On the Practice of B-ing Earley

By

Daniel C. Zingaro, B.Sc.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

Master of Computer Science
Department of Computing and Software
McMaster University

MASTER OF COMPUTER SCIENCE (2007) McMaster University
(Department of Computing and Software) Hamilton, Ontario

TITLE: On the Practice of B-ing Earley

AUTHOR: Daniel C. Zingaro, B.Sc. (McMaster University)

SUPERVISOR: Dr. Emil Sekerinski

NUMBER OF PAGES: viii, 96

Abstract

Earley's parsing algorithm is an $O(n^3)$ algorithm for parsing according to *any* context-free grammar. Its theoretical importance stems from the fact that it was one of the first algorithms to achieve this time bound, but it has also seen success in compiler-compilers, theorem provers and natural language processing. It has an elegant structure, and its time complexity on restricted classes of grammars is often as good as specialized algorithms. Grammars with ϵ-productions, however, require special consideration, and have historically lead to inefficient and inelegant implementations.

In this thesis, we develop the algorithm from specification using the B-Method. Through refinement steps, we arrive at a list-processing formulation, in which the problems with ϵ-productions emerge and can be understood. The development highlights the essential properties of the algorithm, and has also lead to the discovery of an implementation optimization. We end by giving a concept-test of the algorithm as a literate Pascal program.

Acknowledgements

I am most indebted to my supervisor, Dr. Emil Sekerinski, not only for his help in these two years, but throughout my undergraduate degree as well. I appreciate his guidance and patience, the opportunity to be involved in course teaching, and the innumerable meetings from which I learned so much. I cannot imagine a more genuine, knowledgeable and enthusiastic supervisor — especially not one who would allow the title of this thesis to stand.

Thanks to Dr. Zucker for his entertaining and informative undergraduate lectures, and willingness to help with reference letters (which haven't negatively affected me, so they couldn't have been that bad). Thanks also to Dr. Carette for teaching me theory of programming languages;Dr. Janicki for introducing me to formal languages; Dr. Schneider for his wonderful B book and helpful email discussions; Dr. Ryan and Dr. Jones for introducing me to computer science; Dr. Maibaum and Dr. Kahl for their comments on this manuscript; and Dr. Day for broadening my education with psychology.

Thanks to my former colleagues, now friends: Ewa Romanowicz (for being optimistic and confident in my work, even when I wasn't), Pouya Larjani (for livening up the office), Scott West (for his help with various parts of the thesis, and for surviving three summers as my office neighbor) and Vera Pantelic (for her interest in my work and its associated challenges). Much thanks to my long-time best friend, Mark Castrodale, for the cheese parties, editing, journal searching, academic advice, and overall support throughout the process.

Finally, thanks to Mom, Dad, Steph, Joe and Spino, for providing a healthy and supportive environment throughout my education.

> Only he that has traveled the road knows where the holes are deep - Chinese proverb
>
> It is not the knowing that is difficult, but the doing - Chinese proverb

Contents

List of Figures

Chapter 1

Introduction

In 1968, Earley [4] developed a general context-free parsing algorithm. It has the property that it can determine, in $O(n^3)$ time, whether or not a sentence belonged to a given grammar, and produce the parse trees that realized the recognition. In terms of time bounds, this was nothing new, as previous work had already attained the $O(n^3)$ milestone. In fact, there are context-free parsers which run in $O(n^{2.55})$ time [9], so Earley's algorithm is not even the asymptotically fastest algorithm anymore. Additionally, linear-time parsers for large subsets of context-free grammars had already been developed and were widely used in compilers and compiler-compilers.

Despite this, there are two compelling reasons why Earley's algorithm is, in fact, of great theoretical and practical interest. The first is that no preprocessing of the context-free grammar is necessary. In previous $O(n^3)$ algorithms, conversion to Chomsky Normal Form (or some other normal form) is required prior to execution. With Earley's algorithm, this step is unnecessary — we can operate directly on the given grammar. Second, Earley's algorithm can be seen as a unification of the general $O(n^3)$ techniques and the restricted linear parsing techniques. Specifically, when a linear parsing algorithm is applicable to a given grammar, Earley's algorithm also runs in $O(n)$ time on that grammar. In a practical sense, we have the best of both worlds — general context-free parsing when required, and linear parsing when other algorithms would have given us linear parsing.

The bête noire of an otherwise elegant algorithm, grammars with ϵ-productions necessitated Earley to augment his implementations with additional, dynamically-maintained data structures. Other approaches were suggested, but all obscured the simple pattern that the algorithm exhibited. In 2002, Aycock and Horspool solved

the problem in a very unobtrusive way, preserving the structure that Earley had in mind.

While Earley provided a proof of correctness in his Ph.D thesis, it was somewhat informal and did not precisely correspond with an implementation — only an abstract description — of the algorithm. Our first contribution, then, is to employ invariants within the B-method to study the algorithm, which gives another argument for its correctness and clarifies Earley's original proof. Additionally, the invariants give rise to some insight into the problems caused by ϵ-productions. We therefore investigate the Aycock-Horspool algorithm and reason about it through invariants as well. Finally, we present a literate, Pascal implementation of this algorithm.

The rest of this thesis is organized as follows. We begin in Chapter 2 with relevant terminology, a systematic rundown of various parsing techniques for comparison purposes, and an overview of the B notation and weakest preconditions that will be used later. Previous work in proving correctness of parsing algorithms in general — and Earley's in particular — is given in Chapter 3. In Chapter 4 we develop the framework necessary to arrive at a specification of the algorithm in B. To move towards an implementation, Chapter 5 considers a list-processing refinement. Chapter 6 contains a literate development of our version of Earley's algorithm. In Chapter 7 we offer some conclusions and expound on future work.

Chapter 2

Formalities and Background

2.1 Parsing Terminology

Our presentation relies on (mostly) standard terminology surrounding parsing algorithms and context-free grammars. Figure 2.1 below will be used to outline our definitions.

The figure depicts a *context-free grammar* (CFG), in the format used throughout the paper. There is one *production* per line, except on the last line where multiple production right-hand-sides are separated by $\mid$. The single symbols on the left of the arrows are *nonterminals* and the symbols on the right are *terminals* or *nonterminals*. For example, T in the above grammar is a nonterminal, and a is a terminal. We use uppercase letters for nonterminals, lowercase letters for terminals, and Greek lowercase letters for arbitrary strings of terminals and nonterminals. A CFG G is a quadruple (T, N, P, S). We have that T is the finite set of terminal symbols, N is the finite set of nonterminal symbols, P is the finite set of productions, and S is a nonterminal designated as the start symbol. We assume that T and N are disjoint. We make the restriction that the start symbol, S' in the grammar above, is the left-

$$
\begin{aligned}
S' &\rightarrow E \\
E &\rightarrow T \\
E &\rightarrow E + T \\
T &\rightarrow F \\
T &\rightarrow T * F \\
F &\rightarrow a \mid b \mid c
\end{aligned}
$$

Figure 2.1: Sample context-free grammar.

hand-side of only one production in the grammar, a simplifying assumption made by Earley as well [4].

Definition 2.1. *We say that sequence χ is directly derivable from π, written $\pi \Rightarrow \chi$, if we can replace an occurrence of the left-hand side of some production P in π with the corresponding right-side of production P, and arrive at χ. That is, if $\pi = \mu\sigma\nu$, and $\chi = \mu\tau\nu$, $\sigma \rightarrow \tau$ is a production, and σ, τ, μ, ν are all sequences, then $\pi \Rightarrow \chi$.*

As an example, in the grammar above we have that $aFc \Rightarrow abc$.

Definition 2.2. *For χ to be derivable in zero or more steps from π, written $\pi \Rightarrow^* \chi$, we require sequences $\alpha_0, \alpha_1, \ldots, \alpha_k (k \geq 0)$, such that $\pi = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_k$.*

The string $a + b$ is derivable from S' in the example grammar above.

A production of the form $A \rightarrow \epsilon$ is called an ϵ-production; it signifies that the nonterminal A can derive the empty string.

A *sentential form* is a string which is derivable from the start (or root) symbol of the grammar. Since nonterminals are only present to properly construct a grammar, we are usually interested in sentential forms which are composed of only terminal symbols. These are called *sentences*, and the *language* of a CFG is the set of its sentences.

Parsers and recognizers take, as input, a CFG and a string of terminals. Recognizers determine if the string is a sentence of the CFG, and so are just boolean-valued functions. Parsers additionally return information on why given strings are sentences, typically via *parse trees*. A parse tree is a tree representation of the steps in a derivation, independent of the order that the steps were used. If an (interior) node is labelled with a nonterminal A, its children, concatenated in order from left to right, are one right-hand-side of a production of A. The leaves of the tree correspond to the sentence. We focus on parsers as recognizers — in other words, the recognition of a sentence, not its derivation. We represent an input sentence of length n as $x_1 x_2 \ldots x_n$.

There are various normal forms for CFGs. One of the most common — and the only one of importance here — is Chomsky Normal Form (CNF) [10]. It requires that the right-hand-sides of productions be in one of two formats: a solitary terminal, or a sequence of exactly two nonterminals.

2.2 Comparison of Parsing Algorithms

To gain an appreciation for the nature and operation of Earley's algorithm, it is instructive to compare it to a number of other parsing algorithms that have been proposed. Earley's algorithm can be used as a general context-free parser, and can also efficiently parse according to grammars that specialized (i.e. restrictive) parsers can deal with in linear time. We sample from both of these types of parsers, not only to exhibit the strengths of Earley's algorithm, but also to expose its comparatively simple structure — another reason to preferentially study it in depth.

2.2.1 Unger Parsers

The class of parsers attributed to Unger [22] work top-down and, in their simplest form, can deal only with grammars containing no ϵ-productions or loops. (A loop is a production of the form $A \rightarrow A$, or a sequence of productions $A \rightarrow B, B \rightarrow C, \ldots, G \rightarrow H, H \rightarrow A$.) Earley's algorithm, in the form we work with in Chapter 5, has no problems with ϵ-productions, and Earley parsers never have issues with loops. Unger parsers require extra, dynamic checks to avoid infinite loops when processing these CFGs, as we will see.

The intuition behind the algorithm is as follows. Assume that we are given a CFG and a sentence for which we want to determine derivability from the start symbol. This means that one of the right-hand-sides of the start symbol must derive the entire sentence. If a right-hand-side is $\alpha\beta\gamma\ldots$ and the sentence is $abc\ldots$, then α must derive some first part of the input, β must derive some subsequent part of the input, and so on. If this can be done, then we know that this particular right-hand-side can derive the entire sentence, and so too can the start symbol. The problem, of course, is that we do not know how to *partition* the input sentence among the symbols $\alpha\beta\gamma\ldots$ — this is where the main observation (and the main downfall) of the algorithm lies.

The observation is simple: since we do not know how to partition the sentence, we must try all possible partitions. If we are assuming that there are no ϵ-productions, this amounts to dividing $abc\ldots$ in such a way that $\alpha, \beta, \gamma, \ldots$ are individually "responsible" for deriving non-empty consecutive parts of the input. Assuming there are p ways of accomplishing this partition, we have p new problems to solve, all of which may require further partitioning. We can only stop this partitioning when we reach the level of terminals; if we are trying to ascertain whether one terminal derives another, we can answer this directly, without partitioning anything. This approach

E	$+$	T
a	$+$	$b+c$
a	$+b$	$+c$
a	$+b+$	c
$a+$	$b+$	c
$a+$	b	$+c$
$a+b$	$+$	c

Figure 2.2: Partitions of $a + b + c$ on expression grammar of Figure 2.1.

is a classic realization of the depth-first search technique. We start with a problem to solve, break it into subproblems, and recursively solve those subproblems in turn. The recursion ends when the subproblems are trivially solved.

Let us reconsider Figure 2.1 to see how this algorithm might proceed on input $a + b + c$. The root of the CFG is S', so we are looking to partition $a + b + c$ among the symbols of a right-hand-side of S'. At this level, we have just one choice: E must derive $a + b + c$, and so this is our new problem to solve. There are two alternatives for E, so we must consider both and partition $a + b + c$ in all possible ways among the right-hand-sides of these partitions. We begin with the first one, $E \rightarrow E + T$. The partitions of the input are below.

We thus have six subproblems to consider; solving one of them solves the original problem. The first one is asking whether e can derive a, $+$ can derive $+$, and e can derive $b + c$. In this particular case we have three more subproblems to solve, and all seem promising. There are some subproblems, though, that look especially dubius. The most naive Unger parser would indeed attempt to enumerate all partitions in Figure 2.2, even those which are fated from the start, such as the second partition. It is effectively asking if the terminal $+$ can derive $+b$, which it most certainly cannot.

What we do not like in a depth-first search, of course, is when a so-called "subproblem" is the same as a problem we are trying to solve at a higher level. This prevents the recursion from terminating, because we repeatedly try to solve the same thing as we proceed deeper and deeper into the problem space. Unfortunately, this is exactly what can happen when loops or ϵ-productions are present in a CFG. For example, consider adding the production $E \rightarrow E$ to Figure 2.1. We will eventually try to match the input sentence to this rule, and ask whether E can derive it. This, sadly, will require knowing whether E derives the input sentence, and we have the genesis of infinite recursion.

It is conceptually not difficult to solve this problem. As evidenced in [13], all we have to do is check the current stack of "goals" before adding a new one. If it already exists, we do not add it again; if it is a new goal, we proceed as always. This is unfortunate, however, as we are already beginning to obscure the core of the algorithm. We can no longer use straightforward recursion as an implementation method; indeed, Kwiatkowski manually maintains stacks, which balloons the resultant code.

We now have a general context-free parser, and so it is interesting to compare its behavior with Earley parsers. Ignoring the implementation difficulties incurred by the stack searching, there is still one major drawback of the algorithm: it has an exponential worst-case running time in its current form. What can be done about this?

The insight to bring the algorithm from exponential to polynomial is similar to the method by which we brought the algorithm out of the depths of infinite recursion [10]. Every time we successfully derive a partition of the input from the right-hand-side of a nonterminal, we record this in a *known parse table.* Later, if we are presented with the question of whether the same nonterminal derives the same part of the input, we know it does — and we know how, by looking it up from the table. This results in an $O(n^{k+1})$ algorithm, where k is the maximum number of nonterminals on any right-hand-side of a production.

Let us take stock of what Unger parsers give us. It has been found necessary to obscure the once-elegant parser twice — once to solve infinite recursion problems, the other to make it polynomial. Still, to have an $O(n^3)$ Unger parser, it is evident from the time complexity above that k must be 2. We can do this via a conversion to Chomsky Normal Form, but this may not always be practical as it hides the original structure and meaning of the grammar. If we are willing to compromise on this, then we might wonder if there are more direct algorithms that attain the same time bounds; we survey one next.

2.2.2 CYK Parsers

The parsing technique attributed to Cocke, Younger and Kasami (CYK) [10] once again has a very simple core idea, if we are willing to restrict it to CNF grammars. The observation is that we can determine whether a given nonterminal can derive a given segment of the input sentence by looking at the (non-empty!) substrings that

its right-hand-side can recognize. Consider a production $A \rightarrow BC$, and assume we want to know if $A \Rightarrow^* a_1 a_2 \ldots a_n$. This boils down to the task of finding a suitable k such that $B \Rightarrow^* a_1 a_2 \ldots a_k$ and $C \Rightarrow^* a_{k+1} a_{k+2} \ldots a_n$. We do not know this k, so we try all possibilities. Crucial here is that there are no ϵ-productions, so that for every k, all necessary "lookups" about B and C focus on shorter substrings. At some point, we will come to the "base cases" of the algorithm: productions of the form $A \rightarrow a$, for which we can trivially ascribe derivability.

The cannonical CYK parser is a bottom-up, dynamic-programming realization of the above idea. An $n \times n$ matrix is the main data structure used to recognize sentence $a_1 a_2 \ldots a_n$, where an element N in row i and column j means it has been found that $N \Rightarrow^* a_{i+1} a_{i+2} \ldots a_j$. The matrix is filled in shortest-length-first, so that for all productions $N \rightarrow c$, we have that N is a member of element $(i-1, i)$ if $a_i = c$. After this, we proceed with the above recurrence, determining derivability of longer subsequences from the information about shorter subsequences that we have stored in the matrix.

What goes wrong with arbitrary CFGs? Most of the above reasoning still works, if we extend it appropriately. For example, if we have a production $A \rightarrow BCD$, we now split the substring into three parts instead of two, and determine if B derives the first, C derives the second, and D derives the third. The bigger problem, reminiscent of what we found with Unger parsers, is uncovered when we introduce ϵ-productions or loops.

Consider a grammar with loop $A \rightarrow A$, and assume that we are just about to start deriving information about substrings of length l. We could previously assume that all necessary information was already computed, since we were only concerned with building up derivability of longer substrings from shorter ones. Now, for the first time, we are asking whether A derives the substring of length l, and of course this information is not available, as we have not processed any of this iteration yet. (Incidentally, this is precisely the problem we run into with Earley's algorithm, and is the sole reason for the Aycock-Horspool reformulation.) Implementation techniques to mitigate the problem rely, in some form or another, on re-computing the information about length l over and over until "nothing changes". In other words, we run through the grammar productions once, gleaning some information about substrings of length l. We then run over the productions again, and we may now have information to add when loop productions require information about substrings of length l. When nothing is added in an entire sweep of the productions, we have reached the

fixed-point and can move on to length $l + 1$.

2.2.3 $LL(k)$ and $LR(k)$ Parsers

There are efficient algorithms which can perform recognition or parsing on "reasonably-sized" subsets of the CFGs. We briefly look at two methods — $LL(k)$ and $LR(k)$ — in this section.

For $LL(k)$ parsing, we begin with the general idea of taking the start symbol, and choosing a right-hand-side to effect the matching. If the chosen alternative fails, then we backtrack and try the next alternative, until we find the one that succeeds, or until all alternatives have been exhausted. In the worst case, the backtracking can result in recognition time exponential in the length of the sentence. Determining whether a right-hand-side recognizes a portion of the input is a recursive problem. For example, if the grammar has a rule $S \rightarrow ABC$, then we will quickly want to know whether ABC can recognize the input sentence. This involves trying, in turn, right-hand-sides of A to see which part of the input they can recognize, then trying alternatives of B and C to try to recognize the rest. This is the essence of top-down backtracking parsers, of which $LL(k)$ parsers are a subset: we continue to expand the first nonterminal (starting from the left), and if it fails to parse the next part of the sentence, we backtrack until a point where we do succeed. These parsers, in general, do not deal with all CFGs. For example, consider the grammar that consists of rules $S \rightarrow Sb \mid a$. We will expand S to Sb, then expand this new S again to Sb, and so on, expanding S *ad nauseam*, and never reaching terminal symbols to match. These appear to be dark times: we have an exponential running time, and now we cannot even use it on an arbitrary CFG.

$LL(k)$ parsers further cull from the set of CFGs that we can parse, because they entirely remove backtracking from the general top-down parser. However, this modification makes the algorithm run in linear time on the size of the input sentence, so this seems to be a fair compromise. Backtracking is used in general so that we are not committed to choices that we make: if we make a bad choice and get stuck in the recognition, we can just back out and try a different route. If we know that we will make correct choices, backtracking becomes unnecessary, and we have the class of grammars that $LL(k)$ parsers can handle.

How do we know a choice will be correct? Consider the grammar fragment $Q \rightarrow aB \mid aC$, and assume that we are looking only at the next terminal in the sentence.

$$
\begin{aligned}
S' &\rightarrow E \\
E &\rightarrow T \\
E &\rightarrow T + E \\
T &\rightarrow F \\
T &\rightarrow F * T \\
F &\rightarrow a \mid b \mid c
\end{aligned}
$$

Figure 2.3: Example of an $LL(1)$ grammar.

We have no way of determining which of these two alternatives to try, so we cannot be certain the choice is correct. On the other hand, if the two alternatives of Q began with different terminals, then at most one has a possibility of succeeding (since the other cannot match the next input symbol). This is one property that must hold of our grammars if we hope to parse them via $LL(k)$ techniques: that, of all right-hand-sides of a nonterminal, no two can generate strings with a common prefix of length $\geq k$.

We can often transform grammars that do not obey this property into grammars that do. The grammar in Figure 2.3 accepts the same language as that given in figure 2.1, except it is $LL(1)$-parseable. It is this type of grammar massaging that is unnecessary if an Earley parser was being employed instead of an $LL(k)$ parser: at the cost of some efficiency, we have no concerns with whether the grammar is $LL(k)$ or not.

The $LR(k)$ parsers also run in linear time, but can deal with a larger subset of grammars than the $LL(k)$ parsers. Less powerful predecessors of $LR(k)$ parsers, known as precedence parsers [6] exist, but in their present form, $LR(k)$ parsers are in widespread use and can cope with most programming language constructs. The strategy this time is to begin with the input sentence, and determine the nonterminals which can derive portions of it (see [3], Chapter 3). Conceiving of the process as building a tree from the leaves to the root, each step in the process replaces some sequence of terminals on the topmost layer with a nonterminal that can derive them in a right-most derivation of the sentence. If this results in the situation where the topmost level contains just the start nonterminal and the entire input has been processed, we have a successful parse. If no further matching can be done, and we have not reduced the sentence to the start nonterminal, the parse has failed. The critical decision to make throughout the execution of the algorithm is whether to consume the next input symbol from the scanner (a *shift* step) or replace terminals with a

$$S \rightarrow iEtS$$
$$S \rightarrow iEtSeS$$

Figure 2.4: Example of Non-LR grammar.

nonterminal (a *reduce* step). Tables, generated by parser-generators, are typically used to drive this decision process, since these parsers are prohibitive to write by hand. A grammar which cannot be parsed via an LR parser will have one or more conflicts in these tables. For example, consider Figure 2.4, which gives a natural, though non-LR grammar, for "if" and "if-else" statements. The input symbol i is used to represent an "if", e an "else" and t a "then"; the nonterminal E representing expressions is not elaborated. At any point in the parse, if the next input symbol is e, it is not known whether we have just completed an if-statement (so this e is part of some enclosing if-statement), or we are in the process of parsing an if-then-else statement. The first of these options is realized through a reduce step; the second through a shift step. The parser does not know which is correct: we have a *shift-reduce conflict*, and the grammar is non-LR.

2.3 Earley's Recognizer

2.3.1 The Algorithm

We present Earley's recognizer in the spirit of its original presentation [5]. Like CYK, it is iterative; unlike CYK, it recognizes increasingly longer prefixes of the input sentence, not arbitrary portions of it.

Assume the input sentence is of length n and consists of terminals $x_1 x_2 \ldots x_n$. The data structure built this time, called s, is a length $n+1$ sequence of *state sets* — one state set per input symbol, and one initial set. Inhabiting these state sets are states, which record the history of what the recognizer has done so far. States are triples[1]: the first component is a grammar production, the second is a pointer to somewhere in the right-hand-side of this production and the third is an integer indicating when we began recognizing this particular production. Earley uses dot-notation to represent items, so that they look more like pairs when written. For example, an item that

[1]States are quadruples in Earley's formulation, but since we drop the fourth "lookahead" component later anyway, we opt not to obscure the current description.

could be generated from Figure 2.1 is

$$[E \rightarrow E \bullet + T, 1]$$

where the dot indicates "where we are" in the recognition of the production.

Assuming that the root production of the grammar is $S' \rightarrow S$, we begin with the state

$$[S' \rightarrow \bullet S, 0]$$

in $s(0)$. We then continue applying three operations to the items in $s(0)$ until nothing further can be done. These three operations are mutually exclusive on a given state, and are as follows [1]:

SCANNER. If $[A \rightarrow \cdots \bullet a \cdots, j]$ is in $s(i)$ and $a = x_{i+1}$, add $[A \rightarrow \cdots a \bullet \cdots, j]$ to $s(i+1)$.

PREDICTOR. If $[A \rightarrow \cdots \bullet B \cdots, j]$ is in $s(i)$, add $[B \rightarrow \bullet\alpha, i]$ to $s(i)$ for all productions $B \rightarrow \alpha$.

COMPLETER. If $[A \rightarrow \cdots \bullet, j]$ is in $s(i)$, add $[B \rightarrow \cdots A \bullet \cdots, k]$ to $s(i)$ for all items $[B \rightarrow \cdots \bullet A \cdots, k]$ in $s(j)$.

Note how the *scanner* adds items to $s(i+1)$. Once we have reached the fixed-point of these three operations on $s(i)$, we begin doing the same on $s(i+1)$. If the scanner adds nothing to $s(i+1)$, and we are not yet in state set n, then recognition has failed. The scanner can be conceptualized as adding all states which have recognized one more symbol from the input, by moving the dot one position to the right. If there is no way to recognize any more of the sentence, the scanner can do nothing. The predictor is responsible for expanding all nonterminals which we are interested in recognizing, by enumerating their productions and adding them to the current state set. Finally, the completer's job is analogous to the scanner's, but on the nonterminal level rather than the terminal level. That is, if we have recognized the next nonterminal in a right-hand-side of a production, the completer notes this by moving the dot over that nonterminal. If the item

$$[S' \rightarrow S \bullet, 0]$$

is added to $s(n)$ after $n+1$ iterations, then recognition was successful, otherwise recognition has failed. For an example of constructing state sets, consider Figure 2.5,

$s(0)$

$(a1)$	$S' \rightarrow \bullet E$	,0	Initialisation
$(a2)$	$E \rightarrow \bullet T$	,0	Predictor (a1)
$(a3)$	$E \rightarrow \bullet E + T$	,0	Predictor (a1)
$(a4)$	$T \rightarrow \bullet F$	,0	Predictor (a3)
$(a5)$	$T \rightarrow \bullet T * F$	,0	Predictor (a3)
$(a6)$	$F \rightarrow \bullet a$	,0	Predictor (a4)
$(a7)$	$F \rightarrow \bullet b$	,0	Predictor (a4)
$(a8)$	$F \rightarrow \bullet c$	,0	Predictor (a4)

$s(1)$

$(b1)$	$F \rightarrow a\bullet$	,0	Scanner (a6)
$(b2)$	$T \rightarrow F\bullet$	,0	Completer (b1)
$(b3)$	$E \rightarrow T\bullet$	,0	Completer (b2)
$(b4)$	$T \rightarrow T \bullet *F$	,0	Completer (b2)
$(b5)$	$E \rightarrow T\bullet$	,0	Completer (b3)
$(b6)$	$E \rightarrow E \bullet +T$	,0	Completer (b3)
$(b7)$	$S' \rightarrow E\bullet$	,0	Completer (b5)

Figure 2.5: First two Earley sets for the grammar of Figure 2.1 using sentence a.

where we produce $s(0)$ and $s(1)$ for the venerable Figure 2.1, operating on the (accepted) sentence a. We have named the items in the state sets; the second column indicates the operation and item name used to add the item in its row.

One of our gripes about the CYK algorithm was that it requires redoing an iteration until nothing new was added to the matrix. Implementations of Earley's algorithm use lists instead of sets, and process the lists in order, adding new items to the end only if they were not present in the list already. We have "executed" the algorithm in this way in Figure 2.5, and the result (modulo the order that the items were added) is the same as it would have been had we thought of the state lists as state sets. This does not work when ϵ-productions are present; we elegantly deal with this snag in the reformulation given in Chapter 5.

2.3.2 Practical Uses

Earley's algorithm has found application in several areas of language processing. It is used to drive Accent, a compiler-compiler similar to yacc [17]. This allows natural grammars, not supported by yacc or other tools restricted to subsets of CFGs, to be used in language descriptions. For example, portions of Java are not inherently

$LR(k)$ for any k (see [8], Chapter 19). The grammar had to be massaged into such a format for use with classic compiler tools, essentially forcing (what should be) parse tree decisions into the context-sensitive portions of the analysis. Earley's algorithm has also been used in natural language processing [20]. Stolcke found that it was more efficient in answering typical questions about language grammars than the often-used CYK parsers, and could often produce information about a sentence "on-line". One question that exhibits these benefits is: given a string, what is the probability that it is the prefix of any sentence generated by the grammar?[2] Once a prefix has been seen, its probability is immediately available, as a direct result of how the algorithm processes sentences.

2.4 The B-Method

2.4.1 Overview

B is a formal method for specifying, refining and eventually implementing software [16]. These three activities are all based around the *abstract machine*, which is a component of a development encapsulating state, and including operations acting on that state. Abstract machines make use of *abstract machine notation (AMN)*, superficially resembling imperative programming languages. There are two important differences, however, which contribute to B's expressiveness, and allow a greater abstraction from implementation detail. First, we are not restricted to simple data types like arrays and records. While these are supported, mathematical objects — such as sets, sequences, relations, and functions — are both available and more powerful. Second, B provides a rich set of nondeterministic statements which allow decisions, which should not be made at specification time, to be deferred to a later step. We use the first of these features to give succinct (and not overspecified) versions of Earley later, and is central to our decision to use the B-Method in the first place.

There are two types of related refinements supported by B: data refinement and algorithmic refinement. Both involve creating a new machine based on some specification machine, and making it "more concrete". In data refinement, for example, we may decide to use an array as a representation for a set. Algorithmic refinement, on the other hand, yields algorithms which are closer to implementable code, often

[2]This relies on an extension to CFGs, where right-hand-sides of productions are given probabilities, indicating how likely they are to be used in parsing their associated nonterminals.

facilitated by new representations of data from a data refinement step (see [18], Chapter 1). That is, if we now use an array as a set, we have to reinterpret operations like union and intersection, to manipulate array elements, instead of using abstract set operations.

2.4.2 Machine Clauses

To illustrate the B machine components which we shall use, and give the flavor of a simple machine description, we introduce an example in Listing 2.1. The machine is meant to track the permissible moves in a water pouring game, in which we have just a five-litre jar and a three-litre jar, and wish to measure a given number of litres of water (usually four).

Listing 2.1: Water Pouring Game

MACHINE *water*

DEFINITIONS *combine* == *three* + *five*

CONSTANTS *target*

PROPERTIES *target* $\in \mathbb{N}_1$

VARIABLES *three*, *five*

INVARIANT
three $\in 0..3 \wedge$ *five* $\in 0..5$

INITIALISATION *five*, *three* :=0, 0

OPERATIONS

fill3 = *three* := 3;
fill5 = *five* := 5;

empty3 = **PRE** *three* > 0 **THEN** *three* :=0 **END**;
empty5 = **PRE** *five* > 0 **THEN** *five* :=0 **END**;

pourToThree =
  PRE *three* $\leq 3 \wedge$ *five* ≥ 0 **THEN**
    three, *five* := $\min(\{3, \mathit{combine}\}), \max(\{0, \mathit{five} - (3 - \mathit{three})\})$
END;

pourToFive =
  PRE *five* $\leq 5 \wedge$ *three* ≥ 0 **THEN**
    five, *three* := $\min(\{5, \mathit{combine}\}), \max(\{0, \mathit{three} - (5 - \mathit{five})\})$
  END;

ans $\longleftarrow$ *containsTarget* = *ans* :=bool(*five* = *target* $\vee$ *three* = *target*)
END

The **MACHINE** clause gives the machine a name; analogous **REFINEMENT** clauses are used to do the same for refinements. Typically, names of refinements are the same as those of the machines they refine, appended with *R*.

The **DEFINITIONS** clause assigns replacement text to identifier or function names, much as the C preprocessor does[3]. They crop up frequently in B developments, largely because it is prohibited for an operation of a machine to call other operations of the same machine. Definitions can also capture expressions that are used more than once, economizing on code length. In the present machine, there is just one (mostly contrived) definition, giving a name to the sum of two variables.

The **CONSTANTS** clause allows the names for global constants to be specified. The **PROPERTIES** clause is responsible for giving the type of such constants. The current machine has one constant, *target*, meant to represent the number of litres that a jar must contain to win the game. The machine must "work" regardless of the value chosen for the constants; we see this when looking at proof obligations.

The **VARIABLES** clause lists the variables of the machine. Here we have two, which will keep track of how many litres of water are present in the jars. Note, again, that no typing information is given in this clause, but is instead deferred, this time to the **INVARIANT** clause. The invariant usually does more than just type variables, although at a minimum it must do this. It also usually gives constraints on how the variables relate to one another, or asserts some property of a variable which cannot

[3]This is true even to the point that B definitions require extra parenthesization so that veiled operator precedence problems do not present themselves.

be captured simply by a type ascription. The present invariant asserts that the three-litre jar can have between 0 and 3 litres of water in it, and the five-litre jar can have between 0 and 5 litres.

The **INITIALISATION** clause is effectively executed on machine startup, and must give values to variables to satisfy the machine invariant (which must always be true throughout operation). Our initialisation uses an assignment statement to empty the jars. The **OPERATIONS** clause contains what can be thought of as methods or procedures, whose bodies modify the machine state. Some operations may be executed only in certain circumstances: for these, **PRE** statements indicate the preconditions that must be true for the result of the operation to be well-defined. When a precondition is met, the body of the operation is required to re-establish the machine invariant. Preconditions can also be used to impose logical constraints on when operations should be called; for example, *empty3* can only be called when there is water in the 3-litre jar.

The *pourToThree* and *pourToFive* operations contain a *multiple assignment*, which simultaneously sets the two variables on the left of the := to the two expressions on the right, respectively. We also see the use of min and max, which operate on sets of integers — in the present case, to ensure that the right amount of water is poured (literally preventing overflow). These simple features are the first that are not typical of imperative languages. The final operation is a *query operation*, so-named because it returns a value (like a typical function). The bool notation causes a boolean truth value to be given as output. Note how *ans* can be used as a variable in an assignment statement: its final value is the one which the function returns.

2.4.3 Structuring Mechanisms

B includes several structuring mechanisms which allow multiple machines to interact within a single development.

The **INCLUDES** clause allows one machine m to control machine n. This means that m can access all aspects of n (including the values of variables), and can relate its own state to the state of n. It can call operations of n, and is assured that no other machine in the development can change the state of n. A restricted form of this relationship is the **IMPORTS** clause, which can be used by an implementation machine to control abstract machines. To facilitate data hiding, the values of the imported machine's variables cannot be accessed — they can only be obtained via

$=$	equality
$\wedge$	conjunction
or	disjunction
$\neg$	negation
$\Rightarrow$	implication
$\forall$	universal quantifier
$\exists$	existential quantifier
$\in$	set membership
$\notin$	set exclusion
$\cup$	set union
$\cap$	set intersection
$\mathbb{P}$	power set
$\subseteq$	subset
card	set cardinality
closure	transitive reflexive closure
closure1	transitive nonreflexive closure
$\rightarrow$	total function
$\rightarrowtail$	total injection
$\mapsto$	maps to
$\lhd\!\!+$	relational override
$\otimes$	direct product
$\leftrightarrow$	relation
dom	domain of relation
ran	range of relation

Figure 2.6: Set-related and logic symbols.

query operations.

The **SEES** clause allows a machine m read-only access to machine n. This means that m can access the constants and state of n, and can execute query operations of n. It cannot relate its variables to those of n in its invariant, because another machine in the development could change the state of n. Similarly, m cannot call any non-query operation of n, because the machine which includes n is the only one which can modify its state.

2.4.4 Set Theory and Logic

Much of the B notation is based on set-theoretic operations and logical connectives. We present the frequently-encountered symbols in Figure 2.6, and assume a general familiarity with the majority of the concepts.

first (s)	first element of sequence s
tail (s)	all but the first element of sequence s
last (s)	last element of sequence s
front (s)	all but the last element of sequence s
$s \leftarrow e$	sequence s appended with element e
$s \frown t$	concatenation sequences s and t
size (s)	number of elements in sequence s
$s \uparrow n$	first n elements of sequence s
$s \downarrow n$	sequence s with first n elements removed

Figure 2.7: Operations on sequences.

The "maps to" notation $a \mapsto b$ is used to represent the pair $(a,\ b)$. The direct product $(\otimes)$ of relations $a \in b \leftrightarrow c$ and $d \in b \leftrightarrow f$ is a relation with elements $(g,\ (h,\ i))$ where $g \mapsto h \in a$ and $g \mapsto i \in d$.

2.4.5 Sequences

It is often convenient to impose an order on a set of elements; B provides the seq (sequence) and iseq (injective sequence) types for this purpose. A sequence is a total function from $1 \ldots N$ to an element type, where N is a positive natural number; injective sequences additionally require that no element is repeated. The range of a sequence is therefore the set consisting of its elements. Furthermore, this representation lets us extract elements of a sequence by using function notation: if s is a sequence, we can extract element n $(1 \leq n \leq N)$ with $s(n)$. The empty sequence is written $[]$, and we can explicitly list the elements of a sequence as $[e1,\ e2,\ \ldots,\ en]$, where $e1,\ e2,\ \ldots,\ en$ are elements of the sequence type. The operations we use on sequences are summarized in Figure 2.7.

2.4.6 Correctness Criteria

To prove correctness of a machine requires that various proof obligations be discharged; for example, we said previously that the machine invariant must always be maintained. The core idea of the proofs is to use Dijkstra's weakest preconditions to reason logically about the state space of the machine. If S is an AMN statement and P is a predicate characterizing a postcondition, we use the syntax $[S]P$ to denote the weakest precondition of S to establish P. All AMN statements have their own rules for calculating weakest preconditions; the ones we use are given in Figure 2.8. The ;

WPA	$[x := E]\ P$	$= P[E\ /\ x]$
WPM	$[x,\ y := E,\ F]P$	$= P[E,\ F\ /\ x,\ y]$
WPI	$[\textbf{IF}\ E\ \textbf{THEN}\ S\ \textbf{END}]P$	$= (E \Rightarrow [S]P) \wedge (\neg E \Rightarrow P)$
WPB	$[\textbf{BEGIN}\ S\ \textbf{END}]P$	$= [S]P$
WPS	$[S;\ T]\ P$	$= [S]\ ([T]\ P)$
WPP	$[\textbf{PRE}\ Q\ \textbf{THEN}\ S\ \textbf{END}]P$	$= Q \wedge [S]\ P$

Figure 2.8: AMN weakest preconditions.

MACHINE N
CONSTANTS k
PROPERTIES B
VARIABLES v
INVARIANT I
INITIALISATION T
OPERATIONS
$y \longleftarrow op(x) =$
  PRE P **THEN** S
  END ;
...
END

Figure 2.9: Machine template.

operator is used for sequencing B statements.

The syntax $P[E/x]$ is meant to represent the result of substituting all free occurrences of variable x in P with expression E.

We have that $[S](P \wedge Q) = [S]P \wedge [S]Q$. That is, if we want to find the weakest precondition of a conjunction, we can find the weakest preconditions for the pieces of the conjunction separately.

Equipped with these rules, we can understand the proof obligations that must be discharged to prove consistency of a machine. We consider the machine template in Figure 2.9, whose proof obligations we give in Figure 2.10 (adapted from [16]). The template shows only one operation; however, the last obligation in Figure 2.10 must be discharged for all operations.

The first two proof obligations deal with the static specification of the machine. The latter two prove the consistency of the machine as it executes, and are concerned with preservation of the machine invariant. We must first prove that the initialisation clause establishes the machine invariant. Next, under the assumption that the

(1) $(\exists k)\ .\ B$ — there are valid instantiations of the constants

(2) $B \Rightarrow (\exists v)\ .\ I$ — invariant is not falsehood

(3) $B \Rightarrow [T]I$ — initialisation establishes invariant

(4) $(B \wedge I \wedge P) \Rightarrow [S]I$ — operation preserves invariant

Figure 2.10: Proof obligations for template of Figure 2.9.

invariant and a given operation's precondition hold, we must show that execution of that operation leaves the invariant intact. The initialisation of the water example establishes the invariant, since 0 is between 0 and 3, and 0 is between 0 and 5. The operations all maintain the invariant; the last one is a query operation, which by definition can never invalidate anything.

2.4.7 Proof Obligations for Refinements

A different set of proof obligations is required to prove that one machine is a refinement of another. Anything that the refined machine does must be a possibility for what the abstract machine could have done in the same situation. In terms of initialisation, this means that any initial state of the refinement must be a possible initial state for the abstract machine. The refinement machine has its own invariant, which includes a *linking invariant.* The linking invariant relates the state spaces of the two machines. For operations, we must have that executing the refined version and then the abstract version results in a state where the invariants (including the linking one) are still true. For query operations, the required condition is that any result returned by the refined machine must be a possible result of the operation in the machine being refined. This makes refinements and their abstract counterparts indistinguishable to the end user. Figure 2.11 gives a template for a machine assumed to refine a machine of the form in Figure 2.9. The proof obligations for the refinement are summarized in Figure 2.12; obligations (3) and (4) must be discharged for all operations. In proofs, we drop the part of the proof obligation linking the results of an operation if an operation does not return a value. Also, the two negations in proof obligations (2) and (3) cancel if the abstract operation is deterministic [16], so we elide this as well.

MACHINE *NR*
REFINES *N*
CONSTANTS *k2*
PROPERTIES *B2*
VARIABLES *v2*
INVARIANT *J*
INITIALISATION *T1*
OPERATIONS
$y \longleftarrow op(x) =$
  PRE *P1* **THEN** *S1*
  END ;
...
END

Figure 2.11: Template for refining machine of Figure 2.9.

(1) $(\exists k1,\ k2)\ .\ B1 \wedge B2$ — there are constants satisfying the properties

(2) $B1 \wedge B2 \Rightarrow [T1]\ \neg([T])\ \neg(J)$ — initialisation is a proper refinement

(3) $B1 \wedge B2 \wedge I \wedge J \wedge P \Rightarrow [S1[y'/y]]\ \neg[S]\ \neg(J \wedge y' = y)$ — operation is a valid refinement

(4) $B1 \wedge B2 \wedge I \wedge J \wedge P \Rightarrow P1$ — refined precondition is not strengthened

Figure 2.12: Proof obligations for refinement of Figure 2.11.

(1) $(I \wedge E \Rightarrow [S]I)$ — body preserves invariant

(2) $(I \wedge \neg(E) \Rightarrow P)$ — invariant and negation of guard imply postcondition

(3) $(I \wedge E \Rightarrow v \in \mathbb{N})$ — variant is a natural number

(4) $(I \wedge E \wedge v = g \Rightarrow [S](v < g))$ — loop decreases variant

(5) I — invariant is true before loop execution

Figure 2.13: Proof obligations for a while loop.

2.4.8 Correct Loops

Since it is not necessarily statically known how many times a loop will execute, there is no rule for calculating the precise weakest precondition of a loop in B. Instead, various proof obligations must be discharged that, together, verify that the loop is correct and that it yields the desired postcondition when called starting in a state satisfying a suitable precondition. The template for the while loop is **WHILE** E **DO** S **INVARIANT** I **VARIANT** v **END**. The invariant here is a *loop invariant*, which has three proof obligations associated with it. First, it must be true when the loop begins execution. Second, under the assumption that the invariant is true and the loop guard E is also true, the loop body S must maintain the invariant. Third, together with the negation of E, the invariant must imply the desired postcondition. The loop template also includes a variant, which is used to prove termination. It incurs two loop obligations: first, that it is decreased on every iteration and second, that all valid states of the loop are associated with nonnegative values for the variant. These proof obligations are summarized in Figure 2.13.

2.4.9 Proof Method

We follow the proof method of Schneider [16], in which we begin with a statement involving weakest preconditions, eliminate them, and then use logical reasoning to establish truth of the original statement. Justification of steps in a chain of equivalences is given via comments in curly braces; we refer to rules of weakest preconditions via the mnemonics in Figure 2.8. When the proof requires more justification than can be given in such a way, we also use supporting lemmas.

The notation $x := \text{bool}(b)$ is a shorthand for

IF b **THEN** $x := TRUE$ **ELSE** $x := FALSE$ **END**. We do not make this expansion in proofs, since this unnecessarily complicates the structure. We therefore have references to bool in post-conditions, where we are expecting **IF** statements. In refinement proofs, we may be interested in determining whether the refined and refining machines output the same boolean value for a certain operation. In this case, asking whether two arguments to bool are the same is equivalent to showing that both corresponding **IF** statements always execute the same branch (**IF** or **ELSE**).

2.4.10 Supporting Software

Two software packages were used to complement the given hand-proofs. First, B4free [2] was used to syntax-check and typecheck the machines. B4free also generates proof obligations, and can prove some of them with its automatic provers. Second, ProB [14] was used to perform model-checking. It includes both temporal and constraint-based checkers, which can find different types of errors in specifications. Temporal checking involves exploring the state space of a B machine, by first initialising the machine and then executing operations which modify the machine's state. While it is generally not possible to visit all configurations of a machine, it does tend to quickly find consistency problems (such as a self-contradicting properties clause) and invariant violations. As a case in point, ProB can be used to quickly and automatically show consistency of the puzzle given in Listing 2.1, largely because the state space is small. Additionally, one can supply the goal of the puzzle to ProB, and it can automatically show the sequence of operation calls which solve the puzzle. The constraint-based checker, on the other hand, does not begin by initialising the machine. Instead, it tries to find states which satisfy the invariant, but which are just one operation call away from violating it. Errors found in this way, and not by the temporal-checker, involve states which are not reachable through the initialisation, but are errors nonetheless according to B proof obligations.

The input formats accepted by B4free and ProB have several syntactic differences. We have used a common subset of B in order to achieve compatibility with both tools. This leads to some unnecessary verbosity at several points in machine descriptions. Specifically, ProB does not support the closure(s) operator on sets; it is instead simulated by using closure1(s) $\cup$ id(x), where x is the type of s. Also unsupported by ProB is the retrieval of definitions from files; we therefore must repeat common definitions in all machines which use them. Finally, we are required to

include type information which could conceivably be extracted from the surrounding context, because ProB explicitly requires it in source files.

Chapter 3

Correctness of Parsing Algorithms

In this chapter we chronologically outline the previous work on the correctness of Earley's algorithm. We consider the original, unstructured proof given by Earley; Jones' subsequent structured development; and Sikkel's parsing schemata framework.

3.1 Earley's Proof

Earley's Ph.D. thesis [5] contains the first description and associated proof of the algorithm. Earley immediately commits to a list representation of state sets, by running the Earley operations "in order" on the lists. When an operation is to add a new state, it is added to the end of the list, assuming it does not exist already. Using this version of the algorithm, an if-and-only-if argument is given, showing that a sentence is derivable from the root iff Earley's recognizer returns true.

The given proof is unsatisfactory for several reasons. The version of the algorithm used in the proof includes a lookahead feature, which adds complexity to both the predictor and completer. Instead of only expanding the nonterminal after the dot, the predictor also considers the possible strings of terminals that can follow this nonterminal. The completer then uses this information to determine if a complete step should be executed or not. That is, we can avoid some complete steps if the lookahead expected by some state does not match with what is known to come next in the supplied sentence. It is not clear if this lookahead feature is beneficial in practice [1]. Some researchers have found more propitious variations of lookahead, where the predictor makes the lookahead decisions instead of the completer; others posit that it is not necessary at all. Unfortunately, this dubious feature dominates

Earley's proof and hides its comparitively elegant underlying structure.

Earley's proof also gives no mention of the issues caused by ϵ-productions; a parenthetical warning that we should only complete when all states have been added is the only foreboding comment. In a later implementation discussion, Earley comes back to this point in stating that the completer cannot be implemented in a straightforward way as lists are being processed. He offers a solution involving dynamic bookkeepping of the nonterminals that have been completed, but there is no further proof that this is sufficient for preserving the validity of the earlier proof.

3.2 Jones' Structured Development

Cliff Jones [11] provides a correctness proof of the algorithm by breaking the problem into several phases, beginning with specification and ending with implementation. He gradually introduces properties that any correct parsing algorithm would have to possess, and adds data structures and associated operations to achieve this.

For specifying the problem, Jones gives the definitions of grammar and derivability, then defines what it means to be a recognizer. For example, a grammar is formally defined as a set of rules, rules are defined as (*nonterminal, element-list*) pairs and elements are defined as the union of the disjoint terminal and nonterminal sets. Having defined derivability, a recognizer is defined as a function from grammars and strings to booleans. The function returns true when the string is derivable from the root of the grammar, and false otherwise. We now know what Earley's recognizer is supposed to do.

In the next step, Jones serendipitously introduces state sets, in the form that they are used in Earley's algorithm. Importantly, though, he abstracts from how the states are created, and only gives the lower and upper bounds of the state sets to ensure that the algorithm can still act as a recognizer. Consider states as (r, j, f) triples, where r is a production and j and f are natural numbers. The upper bound is that if a state (r, j, f) exists in $s(i)$, it means that $x_1 x_2 \ldots x_f \alpha$ is derivable from the start symbol, where $x_1 x_2 \ldots x_n$ is the sentence and α is some string of terminals. It also means that the first j symbols on the right-side of production r can derive $x_{f+1} x_{f+2} \ldots x_i$. There are two lower bounds that must be satisfied: first, that the start item is present in $s(0)$, and second, that for all states in $s(i)$, if the next symbol after the dot in a state can derive $x_{i+1} x_{i+2} \ldots x_l$, then the state, with the dot moved one symbol to the right, exists in $s(l)$. Any algorithm that generates state sets between this lower and upper

bound will suffice — the next step shows that Earley gives us one such method.

The third step is to formulate Earley's algorithm in terms of the complete, predict and scan steps we are familiar with. To avoid any problems dealing with treating the sets as lists, Jones uses closure on these operations to construct sets that are fixed-points. All that has to be shown to prove correctness at this step is that the states produced by this method fall between the bounds given previously.

We now know that any algorithm that produces sets which are fixed-points of Earley's operations can act as a general context-free recognizer. The fourth and final step of Jones' development is to use the list representation of states, scanning them in order and applying the operation which applies. If this method ensures that no other operation can extend a list which we have finished processing, then we have reached the same fixed-point as in the previous step, and can conclude that the algorithm is correct. Naturally, we can't. Jones finds that the list-processing version is equivalent to the set-building version when there are no ϵ-productions in the grammar, but problems arise if there are.

This structured development improves on Earley's proof in several respects. Since a recognizer is formally defined, we have a specification of what Earley's algorithm is supposed to do. Additionally, the differences between treating collections of states as sets or lists are made explicit, and issues with ϵ-productions naturally appear in the development.

While the progression from one step to the next is intuitively clear, there is no formal means given to verify that the steps are justified. In fact, Jones notes that finding such general principles are one reason for carrying out this proof, and hopes that future structured developments can be simplified by relying on such principles.

3.3 Sikkel's Parsing Schemata

Sikkel's work on parsing schemata [19] uses abstract descriptions of parsers to prove their correctness. The goal is to prove that a given parsing scheme is correct, and then show that a parsing algorithm uses such a scheme. Since schemata abstract from implementation details, the framework makes proving parsing algorithms easier. We illustrate the theory by exploring how it applies to CYK, and then discuss its application to Earley parsers.

3.3.1 Correctness Criteria

To begin, it is necessary to give the domain of items, a set of hypotheses, and a set of deduction rules, collectively making up a *parsing system.* The items characterize the execution of the parser; their presence will come to represent some "useful" property of derivability which has been deduced. For the CYK algorithm, the domain of items is as follows: $\{[A,i,j] \mid A \in N \ \wedge \ 0 \leq i < j\}$, where N is the set of nonterminals of the grammar. This means that throughout the execution of the parser, it is only permitted to generate items which are triples consisting of a nonterminal and two integers. It is hoped that an item $[A,i,j]$ will have the property that terminals $i, i+1, ..., j$ of the input can be recognized by nonterminal A; we return to this below when discussing valid items. For a string of length n, if we find $[S',0,n]$ in the set of items, then we have successfully recognized the input string. The hypotheses and deduction rules must be given to make this last statement true; that is, they add $[S',0,n]$ if and only if the sentence is recognizable. The hypotheses state that we immediately have items $[a,i-1,i]$ if a is the ith input symbol. (It is evident here that hypotheses do not have to be in the item domain.) For the deduction rules, we have two types, corresponding to the two forms of rules in a CNF grammar. First, if $[a,i-1,i]$ is present, then $[A,i-1,i]$ can be added, assuming that $A \rightarrow a$. From the hypotheses, this immediately gives us one item per grammar rule of the form $A \rightarrow a$ for use in adding more items. Second, if we have both $[B,i,j]$ and $[C,j,k]$, then we can add $[A,i,k]$ assuming there is a rule $A \rightarrow BC$.

All items that can be added, in one or more steps, from the hypotheses are termed *valid items.* Additionally, some items will be designated *final items* — in the case of CYK, the final item is $[S',0,n]$. Final items correspond to the situations where the entire string has been processed (i.e. we have a parse tree for the entire sentence). A final item is termed a *correct item* for a given grammar if there is a valid parse tree for recognizing the given sentence (and hence a reason for the final item being added to the item set). To prove correctness then amounts to showing that all valid final items are correct and that all correct final items are valid.

One issue not dealt with is how to find a closed-form W for the set V of valid items — so far we just know that they are derivable from the hypotheses, using the deduction rules, in one or more steps. We have to show that V and W are equal, so for proving that V is a subset of W, we can use induction on the number of deductions used to add the item. For proving that W is a subset of V, we use induction on

a *derivation length function (DLF) d.* The function d must have the property that for every item ν in W, there is a sequence of deduction steps that can add ν to V, and all η_i involved in such a derivation satisfy $d(\eta_i) < d(\nu)$. It is also required that $d(h) = 0$ for all hypotheses h. Finding such a function is proof that $W \subseteq V$. Going back to CYK, we are looking for a value v we can assign to each item $[A, i, j]$ such that all items used in its derivation have values less than v. We know that deduction rules combine two nonterminals that each generate less of the input string, to find a nonterminal that generates more. Thus, if we say $d([A, i, j]) = j - i$ (because A generates $j - i$ terminals of the sentence), then no item used to create this one can have a bigger value for d, and we have a suitable DLF.

3.3.2 From CYK to Earley

To prove Earley's algorithm correct, Sikkel first takes a detour through a simpler version he calls Bottom-up Earley, because it introduces the main ideas for the more complicated case. Bottom-up Earley capitalizes on the fact that the predictor operation is unnecessary if we "predict" every such item from the start. In other words, instead of only predicting on nonterminals that can be expanded at the current time, we take the pessimistic approach and add all productions of the grammar, with the dot at the beginning, to all state sets. In characterizing the domain of items, Sikkel uses one set of quadruples instead of multiple sets of triples; the new component indicates which set the item would have been in. Items are thus of the form $[A \rightarrow \alpha \bullet \beta, i, j]$, with $A \rightarrow \alpha\beta \in P$. The valid items generated by using the initial item and then employing the scanner and completer are characterized as follows: $\{[A \rightarrow \alpha \bullet \beta, i, j] \mid \alpha \rightarrow^* x_{i+1} \ldots x_j\}$. Note that this is one of the two upper bound conditions found in Jones' formulation, showing that correctness can be proven in spite of the other being dropped. Where CYK has two types of derivation rules, Bottom-up Earley has three, corresponding to adding the initial items, using the scanner and using the completer. In this context, a sentence is accepted if we generate the item $[S' \rightarrow \gamma\bullet, 0, n]$. The hypotheses are the same as in CYK, and are used only by the scanner to effectively compare the next symbol in a production with the next token of the sentence. In the standard Earley algorithm, things are more conventional. The deduction rule for the initialization adds only items $[S' \rightarrow \bullet\gamma, 0, 0]$, which corresponds to one item if our restriction on S' is enforced. The rest is the same as Bottom-up Earley, except the predictor is of course necessary now.

It is interesting to compare these two variations on their relative efficiencies. In degenerate grammars, the standard Earley algorithm could be forced to add all possible predictor items anyway, so the Bottom-up version is still $O(n^3)$. The standard version does add less items in general, but at the cost of a far more complicated DLF. In the bottom-up case, associating each item with the number of complete and scan operations it took to generate it would suffice, since items in its derivation would necessarily have at least one less. For item $[A \rightarrow \alpha \bullet \beta, i, j])$, we have recognized $j - i$ terminals, so it took $j - i$ scan steps. To quantify the work of the completer, it turns out that the number of complete steps corresponds with the number of steps necessary to realize the derivation $\alpha \rightarrow^* x_{i+1} \ldots x_j$; we can therefore add to $j - i$ the number of steps in this derivation to yield a DLF. We have implicitly used the fact that $[A \rightarrow \alpha \bullet \beta, i, i])$ took 0 steps to be created (since it would be added initially). In the standard version of the algorithm, this is not true: its value depends on execution until that point. This is what complicates the DLF, whose definition and justification can be found in [19].

Sikkel shows that parsing schemata are a generalization of chart parsers, and can easily be converted to such an implementation. Chart parsers consist of an agenda and a chart: the agenda is the queue of items still to consider, and the chart holds all previously found items. Using deduction rules, items on the agenda are used together with the contents of the chart to add new items to the agenda. We can therefore produce a correct chart Earley parser, once we have established that the Earley parsing schemata is correct. The problem is that chart parsers are inefficient, and special-purpose data structures must be introduced to remedy this. Parsing schemata say nothing about these parts of the implementation, though, and provide no help in correctly implementing these optimizations.

Chapter 4

Proof of Earley's Recognizer

4.1 Overview of Development

The previous chapter outlined the current proof approaches — some less formal, some more formal — for Earley's algorithm. Our goal is to provide a formal proof from specification, taking into account new developments in the treatment of ϵ-productions, and arriving at an efficient implementation. A proof exhibiting all these criteria has so far been elusive.

Our first task is to define machines for representing grammars and sentences, since these are the two data structures required by the algorithm. Using these two machines, we can then describe a context-free recognizer machine whose sole operation tells us whether the given sentence can be recognized by the given grammar. This machine will have nothing to do with Earley at all: it will indicate, in the most abstract way, how to test derivability, giving no hint on how it can actually be implemented. The implementation of this operation, of course, is exactly what Earley provides, and so Earley's algorithm will naturally be a refinement of this general recognizer machine. It is difficult to implement this algorithm directly, however, so first we develop a machine to encapsulate state sets, and the associated operations of prediction, scanning and completion. Importing this machine into the refined version of the recognizer allows us to express Earley's algorithm using a while loop. It then remains to prove that we have a valid refinement — that is, both machines return the same boolean value for their respective recognition operations.

4.2 A Grammar Machine

There are several design decisions that must be made in order to effectively represent a context-free grammar via a machine. Many of the decisions parallel those used in Chapter 3 of [18]. There, a machine was developed to represent an undirected graph, and it was decided to use constants for the edges and weights, and the first N natural numbers for the nodes. To that end, we represent nonterminals and terminals as consecutive segments of the naturals, and the grammar productions as a relation between nonterminals and sequences of symbols. We also require that the root of the grammar appear as the left-hand-side of just one production, not appearing at all in the right-hand-sides of any productions[1]. To impose these requirements, we state that *productions*, under the image of Root— has just one member. Also, *Root* is prevented from occurring in the range of any element in *productions*, which represents the right-hand-sides of all productions. The machine is presented in Listing 4.1.

Listing 4.1: Grammar Machine

MACHINE *gram*
CONSTANTS *numT*, *numNT*, *productions*, *ls*, *rs*
DEFINITIONS
$Nonterminals == (1 \mathinner{..} numNT)$;
$Terminals == (numNT + 1 \mathinner{..} numNT + numT)$;
$Symbols == (Terminals \cup Nonterminals)$;
$Root == 1$;
$directlyDerivable ==$
$\{xx, yy \mid xx \in \mathrm{seq}(Symbols) \wedge yy \in \mathrm{seq}(Symbols) \wedge$
$(\exists \mu, \sigma, \nu, \tau).$
$(\mu \in \mathrm{seq}(Symbols) \wedge \sigma \in Symbols \wedge$
$\nu \in \mathrm{seq}(Symbols) \wedge \tau \in \mathrm{seq}(Symbols) \wedge$
$xx = (\mu \frown [\sigma] \frown \nu) \wedge (yy = \mu \frown \tau \frown \nu) \wedge (\sigma \mapsto \tau \in productions))\}$;
$derivable == (\mathrm{closure1}(directlyDerivable) \cup \mathrm{id}(\mathrm{seq}(Symbols)))$

PROPERTIES
$numNT \in \mathbb{N}_1 \wedge numT \in \mathbb{N} \wedge$
$productions \in Nonterminals \leftrightarrow \mathrm{seq}(Symbols) \wedge$

[1]These restrictions simplify proofs later and do not restrict the power of the recognizer in any way; i.e. we can take any CFG and add a new root production.

$\mathrm{card}(productions[\{Root\}]) = 1 \wedge$
$(\forall xx) \cdot (xx \in \mathrm{ran}(productions) \Rightarrow (Root \notin \mathrm{ran}(xx))) \wedge$
$\mathrm{card}(productions) \in \mathbb{N} \wedge$
$(SIGMA\ (z) \cdot (z \in \mathrm{ran}(productions) \mid \mathrm{size}(z))) + \mathrm{card}(productions) \in \mathbb{N} \wedge$
$ls \in (1 \mathinner{..} \mathrm{card}(productions)) \rightarrow Nonterminals \wedge$
$rs \in (1 \mathinner{..} \mathrm{card}(productions)) \rightarrow \mathrm{seq}(Symbols) \wedge$
$(ls \otimes rs) \in 1 \mathinner{..} \mathrm{card}(productions) \rightarrowtail\!\!\!\!\!\!\rightarrow productions \wedge$
$ls(1) = Root$

OPERATIONS

$ans \longleftarrow getLS(ii) =$
PRE $ii \in 1 \mathinner{..} \mathrm{card}(productions)$ **THEN**
$ans := ls(ii)$
END;

$ans \longleftarrow getRS(ii, jj) =$
PRE $ii \in 1 \mathinner{..} \mathrm{card}(productions) \wedge jj \in 1 \mathinner{..} \mathrm{size}(rs(ii))$ **THEN**
$ans := rs(ii)(jj)$
END;

$ans \longleftarrow numRules =$
$ans := \mathrm{card}(productions)$;

$ans \longleftarrow ruleLength\ (ii) =$
PRE $ii \in 1 \mathinner{..} \mathrm{card}(productions)$ **THEN**
$ans := \mathrm{size}(rs(ii))$
END;

$ans \longleftarrow nullable(ii) =$
PRE $ii \in Symbols$ **THEN**
$ans := \mathrm{bool}\ ([ii] \mapsto [] \in derivable)$
END

END

The representation of grammars just described is sufficient to specify the workings of a context-free recognizer, and its first refinement to Earley's algorithm. For instance, consider executing the predictor on an item in an Earley state set. What we require here is easy access to those productions whose left-hand-side nonterminal is the symbol after the dot in the item. We can obtain the right-hand-sides of these items by a relational image of *productions* under the nonterminal in question. However, in terms of eventual implementation, these set-theoretic operations are not permitted and so we require more conventional access to the components of the grammar. For this reason, we introduce constants *ls* and *rs*, which impose an arbitrary ordering on the grammar productions. They are defined as total functions whose domains consist of card(*productions*) elements. The range of *ls* contains the left-hand-sides of all productions, and the range of *rs* contains the corresponding right sides. The constraint $(ls \otimes rs) \in 1 \,..\, \mathrm{card}(productions) \rightarrowtail\!\!\!\!\!\rightarrow productions$ says that the direct product of *ls* and *rs* is a bijection between 1 .. card(*productions*) and the productions themselves — in other words, that the natural numbers are related to exactly one production. We only require that $ls(1)$ is the root, so that we know that the first production in the ordering is the single production involving the root.

Now that we have imposed an arbitrary ordering, we can provide operations for retrieving meaningful portions of grammar rules. We have *getLS* which, given a suitable index, returns the left-hand-side nonterminal for the associated production; *getRS* does similarly for right-hand-sides, although this time an additional index into its sequence of symbols is necessary. Operations for obtaining the number of rules, and the length of a specified rule, are provided so that later machines can be sure to call the query operations within their preconditions.

4.2.1 Correctness

Our only task here is to show that the machine constants can be instantiated in at least one way to yield a consistent properties clause; that is, we must show proof obligation (1) in Figure 2.10. We can start by giving *numT* and *numNT* the value 1, so that $numNT \in \mathbb{N}$ and $numT \in \mathbb{N}$. We then have to instantiate *productions* with (s, t) pairs, where s is a nonterminal and t is a sequence of symbols. We can use $\{(Root, [])\}$, which represents the grammar with one production $Root \rightarrow \epsilon$. This satisfies the constraints dealing with the root as well, and so all constraints

on *productions* are satisfied. For *ls* and *rs*, we can use $\{(1,\ \mathit{Root})\}$ and $\{(1,\ [])\}$, respectively. The types of these expressions are correct (they are total functions into nonterminals and sequences of symbols), their direct product yields *productions* and the first (and only) entry in the domain maps to the root production.

4.3 A Sentence Machine

A sentence is a sequence of terminals that we are interested in testing for derivability. We thus define a constant, *sentence*, by *seeing* the *gram* machine, and making use of the same definition of *Terminals* that was present there. Listing 4.2 contains the sentence machine which encapsulates this idea. As with the grammar machine, it is necessary to provide queries for retrieving the length of the sentence and the terminals from which it is built.

Listing 4.2: Sentence Machine

MACHINE *sent*
SEES *gram*
DEFINITIONS
$\mathit{Terminals} == (\mathit{numNT} + 1\ ..\ \mathit{numNT} + \mathit{numT})$
CONSTANTS *sentence*
PROPERTIES $\mathit{sentence} \in \mathrm{seq}(\mathit{Terminals}) \wedge \mathrm{size}(\mathit{sentence}) \in \mathbb{N}$

OPERATIONS
$ss \longleftarrow \mathit{senLength} = ss := \mathrm{size}(\mathit{sentence});$

$bb \longleftarrow \mathit{senGet}(xx) =$
PRE $xx \in \mathbb{N}_1 \wedge xx \leq \mathrm{size}(\mathit{sentence})$ **THEN**
$bb := \mathit{sentence}(xx)$
END

END

4.3.1 Correctness

All that is necessary is to exhibit an assignment to the *sentence* constant which is in line with the machine's properties clause; this is proof obligation (1) in Figure 2.10. We can do this easily by giving it the value [] — that is, an empty sequence — which is trivially a sequence of terminals. This corresponds to the sentence ϵ. In the nontrivial case, $numT$ in the grammar machine will be positive, so we would have a nonempty set of terminals to draw from to construct the sentence.

4.4 A General Recognizer Machine

Consider Listing 4.3, which contains the definition of a machine that can recognize whether the supplied sentence can be generated from the given grammar.

Listing 4.3: CFG Recognizer Machine

MACHINE *recm*
SEES *gram, sent*
DEFINITIONS
$Symbols == (1..(numNT + numT))$;
$Root == 1$;
$directlyDerivable ==$
$\{xx, yy \mid xx \in \text{seq}(Symbols) \wedge yy \in \text{seq}(Symbols) \wedge$
$(\exists mu, sigma, nu, tau).$
$(mu \in \text{seq}(Symbols) \wedge sigma \in Symbols \wedge$
$nu \in \text{seq}(Symbols) \wedge tau \in \text{seq}(Symbols) \wedge$
$xx = (mu \frown [sigma] \frown nu) \wedge (yy = mu \frown tau \frown nu) \wedge (sigma \mapsto tau \in productions))\}$;
$derivable == (\text{closure1}(directlyDerivable) \cup \text{id}(\text{seq}(Symbols)))$

OPERATIONS
$ans \longleftarrow isSentence =$
$ans := \text{bool}([Root] \mapsto sentence \in derivable)$

END

The machine formalizes the notion of derivable, and then uses it to return whether or not the sentence can be derived from the root. We first define *directlyDerivable*

which is a relation whose domain type and range type is seq (*symbols*). The pair (xx, yy) will be in *directlyDerivable* if yy is derivable from xx in exactly one step. We rely on Definition 2.1, and almost literally transcribe it into B notation. We use existential quantifiers to implicitly break up the sequences xx and yy into three pieces, being careful to type σ as one symbol (not a sequence) since it is required to be the same type as the domain of *productions*. We can use this definition to define the notion of derivable, given in Definition 2.2. There, we said that yy was derivable from xx if we could effectively iterate *directlyDerivable* enough times to obtain yy. This corresponds to taking the reflexive, transitive closure of the *directlyDerivable* relation just described.

With the given definition of derivable, we can succinctly describe the *isSentence* operation: it returns *true* exactly when the root nonterminal and the sentence are related by *derivable*.

We have nothing to prove to show consistency of this machine: there is no invariant and no newly introduced constants. Furthermore, the included operation is a query operation which incurs no proof obligations. In situations like this it is necessary to convince oneself that the operations are meaningful, and carry out what is intended. The machine will be refined by Earley's algorithm in the next sections.

4.5 Introducing State Sets

We now write an implementation for the CFG recognizer of the previous subsection. The *isSentence* operation is to be refined by a version using Earley's algorithm instead of an abstract assertion about derivability. What we are striving for is the use of a loop which consecutively builds Earley state sets one at a time, until we have created all $n + 1$ sets. At this point, we can check the last state set to see if it contains the acceptance element, and return the corresponding Boolean value.

Since implementation machines can only make use of scalar variables and arrays, we can separate out the state set data structure and its operations into a new machine, and import it into the implementation. The state sets machine, on its own, maintains some invariants, and so this separation also lets us locally prove properties of the state sets. When used as part of the implementation, we can rely on these invariants to hold, and they will be crucial for proving that Earley is a correct refinement. This general technique is referred to as Design for Provability; see, for example, Chapter 3 of [18].

To represent Earley states, we split the production component in two pieces corresponding to the portions before and after the dot. We thus define a state as $state == (Nonterminals \times \text{seq}(Symbols) \times \text{seq}(Symbols) \times \mathbb{N})$. We commonly refer to the four components, in code snippets and in proofs, as $(a,\ l,\ r,\ f)$, where a is a nonterminal, l and r are the components of a right-hand-side of a which are to the left and right of the dot, respectively, and f is the pointer to the production's starting state set.

Next, we consider the scan, predict and complete operations. Considering the scanner first, we know that when it acts on a state, the most it can do is add one more state. In particular, if the next symbol in a state is a terminal, and it matches the next sentence symbol, then it relates this state to the same state with the dot moved one position to the right. We also require access to the sentence, so that we can inspect its $(i+1)$-st element in order to determine whether an item is in the domain of the scanner. We can therefore conceive of the scanner as a function which, when applied to an integer, is a partial function from states to states. In our definition, we assert that two states are related by the scanner if their a and f components are the same, and their l and r components are the same except the dot is moved over one position to the right in the function's range. The core of the function is the following:

$$\begin{aligned}&scan(ii) == \{a1 \mapsto l1 \mapsto r1 \mapsto f1,\ a2 \mapsto l2 \mapsto r2 \mapsto f2 \mid \\ &\quad r1 \neq \{\} \land \text{first}(r1) \in Terminals \land \text{first}(r1) = sentence(ii{+}1) \land \\ &\quad a2 = a1 \land l2 = l1 \leftarrow \text{first}(r1) \land r2 = \text{tail}(r1) \land f2 = f1)\}\end{aligned}$$

The predictor and completer are similar to the scanner in that they relate state sets to state sets. While the scanner is a partial function, the predictor and completer are partial relations. The predictor can add multiple items from a single state in the situation where the next nonterminal has multiple right-hand-sides; the completer can add multiple items if the fth state set has more than one item with the required nonterminal after the dot.

The predictor relates $s = (a1,\ l1,\ r1,\ f1)$ to $t = (a2,\ l2,\ r2,\ f2)$ if $a2$ is the first symbol in $r1$. Additionally, the dot must be at the beginning in t, so $l2$ must be empty, and $r2$ must correspond to a full right-hand-side of a production of $a1$. Again, we require knowledge about which state set we are constructing, since this is the value to give $f2$. This results in the following:

$$\begin{aligned}&predict(ii) == \{a1 \mapsto l1 \mapsto r1 \mapsto f1,\ a2 \mapsto l2 \mapsto r2 \mapsto f2 \mid \\ &\quad r1 \neq \{\} \land a2 = \text{first}(r1) \land l2 = [] \land r2 \in productions[\{\text{first}(r1)\}] \land f2 =\end{aligned}$$

$ii)\}$

The most complicated definition involves the completer. While the scanner requires knowledge of our progress in order to inspect the input sentence and the predictor uses it for populating the f components, the completer has no use for it. Another difference is that we can no longer relate state $s = (a1,\ l1,\ r1,\ f1)$ to state $t = (a3,\ l3,\ r3,\ f3)$ directly, but must use another state $q = (a2,\ l2,\ r2,\ f2)$ as an intermediary. Specifically, we require that $r1$ is the empty sequence, corresponding to the case where we have finished processing $a1$. State q is then confined to exist in state set $f1$, and have $a1$ as its next symbol to process. With these conditions in place, the completer can add a modified q to state set i, obtained by moving the dot in the same way that the scanner did. This gives the following:

$$
\begin{aligned}
&complete == \{a1 \mapsto l1 \mapsto r1 \mapsto f1,\ a3 \mapsto l3 \mapsto r3 \mapsto f3 \mid \\
&\quad (\exists a2,\ l2,\ r2,\ f2)\ . \\
&\qquad r1 = [] \ \land r2 \neq [] \land \\
&\qquad (a2 \mapsto l2 \mapsto r2 \mapsto f2) \in ss(f1) \ \land a1 = \text{first}(r2) \ \land \\
&\qquad a3 = a2 \land l3 = l2 \leftarrow \text{first}(r2) \ \land r3 = \text{tail}\ (r2) \ \land f3 = f2)\}
\end{aligned}
$$

The machine contains two variables: ss for the state sets, and ii to maintain which state set we are currently constructing. When executing the algorithm, we know that the root element must first be added to $ss(0)$, and all other $ss(i)$ should be empty. We can then construct the root element as $rootElement == (Root \mapsto [] \mapsto rhs \mapsto 0)$, where rhs is $productions(Root)$. The initialisation of the machine uses this to construct the initial configuration of the state sets, and also sets ii to 0. We include an operation $init$ whose definition is the same as the initialisation. The reason is pointed out in Chapter 3 of [18]: if we were to use this machine to run Earley's algorithm twice in succession, there would be no way to re-initialise the machine in between the two runs.

The machine maintains some complex invariants, whose basis we have already discovered in the expositions by Jones and Sikkel given in Chapter 2. First, we assert that the root element is always present in $ss(0)$. This is not hard to believe: the initialisation adds it, $init$ adds it, and there are no set operations besides union anywhere else. Next, we make a claim about $ss(0)$ — that it is the transitive closure of the predictor and completer on the root element — but only when we have $ii > 0$; if $ii = 0$, then we may be in the state immediately following the initialisation and so the claim cannot possibly hold. To characterize the remaining state sets that we

have constructed so far, we say that state set i results from running the scanner on set $i-1$, and then closing the predictor and completer over the result.

We then give properties that are expected to hold on the state sets we have constructed. We have that, if $(a1,\ l1,\ r1,\ f1)$ is present in state set i, then $l1$ derives $sentence\{f1+1\}$ .. $sentence(i)$. Additionally, if the next symbol in $r1$ derives $sentence(i+1)$ .. $sentence(l)$, then the same state, with the dot moved to the right, exists in set l. We also have a sanity condition on all states: that the l and r components, together, are a valid right-hand-side of a.

We can then give the machine operations. *inc* increments ii, but only in the situation where state set i is properly constructed to adhere to the machine invariant. In other words, at this level we are hoping that sequences of other operations can create $ss(ii)$ correctly, so that the precondition of *inc* is true. It will be up to the machine that imports this one to ensure that this is the case. *getIi* is a query operation to obtain ii. *nextOps* runs the predictor and completer on the current state set, while *nextScan* runs the scanner on it and deposits states into $ss(ii+1)$. Finally, *accept* encapsulates the acceptance criterion. The state sets machine is in 4.4.

Listing 4.4: State Sets Machine

MACHINE *stateSets*
SEES *sent*, *gram*
DEFINITIONS
$Nonterminals == (1 .. numNT)$;
$Terminals == (numNT + 1 .. numNT + numT)$;
$Symbols == (Terminals \cup Nonterminals)$;
$Root == 1$;
$rhs == (productions(Root))$;
$directlyDerivable ==$
$\{xx,\ yy \mid xx \in \text{seq}(Symbols) \wedge yy \in \text{seq}(Symbols) \wedge$
$(\exists \mu,\ \sigma,\ \nu,\ \tau).$
$(\mu \in \text{seq}(Symbols) \wedge \sigma \in Symbols \wedge$
$\nu \in \text{seq}\ (Symbols) \wedge \tau \in \text{seq}(Symbols) \wedge$
$xx = (\mu \frown [\sigma] \frown \nu) \wedge (yy = \mu \frown \tau \frown \nu) \wedge (\sigma \mapsto \tau \in productions))\}$;
$derivable == (\text{closure1}(directlyDerivable) \cup \text{id}(\text{seq}(Symbols)))$;
$state == (Nonterminals \times \text{seq}\ (Symbols) \times \text{seq}\ (Symbols) \times \mathbb{N})$;
$type1 == a1 \in Nonterminals \wedge l1 \in \text{seq}(Symbols) \wedge$

$$
\begin{array}{l}
r1 \in \mathrm{seq}(Symbols) \wedge f1 \in \mathbb{N}; \\
type2 == a2 \in Nonterminals \wedge l2 \in \mathrm{seq}(Symbols) \wedge \\
r2 \in \mathrm{seq}(Symbols) \wedge f2 \in \mathbb{N}; \\
type3 == a3 \in Nonterminals \wedge l3 \in \mathrm{seq}(Symbols) \wedge \\
r3 \in \mathrm{seq}(Symbols) \wedge f3 \in \mathbb{N};
\end{array}
$$

$$
\begin{array}{l}
scan(ii) == \{xx, yy \mid xx \in state \wedge yy \in state \wedge \\
\quad \#(a1, l1, r1, f1, a2, l2, r2, f2) \cdot \\
\qquad (type1 \wedge type2 \wedge \\
\qquad xx = (a1 \mapsto l1 \mapsto r1 \mapsto f1) \wedge yy = (a2 \mapsto l2 \mapsto r2 \mapsto f2) \wedge \\
\qquad r1 \neq \{\} \wedge \mathrm{first}(r1) \in Terminals \wedge \mathrm{first}(r1) = sentence(ii) \wedge \\
\qquad a2 = a1 \wedge l2 = l1 \leftarrow \mathrm{first}(r1) \wedge r2 = \mathrm{tail}(r1) \wedge f2 = f1)\};
\end{array}
$$

$$
\begin{array}{l}
predict(ii) == \{xx, yy \mid xx \in state \wedge yy \in state \wedge \\
\quad \#(a1, l1, r1, f1, a2, l2, r2, f2) \cdot \\
\qquad (type1 \wedge type2 \wedge \\
\qquad xx = (a1 \mapsto l1 \mapsto r1 \mapsto f1) \wedge yy = (a2 \mapsto l2 \mapsto r2 \mapsto f2) \wedge r1 \neq \{\} \wedge \\
\qquad a2 = \mathrm{first}(r1) \wedge l2 = [] \wedge r2 \in productions[\{\mathrm{first}(r1)\}] \wedge f2 = ii)\};
\end{array}
$$

$$
\begin{array}{l}
complete == \{xx, yy \mid xx \in state \wedge yy \in state \wedge \\
\quad \#(a1, l1, r1, f1, a2, l2, r2, f2, a3, l3, r3, f3) \cdot \\
\quad (type1 \wedge type2 \wedge type3 \wedge \\
\qquad xx = (a1 \mapsto l1 \mapsto r1 \mapsto f1) \wedge yy = (a3 \mapsto l3 \mapsto r3 \mapsto f3) \wedge \\
\qquad r1 = [] \wedge r2 \neq [] \wedge \\
\qquad (a2 \mapsto l2 \mapsto r2 \mapsto f2) \in ss(f1) \wedge a1 = \mathrm{first}(r2) \wedge \\
\qquad a3 = a2 \wedge l3 = l2 \leftarrow \mathrm{first}(r2) \wedge r3 = \mathrm{tail}(r2) \wedge f3 = f2)\};
\end{array}
$$

$$
\begin{array}{l}
ops(ii) == (predict(ii) \cup complete \cup \mathrm{id}(state)); \\
rootElement == (Root \mapsto [] \mapsto rhs \mapsto 0); \\
indef == ss, ii := \{xx, yy \mid xx \in (0 \,..\, \mathrm{size}(sentence)) \wedge yy \in \mathbb{P}(state) \wedge \\
(xx = 0 \Rightarrow yy = \{rootElement\}) \wedge (xx \neq 0 \Rightarrow yy = \{\})\}, 0
\end{array}
$$

CONSTANTS *zero*
PROPERTIES $zero = 0$

VARIABLES *ss*, *ii*

INVARIANT

$ss \in (0 \,..\, \text{size}(sentence)) \rightarrow \mathbb{P}(state) \wedge ii \in \mathbb{N} \wedge ii \leq \text{size}(sentence) + 1 \wedge$
$rootElement \in ss(0) \wedge$

$(ii > 0 \Rightarrow (ss(0) = ((\text{closure1}(ops(zero)))[\{rootElement\}]))) \wedge$

$(\forall jj, kk) \,.\, (jj \in \mathbb{N}_1 \wedge kk \in \mathbb{N} \wedge kk = jj - 1 \wedge jj < ii \Rightarrow$
$(\exists sc) \,.\, (sc \subseteq state \wedge sc = scan(kk)[ss(kk)] \wedge ss(jj) =$
$(\text{closure1}(ops(jj))[sc]))) \wedge$

$(\forall a1, l1, r1, f1, ind) \,.\, (type1 \wedge ind \in \mathbb{N} \wedge ind < ii \wedge (a1 \mapsto l1 \mapsto r1 \mapsto f1) \in$
$state \wedge$
$(a1 \mapsto l1 \mapsto r1 \mapsto f1) \in ss(ind) \Rightarrow$
$((l1 \mapsto ((sentence \downarrow f1) \uparrow (ind - f1))) \in derivable)) \wedge$

$(\forall z, j, a1, l1, r1, f1) \,.\, (type1 \wedge z \in \mathbb{N} \wedge z < ii \wedge j \in \mathbb{N} \wedge$
$j \leq z \wedge (a1 \mapsto l1 \mapsto r1 \mapsto f1) \in state \wedge (a1 \mapsto l1 \mapsto r1 \mapsto f1) \in ss(j) \wedge$
$([\,\text{first}\,(r1)] \mapsto ((sentence \downarrow j) \uparrow (z - j))) \in derivable \Rightarrow$
$((a1 \mapsto (l1 \frown [\,\text{first}\,(r1)]) \mapsto \text{tail}(r1) \mapsto f1)) \in ss(z)) \wedge$

$(\forall a1, l1, r1, f1, ind) \,.\, (type1 \wedge ind \in \mathbb{N} \wedge ind < ii \wedge (a1 \mapsto l1 \mapsto r1 \mapsto f1) \in$
$ss(ii) \Rightarrow$
$((a1 \mapsto (l1 \frown r1)) \in productions \wedge f1 \leq ind)) \wedge$

$(\forall ind) \,.\, (ind \in (ii{+}1)..\text{size}\ (sentence) \Rightarrow ss(ind) = \{\})$

INITIALISATION

indef

OPERATIONS

$init = indef;$

$inc =$
PRE $ss(0) = ((\text{closure1}(ops(zero)))[\{rootElement\}]) \wedge$

$(ii > 0 \Rightarrow$
$(\exists sc,\ kk)\ .\ (kk \in \mathbb{N} \wedge kk = ii - 1 \wedge sc \subseteq state \wedge sc = scan(kk)[ss(kk)] \wedge$
$ss(ii) = (\text{closure1}(ops(ii))[sc]))) \wedge$
$ii \leq \text{size}\ (sentence)$ **THEN**
$ii := ii + 1$
END;

$bb \longleftarrow getIi = bb := ii$;

$nextOps =$
$ss(ii) := \text{closure1}(ops(ii))[ss(ii)]$;

$nextScan =$
IF $ii > 0$ **THEN** $ss(ii) := scan(ii)[ss(ii-1)]$ **END**;

$bb \longleftarrow accept =$
$bb := \text{bool}\ ((Root \mapsto rhs \mapsto [] \mapsto 0) \in ss(\text{size}\ (sentence)))$

END

4.5.1 Correctness

Proving correctness of this machine requires showing, first, that the initialisation establishes the invariant (obligation (3) of Figure 2.10) and, second, that the operations maintain it when called in a state satisfying their preconditions (proof obligation (4) of Figure 2.10). The strategy is to show that the initialisation and operations establish all conjuncts of the invariant separately, from which we can conclude that they establish the whole invariant.

Initialisation Establishes Invariant

We start with the first three conjuncts of the invariant: ss is total and ii is within bounds of the sentence. We begin with the following lemma:

Lemma 4.1. *If* $e \in E$, $f \in F$ *and* $g \in F$, *then:*

$\{xx, yy \mid xx \in E \wedge yy \in F \wedge$

$$(xx = e \Rightarrow yy = f) \wedge (xx \neq e \Rightarrow yy = g)\} \in E \rightarrow F$$

Proof. All elements in E are related to exactly one element in F: one element to a specific element of F, and all the rest to another element in F. □

We can then complete the proof:

$$[ss,\ ii := \{xx,\ yy \mid xx \in (0 \mathinner{..} \mathrm{size}(sentence)) \wedge yy \in \mathbb{P}(state) \wedge$$
$$(xx = 0 \Rightarrow yy = \{rootElement\}) \wedge (xx \neq 0 \Rightarrow yy = \{\})\},\ 0]$$
$$ss \in (0 \mathinner{..} \mathrm{size}(sentence)) \rightarrow \mathbb{P}(state) \wedge ii \in \mathbb{N} \wedge ii \leq \mathrm{size}(sentence) + 1$$

$$= \{WPM\}$$
$$\{xx,\ yy \mid xx \in (0 \mathinner{..} \mathrm{size}(sentence)) \wedge yy \in \mathbb{P}(state) \wedge$$
$$(xx = 0 \Rightarrow yy = \{rootElement\}) \wedge (xx \neq 0 \Rightarrow yy = \{\})\}$$
$$\in (0 \mathinner{..} \mathrm{size}(sentence)) \rightarrow \mathbb{P}(state) \wedge 0 \in \mathbb{N} \wedge 0 \leq \mathrm{size}(sentence) + 1$$

$$= \{\mathrm{size}(sentence) \in \mathbb{N},\ Lemma\ 4.1\}$$
$$true$$

We now show that the initialisation establishes the fact that the root element is in $ss(0)$:

$$[ss,\ ii := \{xx,\ yy \mid xx \in (0 \mathinner{..} \mathrm{size}(sentence)) \wedge yy \in \mathbb{P}(state) \wedge$$
$$(xx = 0 \Rightarrow yy = \{rootElement\}) \wedge (xx \neq 0 \Rightarrow yy = \{\})\},\ 0]$$
$$rootElement \in ss(0)$$

$$= \{WPA\}$$
$$rootElement \in \{xx,\ yy \mid xx \in (0 \mathinner{..} \mathrm{size}(sentence)) \wedge yy \in \mathbb{P}(state) \wedge$$
$$(xx = 0 \Rightarrow yy = \{rootElement\}) \wedge (xx \neq 0 \Rightarrow yy = \{\})\}(0)$$

$$= \{\{0 \mapsto rootElement\}\ is\ in\ the\ constructed\ set\}$$
$$true$$

The next piece of the invariant is an implication with premise $ii > 0$; we know that the invariant sets ii to 0 so this is vacuously true.

Next, a vacuous universal quantification:

$$[ss,\ ii := \{xx,\ yy \mid xx \in (0 \mathinner{..} \mathrm{size}(sentence)) \wedge yy \in \mathbb{P}(state) \wedge$$
$$(xx = 0 \Rightarrow yy = \{rootElement\}) \wedge (xx \neq 0 \Rightarrow yy = \{\})\},\ 0]$$

$(\forall jj,\ kk)\ .\ (jj\ \in \mathbb{N}_1 \wedge kk\ \in \mathbb{N} \wedge kk\ = jj - 1\ \wedge jj < ii\ \Rightarrow ...$

$= \{WPM\}$
$(\forall jj,\ kk)\ .\ (jj\ \in \mathbb{N}_1 \wedge kk\ \in \mathbb{N} \wedge kk\ = jj - 1\ \wedge jj < 0\ \Rightarrow ...$

$= \{empty\ range\ for\ jj\}$
$true$

Proceeding, we have a derivability claim on all elements in ss where the index is less than ii. We have none.:

$[ss,\ ii\ :=\ \{xx,\ yy\ |\ xx\ \in (0\ ..\ \mathrm{size}(sentence))\ \wedge yy\ \in \mathbb{P}(state)\ \wedge$
$(xx\ = 0\ \Rightarrow yy\ = \{rootElement\})\ \wedge (xx\ \neq 0\ \Rightarrow yy\ = \{\})\},\ 0]$
$(\forall a1,\ l1,\ r1,\ f1,\ ind)\ .\ (type1\ \wedge ind\ \in \mathbb{N} \wedge ind < ii\ \wedge (a1 \mapsto l1 \mapsto r1 \mapsto f1)\ \in$
$state\ \wedge$
$\quad (a1 \mapsto l1 \mapsto r1 \mapsto f1)\ \in ss(ind)\ \Rightarrow$
$\quad ((l1 \mapsto ((f1{+}1..ind)\ \lhd sentence))\ \in derivable))$

$= \{WPA\}$
$(\forall a1,\ l1,\ r1,\ f1,\ ind)\ .\ (type1\ \wedge ind\ \in \mathbb{N} \wedge ind < 0\ \wedge (a1 \mapsto l1 \mapsto r1 \mapsto f1)\ \in$
$state\ \wedge$
$\quad (a1 \mapsto l1 \mapsto r1 \mapsto f1)\ \in$
$\quad \{xx,\ yy\ |\ xx\ \in (0\ ..\ \mathrm{size}(sentence))\ \wedge yy\ \in \mathbb{P}(state)\ \wedge$
$\quad (xx\ = 0\ \Rightarrow yy\ = \{rootElement\})\ \wedge (xx\ \neq 0\ \Rightarrow yy\ = \{\})\}(ind)\ \Rightarrow$
$\quad ((l1 \mapsto ((f1{+}1..ind)\ \lhd sentence))\ \in derivable))$

$= \{empty\ domain\}$
$true$

Two easily-proved conjuncts are left, both vacuous universal quantifications. The last conjunct asserts that all state sets are empty except for $ss(0)$, also immediate. This completes the proof that the invariant is established by the initialisation.

Operations Preserve Invariant

We now consider the machine operations, and show that they preserve the invariant. The operation *init* is defined to be the same as the initialisation. Since the

initialisation establishes the invariant under no preconditions at all, *init* preserves the invariant as well.

For *inc*, we first show that the first four conjuncts of the invariant are maintained. We let *inv* be the machine invariant in the following proofs.

$$[\, ii := ii + 1]$$
$$ss \in (0 \,..\, \text{size}(sentence)) \rightarrow \mathbb{P}(state) \wedge ii \in \mathbb{N} \wedge ii \leq \text{size}(sentence) + 1 \wedge rootElement \in ss(0)$$

$$= \{WPA\}$$
$$ss \in (0 \,..\, \text{size}(sentence)) \rightarrow \mathbb{P}(state) \wedge ii{+}1 \in \mathbb{N} \wedge ii{+}1 \leq \text{size}(sentence) + 1 \wedge rootElement \in ss(0)$$

$$\leq =$$
$$inv \wedge ii \leq \text{size}(sentence)$$

Next, we show that *inc* maintains that the zeroth state set is as expected:

$$[\, ii := ii + 1]$$
$$ii > 0 \Rightarrow (ss(0) = ((\text{closure1}(ops(zero)))[\{rootElement\}]))$$

$$= \{WPA\}$$
$$ii{+}1 > 0 \Rightarrow (ss(0) = ((\text{closure1}(ops(zero)))[\{rootElement\}]))$$

$$\leq =$$
$$inv \wedge ss(0) = ((\text{closure1}(ops(zero)))[\{rootElement\}])$$

Next, that executing *inc* gives us one more state set whose contents we know:

$$[\, ii := ii + 1]$$
$$(\forall jj,\ kk) \cdot (jj \in \mathbb{N}_1 \wedge kk \in \mathbb{N} \wedge kk = jj - 1 \wedge jj < ii \Rightarrow (\exists sc) \cdot (sc \subseteq state \wedge sc = scan(kk)[ss(kk)] \wedge ss(jj) = (\text{closure1}(ops(jj))[sc])))$$

$$= \{WPA\}$$
$$(\forall jj,\ kk) \cdot (jj \in \mathbb{N}_1 \wedge kk \in \mathbb{N} \wedge kk = jj - 1 \wedge jj < ii + 1 \Rightarrow (\exists sc) \cdot (sc \subseteq state \wedge sc = scan(kk)[ss(kk)] \wedge ss(jj) = (\text{closure1}(ops(jj))[sc])))$$

$$\leq =$$
$$inv \wedge (ii > 0 \Rightarrow$$

$$(\exists sc,\ kk)\ .\ (kk \in \mathbb{N} \wedge kk = ii - 1 \wedge sc \subseteq state \wedge sc = scan(kk)[ss(kk)] \wedge ss(ii) = (\text{closure1}(ops(ii))[sc]))$$

Here, we have that the invariant was previously established throughout the first $ii-1$ state sets; the precondition of *inc* allows us to conclude that it is now true for the next one as well.

We now know the precise contents of all state sets from 0 to $ii - 1$. The next part of the invariant says for all such states, the symbols before the dot can derive the terminals $sentence(f{+}1)\ ..\ sentence(i)$. We can prove this using only the parts of the invariant we have already shown *inc* to preserve, and so we can conclude that *inc* preserves this conjunct as well. The proof obligation, with the previously proved parts of the invariant numbered, is:

(1) $ss \in (0\ ..\ \text{size}(sentence)) \to \mathbb{P}(state) \wedge ii \in \mathbb{N} \wedge ii \leq \text{size}(sentence) + 1 \wedge$
(2) $rootElement \in ss(0) \wedge$
(3) $(ii > 0 \Rightarrow (ss(0) = ((\text{closure1}(ops(zero)))[\{rootElement\}]))) \wedge$
(4) $(\forall jj,\ kk)\ .\ (jj \in \mathbb{N}_1 \wedge kk \in \mathbb{N} \wedge kk = jj - 1 \wedge jj < ii \Rightarrow$
$(\exists sc)\ .\ (sc \subseteq state \wedge sc = scan(kk)[ss(kk)] \wedge ss(jj) =$
$(\text{closure1}(ops(jj))[sc])))$

$\Rightarrow$
$(\forall a1,\ l1,\ r1,\ f1,\ ind)\ .\ (type1 \wedge ind \in \mathbb{N} \wedge ind < ii \wedge (a1 \mapsto l1 \mapsto r1 \mapsto f1) \in state \wedge$
$(a1 \mapsto l1 \mapsto r1 \mapsto f1) \in ss(ind) \Rightarrow$
$((l1 \mapsto (sentence \downarrow f1) \uparrow (ind - f1)) \in derivable))$

We prove this with the following theorem. We allow $sentence(a)\ ..\ sentence(b)$ to represent the subsequence beginning at position a of *sentence* and ending at position b. We also allow $(a,\ l,\ r,\ f)$ instead of $(a \mapsto l \mapsto r \mapsto f)$.

Theorem 4.1. *Under conjuncts (1) to (4) of the invariant of Listing 4.4, if* $(a,\ l,\ r,\ f) \in ss(i)$ *and* $i < ii$, *then* $(l \mapsto sentence(f{+}1)\ ..\ sentence(i)) \in derivable$.

Proof. The main ideas for this proof are from [5], but ours is simpler[2].

[2]Earley's uses lookahead, which adds complexity, and also proves an unnecessary property about the start symbol.

Invariant (2) establishes one element, the basis, to exist in the zeroth state set. This state is $(Root \mapsto [] \mapsto rhs \mapsto 0)$; the claim is that the empty sequence can derive the empty sequence, and since *derivable* is a reflexive relation, this is true.

Invariants (3) and (4) posit that all other elements come from a scan, complete or predict step. We assume that all other states satisfy the property, and show that any new state added via one of these operations does so as well. The predictor adds elements whose l component is the empty sequence, so they all satisfy the theorem in the same way that the element of invariant (2) does. The scanner adds $(a,\ l,\ r,\ f)$ to the ith state set, from $(a,\ \text{front}(l),\ \text{last}(l)\ ->\ r,\ f)$ in the $i-1$th state set. We have $\text{last}(l) = sentence(i)$. We know that $(\text{front}(l) \mapsto sentence(f{+}1)\ ..\ sentence(i{-}1)) \in derivable$. From this, and the fact that $\text{last}(l) = sentence(i)$, we have $(l \mapsto sentence(f{+}1)\ ..\ sentence(i)$. The completer adds $(a,\ l,\ r,\ f)$ to the ith state set from $(a,\ \text{front}(l),\ \text{last}(l)\ ->\ r,\ f)$ in state set g, and $(b,\ m,\ [],\ g)$ in state set i. From the definition of completer, $\text{last}(l) = b$, and $(b,\ m) \in productions)$. We have that (I) $(m \mapsto sentence(g{+}1)\ ..\ sentence(i)) \in derivable$ and (II) $(\text{front}(l) \mapsto sentence(f{+}1)\ ..\ sentence(g)) \in derivable$. From (I) we have $(b \mapsto sentence(g{+}1)\ ..\ sentence(i)$. Next, from (II), and the fact that $\text{last}(l) = b$, we have $(\text{front}(l) \leftarrow b \mapsto sentence(f{+}1)\ ..\ sentence(g)\ ^\frown$ $sentence(g{+}1)\ ..\ sentence(i)) \in derivable$. So, $(l \mapsto sentence(f{+}1)\ ..$ $sentence(i)) \in derivable$. □

Next, a proof that if an item exists in state set i and the next symbol derives the next $k \geq 1$ sentence terminals, then the item with the dot moved one position to the right must exist in the $i+k$th state set. The proof obligation is similar to the one above: we assume the part of the invariant we have shown *inc* to preserve, and then show that this property follows. That is, we prove:

(1) $ss \in (0\ ..\ \text{size}(sentence)) \rightarrow \mathbb{P}(state) \wedge ii \in \mathbb{N} \wedge ii \leq \text{size}(sentence) + 1 \wedge$
(2) $rootElement \in ss(0) \wedge$
(3) $(ii > 0 \Rightarrow (ss(0) = ((\text{closure1}(ops(zero)))[\{rootElement\}]))) \wedge$
(4) $(\forall jj,\ kk)\ .\ (jj \in \mathbb{N}_1 \wedge kk \in \mathbb{N} \wedge kk = jj - 1 \wedge jj < ii \Rightarrow$
$(\exists sc)\ .\ (sc \subseteq state \wedge sc = scan(kk)[ss(kk)] \wedge ss(jj) =$
$(\text{closure1}(ops(jj))[sc]))$

$\Rightarrow$

$$(\forall z,\ j,\ a1,\ l1,\ r1,\ f1)\ .\ (type1\ \wedge z \in \mathbb{N} \wedge z < ii\ \wedge j \in \mathbb{N} \wedge$$
$$j \leq z\ \wedge (a1 \mapsto l1 \mapsto r1 \mapsto f1)\ \in state\ \wedge (a1 \mapsto l1 \mapsto r1 \mapsto f1)\ \in ss(j)\ \wedge$$
$$([\ \text{first}\ (r1)] \mapsto ((sentence \downarrow j) \uparrow (z - j)))\ \in derivable \Rightarrow$$
$$((a1 \mapsto (l1 \frown [\ \text{first}\ (r1)]) \mapsto \text{tail}(r1) \mapsto f1))\ \in ss(z))\ \wedge$$

We prove this with two theorems, corresponding to nonempty and empty portions of the input sentence.

Theorem 4.2. *Under conjuncts (1) to (4) of the invariant of Listing 4.4, if* $(a,\ l,\ r,\ f)$ *is in state set* i *and* $(\ \text{first}\ (r) \mapsto sentence(i{+}1)\ ..\ sentence(p))\ \in derivable$ *and* $p < ii$, *then* $(a,\ l\ \leftarrow \text{first}(r),\ \text{tail}\,(r),\ f)$ *is in state set* p.

Proof. The proof is by induction on the number of iterations of *derivable* necessary to show first (r) derives $sentence(i{+}1)\ ..\ sentence(p)$. In the basis, we have zero derivation steps; first $(r)\ = sentence(i{+}1)\ ..\ sentence(i{+}1)\ = sentence(i{+}1)$ in this case. From invariant (3), the scanner must have acted on this state, adding $(a,\ l\ \leftarrow \text{first}(r),\ \text{tail}\,(r),\ f)$ to state set p.

For the induction step, first (r) must be a nonterminal Q, since a step in a derivation replaces a nonterminal by its right-hand-side, and we know there is at least one such step. Since $(Q \mapsto sentence(i{+}1)\ ..\ sentence(p)$, one of its right-hand-sides $q1\ q2\ ...\ q'$ must do so as well. This must be the first step in the derivation, so the right-hand-side uses one less step to derive the input, and therefore so do any of its subderivations. There are also $t0 \leq t1 \leq ... \leq tq'$ with $t0 = i$, $tq' = p$, and $(q1 \mapsto sentence(t0)\ ..\ sentence(t1) \in derivable$, $(q1 \mapsto sentence(t1)\ ..\ sentence(t2)) \in derivable$ and so on, until $(q(q') \mapsto sentence(t(q'{-}1))\ ..\ sentence(t(q'))$. From invariant (3), we know that the predictor adds $(Q,\ [\,],\ q1\ q2\ ..,\ i)$ to state set i. From the inductive hypothesis, $(Q,\ q1,\ q2\ ..,\ i)$ is added to state set $t1$. The inductive hypothesis keeps applying in this way, until we have $(Q,\ q1\ q2\ ..,\ [\,],\ i$ in state set p. From invariant (3), The completer uses this last state to add $(a,\ l\ \leftarrow \text{first}(r),\ \text{tail}\,(r),\ f)$ to state set p. □

Theorem 4.3. *Under conjuncts (1) to (4) of the invariant of Listing 4.4, if* $(a,\ l,\ r,\ f)$ *is in state set* i, $i \leq ii$ *and* $\text{first}\,(r)$ *is nullable, then* $(a,\ l\ \leftarrow \text{first}(r),\ \text{tail}\,(r),\ f)$ *is also in state set* i.

Proof. A right-hand-side R of first (r) must be nullable. We know from invariant (3) that the predictor added (first (r), [] , R, 0) to the state set, and can use an induction on the number of iterations of *derivable* used to find R is nullable. In the basis, R is ϵ, and the completer used first (r), [] , [] , i) to add (a, l $\leftarrow$ first(r), tail (r), f). Otherwise, we have a production of the form (first (r), *r1 r2 .. rk*), whose components are all nullable. Iterating the inductive hypothesis k times shows that (first (r), *r1 r2*, *rk*, [] , i) is added, which the completer used to add the required state. □

We now show the last part of the invariant — the sanity conditions on the elements of the state sets. Since our root element has an empty *l1* and *r1* is the only right-hand-side of *Root*, the root element satisfies the invariant. Anything added by the predictor has the whole right-hand-side of a production in *r1* and an empty *l1*, so such states satisfy the invariant as well. Finally, the scanner and completer add acceptable items because they change the split between *l1* and *r1*, not their concatenation.

Nothing else about this machine must be proved. We have two query operations *getIi* and *accept*, so they cannot violate any invariants. We want *accept* to correspond to whether or not the sentence belongs to the grammar, but this will only emerge when we include this machine into an implementation of *recm*. The other operations add items to *ss*(*ii*) and no invariant says anything about this state set.

4.6 Refinement of the General Recognizer

To use the state sets to implement the CFG recognizer machine, we want *ii* to attain a value of size(*sentence*) + 1, which is its maximum value allowed by the invariant. To do this, we initialise *ii* to 0, and call *inc* in a loop. We use *nextOps* and *nextScan* in order to make the precondition of *inc* true prior to its call. After this, we return the same value that *accept* returns. The implementation machine resulting from these ideas is below.

Listing 4.5: Recognizer Implementation

IMPLEMENTATION *reci*
REFINES *recm*
SEES *gram*, *sent*
IMPORTS *stateSets*

OPERATIONS
ans ⟵ *isSentence* =
VAR *jj*, *len* **IN**
init; *jj* := 0; *len* ⟵ *senLength*;
WHILE *jj* ≤ *len* **DO**
nextScan; *nextOps*; *inc*; *jj* := *jj* + 1
INVARIANT
jj ∈ ℕ ∧ *jj* = *ii* ∧ *jj* ≤ size(*sentence*) + 1 ∧
len = size(*sentence*) ∧ *len* ∈ ℕ ∧
(*jj* = 0 ⇒ *ss*(0) = *rootElement*)
VARIANT *len* − *jj*
END;
ans ⟵ *accept*
END
END

4.6.1 Correctness

We only have to show that this implementation of *isSentence* is a valid refinement of the abstract version; this is proof obligation (3) in Figure 2.12. The other proof obligations concern constants and initialisations, of which we have none. We use the loop invariant upon termination of the loop, so the first task is to prove the loop correct. We begin with the fact that the loop invariant is established prior to the first iteration (proof obligation (5) of Figure 2.13). We use *initSets* as shorthand for the function that *init* assigns to *ss*.

[*init*; *jj* := 0; *len* ⟵ *senLength*]
jj ∈ ℕ ∧ *jj* = *ii* ∧ *jj* ≤ size (*sentence*) + 1 ∧ *len* = size (*sentence*) ∧ *len* ∈ ℕ ∧
(*jj* = 0 ⇒ *ss*(0) = *rootElement*)

= {*WPA*, *WPS*}
[*init*]
0 ∈ ℕ ∧ 0 = *ii* ∧ 0 ≤ size (*sentence*) + 1 ∧ size (*sentence*) = size (*sentence*) ∧ size (*sentence*) ∈ ℕ ∧ *ss*(0) = *rootElement*

$= \{expand\ init,\ WPM\}$
$0 \in \mathbb{N} \wedge 0 = 0 \wedge 0 \leq \text{size}\ (sentence) + 1 \wedge \text{size}\ (sentence) = \text{size}\ (sentence) \wedge$
$\text{size}\ (sentence) \in \mathbb{N} \wedge initSets(0) = rootElement$

$=$
$true$

Next, we show proof obligation (1) of Figure 2.13. Assuming that *inc* succeeds, the fact that the loop maintains the invariant is readily apparent because then both ii and jj are incremented. For *inc* to succeed, its precondition must be true, and it is once we call *nextScan* and *nextOps*. We let *closeRoot* represent the closure of the predictor and completer on the root element, $ops(i)$ the result of the closure and predictor on state set i and $scan(i)$ the result of the scanner.

$[nextScan;\ nextOps;\ inc;\ jj := jj + 1]$
$jj \in \mathbb{N} \wedge jj = ii \wedge jj \leq \text{size}\ (sentence) + 1 \wedge len = \text{size}\ (sentence) \wedge len \in \mathbb{N}$

$= \{WPA,\ WPS\}$
$[nextScan;\ nextOps;\ inc]$
$jj + 1 \in \mathbb{N} \wedge jj + 1 = ii \wedge jj + 1 \leq \text{size}\ (sentence) + 1 \wedge len =$
$\text{size}\ (sentence) \wedge len \in \mathbb{N}$

$= \{expand\ inc\ body\}$
$[nextScan;\ nextOps;\ \textbf{PRE}\ ss(0) = CLOSEROOT \wedge (ii > 0 \Rightarrow ss(ii) =$
$OPS(SCAN(ii{-}1))) \wedge ii \leq \text{size}\ (sentence)\ \textbf{THEN}\ ii := ii + 1]$
$jj + 1 \in \mathbb{N} \wedge jj + 1 = ii \wedge jj + 1 \leq \text{size}\ (sentence) + 1 \wedge len =$
$\text{size}\ (sentence) \wedge len \in \mathbb{N}$

$= \{WPP,\ WPS\}$
$[nextScan;\ nextOps]$
$ss(0) = CLOSEROOT \wedge (ii > 0 \Rightarrow ss(ii) = OPS(SCAN(ii{-}1))) \wedge ii \leq$
$\text{size}\ (sentence) \wedge$
$jj + 1 \in \mathbb{N} \wedge jj + 1 = ii + 1 \wedge jj + 1 \leq \text{size}\ (sentence) + 1 \wedge len =$
$\text{size}\ (sentence) \wedge len \in \mathbb{N}$

In other words, running *nextScan* and *nextOps* has to establish the given postcondition; there are two cases. If ii is 0, then from the loop invariant we know that $ss(0)$

is just the rootElement. In this case, the scanner does nothing and the predictor performs the closure on the set, which is exactly what is required. Otherwise, we first perform the scanner operation, so we have the intermediate assertion that the current state set is the result of applying the scanner to the previous one. Next, *nextOps* does the closure on this resulting set, satisfying the second conjunct of the postcondition. We know that $ii \leq \text{size}(sentence)$ because $ii = jj$, and jj is constrained to be at most the size of the sentence by the loop guard.

Upon termination of the loop, we have that $jj = \text{size}(sentence) + 1$, and so by the invariant, ii also holds this value. With this knowledge, we can prove that the result of the refined *isSentence* (i.e. the return value of *accept*) returns the same value as the abstract operation.

We can show this equivalence in two parts. First, if the abstract *isSentence* returns *true*, then the sentence is derivable from the root. This means the sentence is derivable from the one right-hand-side of the root. If the refined version returns *true*, we know that the element $(Root \mapsto rhs \mapsto [] \mapsto 0)$ is in the state set corresponding to the size of the sentence, and from Theorem 4.1 we know that *rhs* must therefore derive the whole sentence.

Second, if the sentence is derivable from the root, then the abstract *isSentence* returns *true*. The refinement does the same, as follows. We know the root element is in $ss(0)$, and that the first symbol of *rhs* derives some portion of the input sentence, say the first $k \geq 0$ terminals. Then, by Theorem 4.2, we know that the same item, with the dot moved one position to the right, will be in $ss(k)$. We can repeat this argument until we have the acceptance element in $ss(\text{size}(sentence))$ which causes *true* to be returned.

4.6.2 Termination Argument

Proof obligations (3) and (4) concern the variant: (3) is discharged because $jj \leq len$ so $len - jj \in \mathbb{N}$. Obligation (4) is immediate because jj is incremented on every iteration, thus decreasing the variant. It is also necessary to show that the closure operations in the abstract machine terminate, since if they do not, one iteration of the while loop may never finish. This is especially important in our later refinement where loops are used to simulate these closures. There is a bound on the number of items that can be added to the state sets, though, and since the closure just adds new items, eventually we must reach a fixed-point. Assume we have p productions, the

longest associated right-hand-side being p'. Then for state set i there are a maximum of $p(p'+1)(i+1)$ items. The first three components of a state combine to form a production, of which there are p, and we can perform the split between the left and right sides in $p'+1$ ways. The last component is constrained to be between 0 and i, so it has $i+1$ possibilities.

4.7 Result of Model Checking

ProB successfully analyzed the *sent*, *gram* and *recm* machines, finding no inconsistencies. The state space of the *stateSets* machine could not be explored in its entirety, however, and we assume this to be a result of the complexity of invariants which it maintains. This general problem filters down to machines in the next chapter, by their virtue of refining the already-complicated *stateSets*.

Chapter 5

Refining to Lists

5.1 Overview of Development

In this chapter, we consider a possible implementation of Earley's algorithm by using lists (B sequences) to model the sets of the previous chapter. We first show that a naive implementation would be incorrect, but only when ϵ-productions are present. A slight modification of this proposal yields the Aycock-Horspool version, which we specify and prove here, and for which we give an executable implementation in the next chapter.

5.2 Epsilon-Productions and Lists

We now move from using sets of states to lists of states, in order to come up with an efficient method for implementing the recognizer. The most naive list-processing version would insert the basis element, and then move linearly through the list applying the Earley operations to the states. New items would be added to the end of the list, but only if they were not present already. If we indiscriminantly added duplicates, we would run into infinite loops because we would process the same item over and over. For example, if a grammar contained the rule $A \rightarrow A$, then the predictor would be applicable to a state made from this item, resulting in the same state being added. However, again we could apply the predictor, thus adding and processing this state infinitely often. Besides, we previously used sets, so we know we can do without duplicates.

This naive list-processing version does work for grammars with no ϵ-productions.

$$S \rightarrow AA$$
$$A \rightarrow \epsilon$$

$s(0)$

(1)	$S \rightarrow \bullet AA$, 0	Initialisation
(2)	$A \rightarrow \bullet$, 0	Predictor (1)
(3)	$S \rightarrow A \bullet A$, 0	Completer (2)

Figure 5.1: Naive list-processing Earley, failing to detect the root deriving the empty string.

We can traverse state list $i-1$, and apply the scanner to all of its elements to give the "kernel" of list i. From here, we must ensure that we have all possible items resulting from running the predictor and completer over and over (previously accomplished by taking the closure). Our linear scan of the list, then, cannot be permitted to pass by a state which could later be used by an operation to generate new items. Assume we are about to scan past a state to which the predictor is applicable. We know that the completer cannot be used on this state, since there are symbols to the right of the dot. Also, running the predictor again at some later point can add no new items, since the action of the predictor does not in any way depend on the current contents of the state list. We can therefore safely run the predictor and move to the next item. Now, assume we are about to move past an item $(a,\ l,\ r,\ f)$ in state list i which should be acted upon by the completer. If $f < i$, then we are searching a state list which we know can never have any new items added to it, since it has already been processed. Therefore, once we run the completer, we can skip past this item since there is no chance a new item can materialize in list f that could have been used here. On the other hand, if $f = i$, we have a problem: list i is still being constructed at the point that the completer is called. A new state could thus be added that would have resulted in our complete step deriving a new state from it. Unfortunately, at this time, we would have already executed the completer on the current item, and will not run the completer on it again. We therefore do not have the closure of the predictor and completer in this case. Figure 5.1 contains a grammar with an ϵ-production and the ill-starred progression of a naive list-processor not recognizing the acceptance of a sentence. The reason is that we could still use the completer on (2), using (3) to add

$$[S \rightarrow AA\bullet, 0]$$

to the list.

5.3 A List Machine

Our implementation of state sets performs several operations on state lists, including testing membership of a state in a list, adding an item to a specified list, and extracting information about the states that have already been added. We separate this list data structure and its associated operations into a machine, which we import into the state set implementation. It is shown in Listing 5.1.

Listing 5.1: State Lists Machine

MACHINE *listData*
SEES *sent*
DEFINITIONS
$state == (\mathbb{N} \times \mathbb{N} \times \mathbb{N})$;
$indef == lists := (0..\ \text{size}\ (sentence)) \times \{[]\}$

VARIABLES *lists*
INVARIANT
$lists \in (0\ ..\ \text{size}(sentence)) \rightarrow \text{iseq}(state)$

INITIALISATION *indef*

OPERATIONS
$initial = indef$;

$addState(r,\ j,\ f,\ i) =$
PRE $r \in \mathbb{N} \wedge j \in \mathbb{N} \wedge f \in \mathbb{N} \wedge i \in 0\ ..\ \text{size}(sentence)$ **THEN**
IF $(r \mapsto j \mapsto f) \notin \text{ran}(lists(i))$ **THEN**
$lists\,(i) := lists\,(i) \leftarrow (r \mapsto j \mapsto f)$
END
END;

$ans \longleftarrow numStates\ (i) =$
PRE $i \in 0\ ..\ \text{size}(sentence)$ **THEN**
$ans := \text{size}(\ lists\ (i))$
END;

$kill\ (i)\ =$
PRE $i \in 0 .. \text{size}(sentence)$ **THEN**
$lists\,(ii)\ :=\ []$
END;

$ans \longleftarrow getR\,(i, j)\ =$
PRE $i \in 0 .. \text{size}(sentence) \wedge j \in \mathbb{N} \wedge j \leq \text{size}(\ lists\,(i))$ **THEN**
ANY rr, jj, ff
WHERE $rr \in \mathbb{N} \wedge jj \in \mathbb{N} \wedge ff \in \mathbb{N} \wedge lists(i)(j)\ = (rr \mapsto jj \mapsto ff)$
THEN $ans := rr$
END
END;

$ans \longleftarrow getJ\,(i, j)\ =$
PRE $i \in 0 .. \text{size}(sentence) \wedge j \in \mathbb{N} \wedge j \leq \text{size}(\ lists\,(i))$ **THEN**
ANY rr, jj, ff
WHERE $rr \in \mathbb{N} \wedge jj \in \mathbb{N} \wedge ff \in \mathbb{N} \wedge lists(i)(j)\ = (rr \mapsto jj \mapsto ff)$
THEN $ans := jj$
END
END;

$ans \longleftarrow getF\,(i, j)\ =$
PRE $i \in 0 .. \text{size}(sentence) \wedge j \in \mathbb{N} \wedge j \leq \text{size}(\ lists\,(i))$ **THEN**
ANY rr, jj, ff
WHERE $rr \in \mathbb{N} \wedge jj \in \mathbb{N} \wedge ff \in \mathbb{N} \wedge lists(i)(j)\ = (rr \mapsto jj \mapsto ff)$
THEN $ans := ff$
END
END;

$ans \longleftarrow inList(r, j, f,\ i)\ =$
PRE $r \in \mathbb{N} \wedge j \in \mathbb{N} \wedge f \in \mathbb{N} \wedge i \in 0 .. \text{size}(sentence)$ **THEN**
$ans\ :=\ \text{bool}((r \mapsto j \mapsto f)\ \in \text{ran}(lists(i)))$
END

END

We have changed our representation of a state, from a quadruple involving grammar symbols and productions, to one simply comprising three naturals. Using the ordering imposed on the grammar in the grammar machine, the first natural represents the production to which the state belongs. The second natural tells us where in the right-hand-side we are in processing the item; it corresponds with the position of the split between l and r in the previous representation. The third natural conveys the same information as the previously-used f component. The reason for the change here is that, in moving towards an implementation, it is not practical to carry around the grammar productions in their entirety. When inspecting or adding new states, it is much more convenient to use natural numbers than sequences of grammar symbols.

5.3.1 Correctness

The invariant here is that *lists* is a total function from 0 .. size(*sentence*) to injective sequences of states. The initialisation establishes this (proof obligation (3) in Figure 2.10), because $(0 \,..\, \mathrm{size}(sentence)) \times \{[]\}$ is a type-correct value for *lists*. For *addState*, we calculate the weakest precondition to establish the invariant (proof obligation (4) in Figure 2.10):

$$
\begin{array}{l}
[\textbf{IF } (r \mapsto j \mapsto f) \notin \mathrm{ran}(lists(i)) \textbf{ THEN} \\
\quad lists(i) := lists(i) \leftarrow (r \mapsto j \mapsto f) \\
\textbf{END}] \\
\quad lists \in (0 \,..\, \mathrm{size}(sentence)) \rightarrow \mathrm{iseq}(state)
\end{array}
$$

$$
\begin{array}{l}
= \{WPI\} \\
((r \mapsto j \mapsto f) \notin \mathrm{ran}(lists(i)) \Rightarrow \\
[\, lists(i) := lists(i) \leftarrow (r \mapsto j \mapsto f)] \\
\quad lists \in (0 \,..\, \mathrm{size}(sentence)) \rightarrow \mathrm{iseq}(state)) \wedge \\
((r \mapsto j \mapsto f) \in \mathrm{ran}(lists(i)) \Rightarrow \\
\quad lists \in (0 \,..\, \mathrm{size}(sentence)) \rightarrow \mathrm{iseq}(state))
\end{array}
$$

$$
\begin{array}{l}
= \{WPA\} \\
((r \mapsto j \mapsto f) \notin \mathrm{ran}(lists(i)) \Rightarrow \\
\quad lists \lhd\!\!+ \{i \mapsto lists(i) \leftarrow (r \mapsto j \mapsto f)\} \in (0 \,..\, \mathrm{size}(sentence)) \rightarrow \\
\mathrm{iseq}(state)) \wedge \\
((r \mapsto j \mapsto f) \in \mathrm{ran}(lists(i)) \Rightarrow
\end{array}
$$

$$lists \in (0 \,..\, \mathrm{size}(sentence)) \rightarrow \mathrm{iseq}(state))$$

The second conjunct is exactly the invariant; the first follows from the invariant because all sequences are injective, and the new element does not exist in the sequence before it is added.

5.4 State List Refinement

We want state lists to behave exactly as state sets, so that they can be used in the *reci* recognizer machine. This leaves us little choice for the invariant: the state sets must be somehow "equal to" the contents of the state lists. Since we've changed the way in which states are represented, we cannot have direct equality. What we can have, though, is equality in the sense that, if we translate between representations, the lists and sets are equal. We define a total bijection which does this conversion between our three naturals representation and quadruples notation:

$$\begin{array}{l} rep == \{xx, yy \mid \\ \quad (\exists rr, jj, ff, a, l, r, f) \cdot \\ \quad (rr \in \mathbb{N} \land jj \in \mathbb{N} \land ff \in \mathbb{N} \land a \in Nonterminals \land l \in \mathrm{seq}(Symbols) \land r \in \\ \mathrm{seq}(Symbols) \land f \in \mathbb{N} \land \\ \quad xx = (rr \mapsto jj \mapsto ff) \land yy = (a \mapsto l \mapsto r \mapsto f) \land \\ \quad a = ls(rr) \land l = (rs(rr) \uparrow jj) \land r = (rs(rr) \downarrow jj) \land f = ff)\}; \end{array}$$

This embodies the idea that rr, the first component, is a natural number representing a production. The left side of rr should be a. We want the jj component to represent the portion before the dot, which is the same role that l has. Similarly, r represents the portion after the dot, which corresponds to everything following jj.

The initialisation first empties all state lists, then adds the root element. This corresponds with the abstract operation of having all sets empty except the first, where the root element is placed. The *accept* operation is the same as its abstract counterpart, except it is evaluated on the new representation for states.

To refine *nextScan*, we use a loop to iterate over the elements of state set $i-1$. When we come to a state whose next symbol matches the next symbol in the sentence, we add the state, with the dot moved one position to the right, to state set i. The loop invariant is that the contents of the current state set are the elements we get by running the scanner on the first *count* elements of the previous set, where *count* keeps our position in the loop. This corresponds with the destructive abstract *nextScan*,

whose result is always a set of the scanned items, regardless of what was previously in the set.

Refining *nextOps* amounts to iterating through the state list, applying predictor and completer to the states. At the end of this loop, we want the state list to have the elements that the corresponding state set started with, plus those added by the closure of the predictor and completer. At the end of the loop, we have visited all items, and claim that predictor and completer working on the whole list can add nothing. Another part of the invariant claims that these are the only states in the lists — that no rogue states have been added in some other way. The machine is in Listing 5.2.

Listing 5.2: State Lists Refinement

IMPLEMENTATION *stateLists*
REFINES *stateSets*
SEES *sent, gram*
IMPORTS *listData*
DEFINITIONS

$$state == (\mathbb{N} \times \mathbb{N} \times \mathbb{N});$$
$$Nonterminals == (1 \mathinner{..} numNT);$$
$$Terminals == (numNT + 1 \mathinner{..} numNT + numT);$$
$$Symbols == (Terminals \cup Nonterminals);$$
$$Root == 1;$$
$$MaxLength == (\max(\{x \mid (\exists y) \cdot (y \in \mathrm{ran}(productions) \land x = \mathrm{size}(y))\}));$$
$$rep == \{xx, yy \mid (\exists rr, jj, ff, a, l, r, f) \cdot (rr \in \mathbb{N} \land jj \in \mathbb{N} \land ff \in \mathbb{N} \land a \in Nonterminals \land l \in \mathrm{seq}(Symbols) \land r \in \mathrm{seq}(Symbols) \land f \in \mathbb{N} \land xx = (rr \mapsto jj \mapsto ff) \land yy = (a \mapsto l \mapsto r \mapsto f) \land a = ls(rr) \land l = (rs(rr) \uparrow jj) \land r = (rs(rr) \downarrow jj) \land f = ff)\};$$

$$scan(ii) == \{xx, yy \mid (\exists r1, j1, f1, r2, j2, f2) \cdot (r1 \in \mathbb{N} \land j1 \in \mathbb{N} \land f1 \in \mathbb{N} \land r2 \in \mathbb{N} \land j2 \in \mathbb{N} \land f2 \in \mathbb{N} \land xx = (r1 \mapsto j1 \mapsto f1) \land yy = (r2 \mapsto j2 \mapsto f2) \land rs(r1)(j1+1) = sentence(ii) \land r2 = r1 \land j2 = j1 + 1 \land f2 = f1)\};$$

$$
\begin{aligned}
&predict(ii,\ upto) \ == \{xx,\ yy \mid \\
&\quad (\exists r1,\ j1,\ f1,\ r2,\ j2,\ f2)\ . \\
&\quad (r1 \in \mathbb{N} \land r1 \leq upto \land j1 \in \mathbb{N} \land f1 \in \mathbb{N} \land r2 \in \mathbb{N} \land j2 \in \mathbb{N} \land f2 \in \mathbb{N} \land \\
&\quad xx = (r1 \mapsto j1 \mapsto f1) \land yy = (r2 \mapsto j2 \mapsto f2) \land \\
&\quad rs(r1)(j1{+}1) = ls(r2) \land j2 = 0 \land f2 = ii)\};
\end{aligned}
$$

$$
\begin{aligned}
&completer(upto) \ == \{xx,\ yy \mid \\
&\quad (\exists r1,\ j1,\ f1,\ r2,\ j2,\ f2,\ r3,\ j3,\ f3)\ . \\
&\quad (r1 \in \mathbb{N} \land j1 \in \mathbb{N} \land f1 \in \mathbb{N} \land r2 \in \mathbb{N} \land j2 \in \mathbb{N} \land f2 \in \mathbb{N} \land r3 \in \mathbb{N} \land j3 \in \mathbb{N} \land \\
&f3 \in \mathbb{N} \land \\
&\quad xx = (r1 \mapsto j1 \mapsto f1) \land yy = (r3 \mapsto j3 \mapsto f3) \land \\
&\quad j1 = \mathrm{size}(rs(r1)) \land j2 < \mathrm{size}(rs(r2)) \land \\
&\quad (r2 \mapsto j2 \mapsto f2) \in \mathrm{ran}(lists(f1)) \land \\
&\quad (\exists index)\ .\ (index \in \mathbb{N} \land index \leq upto \land lists(f1)(index) = (r2 \mapsto j2 \mapsto f2)) \land \\
&\quad r3 = r2 \land j3 = j2 + 1 \land f3 = f2)\};
\end{aligned}
$$

$$
\begin{aligned}
&complete \ == \{xx,\ yy \mid \\
&\quad (\exists r1,\ j1,\ f1,\ r2,\ j2,\ f2,\ r3,\ j3,\ f3)\ . \\
&\quad (r1 \in \mathbb{N} \land j1 \in \mathbb{N} \land f1 \in \mathbb{N} \land r2 \in \mathbb{N} \land j2 \in \mathbb{N} \land f2 \in \mathbb{N} \land r3 \in \mathbb{N} \land j3 \in \mathbb{N} \land \\
&f3 \in \mathbb{N} \land \\
&\quad xx = (r1 \mapsto j1 \mapsto f1) \land yy = (r3 \mapsto j3 \mapsto f3) \land \\
&\quad j1 = \mathrm{size}(rs(r1)) \land j2 < \mathrm{size}(rs(r2)) \land \\
&\quad (r2 \mapsto j2 \mapsto f2) \in \mathrm{ran}(lists(f1)) \land \\
&\quad r3 = r2 \land j3 = j2 + 1 \land f3 = f2)\};
\end{aligned}
$$

$$
ops(ii,\ upto) \ == (predict(ii,\ upto) \cup complete);
$$

$$
\begin{aligned}
&prevItem \ == (\forall uu)\ .\ (uu \in state \land uu \in \mathrm{ran}(lists(ii)) \land uu \notin rep^{-1}[ss(ii)] \land \\
&\quad \neg\,(ii = 0 \land uu = (Root \mapsto 0 \mapsto 0)) \Rightarrow uu \in \mathrm{closure1}(ops\ (ii,\ numR))[\mathrm{ran}(lists(ii))]);
\end{aligned}
$$

$$
\begin{aligned}
&restEqual \ == (\forall i)\ .\ (i \in 0\ ..\ \mathrm{size}(sentence) \land i \neq ii \Rightarrow ss(i) = \\
&rep[\mathrm{ran}(\ lists\ (i\ ))]);
\end{aligned}
$$

$$
setInList \ == rep^{-1}[ss(ii)] \subseteq \mathrm{ran}(lists(ii));
$$

$directlyDerivable$ ==
  $\{xx,\ yy \mid xx \in \text{seq}(Symbols) \wedge yy \in \text{seq}(Symbols) \wedge$
  $(\exists \mu,\ \sigma,\ \nu,\ \tau).$
  $(\mu \in \text{seq}(Symbols) \wedge \sigma \in Symbols \wedge$
  $\nu \in \text{seq}\ (Symbols) \wedge \tau \in \text{seq}(Symbols) \wedge$
  $xx = (\mu \frown [\sigma] \frown \nu) \wedge (yy = \mu \frown \tau \frown \nu) \wedge (\sigma \mapsto \tau \in productions))\}$;
$derivable$ == (closure1($directlyDerivable$) $\cup$ id(seq($Symbols$)))

CONCRETE_VARIABLES *ii*, *numR*

INVARIANT
$(\forall i)$. $(i \in 0 .. \text{size}(sentence) \Rightarrow ss(i) = rep[\text{ran}(\ lists\ (i\ ))])$ $\wedge$
$numR = \text{card}\ (productions)$

INITIALISATION
  BEGIN *initial*; $ii := 0$; *addState* (*Root*, 0, 0, 0); $numR \longleftarrow numRules$ **END**

OPERATIONS
init =
  BEGIN *initial*; $ii := 0$; *addState* (*Root*, 0, 0, 0); $numR \longleftarrow numRules$ **END**;

inc = $ii := ii + 1$;

$bb \longleftarrow getIi = bb := ii$;

nextScan =
  IF $ii > 0$ **THEN**
  VAR *count*, *len*, *rr*, *jj*, *ff*, *rLen*, *symbol*, *match* **IN**
    $count := 0$; $len \longleftarrow numStates\ (ii{-}1)$; *kill* (*ii*);
    WHILE $count < len$ **DO**
      $count := count + 1$;
      $rr \longleftarrow getR(ii{-}1,\ count)$;
      $jj \longleftarrow getJ(ii{-}1,\ count)$;
      $ff \longleftarrow getF(ii{-}1,\ count)$;
      $rLen \longleftarrow ruleLength\ (rr)$;

```
      IF jj < rLen THEN
        symbol ⟵ getRS (rr, jj+1);
        match ⟵ senGet (ii);
        IF symbol = match THEN
          addState (rr, jj+1, ff, ii)
        END
      END
      INVARIANT
      count ∈ ℕ ∧ count ≤ len ∧ len = size( lists ( ii−1)) ∧
      ran( lists ( ii)) = scan( ii )[( lists ( ii −1))[1.. count]] ∧
      (∀i) . (i ∈ 0 .. size(sentence) ∧ i ≠ ii ⇒ ss(i) = rep[ran( lists (i))])
      VARIANT len − count
    END
  END
  END;

nextOps =
VAR count, inner, len, len2, rLen, rr, jj, ff, r, j, f, symbol, match, nl IN
  count := 0; len ⟵ numStates (ii);
  WHILE count < len DO
    count := count + 1;
    rr ⟵ getR(ii, count);
    jj ⟵ getJ(ii, count);
    ff ⟵ getF(ii, count);
    rLen ⟵ ruleLength (rr);
    IF jj < rLen THEN
      inner := 0;
      WHILE inner < numR DO
        inner := inner + 1;
        match ⟵ getLS (inner);
        symbol ⟵ getRS (rr, jj+1);
        IF symbol = match THEN
          addState (inner, 0, 0, ii)
        END
        INVARIANT
```

```
    inner ∈ ℕ ∧ inner ≤ numR ∧
    (rr ↦ jj ↦ ff) = lists(ii)(count) ∧ setInList ∧
    predict(ii, inner)[{(rr ↦ jj ↦ ff)}] ⊆ ran(lists(ii)) ∧
    prevItem ∧ restEqual
    VARIANT numR − inner
  END;
  symbol ⟵ getRS (rr, jj+1);
  nl ⟵ nullable (symbol);
  IF nl = TRUE THEN
    addState (rr, jj+1, ff, ii)
  END
END;
IF jj = rLen THEN
  symbol ⟵ getLS (rr);
  inner := 0; len2 ⟵ numStates(ff);
  WHILE inner < len2 DO
    inner := inner + 1;
    r ⟵ getR(ff, inner);
    j ⟵ getJ(ff, inner);
    f ⟵ getF(ff, inner);
    rLen ⟵ ruleLength (r);
    IF j < rLen THEN
      match ⟵ getRS (inner, j+1);
      IF symbol = match THEN
        addState (inner, j + 1, f, ii)
      END
    END;
    len2 ⟵ numStates (ff)
    INVARIANT
    inner ∈ ℕ ∧ inner ≤ len2 ∧ len2 = size( lists (ff)) ∧
    (rr ↦ jj ↦ ff) = lists(ii)(count) ∧ setInList ∧
    completer(inner)[{(rr ↦ jj ↦ ff)}] ⊆ ran(lists(ii)) ∧
    prevItem ∧ restEqual
    VARIANT (MaxLength × (ff + 1) × (card (productions) + 1)) − inner
  END
```

END;
len ⟵ *numStates* (*ii*)
INVARIANT
count ∈ ℕ ∧ *count* ≤ *len* ∧ *len* = size(*lists* (*ii*)) ∧ *setInList* ∧
(∀*r*, *j*, *f*). (*r* ∈ ℕ ∧ *j* ∈ ℕ ∧ *f* ∈ ℕ ∧ (*r* ↦ *j* ↦ *f*) ∈ *lists*(*ii*)[1.. *count*] ⇒
((*ops*(*ii* , *numR*)[{(*r* ↦ *j* ↦ *f*)}] ⊆ ran(*lists*(*ii*))) *or*
((*f* = *ii*) ∧ ((∀*rr*, *jj*, *ff*) . (*rr* ∈ ℕ ∧ *jj* ∈ ℕ ∧ *ff* ∈ ℕ ∧
(*rr* ↦ *jj* ↦ *ff*) ∈ (*ops*(*ii*, *numR*)[{(*r* ↦ *j* ↦ *f*)}]) ∧
(*rr* ↦ *jj* ↦ *ff*) ∉ ran(*lists*(*ii*)) ⇒
(([*rs*(*rr*)(*jj*)] ↦ []) ∈ *derivable*) ∧
(*rr* ↦ (*jj*−1) ↦ *ff*) ∈ ran(*lists*(*ii*)))))))) ∧

(∀*r*, *j*, *f*) . (*r* ∈ ℕ ∧ *j* ∈ ℕ ∧ *f* ∈ ℕ ∧ (*r* ↦ *j* ↦ *f*) ∈ *lists*(*ii*)[1.. *count*] ∧
([*rs*(*r*)(*j*+1)] ↦ []) ∈ *derivable* ⇒ (*r* ↦ *j*+1 ↦ *f*) ∈ ran(*lists*(*ii*))) ∧
prevItem ∧ *restEqual*
VARIANT (*MaxLength* × (*ii* + 1) × (card (*productions*) + 1)) − *count*
END
END;

bb ⟵ *accept* =
VAR *ruleLen*, *senLen* **IN**
ruleLen ⟵ *ruleLength*(*Root*);
senLen ⟵ *senLength*;
bb ⟵ *inList*(*Root*, *ruleLen*, 0, *senLen*)
END

END

5.4.1 Correctness

Refinement of Initialisation

We calculate the weakest precondition for the abstract initialisation to establish the linking invariant (proof obligation (2) in Figure 2.12):

[*ss*, *ii* := {*xx*, *yy* | *xx* ∈ (0 .. size(*sentence*)) ∧ *yy* ∈ ℙ(*state*) ∧

$(xx = 0 \Rightarrow yy = \{rootElement\}) \wedge (xx \neq 0 \Rightarrow yy = \{\})\}, 0]$
$(\forall i) \cdot (i \in 0 .. \text{size}(sentence) \Rightarrow ss(i) = rep[\text{ran}(lists(i))]) \wedge$
$numR = \text{card}(productions)$

$= \{WPM\}$
$(\forall i) \cdot (i \in 0 .. \text{size}(sentence) \Rightarrow$
$\{xx, yy \mid xx \in (0 .. \text{size}(sentence)) \wedge yy \in \mathbb{P}(state) \wedge$
$(xx = 0 \Rightarrow yy = \{rootElement\}) \wedge (xx \neq 0 \Rightarrow yy = \{\})\}(i) =$
$rep[\text{ran}(lists(i))]) \wedge$
$numR = \text{card}(productions)$

Next we calculate the weakest precondition of the implementation to establish this postcondition:

$[initial; ii := 0; addState(Root, 0, 0, 0); numR \longleftarrow numRules]$
$(\forall i) \cdot (i \in 0 .. \text{size}(sentence) \Rightarrow$
$\{xx, yy \mid xx \in (0 .. \text{size}(sentence)) \wedge yy \in \mathbb{P}(state) \wedge$
$(xx = 0 \Rightarrow yy = \{rootElement\}) \wedge (xx \neq 0 \Rightarrow yy = \{\})\}$
$(i) = rep[\text{ran}(lists(i))]) \wedge$
$numR = \text{card}(productions)$

$= \{WPS\}$
$[initial; ii := 0]$
$(\forall i) \cdot (i \in 0 .. \text{size}(sentence) \Rightarrow$
$\{xx, yy \mid xx \in (0 .. \text{size}(sentence)) \wedge yy \in \mathbb{P}(state) \wedge$
$(xx = 0 \Rightarrow yy = \{rootElement\}) \wedge (xx \neq 0 \Rightarrow yy = \{\})\}$
$(i) = rep[\text{ran}($
$lists \lhd\!\!+ \{0 \mapsto (lists\ (0) \leftarrow (Root \mapsto 0 \mapsto 0))$
$(i))]) \wedge$
$numRules = \text{card}(productions)$

$= \{WPS\}$
$(\forall i) \cdot (i \in 0 .. \text{size}(sentence) \Rightarrow$
$\{xx, yy \mid xx \in (0 .. \text{size}(sentence)) \wedge yy \in \mathbb{P}(state) \wedge$
$(xx = 0 \Rightarrow yy = \{rootElement\}) \wedge (xx \neq 0 \Rightarrow yy = \{\})\}$
$(i) = rep[\text{ran}($

$(0 \;..\; \mathrm{size}(sentence)) \times \{[]\}$
$\lhd\!\!+ \{0 \mapsto (lists\ (0) \leftarrow (Root \mapsto 0 \mapsto 0))$
$(i))]) \wedge$
$numRules = \mathrm{card}\ (productions)$

$= \{\ definition\ \ of\ rep\}$
$true$

The last step follows because all state sets and lists besides 0 are empty. We are thus claiming only that $(Root \mapsto 0 \mapsto 0)$ represents *rootElement*. We find that $rep((Root \mapsto 0 \mapsto 0))$ is indeed the quadruple $(Root \mapsto [] \mapsto rs(Root) \mapsto 0)$. Since the definitions of *init* are the same as the initialisations, we have a proof for this operation as well.

Refinement of Accept

The *accept* operation is a query, so we just have to verify that the outputs match (proof obligation 3 in Figure 2.12). Renaming the output of the refined *accept* to *bb*', we have:

[**VAR** *ruleLen*, *senLen* **IN**
$ruleLen \longleftarrow ruleLength(Root)$;
$senLen \longleftarrow senLength$;
$bb \longleftarrow inList(Root, ruleLen, 0, senLen)$
END]
$\mathrm{bool}\ ((Root \mapsto rhs \mapsto [] \mapsto 0) \in ss(\mathrm{size}(sentence))) = bb'$

$= \{WPV,\ WPC\}$
$[ruleLen \longleftarrow ruleLength(Root);\ senLen \longleftarrow senLength]$
$\mathrm{bool}\ ((Root \mapsto rhs \mapsto [] \mapsto 0) \in ss(\mathrm{size}(sentence))) =$
$inList(Root,\ ruleLen,\ 0,\ senLen)$

$= \{WPC\}$
$[ruleLen \longleftarrow ruleLength(Root)]$
$\mathrm{bool}\ ((Root \mapsto rhs \mapsto [] \mapsto 0) \in ss(\mathrm{size}(sentence))) =$
$inList(Root,\ ruleLen,\ 0,\ \mathrm{size}\,(sentence))$

$= \{WPC\}$
bool $((Root \mapsto rhs \mapsto [] \mapsto 0) \in ss(\text{size}(sentence))) =$
$inList(Root, \text{size}(rs(Root)), 0, \text{size}(sentence))$

$= \{since\ sets\ and\ lists\ are\ equal\ under\ rep\}$
$true$

Refinement of Scanner

We can make some observations prior to the proof to make it less reprehensible. First, the two scanners — abstract and concrete — perform the same function on their respective state representations. Consider an item (rr, jj, ff) existing in a state list, and let (a, l, r, f) be the corresponding item in the state set. From the definition of the scanner on triples, if $jj{+}1$ matches the next symbol in the sentence, then we add $(rr, jj{+}1, ff)$. To perform the same function, the abstract scanner should add the corresponding item to its state set. From the definition of *rep*, this item is $(a, l \leftarrow \text{first}(r), \text{tail}(r), f)$, which is precisely what the abstract scanner would add when operating on (a, l, r, f). Second, if all state lists and sets are equal before execution, then we only must show that the ith state and list are equal afterwards; no other state or list is modified. Finally, on state set 0, both scanners do nothing, so we can restrict our attention to the positive integers. With these in mind, we calculate the weakest precondition of the abstract scanner to establish the linking invariant:

$[ss(ii) := scan(ii)[ss(ii-1)]]$
$ss(ii) = rep[\text{ran}(lists(ii))]$

$= \{WPA\}$
$scan(ii)[ss(ii-1)] = rep[\text{ran}(lists(ii))]$

We therefore have to show that the result of the concrete *nextScan* is the same as that obtained by the scanner on the previous list. To start, we prove obligation (5) in Figure 2.13. There are four conjuncts in the invariant that must be true prior to loop execution; the fifth is always true as argued above:

$[count := 0;\ len \longleftarrow numStates(ii-1);\ kill(ii)]$
$count \in \mathbb{N} \wedge count \leq len \wedge len = \text{size}(lists(ii-1)) \wedge$

$\text{ran}(\mathit{lists}\,(\mathit{ii})) = \mathit{scan}(\mathit{ii})[(\mathit{lists}\,(\mathit{ii}-1))[1..\mathit{count}]]$

$= \{\mathit{WPS}\}$
$0 \in \mathbb{N} \wedge 0 \leq \text{size}(\mathit{lists}\,(\mathit{ii}-1)) \wedge \text{size}(\mathit{lists}\,(\mathit{ii}-1)) = \text{size}(\mathit{lists}\,(\mathit{ii}-1)) \wedge$
$\{\} = \mathit{scan}(\mathit{ii})[(\mathit{lists}\,(\mathit{ii}-1))[1..0]]$

$=$
true

We turn to proof obligation (1) in Figure 2.13. The body of the loop extracts the next element $(\mathit{rr},\ \mathit{jj},\ \mathit{ff})$ from the state set, and if its next symbol matches the next sentence terminal, it adds $(\mathit{rr},\ \mathit{jj}+1,\ \mathit{ff})$ to $\mathit{lists}\,(\mathit{ii})$, emulating what the scanner definition would do on this element.

At the end of the loop, we have postcondition $\text{ran}(\mathit{lists}\,(\mathit{ii})) = \mathit{scan}(\mathit{ii})[(\mathit{lists}\,(\mathit{ii}-1))]$, so this list equals the corresponding set, and the linking invariant is reestablished.

Refinement of Predictor and Completer

To correctly refine *nextOps*, we can finally introduce the modification made in the algorithm due to Aycock and Horspool [1]. The completer remains the same, but the predictor includes a new condition. The definition of the predictor as given by Aycock and Horspool is:

> If $[A \rightarrow \cdots \bullet B \cdots, j]$ is in S_i, add $[B \rightarrow \bullet\alpha, i]$ to S_i for all rules $B \rightarrow \alpha$.
> **If B is nullable, also add $[\mathbf{A \rightarrow \cdots B \bullet \cdots, j}]$ to $\mathbf{S_i}$.**

We can now show the correctness of *nextOps*, starting with proof obligation (5) in Figure 2.13. We have to show that the invariant of the outer loop is true prior to its execution. We include only the first four conjuncts; the others are true because the premises of their respective implications are vacuously true when $\mathit{count} = 0$.

$[\mathit{count} := 0;\ \mathit{len} \longleftarrow \mathit{numStates}\,(\mathit{ii})]$
$\mathit{count} \in \mathbb{N} \wedge \mathit{count} \leq \mathit{len} \wedge \mathit{len} = \text{size}(\mathit{lists}\,(\mathit{ii})) \wedge \mathit{setInList}$

$= \{\mathit{WPA},\ \mathit{WPS},\ \mathit{expand\ operation\ bodies}\}$
$0 \in \mathbb{N} \wedge 0 \leq \mathit{len} \wedge \text{size}(\mathit{lists}\,(\mathit{ii})) = \text{size}(\mathit{lists}\,(\mathit{ii})) \wedge \mathit{setInList}$

The final conjunct, *setInList*, asserts that the elements in the corresponding state list are already present in the list. This follows from the linking invariant, stating that states and lists are equal.

We now show proof obligation (1) by showing that the operation body maintains, piece by piece, the outer loop invariant.

For $count \leq len$, we simply observe that the loop guard is $count < len$. To justify $len = \text{size}(lists(ii))$, we have that the last thing the loop does in its body is ensure this is the case. Next, *setInList* is always true, because the machine invariant asserts that all sets and lists are equal prior to the operation call, and *nextOps* only adds new items.

The next piece of the invariant states, for all $(r, j, f) \in lists(ii)[1 \,..\, count]$:

- If $f \neq i$, then this state cannot be used to add new items
- If $f = i$, and some new item $(q, k{+}1, g)$ could be added from this state, then (q, k, g) is already present and $rs(q)(k)$ is nullable.

The inner loops of *nextOps* add the items that a predict or complete operation would add, and we have already argued that once we use the predictor, or complete on an item where $f < i$, no further item can be added. When $f = i$, we know that (q, k, g) is present, otherwise the completer could not have added $(q, k{+}1, g)$. The item (r, j, f) was initially created in list i and completely recognized without consuming any input, so the right-hand-side of r must be nullable. This means that nonterminal $r = rs(q)(k)$ is nullable as well.

The next conjunct states that for all items (r, j, f) we have visited, if the next symbol is nullable, then $(r, j{+}1, f)$ is added. This addition is exactly made by the part of the predictor corresponding to the Aycock-Horspool modification.

Continuing with *prevItem*, we state that all newly added items are in the closure of the predictor and completer acting on the current contents of the list. The only possibly problematic item is the one added when (r, j, f) is present, and $(r, j{+}1, f)$ is added by moving over a nullable symbol. However, we know that $(r, j{+}1, f)$ must be in the closure of the predictor and completer over any set that includes (r, j, f), via an argument paralleling Theorem 4.3.

Lastly, we want that all other sets and lists are still equal, certainly true because they were equal prior to this call, and we are not changing them.

We calculate the weakest precondition for the abstract *nextOps* to establish the invariant. As with the scanner above, we focus on the set that we know is changed:

$[ss(ii) := \text{closure1}(ops(ii))[ss(ii)]]$
$ss(ii) = rep[\text{ran}(lists(ii))]$

$= \{WPA\}$
$ss \lhd\!\!+ \{ii \mapsto \text{closure1}(ops(ii))[ss(ii)]\}(ii) = rep[\text{ran}(lists(ii))]$

In other words, it adds the closure of the predictor and completer to the state set. The implementation does the same, because on termination of the while loop we have that no predictor or completer operation on *any* state can add a new state. The reason is that even if a state where $f = i$ exists, there is no $(q, k{+}1, g)$ that could be added using it. If there were, we know (q, k, g) exists in the list and its $k+1$st symbol is nullable, so $(q, k{+}1, g)$ must be present. Combining this with the fact that all newly added states result from a predictor or completer step shows we have the required closure on the predictor and completer.

As in the execution of the scanner, the predictor and completer run in their own inner loop, in order to systematically add all resultant states to the list. The loop invariant of the predictor states that, using only the first *inner* productions of the grammar, we have added the elements that the predictor would produce, so that at its completion, we have predicted all items using the current one. The completer loop operates similarly: at any point, we have used the completer on the first *inner* states of state list f. Since the loop iterates until the whole list has been processed, nothing further can be accomplished using the completer at this point. We thus know that if $f \neq i$, this item can never be used again to add a new item; if $f = i$, we know that the modified predictor will add any items that we will miss by running this complete step now and not again at some later point.

To conclude, we require a suitable variant for the loops in *nextOps*. When executing the predictor, the variant is simply the number of rules of the grammar not processed. The variant of the completer and outer loop use the analysis in 4.6.2, which indicates the maximum number of states we can visit before we must exhaust the current list.

5.5 Optimizing via Invariants

The loop invariants given in the previous section for the list refinement are useful for understanding correctness. Interestingly, they have also directly lead to the discovery

of an optimization we can make, which we briefly outline here. For an item where $f = i$, we can process it and move to the next item, even if we could have used it later to add a new state. Assume that we are about to run the inner loop corresponding to the completer on such a state, and consider what happens if we in fact do not run the completer and just move past the item. We have $f = i$, so the loop invariant tells us that, for any $(s, k{+}1, g)$ that could be generated from this state, we have (s, k, g) present in the state list already, and the predictor will add $(s, k{+}1, g)$. Therefore, we do not have to produce any of the $(s, k{+}1, g)$ items at all, since the predictor will do this. Essentially, running the completer at this point accomplishes a subset of the work that the predictor would do. In summary, we can avoid running the completer on any state where $f = i$.

Chapter 6

Literate Implementation

We give a self-contained, literate implementation of Earley's algorithm in Pascal. Literate programming [12] was conceived by Donald Knuth in 1984, as a way to reverse the roles of comments and code. The essense of the approach is that code should be written primarily for human consumption, not in the form compilers and interpreters expect. We can freely mix code chunks and associated documentation in a single file which can be converted (*woven*) into a document (a literary work) to be read and understood. We can also extract (*tangle*) the code from the file, resulting in the removal of documentation and reorganization of code into the form required for computer processing. Since documentation and code cohabit, it is hoped that the documentation will remain up-to-date with the code, rather than becoming unsynchronized.

Weaving and tangling is accomplished by a literate programming tool which supports the desired input and output languages. We have chosen Noweb [15] since it is input-language-independent, thus supporting Pascal, and can generate LaTeX as output.

6.1 Algorithm Description

Earley's algorithm is a context-free recognizer, with the property that it can be used on any context-free grammar, and still run in no worse than $O(n^3)$ time. It takes a sentence and a grammar, and determines whether or not the sentence belongs to the grammar.

The algorithm works conceptually in $n + 1$ phases, where n is the length of the

input string. These phases are all associated with their own set of data, referred to as a *state set.* These state sets are composed of so-called Earley items, which, taken in unison, maintain the entire state of the parser (that is, what has been parsed so far and what can potentially be parsed next).

Earley items can be represented by triples (r, j, f) of integers, assuming that the grammar productions have been arbitrarily ordered. We have that r is the integer for a rule of the grammar, j represents the position we are interested in parsing of the right-hand-side of r and f represents the state set in which parsing this production began. A *final item* is one which has been completely parsed. In such a case, $j = p'$, where p' is the length of the rule, since the last token parsed was the last token in the production.

The algorithm requires that a new rule, S, be introduced, which has only one production. This simplifies the recognition condition, and is a simple transformation that can be performed on all grammars.

The algorithm begins with the item $(S, 0, 0)$ in state set 0. If the sentence has length n and the length of the root production is S', then it accepts the sentence if s_n contains $(S, S', 0)$.

To get an idea of what states represent, consider an item (r, j, f) belonging to state set i. Recall that the f component represents the state set where the production r began to be parsed. Keeping in mind that a state set corresponds with the current position in the input string, f represents the fact that the first f characters of the input string, followed by the nonterminal on the left-hand side of rule r, can be generated.

Next, we turn to the j component, which says that we have parsed j tokens of the right-hand side of rule r. This tells us that the first j tokens can generate the string consisting of the terminals from $f + 1$ to i in the input sentence.

Earley's algorithm seeks to systematically build these state sets. It involves the repeated application of three functions: predictor, scanner, and completer. We review them for completeness, and assume that they are operating on a state set (r, j, f) in state set i.

- Predictor: adds, to state set i, items $(q, 0, i)$, where q is the $j + 1$st symbol in the right side of r
- Completer: if operating on a final state, then for all items of the form (q, l, g), found in state set f, if the $l + 1$th symbol in q is r, then add $(q, l + 1, g)$ to state

set i

- Scanner: If the $j+1$st symbol in the right side of r is the $i+1$st symbol in the sentence, add $(r, j+1, f)$ to state set $i+1$

In order to deal with grammars containing ϵ-productions, it is necessary to add to the predictor step given: if the jth symbol of the right-hand side of rule r is nullable, then we also add $(r, j+1, f)$ to the state set.

6.2 Grammar Module

The first step is to define a module for representing grammars. There must be a way to extract the left side of a rule, part of the right side of a rule, the nonterminal and terminal sets, the first rule belonging to a given nonterminal, the length of a rule, and so on. The following interface is used.

⟨*GrammarInterface*⟩≡

```
unit grammar;

interface

const
  ruleLength = 10;

function getRuleLeft (w : integer) : char;
function getRuleRight (w, p : integer) : char;
function getRuleLength (w : integer) : integer;
procedure newRule (c : char; s : string);
function getRuleNum (w : char) : integer;
function numRules : integer;
procedure addNT (c : char);
function inNT (c : char) : boolean;
procedure addTT (c : char);
function inTT (c : char) : boolean;
procedure findNullable;
function isNullable (c : char) : boolean;
```

The upper limit on the length of a rule is to allow an array to represent the rules, for faster access. The number of rules, however, is not known and so is represented as a linked list in the implementation. A variable pos is introduced, to point at the end of this list. This makes it fast to add a new rule to the end of the list, via newRule. The implementation procedures are then straightforward:

⟨*GrammarImplementation*⟩≡

```
implementation

type
  ruleList = ^ruleElement;
  ruleElement = record
    left : char;
    right: string[ruleLength];
    which : integer;
    next : ruleList
end;

var
  rules : ruleList;
  nRules : integer;
  pos : ruleList;
  tt, nt : set of char;
  nullable : array[char] of boolean;

function getRule (w : integer) : ruleList;
var counter : integer;
var t : ruleList;
begin
  counter := 0;
  t := rules;
  while counter < w do
  begin
    t := t^.next;
    counter := counter + 1
  end;
```

```
  getRule := t
end;

function getRuleLeft (w : integer) : char;
var t : ruleList;
begin
  t := getRule (w);
  getRuleLeft := t^.next^.left
end;

procedure newRule (c : char; s : string);
begin
  new (pos^.next); pos^.next^.left := c; pos^.next^.right := s;
  pos^.next^.which := nRules; pos^.next^.next := nil;
  nRules := nRules + 1; pos := pos^.next
end;

function getRuleRight (w, p : integer) : char;
var t : ruleList;
begin
  t := getRule (w);
  getRuleRight := t^.next^.right[p]
end;

function getRuleLength (w : integer) : integer;
var t : ruleList;
begin
  t := getRule (w);
  getRuleLength := length(t^.next^.right)
end;

function getRuleNum (w : char) : integer;
var t : ruleList;
begin
  t := rules;
```

```
  while t^.next^.left <> w do
    t := t^.next;
  getRuleNum := t^.next^.which
end;

function numRules : integer;
  begin numRules := nRules end;

procedure addNT (c : char);
  begin nt := nt + [c] end;

function inNT (c : char) : boolean;
  begin inNT := c in nt end;

procedure addTT (c : char);
  begin tt := tt + [c] end;

function inTT (c : char) : boolean;
  begin inTT := c in tt end;
```

⟨*nullable*⟩

```
begin
  nt := []; tt := [];
  new (rules); rules^.next := nil;
  pos := rules; nRules := 0
end.
```

We can compute the nullable elements of the grammar with a fixed-point procedure. We know that all nonterminals with ϵ-productions are nullable. Moreover, if all right-hand-side symbols of a production are nullable, then so is the left-hand-side of this production. We continue this until no new information is added.

⟨*nullable*⟩≡

```
procedure findNullable;
var
  ch : char; i, j : integer;
  change : boolean; {did something change on this iteration?}
  temp : boolean;
begin
  for ch := '$' to 'z' do
    nullable[ch] := false;
  i := 0;
  while i < numRules do
  begin
    if getRuleLength (i) = 0 then nullable[getRuleLeft (i)] := true;
    i := i + 1
  end;
  change := true;
  while change do
  begin
    change := false;
    i := 0;
    while i < numRules do
    begin
      temp := nullable[getRuleLeft(i)];
      if getRuleLength (i) > 0 then
      begin
        j := 1;
        while (nullable[getRuleRight(i, j)]) and
         (j < getRuleLength (i)) do
            j := j + 1;
        if (j = getRuleLength (i)) and
         (nullable[getRuleRight(i, j)]) then
```

```
            nullable[getRuleLeft(i)] := true
        end;
              if temp <> nullable[getRuleLeft(i)] then change := true;
        i := i + 1
      end
    end
  end;

function isNullable (c : char) : boolean;
begin isNullable := nullable[c] end;
```

6.3 Reading the Grammar

We have to provide a way for a grammar to be entered by the user, for use in the algorithm. Our input format is one production per line, where a production begins with a nonterminal symbol, followed by a space, the = character, another space, the right-hand-side of the production and a period (.). The right-hand-side is a sequence of nonterminal symbols and quoted terminal symbols.

⟨*ReadGrammar*⟩≡

```
unit readgrammar;

interface

procedure getGrammar;

implementation

uses grammar;

procedure getGrammar;
var
  l, r, temp : char;
  counter : integer;
  text : string[ruleLength];
begin
  while not eof do
  begin
    read (l);
    addNT (l);
    read (temp); read (temp); read (temp); read (r);
    counter := 0;
    while true do
    begin
      if r = '.' then begin readln; break end;
      counter := counter + 1;
      if r = '"' then
```

```
        begin read (r); addTT (r);
      read (temp); end;
      text[counter] := r;
      read (r)
    end;
    SetLength (text, counter);
    newRule (l, text)
  end;
end;
end.
```

6.4 State Set Module

The other data that the algorithm uses is state sets, which will also be presented as a module. The interface primarily consists of procedures to add new Earley items, and to retrieve their components. A constant `maxInput` is introduced, so that state sets can be represented as elements of arrays, improving the access time:

⟨*StateInterface*⟩≡

```
unit statesets;

interface

const
  maxInput = 50;

procedure addToEnd (i, r, j, f : integer);
function inSet (i, r, j, f : integer) : boolean;
function numIn (i : integer) : integer;
function getR (i, j : integer) : integer;
function getJ (i, j : integer) : integer;
function getF (i, j : integer) : integer;
```

⟨*StateImplementation*⟩≡

```
implementation
⟨StateType⟩
⟨StateCode⟩
```

The Earley items consist of three integers, so will be represented as triples. An array `stateObj` will be introduced, whose elements are linked lists of these items. We also keep an array containing the count of the number of elements in the state sets.

⟨*StateType*⟩≡

```
type
  stateList = ^state;
  state = record
    r, j, f : integer;
    next : stateList
  end;

var
stateObj : array[0..maxInput] of stateList;
objCount : array[0..maxInput] of integer;
i : integer;
```

Procedure addToEnd adds the given triple to state set i.

⟨*StateCode*⟩≡

```
procedure addToEnd (i, r, j, f : integer);
var t : stateList;
begin
  t := stateObj[i];
  objCount[i] := objCount[i] + 1;
  while t^.next <> nil do
    t := t^.next;
  new (t^.next);
  t^.next^.next := nil;
  t^.next^.r := r;
  t^.next^.j := j;
  t^.next^.f := f
end;
```

Earley's algorithm requires that new items be added, only if they did not previously exist in the list. inSet thus returns true or false, reflecting whether or not the item is in the list:

⟨*StateCode*⟩+≡

```
function inSet (i, r, j, f : integer) : boolean;
var t : stateList;
begin
  inSet := false;
  t := stateObj[i];
  while t^.next <> nil do
  begin
    if (t^.next^.r = r) and (t^.next^.j = j) and (t^.next^.f = f) then
      inSet := true;
    t := t^.next
  end
end;
```

Returning the number of items in a state set is achieved through the `objCount` array:

⟨*StateCode*⟩+≡

```
function numIn (i : integer) : integer;
  begin numIn := objCount[i] end;
```

To retrieve a specific item from the list, the list number and item number are required. So as not to duplicate code, a general `getItem` procedure is defined, and used by the following procedures to extract a certain part of the rule:

⟨*StateCode*⟩+≡

```
function getItem (i, j : integer) : stateList;
var counter : integer;
pos : stateList;
begin
  pos := stateObj[i];
  counter := 0;
  while counter < j do
  begin
    pos := pos^.next;
    counter := counter + 1
  end;
  getItem := pos
end;

function getR (i, j : integer) : integer;
var pos : stateList;
begin
  pos := getItem (i, j);
  getR := pos^.next^.r
end;

function getJ (i, j : integer) : integer;
var pos : stateList;
begin
  pos := getItem (i, j);
  getJ := pos^.next^.j
end;

function getF (i, j : integer) : integer;
var pos : stateList;
begin
```

```
  pos := getItem (i, j);
  getF := pos^.next^.f
end;

begin
  for i := 0 to maxInput do
  begin
    objCount[i] := 0;
    new (stateObj[i]);
    stateObj[i]^.next := nil
  end
end.
```

6.5 Earley Module

It now remains to use these two modules to execute Earley's algorithm. The main loop is a `for` loop that iterates through all the state sets. An inner while loop iterates until there are no further items in the current state set to process. After the `for` loop ends, the recognition condition is checked and the result is printed. Note that if the sentence does not belong to the grammar, some iterations of the `for` loop will do nothing; it will just loop until it has searched the last (and empty) state set. This is fine — although Earley optimized slightly and short-cuts the algorithm if the next state set is empty. After reading the input string and grammar, the algorithm is executed.

⟨*EarleyAlgorithm*⟩≡

```
program earley (input, output);

uses grammar, readgrammar, statesets;

var
  x : string[maxInput]; {input string}

procedure computeEarley;
var
 nr, i, inner, counter, r, j, f, q, l, g : integer;

begin
  addToEnd (0, 0, 0, 0); findNullable;
  for i := 0 to length (x) do
  begin
    inner := 0;
    while inner < numIn (i) do
    begin
      r := getR (i, inner);
      j := getJ (i, inner);
      f := getF (i, inner);
      {prediction}
      if (j <> getRuleLength(r)) and inNT((getRuleRight(r, j+1))) then
```

```
begin
  if (isNullable (getRuleRight(r, j+1))) and
   (not inSet (i, r, j + 1, f)) then
    addToEnd (i, r, j + 1, f);
  nr := getRuleNum (getRuleRight (r, j+1));
  while nr < numRules do
  begin
    if getRuleLeft (nr) <> getRuleRight (r, j+1) then break;
    if not inSet (i, nr, 0, i) then
      addToEnd (i, nr, 0, i);
    nr := nr + 1
  end
end

{completion}
else if (j = getRuleLength(r)) then
begin
  counter := 0;
  while counter < numIn (f) do
  begin
    q := getR (f, counter);
    l := getJ (f, counter);
    g := getF (f, counter);
    if getRuleRight (q, l+1) = getRuleLeft (r) then
    if not inSet (i, q, l+1, g) then
      addToEnd (i, q, l+1, g);
    counter := counter + 1
  end
end

{scanning}
else if (i < length (x)) and (j <> getRuleLength (r)) and
 (inTT (getRuleRight (r, j+1))) then
begin
  if getRuleRight (r, j+1) = x[i+1] then
```

```
          if not inSet (i+1, r, j+1, f) then
            addToEnd (i+1, r, j+1, f);
        end;
        inner := inner + 1
      end
    end;
    writeln (inSet (length (x), 0, 1, 0));
end;

begin
  readln (x);
  getGrammar;
  computeEarley
end.
```

Chapter 7

In Closing

We have shown a development of Earley's recognizer in the B-Method, from an abstract description of a recognizer, through a set-theoretic version of the algorithm and ending at an efficient list-processing realization. For two reasons, a literate Pascal implementation was also presented. First, we wanted to show that the B implementation using lists closely corresponds to an actual implementation in a commonplace programming language. Second, it gave an opportunity to demonstrate an optimization of the version developed in B. Specifically, the scanner was incorporated into the same loop structure as the predictor and completer, whereas the B development relies on a separate operation to act as the scanner. We chose to carry out the B development in this way to simplify the reasoning of the list refinement, whose proof was already obscured by the complexity of the algorithm and the change of representation of states. This observation directly leads to some possible future work: we can try to incorporate optimizations of Earley's recognizer into the proof. Besides the scanner optimization, various forms of lookahead have been proposed to speed up the algorithm [5, 1]: it would be interesting to show when and where lookahead could be used, but also to find cases where "obvious" lookahead optimizations in fact break the algorithm.

In its full glory, an Earley recognizer can also build parse trees corresponding to the steps used in the recognition of a sentence. It would be instructive to formlize this as well for several reasons. First, it would require a rigorous description of what exactly a parse tree is, so that we can assert that Earley's algorithm indeed generates parse trees for a given sentence. Second, for ambiguous grammars, there may be multiple parse trees for a given sentence. We therefore cannot simply ignore the

addition of an item to a set or list when it is added two or more times, as we did here. It would be instructive to modify the B development to take this into account, and we wonder how much of the current development would readily scale up to this more chalenging problem. There is also work [21] indicating that the method Earley gives for constructing parse trees is incorrect in some instances involving ambiguous grammars. A B development should provide another avenue for exploring the reason for this incorrectness, and directly or indirectly lead to possible solutions.

A litany of other parsing algorithms exist, some of which we outlined at the outset. Formalizing these algorithms may provide common ground for which to analyze their commonalities, or to make clear the subsets of the CFGs that they can deal with. We expect that the CYK algorithm, with its simple matrix-multiplication-like structure, would be particularly amenable to an elegant specification and implementation.

While we have focused on context-free parsing algorithms, which have been widely studied and understood, there are other means by which languages can be specified. In particular, for describing programming languages, a more modern device is the Parsing Expression Grammar (PEG) [7]. These grammars allow only a single production per nonterminal, thus disallowing the very idea of ambiguity from entering a grammar. The languages recognized by the PEGs are an arbitrary-looking mélange of regular, context-free and even non-context-free languages. While these fuzzy boundaries may be disconcerting, an interesting property is that we can have a linear-time parser for any PEG. We wonder if the parse trees generated by these grammars would be easier to reason about than the potential parse forests produced by a context-free recognizer.

Bibliography

[1] J. Aycock and N. Horspool, "Practical Earley Parsing," *The Computer Journal*, vol. 45, pp. 620–630, 2002.

[2] ClearSy, "B4free and clik'n'prove,a set of tools for the development of B models with the B formal method." `http://www.b4free.com`, 2007.

[3] K. D. Cooper and L. Torczon, *Engineering a Compiler*. Morgan Kaufmann, 2004.

[4] J. Earley, "An efficient context-free parsing algorithm," *Communications of the ACM*, vol. 13, pp. 94–102, 1970.

[5] J. C. Earley, *An efficient context-free parsing algorithm*. PhD thesis, Carnegie-Mellon University, 1968.

[6] R. W. Floyd, "Syntactic Analysis and Operator Precedence," *J. ACM*, vol. 10, no. 3, pp. 316–333, 1963.

[7] B. Ford, "Parsing expression grammars: a recognition-based syntactic foundation," *SIGPLAN Not.*, vol. 39, no. 1, pp. 111–122, 2004.

[8] J. Gosling, B. Joy, and G. L. Steele, *The Java Language Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[9] S. L. Graham, M. Harrison, and W. L. Ruzzo, "An Improved Context-Free Recognizer," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 3, pp. 415–462, 1980.

[10] D. Grune and C. J. H. Jacobs, *Parsing techniques a practical guide*. Chichester, England: Ellis Horwood Limited, 1990.

[11] C. Jones, "Formal development of correct algorithms: an example based on Earley's recogniser," *SIGPLAN Notices*, vol. 7, pp. 150–169, 1972.

[12] D. E. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[13] L. Kwiatkowski, "Reconciling Unger's parser as a top-down parser for CF grammars for experimental purposes," Master's thesis, Vrije Universiteit, Amsterdam, July 2005.

[14] M. Leuschel, "Prob." http://www.stups.uni-duesseldorf.de/ProB, 2007.

[15] N. Ramsey, "Noweb home page." http://www.eecs.harvard.edu/~nr/noweb, 2006.

[16] S. Schneider, *The B-Method: An Introduction.* Cornerstones of Computing, Palgrave, 2001.

[17] F. W. Schroer, "The ACCENT compiler compiler, introduction and reference," Tech. Rep. TR 101, German National Research Center for Information Technology, 2000.

[18] E. Sekerinski and K. Sere, *Program Development by Refinement: Case Studies Using the B Method.* London, UK: Springer-Verlag, 1999.

[19] K. Sikkel, "Parsing schemata and correctness of parsing algorithms," *Theoretical Computer Science*, vol. 199, no. 1–2, pp. 87–103, 1998.

[20] A. Stolcke, "An efficient probabilistic context-free parsing algorithm that computes prefix probabilities," *Comput. Linguist.*, vol. 21, no. 2, pp. 165–201, 1995.

[21] M. Tomita, *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems.* Norwell, MA, USA: Kluwer Academic Publishers, 1985.

[22] S. H. Unger, "A global parser for context-free phrase structure grammars," *Commun. ACM*, vol. 11, no. 4, pp. 240–247, 1968.