cfg\ inp\ (earley\text{-}list\ cfg\ inp) = Some\ fs\ f \in set\ fs\ t \in set\ (trees\ f)$   
**shows**  $wf\text{-}rule\text{-}tree\ cfg\ t \wedge root\text{-}tree\ t = \S\ cfg \wedge yield\text{-}tree\ t = inp$   
**theorem**  $soundness\text{-}build\text{-}trees\text{-}earley\text{-}list$ :  
**assumes**  $wf\text{-}cfg\ cfg\ is\text{-}sentence\ cfg\ inp\ nonempty\text{-}derives\ cfg$   
**assumes**  $build\text{-}trees\ cfg\ inp\ (earley\text{-}list\ cfg\ inp) = Some\ fs\ f \in set\ fs\ t \in set\ (trees\ f)$   
**shows**  $derives\ cfg\ [\S\ cfg]\ inp$   
**theorem**  $termination\text{-}build\text{-}tree\text{-}earley\text{-}list$ :  
**assumes**  $wf\text{-}cfg\ cfg\ nonempty\text{-}derives\ cfg\ derives\ cfg\ [\S\ cfg]\ inp$   
**shows**  $\exists fs.\ build\text{-}trees\ cfg\ inp\ (earley\text{-}list\ cfg\ inp) = Some\ fs$

## 7.6 A word on completeness

## 8 Usage

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

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

**datatype**  $t = x \mid \text{plus}$

**datatype**  $n = S$

**datatype**  $s = \text{Terminal } t \mid \text{Nonterminal } n$

**definition**  $\text{nonterminals} :: s \text{ list}$  **where**  
 $\text{nonterminals} = [\text{Nonterminal } S]$

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

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

**definition**  $\text{start-symbol} :: s$  **where**  
 $\text{start-symbol} = \text{Nonterminal } S$

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

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

**lemma**  $\text{wf-cfg}$ :

**shows**  $\text{wf-cfg } \text{cfg}$

**lemma**  $\text{is-sentence-inp}$ :

**shows**  $\text{is-sentence } \text{cfg } \text{inp}$

**lemma**  $\text{nonempty-derives}$ :

**shows**  $\text{nonempty-derives } \text{cfg}$

**lemma**  $\text{correctness}$ :

**shows** *earley-recognized-list* (*earley-list* *cfg inp*) *cfg inp*  $\longleftrightarrow$  *derives* *cfg* [ $\mathfrak{S}$  *cfg*] *inp*

## **9 Conclusion**

### **9.1 Summary**

### **9.2 Future Work**



# List of Figures

3.1 Earley Inference Rules . . . . .	11
--------------------------------------	----

# List of Tables

3.1 Earley Items . . . . .	11
----------------------------	----