Incremental Parsing

Wouter Pasman, August 1991
Contents

Chapter 1 The problem ................................................................. 4
   Abstract ............................................................................. 4
   Notation and conventions .............................................. 4
   Text and parse trees .......................................................... 5
   Incremental parsing ............................................................ 6
   Overview of the thesis ....................................................... 7

Chapter 2 Introduction to incremental parsing ............................. 8
   Finding a node to replace .................................................... 8
   An algorithm for incremental parsing .................................. 9
   How hard is CheckForTree? .............................................. 11

Chapter 3 Partial solutions for CheckForTree ............................... 12
   The CheckForTree algorithm ............................................ 12
   Some non-partial but expensive tests ................................. 12
   Partial testing without parsing ........................................ 13
      Idea 1 ........................................................................ 13
      Idea 2 ........................................................................ 13
      Idea 3 ........................................................................ 16
      Idea 4 ........................................................................ 17
   Conclusions .................................................................... 17

Chapter 4 Bidirectional recognising ............................................ 18
   Introduction .................................................................... 18
   A short description of Earley's recogniser .......................... 18
   Adapting Earley for symmetry ......................................... 19
   Recognising with these notions ....................................... 20
   An algorithm for Left to Right recognising ....................... 27
   Comparison with Earley's algorithm .................................. 28
      Discussion of the symmetry difference .......................... 31
   Example of Right to Left recognising ............................... 31

Chapter 5 Island Parsing .............................................................. 33
   I
      Adapting the bidirectional recogniser for island recognising .... 33
      The context idea .......................................................... 33
      Using a context set ....................................................... 36
      Switching the context to the other side ............................ 39
      Deciding about the result .............................................. 42
      An algorithm for island recognising ............................... 43
      Building parse trees .................................................... 44
   II
      Removing useless items ................................................ 45
      An invariant for the items ............................................. 47
      A solution for removing useless items ............................ 47
      Incremental removal of useless items ............................. 48
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>An implementation of an Island Parser</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>Implementation in pseudo-Pascal</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>A notion of correctness</td>
<td>58</td>
</tr>
<tr>
<td>7</td>
<td>Optimizations</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Goal of this chapter</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Alternative ways to make island parsers</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Relation between Tomita's and our parser</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>Optimization ideas for our island parser</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>An implicit context</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>Efficient parse tree storage</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>A more efficient implementation</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>Definitions</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>A Specification</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>An implementation</td>
<td>68</td>
</tr>
<tr>
<td>9</td>
<td>An implementation in Lisp</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>LeLisp introduction</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>Island parser</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>Improved island parser</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>Help functions</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>Grammar functions</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>Colophon</td>
<td>88</td>
</tr>
</tbody>
</table>
1 The problem

Abstract

During syntax-directed editing, a text and a corresponding parse tree exist. When the user makes a change in the text, it is desirable to update the parse tree in an incremental way, instead of reparsing the whole text. We will do this by means of an island parser that allows us to start parsing at any place in the text, and to parse in both left and right direction until a tree is found that fits into the existing parse tree. The island parser is presented as an adaptation of the Earley parser.

Notation and conventions

In this section, we will give some definitions that we use.
A terminal is a character in [a..z] followed by an arbitrary sequence of characters, or the empty sequence ε. For example, 'apple' and 'aA$9' are terminals. A nonterminal is a character in [A..Z], followed by an arbitrary sequence of characters. So 'Apple' is a nonterminal. A symbol can be either a terminal or a nonterminal. Lower case Greek characters (like α, β, σ, excepted ε since it is the empty sequence) stand for an arbitrary sequence of symbols. Two symbols in a sequence are separated by a white space (" "). To avoid confusion, we do not allow spaces in symbols.

For $T_1$ and $T_2$ being terminals, we can concatenate $T_1$ and $T_2$. We write $T_1$·$T_2$. Two terminals are concatenated by simply sequencing them. For example, golf·club = golfclub.

If $N$ is some nonterminal, and $σ$ is some sequence of symbols, then '$N ::= σ$' is a rule.
So 'P ::= a b c', 'S ::= c' and 'Apple ::= a51 Apple Bear' are rules. A "|" in a rule has a special meaning: $N ::= α | β$ is short for two rules: $N ::= α$ and $N ::= β$. A grammar is a set of rules. For example { S ::= S S, S ::= x, S ::= ε } is a grammar.

Assume we have a grammar $G$. If $N$ is a nonterminal, $α, β$ are arbitrary sequences and there is a rule $N ::= σ$ in $G$, then $α N β ⇒ α σ β$ is called a derivation step. If $α ⇒ β ⇒ .. ⇒ γ$ or $α = γ$ then $α ⇒ γ$. If there is a nonterminal $N$ such that $N ⇒^* S_1 .. S_n$, and each $S_i$ is a terminal, then we say that $G$ produces $S_1 S_2 .. S_n$. $L(G)$ is the set of all sequences that are produced by $G$.

Example: take $G=\{S ::= a M b M, M ::= p L q\}$.
Now the following holds: $S ⇒ a M b M ⇒ a p b M ⇒ a p b q$. Therefore $S ⇒^* a p b q$, and we say that $G$ produces apbq.

1 It would be better, but it seems a nonwritten rule to use 'we'. After all, it is reassuring to know that someone else has exactly the same ideas.
In each derivation step, one nonterminal is substituted by a sequence of symbols. But if different nonterminals occur in the same sequence of symbols, it does not matter in what order these nonterminals are replaced. In the example, we can also make a derivation $S \Rightarrow a \ M \ b \ M \Rightarrow a \ M \ b \ q \Rightarrow a \ p \ b \ q$. There is no real difference between these two derivations: in both cases the first $M$ is replaced by $p$, and the second by $q$.

The derivation we make to proof that $N \Rightarrow^* \sigma$ can be put in a picture called a parse tree. A parse tree is a (directed) acyclic graph. The direction is implicit downwards. A node in a parse tree has children. Unlike acyclic graphs, these children are ordered in a way corresponding to their ordering in the rule. In the pictures, the children are placed under their parent in order from left to right. Each symbol in a derivation has a corresponding symbol in the graph. If, in a derivation, $S_i$ is substituted by $S'_1 \ldots S'_n$, then $S_i$ gets children $S'_1 \ldots S'_n$ in this order.

The parse tree of our example looks as follows (a):

```
   S
  / \  \
 a   M   b   M
   |     |     |
   p     q     
   
(a)
```

Each node in such a parse tree has a symbol, called the sort of the node.

Given a grammar $G$ and a sequence of terminals $\sigma$, an algorithm deciding whether there is an $N$ with $N \Rightarrow^* \sigma$ is called a recogniser. If the algorithm returns a parse tree it is called a parser. If, for some $N$ and $\sigma$, there are two derivations $N \Rightarrow^* \sigma$ with different parse trees, we say that $\sigma$ is ambiguous.

**Text and parse trees**

We assume the reader to be familiar with syntax directed editors. In these editors, the user works with a kind of text editor. The editor knows the grammar the user is working with, and checks whether the text the user types is correct. For example, if the used grammar is Pascal, the editor will only allow correct Pascal text to be entered. Of course, there are moments when this is not possible. For example if the user wants to type 'begin', and he types the first 'b' of it, we do not yet have a correct Pascal text.

Usually, syntax directed editors update the parse tree as soon as possible. For example, the Synthesizer Generator and the Generic Syntax-directed Editor ([DK90]) work in this way. This can be useful for several reasons:

- The user can be informed as soon as possible about syntax errors in his text.
- It is easy to provide parse tree dependent commands to the user, for example a 'go to parent node' or a 'pretty print' command.
- When the text the user typed is needed in a parsed format, the parse tree is directly available.
Incremental parsing

In most cases, the user edits a small part of his text. Usually, these small changes have little effect on the parse tree. For example, when the user changes the statement 'x:=3' in 'x:=6' in a Pascal editor, only the assignment has to be changed. In a picture, this looks as follows (a):

![Diagram showing the parse tree structure.]

As we can see in the picture, to correct the tree, only the 3 has to be replaced by a 6. We do not have to look at the other nodes. However, it is not always as easy as in this picture. Consider the following example.

Grammar:

\[
E ::= \text{Nat} \mid E + E \mid E \cdot \text{Nat} \mid \text{Nat} \\
\text{Nat} ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]

Note that in this grammar, a * multiplies anything before it with the number directly following it.

When the + is changed into an * as in

\[
2+3+4+5 \quad \rightarrow \quad 2+3+4*5
\]

the parse tree before the edit looks like (b), and the one after the edits like (c).
The change of only one character therefore changes the whole tree up to the top node. In general, the work involved in correcting the parse tree depends on the grammar and the specific edit actions.

The job of updating the parse tree with as little work as possible is called 'incremental parsing', and is the subject of this thesis.

Overview of the thesis

In chapter 2, we will give one way of updating the parse tree after an edit action. The algorithm for correcting the parse tree needs to decide about what is the right place to do a correction. There are various ways to make this decision. One way to make an algorithm for deciding it is given in chapter 3. We will show that in general, the decision is at least as hard as a recogniser. And after all, we need to parse if we found the place to do a correction. So it seems better to make an island parser doing both jobs in one time. This is done in chapter 4-6. In chapter 7, we consider some optimizations.
2 Introduction to incremental parsing

Finding a node to replace

Assume that the user of a syntax-directed editor has changed a part of the text. This text can be found at the leaves of the parse tree. We want to replace at most one node to correct the parse tree. The reason for this is that although it may be possible to replace a number of smaller trees, it is not clear to us how an algorithm to do this should be like. So, we have the following picture (a).

![Diagram of OldTree with Node, Edited part, and Replacing Node](image)

(a)

There is always such a replacing node if there is a correct parse tree: the root node. However, replacing that node means that we have to make a new root node, and this implies reparsing of the whole text. This is not really incremental parsing. So we prefer a lower node than the root node.

How do we find the lowest Node that has to be replaced? In general, this is a difficult problem. We choose for the following solution: Start at a certain node, and try to replace it. If this fails, try to replace the parent of the node.

![Diagram of N and Root with Changed part and Part covered by N](image)

(b)
What is a useful start node? Consider picture (b). When the user has changed some text, the characters he changes correspond to some leaf nodes in the parse tree. The node we are looking for must at least cover the leaves that are changed. We find this node by finding the lowest common parent of the leftmost and the rightmost leaf that has been changed. In the picture, this is \( N \).

If there is no correct parse tree, we cannot do the correction step. Instead, we place the new text as unparsed text under the \( N \) node of the old parse tree, just to have the parse tree correspond with the text. We will have to wait for more corrections until we can make up the new parse tree. We remember the sort of the old node, because we may only place trees of that sort at that place in the parse tree without problems. For example when the user again edits this text, and retypes the old text, we can put back the old parse tree at the same place.

If the user does a plain insert, without replacing old leaves of the parse tree, we have a problem with finding the start node. In this case, no leaves have changed, so there is no common parent node. But for our algorithm, we still need a start node. In a picture, the problem looks like (c). \( T_1 \) and \( T_2 \) is the text under the lowest common parent node of the leaf directly at the left and right of the insertion point. To choose Node should be the worst choice, since \( T_3 \) might be placed under node \( A \), but giving \( Node \) as the edit node would prevent us from replacing \( A \).

![Diagram](image)

We solve this problem by just taking the parent of the leaf at the left of the edits as the edit node, node \( A \) in the picture. But this can be the wrong choice, if \( T_3 \) appended to \( T_2 \) could make a new \( B \) node. In this case, we are parsing text \( T_1 \) while this could be avoided by just choosing the other node. But as long as we have no better solution, this seems the only solution.

An algorithm for incremental parsing

With the ideas of the previous sections in mind, we can give an algorithm for incremental parsing. We start looking at the lowest node considering only the last edit action. Previous edit actions have been handled already, and it is useless to hope that the last edit action will solve the unparsed pieces in the parse tree, since previous edit
actions may get important automatically when they get covered during the search for the node that has to be replaced.
In this specification, we first place the unparsed text in the tree. Thereafter, we try to remove the unparsed text from the tree.

```c
inc_parse (last edit action)
    /* first find the lowest shared parent node of last action */
    if only insert has been done in last edit action then
        P := parent of the leaf direct at left of insert point
    else
        L := leftmost leaf that has been changed in last action
        R := rightmost leaf that has been changed in last action
        P := lowest common parent of L and R
    fi
    /* the text covered by P */
    Text := the text covered by P
    replace P by unparsed Text of the same sort as P
    /* and now try to remove the unparsed text from the tree */
    loop
        C := text covered by P
        /* Now decide the allowed sorts at replace point P */
        if P=root node
            TopSort := 'any' else TopSort := sort of P fi
        result := CheckForTree(C,TopSort)
        if result=a parse tree
            then
                replace tree at P by that parse tree
                return /* changes inserted in tree */
            elseif result = 'failure' or P=root node
                then return /* could not make new tree */
            fi
        P:=parent of P
    end loop
```

CheckForTree can be described as follows. A sequence of terminals $T_1 \ldots T_n$ is a **substring** of $L(G) \Rightarrow \exists \rho, \sigma$ such that $\rho \cdot T_1 \ldots T_n \cdot \sigma$ is in $L(G)$. We call $\rho, \sigma$ invented. If $T_1 \ldots T_n$ is in $L(G)$, it is not necessary to invent symbols. TopSort is required nonterminal (in the parse tree, we can only replace a node by another node with the same top sort).

```c
CheckForTree (Text,TopSort)
if a parse tree with right TopSort for Text exists
    then return that parse tree
elseif Text is not a substring of $L(G)$
    then return 'failure'
else return 'more-context'
fi
```

The test for 'failure' is needed in the case that the user is editing in a text for which no parse tree exists. In such cases, we want to stop trying to make a correct parse tree as soon as possible, and not go on trying parent nodes until we are at the root of the parse tree.

In chapter 3, we have separate solutions that can check if we are in a successful, failure or more-context situation or not. In chapter 4, an integrated solution is presented. The island parser only needs the context, (the new covered text *without* the old covered
text) and not the whole text to decide about parse trees and failures. Therefore, the CheckForTree call looks a little different in that case.

How hard is CheckForTree?

With the following algorithm it is shown that the CheckForTree algorithm in general is as hard as the membership question for context-free grammars (the question 'is \( w \in G \)' with \( G \) such a grammar):

```plaintext
is-element-of \( (w,G) \) /* decide if \( w \in G \) */
  if CheckForTree\( (w,'\text{any}') \) is a parse tree
    then return 'accept'
  else return 'reject'
fi
```

It is clear that is-element-of returns 'accept' if \( w \) can be parsed starting with the start nonterminal of \( G \). Furthermore, CheckForTree can never return a parse tree when this is not the case. In that case, is-element-of returns 'reject'.
3 Partial solutions for CheckForTree

The CheckForTree algorithm

In this chapter, we present some partial solutions for implementing the CheckForTree as described in chapter 2. CheckForTree now gets the following form:

```
CheckForTree (Text, TopSort)
  if there is a Test(Text, TopSort) = 'failure'
    then return 'failure' fi
  if there is a Test(Text, TopSort) = 'more-context'
    then return 'more-context' fi
  return parse(Text, TopSort)
```

The strategies to choose a test can differ. It depends a bit on the price (in calculation time) of the different tests, and their chance to return a useful result. How do our tests look like?

Some non-partial but expensive tests

Since CheckForTree has to be certain about the case, as it only can return 'failure', 'more-context' and a parse tree, we need to do a non-partial test in the case that all partial tests failed. An obvious test uses the substring parser ([RK90]). This substring parser makes parse trees given a substring of L(G) by inventing context symbols. It does not seem hard to adapt their substring parser such that it returns information about the need to invent symbols.

```
Test_substring (Text, TopSort)
  <Result, TokensInvented> := substring_parse(Text, TopSort)
  if Result = a parse tree then
    if TokensInvented
      then return 'more-context'
    else return nil /* Correct tree exists */
  fi
  else return 'failure'
fi
```

It does not seem very hard to adapt the substring parser such that it returns a tree of Sort if a normal parse of the text is possible. This way, we do not need to do a normal parse after the test. If this is done, we can make the following test, which decides in all cases:

```
CheckForTree' (Text, SortTop)
  <Result, TokensInvented> := substring_parse(Text, TopSort)
  if Result = a parse tree then
    if TokensInvented then return 'more-context'
    else return Result fi
  else return 'failure' fi
```
These tests all are quite expensive in the case that a parse at the root is needed because of a small change, like in the second example of chapter 1. This is especially the case since these tests each time reparse the total Text, and do not reuse results from previous parses. There are two solutions for this:

- Try cheaper tests first, avoiding parse actions.
- Make a parser that reuses the previous work.

The last solution is worked out in chapter 4-7. We will now consider some examples of the first solution.

Partial testing without parsing

Idea 1

The idea behind the following test is to check whether the nonterminal Sort can produce all symbols in the Text. If this is the case, a parse might succeed, but we cannot be sure of this, so we return nil, forcing a parse of the test. If one or more symbols cannot be produced by the Sort nonterminal, a parse for this nonterminal will fail. But maybe another nonterminal higher in the parse tree can help.

```
TestCharClass (Text,Sort)
    if set of symbols in Text ⊆ ProducedSymbols(Sort)
        then return nil /* parse might succeed */
    else return 'more-context' fi
```

ProducedSymbols(Sort) is a set containing all the symbols that Sort can produce: ProducedSymbols (N) = {S | N ⇒* α S β}. In the special case that Text only contains terminals, this set can be restricted to the terminals in it.

Idea 2

Another idea is to check for standard combinations of symbols. For example, an (’ is usually accompanied by a ’). Caution is needed for example in the if-then and if-then-else case. The if needs a then, but not an else. But when you see an else, there surely is an if and a then. For the resulting test, we need a multiset and some operations on it:

A multiset is a kind of set, but elements can occur multiple times. So {1,1,1,1,2,2} is a multiset, and it is not equal to {1,2}. The ordering is not important, so it is equal to {1,1,2,2,1,1}. Furthermore, we have the following operations:

- M - x: remove one occurrence of x from M, if there is such an x. Otherwise do nothing.
- M + x: add x to M once.
- \( |M|_x \): the number of occurrences of x in M.

\[
\begin{align*}
|M \cap M'| &= M \cap M' \\
|M|_x &= \min(|M_1|_x, |M_2|_x) \quad \text{for each x} \\
|M \cup M'| &= M \cup M' \\
|M|_x &= \max(|M_1|_x, |M_2|_x) \quad \text{for each x} \\
M_1 \subseteq M_2 &\iff |M_1|_x \leq |M_2|_x \quad \text{for each x} \\
M_1 - M_2 &= M \setminus M_2 \\
|M|_x &= |M_1|_x - |M_2|_x \quad \text{for each x (only possible if } M_2 \subseteq M_1) \\
\end{align*}
\]
Examples:

\[ \{2,2,3,4\} - 2 = \{2,3,4\} \]
\[ (2,2) + 2 = (2,2,2) \]
\[ \{1,2,3,4\} - \{1,2,3\} = \{4\} \]
\[ \{1,2,2,3,3,4\} - \{1,2,3,4\} = \{2,3\} \]
\[ \{1,2,3\} \cap \{2,4\} = \{2\} \]
\[ \{2,2,2\} \cap \{1,2,2\} = \{2,2\} \]
\[ \{1,2,2,2,3,4\} \cap \{2,2,3\} = \{2,2,3\} \]

First, we make a multiset for each rule in the grammar. These multisets have to contain all the terminals in that grammar rule.

Example

Syntax of G:

\[ S ::= a \ b \ S \ c \ a \ T \ d \quad (1) \]
\[ S ::= b \ e \ T \ f \ g \quad (2) \]
\[ S ::= b \ a \ h \quad (4) \]
\[ T ::= a \ u \ b \ e \quad (3) \]

Now the multisets become:

\[ M_1 = \{a,b,c,a,d\} \]
\[ M_2 = \{b,e,f,g\} \]
\[ M_3 = \{a,u,b,e\} \]
\[ M_4 = \{b,a,h\} \]

These sets intend to express what tokens have to be produced in one nonterminal. If we want to recognise a nonterminal successfully, all its symbols have to appear in the text, otherwise it cannot succeed. The idea is to see the text as a symbol multiset SM of symbols. We try to remove all symbols from SM. If we don't succeed, then it is certain that the nonterminal cannot be produced.

We take a symbol from SM, and check which symbols are accompanied by this symbol, with the multisets \( M_1 \) to \( M_4 \). Now we strip all these symbols from SM.

Example

Syntax: G as in the previous example.

Text: "a b d c a f e d" \( \rightarrow \) SM = \{a,b,d,c,e,d,f,a\}

Question: is this text in \( L(G) \)?

Approach:

1. Take a symbol from SM, say c.
2. The only set that contains a 'c' is \( M_1 \), therefore remove all symbols that are in \( M_1 \) from SM. This gives SM=\{e,d,f\}
3. Take another symbol from the new SM. For example f.
4. The only set that contains an 'f' is \( M_2 \).
5. Subtract \( M_2 \) from SM. This cannot be done, since not \( M_2 \subseteq SM \). Therefore we can be sure that "a b d c a f e d" is not in \( L(G) \).

In some cases there are more choices. If we had chosen an 'a' in step 1, both \( M_1 \), \( M_3 \) and \( M_4 \) told something about an 'a'. In that case the only thing you can be sure of, is that a 'b' is needed, because any of the 3 multisets contain a 'b'. Anyway, it is
important which symbol is chosen from SM. When there is only 1 set containing the
chosen symbol, it's easy, therefore the first check should be for this case. This idea was
used in the example. Difficulties arise when this is not the case, as shown in
the following example.

Text: "a e b e a" → FM = {a,e,b,e,a}

Question: What symbol is the best to choose from FM?
We can choose 3 symbols in this case: a,b and e. What are the consequences?

<table>
<thead>
<tr>
<th>Chosen symbol</th>
<th>Multisets containing that symbol:</th>
<th>∩</th>
<th>∪</th>
<th>∪∩</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1,4</td>
<td>[a,b]</td>
<td>[a,b,c,d,h]</td>
<td>[c,d,h]</td>
</tr>
<tr>
<td>b</td>
<td>1,2,4</td>
<td>[a,b]</td>
<td>[a,b,c,d,e,f,g,h]</td>
<td>[c,d,e,f,g,h]</td>
</tr>
<tr>
<td>e</td>
<td>2,3</td>
<td>[b,e]</td>
<td>[a,b,e,f,g,u]</td>
<td>[a,f,g,u]</td>
</tr>
</tbody>
</table>

What happens if we choose an 'a', for example? Then we can be sure that there has to
be a 'b' symbol in SM. Thus we remove both 'a' and 'b' from SM. But there are other
symbols that could be produced (this is the '∪∩' set). The view we take is to note
them as 'possibly stripped', and remove them from SM. We note them as 'possibly
stripped' because another symbol, that is left in SM, may need it. In that case it can find
it in the possibly-stripped-set.

With this possibly-stripped approach in mind, a useful tactic is to choose the symbol
with the smallest number of elements in the '∪∩' set. In the case that only one
multiset contains the symbol, it is ∅, so it's a good choice in this case, too. In the case
of example 9, symbol 'a' is chosen with this tactic (the set {c,d,h} corresponding to
choosing an 'a' contains 3 elements, other choices give 4 or 6 elements). The test
becomes:

TestSets (Text,Symbol)
SM := multiset containing the symbols in Text
PossiblyStriped := ∅
/* As long as there are symbols not stripped */
while SM ≠ ∅ do
  for each x in SM do
    Intersection_x := ∩ {M_i | x ∈ M_i}
    Union_x := ∪ {M_i | x ∈ M_i}
    Difference_x := Union_x - Intersection_x
    k := a number such that |Difference_x| is minimal
    /* Strip symbols in the chosen intersection from SM. */
    for each x in Intersection_x do
      Intersection_x := Intersection_x - x
      /* Check if symbol appears in SM*/
      if x in SM
        then /* Yes, strip it */ SM := SM - x
        else /* No, perhaps moved to PossiblyStriped? */
          if x in PossiblyStriped then
            /* Yes, strip it there because now */
            /* We're sure that token is needed */
PossiblyStripped := PossiblyStripped - x

else /* Symbol isn't there! parse will fail */
    return 'more-context'
fi

od /* All symbols in intersection stripped. */
/* Now move symbols that can be stripped to PossiblyStripped */
for each x in Difference_k do
    if x ∈ SM then
        PossiblyStripped := PossiblyStripped + x
        SM := SM - x
    fi
od /* Everything went ok, nothing concluded.. */
return nil

Idea 3

We will describe the following test only briefly. For each nonterminal in the grammar, we can make a regular expression that can recognise 'about the same' as the nonterminal in the grammar. This approximation can never be perfect, since there context-free grammars are more expressive than regular expressions ([HU79] p.61). We aim for a regular expression that accepts as little too much as possible (but never to little). Some examples of the conversion that can be done:

Definition: \( \Sigma_N = \{ S \mid N \Rightarrow^* \alpha S \beta \} \)

Grammar

| S ::= aS | b       | RE_S = a*b    |
| S ::= aSb | c       | RE_S = a*cb* |
| S ::= aSb | c       | RE_S = c | aa*cb*b     |
| E ::= (E) | E+E | E+E          | RE_E = (\( \Sigma_E \) ) | \( \Sigma_E^+ \) | \( \Sigma_E^* \) | \( \Sigma_E^* \) - | \( \Sigma_E^* \) |
| P ::= aPbPc | d     | RE_P = a*d[bΣ_P*c]* |

Hint for conversion of the last: convert P ::= aPbΣ_P*c | d instead.

Now, we make the following test. We return 'more-context' if the Text is not produced by the regular expression that we made for Symbol: we made that expression produce more than really is produced by Symbol, so if our regular expression does not produce the text, Symbol certainly won't.

TestRegExp(Text, Symbol)

if Text is produced by RE_Symbol
    then return nil /* Text seems ok */
else return 'more-context' /* Parse here will fail */

We can also make partial solutions returning 'failure' in stead of 'more-context'. For this, we look for certain restrictions for the whole grammar, and check whether the
given Text part can fulfill them. Test 3 can be transformed in such a test by making a regular expression that accepts all substrings of G. We will not work this out.

Idea 4

This idea uses some knowledge about the order of terminals in the grammar. Such a restriction for the whole grammar appears in the following example:

Grammar: \[ S ::= aSb | c \]

In this grammar, we will never find an 'a' after a 'c'. In this case, we could write \( a \leq c \). A precise definition of \( \leq \):

\[ S \leq T \iff \alpha S \beta T \gamma \in L(G) \]

The test we can make with this relation:

\[ \text{TestOrder (Text, Symbol)} \]

\[ X_1 \ldots X_n := \text{Text} \]

\[ \text{if there are } X_p \text{ and } X_q \text{ with } p < q \text{ and not } X_p \leq X_q \]

\[ \text{then return 'failure'} \]

\[ \text{else return nil fi} \]

Conclusions

All the given tests without parsing are partial, and this is not surprising, since we showed that the test is at least as hard as parsing. If there is a partial solution that works good enough in practice, then this is an attractive way to solve the CheckForTree in this way. But we have no performance indications for these tests, so we can draw no conclusions about practical performance of these tests.
4 Bidirectional recognising

Introduction

Although we have no performance tests, we don't expect the partial solutions of chapter 3 to be good enough in practice. And after all, we still need to parse when we found the right node. We saw in the CheckForTree' algorithm that a slightly adapted substring parser can decide the CheckForTree algorithm by itself. The problem with this solution is that the text that has to be parsed after going to the parent node (in inc_parse of chapter 2) will always include the text that we already parsed, but the substring parser can not reuse the work it has done. In a picture, the situation looks like (a).

![Diagram of parse trees for old and new text]

(a)

In essence the substring parses does work for all parse trees in which the old text can fit. So in fact, the work we will encounter in building the bigger parse tree has already been done. So why not reuse it? If we do so, we have an island parser. We will adapt Earley's parser for island parsing, unlike the substring parser, which adapts Tomita's parser.

A short description of Earley's recogniser

We will describe Earley's recogniser ([E70]) in short. Although a recogniser does not return parse trees, it is not hard to change it into a parser. Earley's recogniser tells whether the string $S_1 .. S_n$ is in $L(G)$. We are not interested in lookahead, so we remove this from Earley's algorithm. To explain his parser, we need a few definitions:

A dotted rule is a rule of the grammar, with a dot (•) placed in it. So if $N := a \ b \ c$ is a rule, then $N := a \ b \ • \ c$ is a dotted rule. The • indicates where we are with recognising that rule.

An item is a tuple containing a dotted rule and a number startPos, with $0 \leq startPos \leq n$ ($n$ is the length of the input string). startPos denotes where the recognition according to this rule started. An item set $U_k$, with $0 \leq k \leq n$, is a set containing items. Only items that are in some way 'useful' according to input symbols $S_1 .. S_k$ are placed in $U_k$. More precise:

An item $<N := \alpha \bullet \beta, i>$ is in $U_k$

$\Rightarrow$

$\exists$ nonterminal $M : M \Rightarrow^* \pi \ N \ \theta \ \land \ \pi \Rightarrow^* S_1 .. S_i \ \land \ \alpha \Rightarrow^* S_{i+1} .. S_k$
In a picture, the item sets are placed as follows around the input symbols (b):

\[ U_0 \quad S_1 \quad U_1 \quad S_2 \quad \ldots \quad U_{n-1} \quad S_n \quad U_n \]  

(b)

Initially, we place all items \(<N::=\alpha,0>\) in item set \(U_0\), where \(N::=\alpha\) is a grammar rule. This represents that we can recognize the first 0 (zero) input symbols using rule \(N::=\alpha\). The 0 in the tuple indicates that we started trying this rule in set \(U_0\). After this initial step, the sets are filled further using one of the following actions:

Scanner: /* continue parsing if next input symbol is ok */
\[
\text{if } \langle N ::= \alpha \beta, \text{StartPos} \rangle \text{ in } U_k \\
\text{then add } \langle N ::= \alpha \beta, \text{StartPos} \rangle \text{ to } U_{k+1} \text{ fi}
\]

Predictor: /* adds items that can produce needed nonterminals */
\[
\text{if } \langle M ::= \alpha \beta, \text{StartPos} \rangle \text{ in } U_k \\
\text{with } N \text{ nonterminal and } N ::= \sigma \text{ in } G \text{ for some } \sigma \\
\text{then add } \langle N ::= \sigma, k \rangle \text{ to } U_k \text{ fi}
\]

Completer: /* 'Scans' other items after a nonterminal is produced */
\[
\text{if } \langle M ::= \alpha \beta, \text{StartPos} \rangle \text{ in } U_k \\
\text{and } \langle N ::= \sigma, k \rangle \text{ in } U_m \\
\text{then add } \langle M ::= \alpha \beta, \text{StartPos} \rangle \text{ to } U_m \text{ fi}
\]

Adapting Earley for symmetry

The first step towards bidirectional recognizing (recognizing in the normal left-to-right way and in the reverse way) is to remove the left-to-right bias from Earley's algorithm. The left-to-right bias appears in the StartPos in the item tuple. We will replace it by the left and right 'immediate neighbour items' of the state. So our item is a triple containing a dotted rule, a set of left neighbour items and a set of right neighbour items. The left neighbours are items having a dotted rule with the \(\bullet\) moved one symbol to the left, and the right neighbours have the \(\bullet\) moved one symbol to the right.

But written in this way, there seems to be double storage of this data: an item \(A\) has neighbour \(B\), and \(B\) has neighbour \(A\). It seems more convenient to think about a relation between \(A\) and \(B\). We express this relation by a link. A link is a tuple \(<\text{Left},\text{Right}>\), where \(\text{Left}\) is an item in an item set, and \(\text{Right}\) is another item with the \(\bullet\) in its dotted rule moved one symbol to the right. In a picture, links look like (c).
If there is a link with 'left'=A and 'right'=B, we say that there is a link between A and B. We also write $A \Rightarrow B$. If $A \Rightarrow \ldots \Rightarrow B$ (or $A=B$), we write $A \Rightarrow^* B$.

A link is an explicit reflection of Earley's Completer action: when we found a production for a useful nonterminal, we trigger a completer action. Earley does not make links, but just adds new items, if it finds old items that need the nonterminal. We also make a link from the old to these new items.

Furthermore, we make shift possibilities. A shift possibility is a triple $<S,U_m,U_n>$. $S$ is some symbol, $U_m$ and $U_n$ are itemsets. A shift possibility indicates that symbol $S$ has been recognised, using the input symbols $S_{m+1} \ldots S_n$. Usually, $S$ will be a nonterminal, but this definition allows us to make shift possibilities for terminals. Formally: $<S,U_m,U_n>$ is a shift possibility $\Rightarrow (S \Rightarrow^* S_{m+1} \ldots S_n)$.

We define the following properties of items:

We call an item left-complete $\Rightarrow$ its dotted rule is of the form $S ::= \bullet \alpha$.

We call an item right-complete $\Leftrightarrow$ its dotted rule is of the form $S ::= \alpha \bullet$.

We call Item complete $\Leftrightarrow$

$\exists$Item' : (Item $\Rightarrow^*$ Item' and Item' is right-complete) and

$\exists$Item" : (Item" $\Rightarrow^*$ Item and Item" is left-complete)

Intuitively, an item is complete if it is between some left- and right-complete item.

Recognising with these notions

We will first give some examples, to show how these notions can be used to make a recogniser.

Grammar: $S ::= a \ b$

Input string: $a \ b$

We start with an initial itemset that is located before the 'a' symbol of the input. We want to try to recognise $S$, so the dot is placed at the left of the production rule. This looks like (a). To save space, we remove the spaces that separate the terminals from the production rules. In our example, this gives no problems, since all symbols consist of only one character.

This is set $U_0$. Used for reference.

an item set

the dotted rule of an item in this set
Now, we want to process the 'a' of the input string. Therefore, we make a new, empty item set and a shift possibility that indicates that an 'a' has been 'recognised':

![Diagram showing a shift possibility for 'a' between 0 and 1.]

The $S::=\ast ab$ item can use this shift possibility, since the left side of the shift possibility points at the set the item is in, and the item just needed to shift an 'a', which is made possible by the shift possibility. So the item shifts its 'a', and makes an item and a link to it in set 1, the other end of the shift possibility. The result is (c).

![Diagram showing a link and a shift possibility.]

As we can see, a link is a kind of reflection of an existing shift possibility. So it is not necessary to make the links explicit. Instead, we can refer to that shift possibility. But we will draw the link in our pictures, because it makes the situation more clear than a reference to a shift possibility.

Next, we want to process the 'b' of the input string. So we make a new item set and a new shift possibility for the 'b'. The $S::=a \ast b$ item in set 1 uses this shift, resulting in a new link and a new item in the new set. The situation now looks like (d).

![Diagram showing a shift possibility and a link.]

The $S::=ab \ast$ item is complete (it is right-complete and $S::=\ast ab \Rightarrow S::=a \ast b \Rightarrow S::=ab \ast$). This means that the $S$ nonterminal has been recognised. Therefore, a new shift possibility for $S$ between set 0 and set 2 is added. The situation now looks like (e). The creation of this shift possibility does not trigger any further actions, since there are no items having the $\ast$ before an $S$ in set 0.
The input sentence 'ab' has been recognised, because we found a shift possibility between the leftmost itemset (0) and the rightmost itemset (2). The top nonterminal of the tree we found is S.

Now we will look at an ambiguous grammar.
Grammar: \[ S ::= x \mid SS \]
Input string: \[ xxx \]
We place both rules with the dot at the left in the initial set, and we make a shift possibility for x between set 0 and 1. The situation looks like (a).

The S::=•x item can use this shift possibility, and makes a new link and item in set 1. This item is complete, so a new shift possibility for S between 0 and 1 is made. The situation is like (b). This new shift possibility is be used by S::=•SS. A new item and link is made (c).

Something more has to be done. The S::=S•S item tries to shift an S, but there are no production rules for S with a • at the left in set 1, so there will never be a shift
possibility with the left side at this set, if we do nothing. So a predictor action like in Earley’s algorithm is needed. In fact, nothing is predicted, but we simply put all the rules that can recognise an $S$ with the dot at the left in set 1. The situation now looks like (d), and nothing more can be done.

Next, we process the second $x$ of the input string. We add another shift possibility for $x$, between set 1 and set 2. The $S::=\cdot x$ item can use it, and makes an $S::=x\cdot$ item in 2. Because $S::=x\cdot$ is complete, a shift possibility for $S$ between 1 and 2 is made. Both the $S::=\cdot SS$ and the $S::=S\cdot S$ item can use it, resulting in two new items and links (e).

![Diagram](image)

The $S::=SS\cdot$ item is complete, so a new shift possibility is made between 0 and 2. The $S::=\cdot SS$ item in 0 can use it, resulting in a new link to the $S::=S\cdot S$ item in set 2. The item already exists, so no new item has to be made (f).

![Diagram](image)

Again, a predictor step is needed, because the $S::=S\cdot S$ item tries to recognise an $S$. Therefore, the items $S::=\cdot x$ and $S::=\cdot SS$ are added to set 2 (g).
Now, we are ready to process the last x of the input string. We make a new shift
possibility for x. The S ::= *x makes a new link and item. The item is complete, so a new
shift possibility for S between sets 2 and 3 is made. Items S ::= SS* and S ::= S*S use that
shift possibility. They make new items S ::= S*S and S ::= SS* (h).

The item S ::= SS* is complete, in two ways. First, because it is linked to S ::= *SS in set 1.
This results in a shift possibility for S between 1 and 3. Second, because it is linked to
S ::= *SS in set 0. This results in a shift possibility for S between 0 and 3 (i).
The shift possibility between 0 and 3 is used by $S := \cdot SS$, resulting in a link to item $S := S \cdot S$ in 3. The shift possibility between 1 and 3 is used by 2 items in set 1: $S := \cdot SS$ makes a link to $S := S \cdot S$ in 3. $S := S \cdot S$ makes a link to $S := SS \cdot$ in 3 (i).

The item $S := SS \cdot$ in set 3 now has another way to be complete: by following the link to $S := S \cdot S$ in 1, and then back to $S := \cdot SS$ in 0. But this gives a shift possibility between 0 and 3, which already exists. This double way to make a shift possibility exactly represents the ambiguity in our grammar. Now, we have processed all input symbols, and we found a shift possibility for $S$ between the leftmost and the rightmost set. This means that we recognised the input string. The two parse trees representing the input are (k):
Finally, we will look at an example with an empty production.

Grammar: \( S ::= x \mid SS \mid \epsilon \)

Input string: \( x \)

Again, we start with placing all rules with a dot at the left in a set (a). The \( S ::= \cdot \) item is complete, so a shift possibility between 0 and 0 is made, indicating that an \( S \) was recognised without using any symbols (b).

The \( S ::= \cdot SS \) item uses this shift possibility. The situation now looks like (c) (We made the link somewhat thinner in the picture, otherwise the new item would be unreadable). The \( S ::= S \cdot S \) item can do its shift directly after it is added. It adds an \( S ::= SS \cdot \) to the set. Although this item is complete, nothing is done, since the resulting shift possibility already exists (d).

The state of the pushdown automaton now looks like (e).
We are ready to process the 'x' of the input string. We add a shift possibility for x between 0 and 1. The S:=•x item uses it. It is complete, and a shift possibility for S between 0 and 1 is made. The S:=•SS and S:=S•S in 0 use this shift possibility, and also make a link to set 1 (e). The S:=SS• is complete, but the shift possibility already exists, so nothing happens.

Because the S:=S•S item tries to shift an S, we add all rules that can produce an S to set 1. Because item S:=• is added, a shift possibility for S between 1 and 1 (f) is added.

The items S:=•SS and S:=S•S were waiting for this, and new links are made (g). The S:=SS• in set 1 now is complete in 2 new ways, but both shift possibilities already exist. The x has been recognised, since we found an S shift possibility between 0 and 1.

An algorithm for Left to Right recognising

In this section, we give a more formal description of the parsing method illustrated in the previous sections. In the algorithms, S is a symbol, N a nonterminal.

recognise \((X_1 \ldots X_n, G)\)
for \(k = 0\) to \(n\) do \(U_k := \emptyset\) od
for each \(N := \alpha\) in \(G\) do add \((N := \cdot \alpha, U_0)\) od
for \(k = 1\) to \(n\) do
make shiftposs between \(U_{k-1}\) and \(U_k\) for \(X_k\)
while an action is possible do action od
di
if there is a shiftposs between \(U_0\) and \(U_k\)
then return 'accept' else return 'reject' fi

The possible actions are as follows. Note that the scanner only makes moving the • to the right possible. The predictor only predicts to the right. This still is a kind of left-to-right bias.
scanner /* makes new items and links */
if there is some set \( U_k \) containing an Item \( N ::= \alpha \cdot S \beta \)
and there is some \( k \leq n \) /* n=#input symbols */
and is a shift possibility for \( S \) between \( U_k \) and \( U_m \)
then
Item' := add(N ::= \( \alpha S \cdot \beta, U_m \))
make a link from Item to Item'
fi

completer /* makes new shift possibilities */
if Item with dotted rule \( N ::= \star \alpha \) in some set A
and Item' with dotted rule \( N ::= \alpha \cdot \) in some set B
and Item \( \Rightarrow^* \) Item' then
make new shift possibility for \( N \) between A and B
fi

predictor /* adds items that produce needed nonterminals */
if Item with dotted rule \( M ::= \alpha \cdot N \beta \) in some set A
and \( N ::= \gamma \) is production of \( G \)
then add Item' with dotted rule \( N ::= \star \gamma \) to A fi

add \( (N=\alpha \cdot \beta, Set) \) /* find the item with this dotted rule */
Item := find an item with dotted rule \( N ::= \alpha \cdot \beta \) in Set
if Item not found
then add Item without links with dotted rule \( N ::= \alpha \cdot \beta \) to Set
fi
return Item

Comparison with Earley's algorithm

In this section, we will compare Earley's algorithm and our left-to-right algorithm. First we look at an example.

Grammar: \[ A ::= x \mid AA \]
Input string: \[ xxxx \]

If we follow Earley's algorithm, the result is this table ([E70] p.99). We removed lookahead data and end-of-input markers, and adapted it to our notation.
<table>
<thead>
<tr>
<th>Item set</th>
<th>Dotted rule</th>
<th>StartPos</th>
<th>Item set</th>
<th>Dotted rule</th>
<th>StartPos</th>
</tr>
</thead>
<tbody>
<tr>
<td>U₀</td>
<td>A := •x</td>
<td>0</td>
<td>U₃</td>
<td>A := x⁺</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>A := •AA</td>
<td>0</td>
<td>A := AA⁺</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>U₁</td>
<td>A := •x⁺</td>
<td>0</td>
<td>A := AA⁺</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A := A⁺A</td>
<td>1</td>
<td>A := A⁺A</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A := •x</td>
<td>1</td>
<td>A := x</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A := •AA</td>
<td>3</td>
<td>A := •AA</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>U₂</td>
<td>A := •x⁺</td>
<td>1</td>
<td>U₄</td>
<td>A := x⁺</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>A := AA⁺</td>
<td>0</td>
<td>A := AA⁺</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A := A⁺A</td>
<td>1</td>
<td>A := AA⁺</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A := A⁺A</td>
<td>0</td>
<td>A := AA⁺</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A := •x⁺</td>
<td>2</td>
<td>A := A⁺A</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A := •AA</td>
<td>2</td>
<td>A := A⁺A</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A := A⁺A</td>
<td>4</td>
<td>A := •AA</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

We want to compare these sets with our pictures. Therefore, we put the items that are produced by Earley's algorithm in a picture, and make links with arrows. The result is (a).

![Diagram](image-url)

If the same grammar and input string is provided to our algorithm, we get picture (b).
As illustrated by these pictures, we notice the following similarities and differences:

**similarities**

- Both use dotted rules.
- Both have a set between each two input symbols.
- Each dotted rule can occur at most one time in a set between two input tokens.
- The invariant\(^1\) is the same for the items\(^2\). This can be seen in the pictures because the sets in Earley's and our recogniser contain states with exactly the same dotted rules.
- We have similar actions as in Earley:
  - 'scanner' handles the shift of some symbols.
  - 'predictor' does exactly the same.
  - 'completer' works out similar

**differences**

- Our scanner handles all shifts, Earley's scanner handles only terminal shifts.
- We have an explicit notation for recognised nonterminals: the shift possibilities. In Earley, this is not the case.
- We do not have a look-ahead possibility.
- We have links between two items, Earley has links between an item and a set.
- Perhaps the most important difference is that our recogniser is symmetric for L-R and R-L recognising.

---

\(^1\) The invariant was given at the beginning of this chapter.

\(^2\) This only holds for the left-to-right version of our parser without removing of useless items.
Discussion of the symmetry difference

Earley's algorithm gives troubles if we try to reverse the parse direction. This is caused by the way the links are made. In Earley, a link is maintained to the set containing the 'left-complete' (in our terminology) item. This works perfect if only is parsed from left to right, since an item is always left-complete in that case. However, in the island parser of our final goal, we need to start somewhere in the middle of recognising an item, resulting in items with only a temporary left-complete item. In this case, it gives problems to link to a left-complete item. In our recogniser, an item maintains links to immediate neighbours of it. Because of this, a recognise action is more complicated and will cost more. We will look at a possible solution for this problem in chapter 7, and work it out in chapter 8.

Example of Right to Left recognising

Because right-to-left recognising is symmetric to left to right recognising, we limit ourselves to an example of right to left recognising.

Grammar: $S ::= b \mid cS$
Input string: $cb$

Just as in the case of left to right recognising, we start by placing all rules in a set, with the dot at the start. In right to left parsing, this means placing the dot at the right (a). To process the first (the rightmost in right to left recognising!) symbol, we make a new shift possibility for $b$ between 0 and a new set 1. Note that this set is left of the old set (b). Also note that our numbering gets a little inconsistent, but that is no real problem.

![Diagram](a)

The $S ::= b \bullet$ item can use the shift, and places an $S ::= \ast b$ item in 1. That item is complete, and it adds an $S$ shift possibility between 0 and 1. The $S ::= cS \bullet$ item uses that shift, resulting in an $S ::= c \ast S \bullet$ item in 1 (c). The predictor does nothing, because there is no nonterminal at the left of the $\ast$. 

![Diagram](b)
Next, we add a shift possibility for the first 'c' (when reading from right to left) between 1 and 2. The $S ::= cS$ uses it, and makes an item $S ::= cS$ in 2. This item is complete, so a new shift possibility for $S$ between 2 and 0 is made (d). The $S ::= cS$ item in 0 uses it, and makes an item $S ::= cS$ in 2 (e).

We expect that the reader now understands the symmetry in our parsing methods.
5 Island Parsing

This chapter consists of two parts. First, we will make an island recogniser. Then, we will turn this island recogniser into an island parser.

I: Adapting the bidirectional recogniser for island recognising

The context idea

The basis idea for island recognising is simple: since the data structures look the same for both directions, we simply parse in the direction we need. But there is a problem when we want to turn the direction: initially, we placed all rules with the dot at the start in an initial set, indicating that we want to recognise these rules. But what is the start side when we may change the direction? In fact, the 'initial' item set may end up somewhere in the middle of the input string, when we allow turning the direction.

The solution is to put the dot at any place in any production rule. The items that are needed certainly exist in this case. But this does not solve all the problems. We are going to give some examples to show that we need some notation for unknown context. Look at the following example:

Grammar:

\[ B ::= b \cdot a \]
\[ S ::= c \cdot B \]

input string:

first an 'a' at the right, and then 'c b' at the left

As said, we put the production rules with the dot at any place in the initial itemset (a). Now, we make a shift possibility for the 'c', at the right side. The item \( B ::= a \cdot b \) uses it, and makes a new item \( B ::= ab \cdot \) in set 1. This item is not complete, so we are ready (b).
Next, we make a shift possibility for 'b', but now on the left side. The $B := b \cdot a$ can use it, is complete and makes a shift possibility (c).

As a last step, a shift possibility for 'c' is made (d). But no action is possible! We expect a shift possibility for $S$ between 3 and 1.

The problem is caused by the absence of an $S := cB \cdot$ item in set 1. That item was not added, because the $B := ba \cdot$ was found to be right-complete, but not left-complete. The $B := ba \cdot$ was located somewhere in an unknown context at that time (b).

As shown, some sets contain too much items (set 0, e.g. the $B := ba \cdot$ is useless, since it will never be able to shift the 'b') and others too little (set 1, where we missed an $S := cB \cdot$ item). We will show how the situation of a set containing too little items can be avoided, by means of a context set.

Another interesting point to use a context is to decide between 'failure'and 'more-context' (chapter 2). Consider the following example:

Grammar:

\[
S ::= A \mid B \\
A ::= a A \mid \varepsilon \\
B ::= b B \mid \varepsilon
\]
Input:
'a' at the right, then 'b' at the left (note that 'b a' is no string in L(G))

After all dotted rules are placed in 0, and a shift possibility for 'a' to set 1 is made, the situation looks like (a) (only items and shift possibilities that are of interest for us to show the point are shown).

We make a shift possibility for 'b', between 0 and 2. Now, the situation looks like (b). Since we found no shift possibility between 1 and 2, we know that we did not recognise the input. But what can we say about usefulness to continue parsing\(^1\), by only looking at this picture? We think that the best solution is to place something representing any possible left context at the left of set 2, and something representing any right context at right of 1. If it is possible to create at least one shift possibility using the left or right context, we know that it still is possible to recognise the input (the 'more-context' case), otherwise we can safely return 'failure'.

We make an object representing any possible context as follows: we place items with all possible dotted rules in a set. This represents that each possible dotted rule can be reached with an appropriate input. Next, we make a shift possibility for \(\gamma\) between that set and itself, and do all possible actions to get the appropriate links. The \(\gamma\) is a symbol that matches any other symbol. This shift possibility represents that it is possible to recognise any symbol in the context\(^2\).

Grammar:
\[
S ::= SSx \gamma p
\]

\(^1\) The question whether we should return 'more-context' or 'failure', see chapter 2

\(^2\) We assume that all nonterminals can produce some terminal-only string. If this is not the case, a \(\gamma\) shift possibility (next section) in stead of an \(S\) shift possibility can be used.
The context set now looks like (c). Note that no additional shift possibilities for $S$ are made, although they are complete. This is because we already have a ? shift possibility.

Using a context set

In this section, we will illustrate the use of a context set. If we are working from left to right, we only need a left context, because we are in a situation very similar to Earley without lookahead: items are added without knowledge about the rest of the symbols that are going to be parsed. For the same reason, we only need a right context if we parse from right to left.

Because of this, we need to swap the context set to the other side, if we want to change the parse direction. This is one of the reasons to keep a separate context set, and not just change the outermost sets into context sets.

So the intention is that the context can only be reached by shifting at least one unknown input symbol. We enforce this by making a shift possibility for $T$ from the outermost sets to a context set. The $T$ matches with any terminal, but not with nonterminals. Working this way, shift possibilities to the context are made explicit, which is just what we needed. The situation is shown in (a).

While parsing, we use only one context. But when we need to decide about failure (chapter 2), we need a context at both sides. We will look at this after the following example of left to right parsing using a context.
Grammar:

\[ S ::= B | SS \]
\[ B ::= a \cdot b \]

Input string: b a b

Now, we start with a situation like (b). Note that both \( B ::= a \cdot b \) and \( B ::= a \cdot b \) can shift their terminal, since both matches with the \( T \) shift possibility. We have not added the right context, since we want to parse to the right. The \( B \) shift possibility was made because the \( B ::= a \cdot b \) item in 0 is complete with appropriate left context. Because of this shift possibility, the \( S ::= B \) is also complete, causing an \( S \) shift possibility.

We start parsing by adding a shift possibility for \( b \) between 0 and 1. Item \( B ::= a \cdot b \) can use it, and makes \( B ::= ab \) in 1. This item is complete because it is linked to \( B ::= a \cdot b \) in \( C \). Note that this was not true if we did not have a context, in that case nothing would happen. So a shift possibility between \( C \) and 1 for \( B \) is made (c).

The \( S ::= B \) in \( C \) can use this shift possibility, and makes an \( S ::= B \) in 1. This item is also complete, so a shift possibility for \( S \) between \( C \) and 1 is made. This shift possibility is used by \( S ::= SS \) and \( S ::= S \cdot S \) in \( C \). Finally, the predictor adds items \( S ::= B \), \( S ::= SS \) and \( B ::= a \cdot b \) to set 1 (d).

At this point, we can see that only the \( b \) gives no complete parse tree, since there is no shift possibility between 0 and 1. We can also see that an appropriate left context can give both a tree with \( B \) and a tree with \( S \) at the top. Now, we process the rest of the input string. First we make a shift possibility for 'a' between 1 and 2. Only \( B ::= a \cdot b \) uses it, and it only makes \( B ::= a \cdot b \) in 2.
Finally, we make a shift possibility for the last 'b' of the input string, between 2 and 3. The B::=a•b in 2 uses it, makes B::=ab• in 3, which is complete. This makes a shift possibility between 1 and 3. This is used by S::=•B in 1. This causes a shift possibility for S between 1 and 3 (e).

This S shift possibility is used by S::=•SS and S::=S•S, resulting in S::=S•S and S::=SS• in 3. The S::=SS• is compete, resulting in a shift possibility for S between C and 3. This is used by S::=•SS and S::=S•S, resulting in new links. At the end, the predictor adds S::=•SS, S::=•B and B::=•ab to 3. Then we are ready (f).

Switching the context to the other side

If we want to change the working direction, the context has to be moved to the other side. An easy way to do this is as follows. To distinguish between the two contexts, we call the left context $\mathcal{L}$ and the right context $\mathcal{R}$. 
SwapContextToRight
S := rightmost set
L := left context set
R := Make a new context set /* the new context set */
put R at rightmost position
Make a T shift possibility between S and R
while an action is possible do action od
remove(L)
remove(IItemSet)
Remove all links to items in IItemSet
Remove all shift possibilities between IItemSet and other sets
Remove the items in IItemSet
Remove the emptied IItemSet

Example:
We start from situation (d) of the previous example. We want to add the missing 'a' at the left. Before this can be done, we need to move the context to the right. Therefore, we add a T shift possibility to a new context R, and we do all actions (e').

Next, we remove L, resulting in (f).
We are ready to parse leftwards, so we add a shift possibility for 'a' between 0 and 2, with 2 at the left of 0, and we do the possible actions. The result is (g').
We found a successful parse for ab because of the shift possibility for S between 1 and 2, and we know that extending the context to the right may be successful because of the shift possibility between 2 and $\mathcal{S}$.

Deciding about the result

If we found a shift possibility between the left and the right normal set, we can accept the input. But if no such a shift possibility has been made, the next question is whether it is useful to try a larger context\(^1\). As we saw in several examples, we can decide for this by looking for shift possibilities using context. During parsing, we only have one context, say the left context. But it may be possible that the parsed input string can only be extended to the right. There can be no shift possibility representing this, as we assumed that there only was a left context. So to decide between failure and more context, we need both a left and a right context. We can make two contexts without problems, as we saw in the previous section. We only need to remove one of them when we want to continue parsing. The algorithm for deciding about the result is as follows:

```plaintext
decide_result (S)
  /* S is the needed topnode sort in the parse tree */
  U_0 := leftmost normal set
  U_n := rightmost normal set
  if there is a shift possibility for S between U_0 and U_n
    then return 'accept' fi

  /* no accept. Find a shift possibility using some context */
  if there is only one context then make other context fi
  L := left context set
  R := right context set
  if there is a shift possibility between L and (R or U_n)
    or there is a shift possibility between U_0 and R
    then return 'more-context'
  else return 'failure' fi
```

With the shift possibilities between left and right context as described here, it is possible to make a more sophisticated incremental parser than the one described in chapter 2. In the algorithm of chapter 2, we go to the parent when more context seems to be useful. The following options are not considered there:

- We can see what nonterminals will possibly result from more context by looking at the shift possibilities. For example if the parent is nonterminal Q, but we have no shift possibility for Q using context, we already know that that parse will not succeed (but both 'fail' and 'more-context' are still possible at node Q).
- If we have no shift possibilities using right context, but the node one higher has right context, we already know that we are in a 'failure' situation.

And there are other similar tests that can be done.

---

\(^1\)The question whether we should return 'more-context' or 'failure', see chapter 2
An algorithm for island recognising

In this section, we will look at the methods that are needed for an island recogniser. 'new' makes a new recogniser, and 'extend' continues the recognising of some text in a given direction. It returns the result, either 'accept' (this is not a parser), 'more-context' or 'failure', conform to the ideas of chapter 2.

In the algorithms in chapter 4, there still is a left-to-right bias. This way, we prevented the addition of too much items. But now it is possible to turn the direction during recognising, returning the problem of addition of many items. For example, in example (g') two pages back, the shifter now can add B=a•b to set 1, because it is possible to shift the b using shift possibility T from S. We did not do so, because this is not what we aimed for: we already added all the items that once might be important, before we continued with the next input symbol.

We solve the problem by only permitting addition of items to the last made itemset. To make this explicit, we use a global variable 'Outermost' indicating that item set.

Some definitions:

• We distinguish between 'normal' and 'context' itemsets. An itemset is 'context' if there is a ? shift possibility between the set and itself.
• left⁻¹ = right; right⁻¹ = left
• S stands for a symbol, that is a (non)terminal. N stands for a nonterminal.
• In the actions, outermost is the rightmost set in left-to-right parsing. It is the only set that can be changed. Direction is the current working direction.
• Direction-most should be read as leftmost when Direction='left', and as 'rightmost' when Direction='right'. Direction is a global variable in this specification.

new (Grammar)
  start in an empty situation
  Direction := 'right'
  G := Grammar /* for later references to it */
  L := a new context itemset, at leftmost position
  Outermost := a new itemset with all dotted rules, at right of L
  make a T shift possibility between L and Outermost
while an action is possible do action od

extend (X₁...Xₙ,Direction,S)
  /* S is the needed sort in the parse tree */
  set_direction(Direction) /* context at correct side */
  for 1≤k≤n with k counting in Direction do
    U := Direction-most itemset
    Outermost := a new Direction-most itemset without items
    make shift possibility for Xₖ between U and Outermost
    while an action is possible do action od
  od
  return decide_result(S)

set_direction (Direction)
  if not Direction-most itemset is a normal set then
/* context at wrong side.. */
  if there is only one context then make other context fi
  remove(Direction-most itemset)
fi

The possible actions, adapted to use Direction and Last:

scanner
  if Item in set A has dotted rule DR with an S at the
    Direction side of the •
    and there is a shift possibility for S between A and Outermost
    then
      Item' := add(DR with • moved 1 symbol to Direction)
      make a link between Item and Item', pointing to item
        with the • at right side of S
    fi
completer
  if Item with dotted rule N=\alpha in set A
    and Item' with dotted rule N=\alpha• in set B
    and (A=Outermost or B=Outermost) and Item \Rightarrow* Item'
    then
      make new shift possibility for N between A and B
  fi

predictor
  if Item in Outermost has N at Direction side of the •
    and N:=\gamma is production of G then
      /* add item with this production and dot at start */
      DottedRule := N:=\gamma, with the • at Direction⁻¹
      add(DottedRule)
  fi

add (N=\alpha•\beta) /* returns item with this dotted rule in Outermost*/
  Item := find an item N:=\alpha•\beta in Outermost
  if Item not found then
    add Item N:=\alpha•\beta to Outermost
    if DottedRule is an empty production then
      make shift possibility for N
        between Outermost and Outermost
    fi
  fi
return Item

We are getting close to the implementation, but before we give it, another problem
has to be solved.

Building parse trees

As usual with parsing problems, we started building a parser, and ended up with a
recogniser. In this section, we will turn the recogniser into a parser.
We have ended up with a situation containing a shift possibility from the leftmost normal to the rightmost normal set, producing the right nonterminal. The recogniser now simply returns 'accept', but the parser has to return a parse tree instead. We know two ways to derive parse trees when we have ended up with this situation:

1. Find out how the shift possibility was made. This can be done by searching a production rule for which there are adjacent shift possibilities, such that the left shift possibility starts in the leftmost normal set, and the right shift possibility ends in the rightmost normal set. This search seems to imply a lot of work. Even removing useless items (next section) seems of little use here.

2. When a shift possibility is made, remember how it was made. We can do so by remembering the links that were used, or better the shift possibilities that made these links possible. This way, the search of the solution 1 is avoided. Furthermore, the resulting data structure is very close to a parse tree.

We will use the second solution, because it makes deriving parse trees easy.

What happens with ambiguities? With the chosen solution, a shift possibility will be added twice, but with different used links. We can handle ambiguities in two ways:

1. Discard the second way to make this shift possibility. Although saving some storage, this does not seem to have any advantages.
2. Store the second way as alternative of the first way.

We will use the second way to handle ambiguities. The result is a way of storing parse trees similar to the structure described in [R92], Chapter 2. This way of storing parse trees may be expensive. For very ambiguous grammars, it will require exponential space to store the parse trees. In chapter 7, an idea to store them in O(n^3) is described.

II: Removing useless items

Useless items are items that do not help in building a useful shift possibility. A shift possibility is useful if it helps us for deciding the result (so if it is a shift between the leftmost normal or context set and the rightmost normal or context set). A shift possibility is also useful if it helps creating a useful shift possibility, by making one of the links for it.

Example. Take the second grammar of this chapter,

Grammar:

\[ S ::= A \mid B \]
\[ A ::= aA \mid \varepsilon \]
\[ B ::= bB \mid \varepsilon \]

As shown there, we get situation (b) after giving a right context 'a', and a left context 'b':
We provided 'b a' to the parser, which is no string in L(G), and cannot be extended to be one either. We can see this by making two contexts. If we do so, the situation looks like (c). In this situation, we can see that more context does not help: there is no shift possibility between (L or 2) and (1 or 3). So there are no useful shift possibilities and no useful items. The aim of this section is to remove all useless items (this means, all the items in set 0,1 and 2 in case (c))!

Another good example is picture (g'), on page 41. In set 0 for example, only B::=a•b is useful, because it is the only item that helps constructing a useful shift possibility between 2 and 1.

An invariant for the items

In fact, we want the items to fulfil the following invariant:

If we have processed input X_1..X_n, and we have itemsets U_0..U_n, then:

\[
\text{Item with dotted rule } N=\alpha\cdot\beta \in U_k \\
\iff \\
\exists N': N' \Rightarrow^* \gamma \Delta \\
\Rightarrow \gamma \alpha \beta \Delta \land \gamma \Delta \\
\Rightarrow \omega X_1..X_k \land \beta \Delta \\
\Rightarrow^* X_{k+1}..X_n \omega' \text{ for some } \omega,\omega'
\]

That is, we want each item to have possibilities to be fit into a derivation that has X_1..X_n as (sub)string of L(G) ([RK90]). We say that an item is useful if it fulfills this invariant. An item is useless if it does not.

Note that the context sets do not fit into this invariant notation because they are not an U_k in our notation. Furthermore, an item in L can contain items that produce many context before actually producing X_1..X_n. So if we want an invariant for the context, it is like

\[
\text{Item with dotted rule } N=\alpha\cdot\beta \in L \\
\iff \exists N': N' \Rightarrow^* \gamma \Delta \\
\Rightarrow \gamma \alpha \beta \Delta \land \gamma \Delta \\
\Rightarrow \rho X_1..X_n \sigma
\]

\[
\text{Item with dotted rule } N=\alpha\cdot\beta \in R \\
\iff \exists N': N' \Rightarrow^* \gamma \Delta \\
\Rightarrow \gamma \alpha \beta \Delta \land \gamma \alpha \\
\Rightarrow^* \rho X_1..X_n \sigma
\]

We will only remove useless items from the normal item sets, and not look at the context sets. It is not very hard to do so, but we introduced a context set as a set with all dotted rules, which conflicts with the idea of removing items from it. Furthermore, we do not see any advantages in doing so.

A solution for removing useless items

A solution to find useful items is to mark the useful shift possibilities, like we noted in the examples. If a shift possibility is marked as useful, we can conclude that the items that were needed for that shift possibility are also useful. For example, if a shift possibility for S between A and B is known to be useful, and we know that that shift possibility was made by using the production S::=aSb, the items S::=•aSb in set A and S::=aSb• in set B are useful. Note that we can easy decide the rule and and shift
possibilities that were used for making a shift possibility by remembering the cause of
a shift possibility when we make it, and do not discard ambiguities, like described in
the section 'Building parse trees'.
But what to do with the other items? In our example, the items between S::=a*SB and
S::=aSB* (some items with dotted rule S::=a*SB and S::=aS*b) are also useful. In our
solution, we do not recognise them as useful, but they will be saved as long as they are
'hanging' with their links between two useful ends.

This approach gives the following algorithm:

remove_useless
mark all items and shiftpossibilities as useless
for each ShiftPoss between (leftmost normal or context set)
and (rightmost normal or context set)
do useful(ShiftPoss) od
for each ShiftPoss marked with useless do remove ShiftPoss od
for each unmarked left- or right-complete Item
do remove(Item) od

useful (ShiftPoss)
if ShiftPoss already marked as useful then return fi
mark ShiftPoss as useful
for each way to make ShiftPoss do
'N::=α' := rule that was used
mark item in left set that ShiftPoss points to
that has dotted rule N::=•α
mark item in right set that ShiftPoss points to
that has dotted rule N::=α*
for each ShiftPoss' used for one of the links
between these two items
do useful(ShiftPoss') od

remove (Item)
for each Link between Item and Item' do
remove Link
if Item' now has no more links at the side
where we removed the link
then remove(Item') fi
od
remove Item

Incremental removal of useless items

In this section, we will try to make the removal of useless items incremental. At first
glance, one thinks that there must be an easy way to remove them, directly after the
parser is extended with one terminal. It seems easy to see which items can not do their
shift, remove them and check whether this influences other items. Once an attempt is
done to make an algorithm for it, this approach turns out to give troubles. We will try
to show the problem with an incorrect algorithm.
Consider the following incremental 'solution'.

Assume that there is a left context, and we are parsing to the right. The general idea is to mark the items that can 'reach the right side'. Since any item reaches the left side because of the way we add items, we expect that the marked items are useful. After the marking process, we remove all unmarked items. We have 2 possible marks: needed (N) and reaches right side (R). An item is marked N if another item uses the shift it produces. An item is marked R if it is uplinked to an item in the rightmost set, or wants to shift a nonterminal and there is a rule producing that nonterminal in the same set, marked with R. Furthermore, we have an ordering on this marks: R>N>no mark. Each item prefers the highest possible mark. If this approach would work, we could try to update the marks in an incremental way.

```plaintext
remove_uncompletables(ItemSets)
  for each Item do Item.mark := no mark od
  for each Item in rightmost itemset do Item.mark := R od
  while an action is possible do action od
  for each Item with Item.mark = no mark
     do remove item and links od
  /* note that we do (can) not remove shift possibilities. */
```

We have the following actions to give Item higher mark. An action is only done if it gives a higher mark to Item.

```plaintext
use_mark_of_neighbour
  if ∃Item → Item' then Item.mark := Item'.mark fi

someone_needs_us
  if ∃Item' with dotted_rule M := βN*γ in U_k
     and Item'.mark = N or R /* item is useful */
     and ∃Item with dotted_rule N := α* in U_k
     /* then mark all rules that produce needed nonterminal */
  then Item.mark := N fi

some_item_is_working_for_us
  if ∃Item with dotted_rule M := α*Nβ in U_k
     and ∃Item' with dotted_rule N := γ in U_k
     and Item'.mark = R
  then mark item with R fi
```

The following example shows where this algorithm fails in removing some items. The algorithm works incorrect in the someone_needs_us action, where all the dotted rules are marked that produce a nonterminal that is shifted by a useful item, which is an overkill.

Counter example:
Grammar:

```
S ::= p L
T ::= L a
L ::= q | p q
```
Input sentence: \( p \rightarrow a \)

The resulting situation after processing input (for ease, we do a normal parse, starting only with productions with the \( \bullet \) on the left) and marking all items is (a):

![Diagram of parsing process]

The \( L ::= q \) item in 2 is useless because it builds no subtree for the \( T ::= L a \) production. So it should not be marked. The problem is that the \( L ::= q \) is marked because \( T ::= L \bullet a \) can reach set 3. But actually, not the \( L ::= q \) but the \( L ::= p \rightarrow q \) production caused the \( T ::= L \bullet a \) to appear.

This problem can be solved by making a link from the \( T ::= L \bullet a \) item to the \( L ::= pq \) item, indicating that this item caused the \( T ::= L \bullet a \) to be added. The problem with this approach is that it is not symmetric. The problem can also be solved by making a link from the \( L ::= pq \) item to the \( L \) shift possibility from 0 to 2, the shift possibility it caused to be added. This looks more symmetrical, since both \( L ::= \bullet pq \) and \( L ::= pq \) can get such a link. But in fact, this looks like another way to get the first solution we gave.

A better way to make removing of useless items incremental is to make the first solution incremental. Therefore, it seems necessary for all shift possibilities and items to remember why they are useful. For the shift possibilities, this implies that they have to remember all shift possibilities that made them useful. For left- and right-complete items, this implies that they have to remember the shift possibilities that used them for constructing their shift.

The algorithm we think of maintains both a left and a right context during parsing. More context symbols are inserted by making a shift possibility for that symbol between the outermost normal set and a new set, and a \( T \) shift possibility from that new set to the old context (a).
Next, the old \( T \) shift possibility from the old outermost normal set to the old context is 'retracted' in an incremental way, with remove_shift (b).

```
remove_shift (ShiftPoss)
remove the ShiftPoss
for each left- and right-complete Item that ShiftPoss could use for its shift do
  forget that Item was useful because of ShiftPoss
  if Item knows no more useful shift possibilities
     then remove(Item) fi
od
for each ShiftPoss' that was useful because of ShiftPoss do
  forget that ShiftPoss' was useful because of ShiftPoss
  if ShiftPoss' has no other reasons to be useful
     then remove_shift(ShiftPoss') fi
od
for each link that uses ShiftPoss do remove link od
```

We will not work out this algorithm in more detail. The reason is that (at least?) two things are remembered in duplicate:
1. Shift possibilities remember their children, and their children remember the shift possibility that made them possible.
2. Links remember the shift possibility that makes them possible, and shift possibilities what links they made.

This duplication seems waste of space and computation time. But it is not clear how this can be done better. This duplication of data seems closely related to the aim of incrementality.
6 An implementation of an Island Parser

Implementation in pseudo-Pascal

In this chapter, we gather all the results of chapter 4 and 5 to make an island parser. The methods we provide are new and extend. With this, we can make an incremental parser in the same way as the DecideResult' test of chapter 3, with the difference that the island parser only needs the new symbols and not the old as well. In Chapter 9, a Lisp implementation is given. The objects we use look as follows:

object 'ParserObject': /* A parse situation consists of */
  itemsets /* a list of our itemsets. Ordering is important */
  outermost /* the direction-most itemset—the only changing set */
  shift_possibilities /* a set of shift possibilities */
  G /* the grammar */
  direction /* the direction we're working in */

object 'itemset': /* a set of items */
  items /* is it a 'normal' or a 'context' set? */
  type

object 'ends': /* many things consist of two sides... */
  left /* a left */
  right /* and a right side */

object 'item': /* an ends-object: */
  dotted_rule /* in 'left' the links to us, in 'right' links from us */
  links
  itemset /* the itemset where this item is in */

object 'link': /* an ends-object: link from left to right */
  arrow /* who made this link possible */

object 'shift_possibility': /* the recognised symbol */
  symbol /* an ends-object containing the sets between which */
  sets /* this shift_possibility is */
  alternatives /* different ways to make this shift possibility */

object 'alternative': /* an alternative to make a shift possibility */
  production /* the production that was used */
  shifts /* the shift possibilities that were needed to complete */
  /* that production */
'no_alternative' represents an object without production and shifts. It is used for the leaves, which are not produced. Self indicates the ParserObject we are talking about. Field names of the ParserObject refer to the 'Self' object that is passed in all functions. For example, 'itemsets' is short for Self.itemsets; 'direction' is short for Self.direction etcetera. Variables (excepted some index variables) and parameters start with an upper case character. We use = to indicate matching symbols. e.g. 'a'=a='b=?' and 'S'=?
The methods now look as follows:

```plaintext
new (Grammar)  /* returns a new, empty ParserObject */
  L := make a context set
  U0 := make a normal set
  Self := make an empty ParserObject, with
           G=Grammar, direction='right', itemsets=list of L and U0,
           outermost=U0
  for each possible DottedRule of G
    do add(Self,DottedRule) od
  add_shift_possibility(Self,T,L,'no_alternative')
  return Self

extend (Self,X1...Xn,Direction,Symbol)
  /* Symbol is the topnonterminal we want in parse tree*/
  /* ? if any top symbol is OK */
  set_direction(Self,Direction)  /* context at correct side */
  for 1≤k≤n with k counting in Direction do
    U := outermost
    outermost := a new Direction-most 'normal' itemset
               without items
    add_shift_possibility(Self,Xk,U,'no_alternative')
  od
  return decide_result(Self,Symbol)
```

We used the following help functions. The actions are implemented as follows: each action tests whether its action triggers other actions. If this is the case, these actions are called.

```plaintext
set_direction (Self,Direction)
  /* first check if we are in the right situation */
  if Direction-most itemset is a normal set then return fi
  /* no. Make 2 contexts and remove the old one */
  make_context(Self)  /* if it already is there nothing happens */
  remove the Direction-most context set
  direction := Direction
  outermost := Direction-most itemset

predictor (Self,N)  /* Add N-producing rules to outermost */
  for each Rule in G do
    DottedRule := Rule with the * at direction-1
    add(Self,DottedRule)
  fi
```
add (Self,N::=α•β)  /* add item to outermost */
Item := find an item with dotted rule N::=α•β in outermost
if Item not found then
  Item := an item-object with dotted_rule=N::=α•β,
       links=an new empty ends-object, itemset=outermost
  add Item to outermost
if α•β=ε then
  Alt := an alternative-object with
       production='N=ε', shifts=empty list
  add_shift possibilità(Self,N,outermost,Alt)
else
  S := symbol directly at direction side of • in N=α•β
if S exists then
  /* Check for predictor and scanner actions */
  UsableShifts := set of all shift possibilities
                  in shift possibilities with
                  sets.direction−1=outermost and symbol=S
  for each ShiftPoss in UsableShifts
    do shift(Self,Item,ShiftPoss) od
  if S is nonterminal then predictor(Self,S) fi
fi
return Item

add_shift possibilité (Self,S,FromSet,Alternative)
  /* make a shift poss. for S between FromSet and outermost */
Ends := a new ends-object
Ends.direction−1 := FromSet
Ends.direction := outermost
find in shift possibilities a ShiftPoss with
sets=Ends and symbol=S
if ShiftPoss is found then
  add Alternative to ShiftPoss.alternatives
  /* it existed already so no check for items using it */
else
  /* create new shift poss and check who uses it */
  ShiftPoss := a new shift possibility object with symbol=S,
              sets=Ends, alternatives={ Alternative }
  add ShiftPoss to shift possibilities
  /* and check whether this causes new links */
  Items := all items in FromSet having a dotted rule
         with a symbol=S at direction of •
  for each Item in Items do shift(Self,Item,ShiftPoss) od
fi

shift is the most difficult function in our situation. It makes a new link to an item. The item can already exist, but the link can not, since shift can be called in two ways:
1. From add_shift possibility, which only calls shift when the shift possibility for the link did not exist, so the link can not exist.
2. From add, which only calls shift when the item did not exist, so a link between that item and other items can not exist.
shift (Self, Item, ShiftPos)  /* Item can use ShiftPos */
  /* ShiftPos.direction should be outermost !! */
  /* shift makes a new link to an item. */
ShiftedRule := Item.dotted_rule,
  with * moved one symbol to the direction
NewItem := add(Self, ShiftedRule)
Arrow := a new ends-object
Arrow.direction :=NewItem
Arrow.direction^{-1} := Item
  /* Now create the link */
L := a new link-object, with arrow=Arrow,
  shift_possibility=ShiftPos
  /* link to right for left item, and to left for right item */
add L to Arrow.left.links.right
add L to Arrow.right.links.left
ToComplete := a new ends-object
ToComplete.direction :=
  shifts_to_complete(Arrow.direction, direction)
  /* Check if item completed in the working direction */
if ToComplete.direction=nil then return fi
  /* complete! now check for the other direction */
ToComplete.direction^{-1} :=
  shifts_to_complete(Arrow.direction^{-1}, direction^{-1})
Production := remove the dot from Item.dotted_rule
Symbol := nonterminal that is produced by Production
for each combination of a LeftToComplete in ToComplete.left
  and a RightToComplete in ToComplete.right
  /* For each such combination, a shift poss can be made! */
do
  if direction='right'
    then From := LeftToComplete.endset
  else From := RightToComplete.endset fi
  Alternative := a new alternative-object, with
    production=Production,
    shifts=append LeftToComplete.shiftlist,
      ShiftPos and RightToComplete.shiftlist
add_shift_possibility(Self, Symbol, From, Alternative)
do

When a new link is made, we need to check if it makes items complete. Therefore, we need to follow all the arrows in two Directions and give all the ways to reach a complete item. This function looks in the given way, starting at a given item. It returns a set containing all possible shift-to-complete objects:

object 'shift-to-complete':  /* one way to reach a dir-complete item */
  shiftlist
  /* shifts to be done up to a complete item */
endset
  /* in which set was the complete item */

shifts to complete (Item, Direction)
if Item.dotted_rule has the * at Direction
then
  ShiftToComplete := a shift-to-complete-object, with
    shiftlist=empty list, endset=Item.itemset
  return { ShiftToComplete }
else /* Item is not complete, look back */
ShiftsToComplete := Ø
for each Link in ItemSet links Direction do
  ShiftsToComplete' :=
    shifts_to_complete(Link.arrow.Direction,
                      Direction)
  LinkShift := Link.shift_possibility
  for each ToComplete' in ShiftsToComplete' do
    append LinkShift at Direction^{-1}-side of
    ToComplete'.shiftlist
  od
  add ShiftsToComplete' to ShiftsToComplete
od
return ShiftsToComplete
fi
decide_result (Self,Symbol)
U₀ := leftmost normal set
Uₙ := rightmost normal set
if there is a ShiftPoss in shift_possibilities
  for a symbol=Symbol between U₀ and Uₙ
  then return select_a_tree(ShiftPoss) fi
/* no tree. Find a shift possibility using some context */
make_context(Self)
L := left context set
R := right context set
if there is a shift possibility between L and (R or Uₙ)
or there is a shift possibility between U₀ and R
  then return 'more-context'
else return 'failure' fi
Since we need two contexts for the remove_useless algorithm, and because there are
two contexts when make_context is called, it is attractive to have make_context to
remove useless items directly. Without it, the algorithm also works correctly.

make_context (Self) /* add a context at end of working dir */
U₀ := outermost
if Uₙ.type='context' then return fi
outermost := a new direction-most context set
add_shift_possibility(Self,L,Uₙ,'no_alternative')
/* The following call is optional */
remove_useless(Self)

select_a_tree (ShiftPoss)
/* returns one tree derived from the cyclic parse forest
/* that we created. This job is not trivial but outside the
/* scope of this algorithm */
If remove_useless is used, the objects 'item' and 'shift_possibility' need an additional
'mark' field.

remove_useless(Self) /* both contexts required */
make_context(Self)
for each normal ItemSet do
  for each Item in ItemSet do Item.mark:=false od
for each ShiftPoss in shift possibilities
do ShiftPoss.mark:=false od
L := leftmost context set
U_0 := leftmost normal set
U_n := rightmost normal set
R := rightmost context set
for each ShiftPoss in shift possibilities
  with ShiftPoss.sets.left=L or U_0
   and ShiftPoss.sets.right=R or U_n
do useful(ShiftPoss) od
shift_possibilities := {SP∈shift_possibilities | SP.mark=true}
for each normal itemset U_k do
  for each Item in U_k with Item.mark=false
    and (Item.links.left=∅ or Item.links.right=∅)
do remove(Item) od
od
useful(ShiftPoss)
if ShiftPoss.mark=true then return fi /* already done */
ShiftPoss.mark := true
/* now mark the other shift possibilities that this ShiftPoss
/* needs to be created (all alternatives) and
/* the items that were used */
for each Alternative≠'no_alternative' in ShiftPoss.alternatives
do
  'N:=α' := Alternative.production
  for each ShiftPoss in Alternative.shifts
do useful(ShiftPoss) od
  mark item with dottedrule N:=•α in ShiftPoss.sets.left
  mark item with dottedrule N:=α• in ShiftPoss.sets.right
od
remove(Item)
for Side in {'left','right'} do
  for each Link in Item.links.Side do
    LinkedItem := Link.arrow.Side
    remove Link from LinkedItem.links.Side⁻¹
    if LinkedItem.links.Side⁻¹=∅
      then remove(LinkedItem) fi
  od
remove Item from Item.itemset

A notion of correctness

We have no proof of correctness for our algorithms. To give an idea of correctness, we can show two things:
1. If the input string is in L(G), a shift possibility representing this can be made by means of (some of) the actions.
2. This implementation does all possible actions.
Point 2 seems easy to proof: each action checks for the need to trigger other actions due to his own action. Proving point 1 seems more complicated.
7 Optimizations

Goal of this chapter

In this chapter, we have a brief look at alternative solutions for several problems, and look whether these ideas can improve our algorithms. This chapter considers the following points:

- alternative ways to make an island parser
- relations with Tomita's parser
- optimizations for our island parser

Alternative ways to make island parsers

The first idea is to adapt the substring parser ([RK90]) for island parsing. This is an adapted Tomita parser, that invents symbols when a reduce beyond the stack boundaries is attempted. The substring parser can parse only from left to right. If we want to extend a parse to the right, we can simply go on parsing from the situation just before the end-of-input symbol was scanned. If we want to extend a parse to the left, we start a new substring parser at the left of the new context, and parse until we reach the part that we already parsed. Then, we need to attach the parse stacks to each other.

The substring parser invents the missing symbols. But in our case, we may not invent symbols, since the actual context may be given in a next call to extend. Instead, we block the parsers that try to reduce beyond their stack boundary, until their stack is large enough to do a complete reduction. This blocking causes troubles because the blocked parsers cannot go on scanning the input. So when they get unblocked their scanning position is wrong. Furthermore, we need to keep any peace of produced stack in stead of only the stacks under the running parsers, to prevent double work. This may cause high overhead.

If we unblock a parser, and after a few actions we can attach its stack to an existing part, we need to replay all reduce actions ever done over that stack, now using the new stack part we just connected.

A specification for this idea is not very difficult, but an efficient implementation for it seems hard. Furthermore, this approach lacks the symmetry that we think to be essential to the problem.

A second idea seems more practical: just start a left-to-right parser and use it when more right context is provided, and start a right-to-left parser using the reversed grammar when left context is provided. This approach has been used in [S90], but an algorithm is not worked out there. With this idea, we still have to keep track of the reductions that are done over the boundary between these two parsers.

Although not clear from [S90], the parsers need to know at what position in the input they started parsing after the last communication. This is because a communication between two parsers is only allowed if the trees they already built cover the same part of the input symbols.
With this, we get the following data structure for a parser:

object 'Parser':
- direction /* working direction, left or right */
- startpos /* Start position after last communication */
- otherstartpos /* Start position of other parser after last communication */
- scanningpos /* our current scanning position */
- stack /* our stack */
- situation /* the things we're doing, usually 'parsing' */

We will not describe a complete algorithm, but only sketch some actions to give an idea of what has to be done. We only describe the allowed actions, not the parsing itself, as this is done in the same way as Tomita's parsing algorithm. stack is short for Parser.stack. Initially, in each possible state S of the L→R table a left-to-right parser with S on the stack is started, and similar for the R→L table, like in the substring parser ([RK90]).

shift (Parser, state')
push state' on stack

error (Parser)
remove Parser from set of active parsers

reduce (Parser, A ::= αβ) /* α,β may be ε. */
if there are at least |αβ|+1 entries on stack
then
  pop |αβ| entries from stack
  push GOTO(top of stack, A) on stack
else
  there are only |β| entries on the stack
  block(Parser,"A ::= α*β")
  /* The block function handles unblocks */
fi

block (Parser, A ::= α*Sβ) /* One state is always on the stack */
/* A ::= α*Sβ means that only |β|+1 entries are on the stack */
/* and the parser tried to reduce according to A ::= αSβ */
Parser.situation := Situation
for each blocked Parser'
with Parser'.direction=(Parser.direction)^-1
and Parser'.situation="A ::= β^-1*Sα^-1"
and Parser'.startpos = Parser.otherstartpos
do
  complete reduction of Parser and Parser', by using the
  α of Parser', S and the β of Parser
  if both parsers have no remaining context
  and A=Start nonterminal of used grammar
  then we found a successful parse fi
  restart(Parser, A, Parser'.scanningpos)
  restart(Parser', A, Parser.scanningpos)
od
restart (Parser, Symbol, OtherParserStartPos)
    if never restarted a parser after recognising Symbol
    with the same scanningpos then
        for each State that can be reached directly by a Symbol
            transition
                do make a new Parser'
                    with direction=Parser.direction,
                    startpos=scanningpos=Parser.scanningpos,
                    otherstartpos=OtherParserStartPos,
                    stack contains only State,
                    situation='parsing'
                od
    fi

Some problems with this algorithm are:

- It is difficult to see that it is correct.
- The synchronization is lost, because some parsers can be delayed while waiting for
  the other side to give some missing parts. Because of this we need to keep parts of
  the stack that could be removed in normal Tomita parsing, to prevent duplication
  of work. This makes the algorithm waste some storage space.

Relation between Tomita's and our parser

Tomita made a parser that uses parse tables in stead of bare grammar ([T85]). It may be
attractive to use parse tables, since it is a way to do some preprocessing over the
grammar. What exactly is preprocessed by building a parse table? It mainly combines
uncertainties related with future input symbols. Say, we want to recognise an S, but
both S=\*ab and S=\*ac can be used when we see the first a. In our case, we shift both
items. Tomita's table catches this case in one, because there is a state in his table
representing that both S=a\*b and S=a\*c are applicable.

Can we adapt this idea for our own parser? It seems hard to do so. The problems we
encountered are caused by the need to parse in two directions, while Tomita's idea
basically works with extension to one side. We have no good ideas to solve this
problem.

Optimization ideas for our island parser

There are a number of possible optimizations and attempts for optimization for our
parser. We will give a short overview, and then work out the last two in more detail.

1. Because a link is a reflection of a shift possibility, the link can reuse the sets object of
   the shift possibility.

2. We expect that many optimizations as described in [GHR80] can be applied to our
   algorithm.

3. In the algorithm for removing useless items, it is possible to use a new mark in
   stead of deleting all old marks.
4. In our shift algorithm, a check is done whether the item is complete, by first looking in the working direction. It is inefficient to use the all-paths scanning function 'shifts_to_complete' in this case, since there will be at most one link from an item in the outermost set in the working direction. Furthermore, if something is returned, the 'endset' of the returned shift-to-complete object is not interesting, since it always will be outermost. An algorithm that returns 'direction-complete' or 'not direction-complete' would be sufficient.

5. A context is basically the same in all cases. Making the context implicit might improve the performance.

6. Because of the way we store parse trees by storing alternatives in the shift possibilities, the space complexity may be exponential in the number of input symbols. This is the case with very ambiguous grammars. We have an idea to solve this problem, by improving the data structure for storing parse trees. We expect that this will also improve the performance of the parser in general, since we do not need to build the alternative-objects.

An implicit context

The idea is implemented by making a special link indicating that a link to a context is ment. This way, we do not need to make the context itself. If we use such an implicit context, we still need to do the same actions as in the case that a context was present. Therefore, we keep the idea that there are $\mathcal{L}$ and $\mathcal{R}$ contexts, but these are not represented by itemsets. Shift possibilities to a context get a special pointer at the side where they need to point to a context. It is easy to correct the specification, by doing as if the link exists if there is a shift possibility to the context by extending the definition for $\Rightarrow$:

Item with dotted rule $N ::= \alpha S \beta$ in $U_k$ and $\exists$ shift possibility for $S$ between $\mathcal{L}$ and $U_k$

$$\Rightarrow$$

GhostItem with dotted rule $N ::= \star \alpha S \beta \Rightarrow$ Item

and

Item with dotted rule $N ::= \alpha \star S \beta$ in $U_k$ and $\exists$ shift possibility for $S$ between $U_k$ and $\mathcal{R}$

$$\Rightarrow$$

Item $\Rightarrow$ GhostItem with dotted rule $N ::= \alpha S \beta \star$

So we do as if there is a link when there is a corresponding a shift possibility. In the usual case, this is not correct. But in the case that one side is a context set, it is correct. This is so because in the island parser, a link to a context exists if and only if there is a corresponding shift possibility. This can be seen in the following way: if, in the island parser, a new shift possibility is made to a context (in this case, we are at work in the context), all items that can make a link will do so. The other way round, if a shift possibility is made from a context, any item that can shift to the context will be unlinked, because the context contains all possible items.
It is less easy to implement this idea. The only working solution we know is to 'reconstruct' (in an implicit way) the context that is missing, and do the actions as if it were there. We will describe what has to be changed in our parser to use implicit contexts.

new: no context has to be made. The $T$ shift possibility between $L$ and $U_0$ now does not result in a link to a context.

extend: stays the same.

set_direction: in stead of testing for Direction-most set to be a normal set, we need to check for the existence shift possibilities to $L$ or $R$.

predictor: stays the same.

add: stays the same.

add_shift_posibility: if FromSet is a context set and the ShiftPoss did not yet exist, we only need to shift all items from the context to the outermost set. No link has to be made, since the shift possibility represents the links. The algorithm now becomes:

```c
add_shift_possibility (Self,S,FromSet,Alternative)
  /* Make shift poss. for S between FromSet and outermost */
  Ends := a new ends-object
  Ends.direction^-1 := FromSet
  Ends.direction := outermost
  find in shift_possibilities a ShiftPoss with
    sets=Ends and symbol=S
  if ShiftPoss is found then
    add Alternative to ShiftPoss.alternatives
    /* it existed already so no check for items using it */
  else
    /* create new shift poss and check who uses it */
    ShiftPoss := a new shift_possibility object with symbol=S,
      sets=Ends, alternatives={ Alternative }
    add ShiftPoss to shift_possibilities
  if outermost = a context then
    /* we made an implicit link. */
    if FromSet = a context then return fi
    /* all items shifting an S now are right-cplt */
    /* check if they are left-complete */
    for each Item N::=α*Sβ in FromSet do
      CompleteToLeft :=
        shifts_to_complete(Item,direction^-1)
      for each ShiftToComplete in CompleteToLeft do
        add_shift_possibility
          (Self,N,ShiftToComplete.endset,
            'no_alternative')
        /* alternative not clear: a part of the */
        /* parse has been invented implicitly */
      od
    od
    elseif FromSet = a context then
      for each DottedRule N::=αS*β with N::=αSβ a
        production of G
        do add(Self,DottedRule) od
    else
      /* normal case: check if new links caused */
      Items := all items in FromSet having a dotted rule
      for each DottedRule N::=αS*β with N::=αSβ a
        production of G
        do add(Self,DottedRule) od
  od
```

with • directly at direction of a symbol=S
for each Item in Items
do shift(Self, Item, ShiftPoss) od
fi

shift: we only call it for shift possibilities between two normal sets. Therefore, it stays the same.

shifts_to_complete: Now, we also need to check for shift possibilities to a context that can replace a normal link. In stead of ShiftsToComplete=∅ we set
if there is a shift possibility between Direction context and Item.itemset
then ShiftsToComplete := a set containing
a shift-to-complete-object with shiftlist=empty list,
endset=Direction context.
else ShiftsToComplete := ∅
fi
decide_result: stays the same.
make_context: stays nearly the same, but no context set has to be made.

What do we gain? In shifts_to_complete, we can stop following links when we find a shift possibility to the context, in stead of going on following arrows. But since the search path along the arrows does not split any more in the context, and the usual grammar rules are not very long, the gain seems marginal.

Something can be gained from add_shiftPossibility, because we can precompute which items can use a certain shift possibility. But if in the old situation the items are sorted on the symbol that they want to shift, the same gain can be archived there.

Concluding, this algorithm will be slightly faster, but the gain is not very high.

Efficient parse tree storage

We expect the following optimization to give better improvements of the performance. As noted before, our way of storing parse trees may need exponential space (our algorithm runs in exponential time in that case). We made another way to store parse trees, such that parse trees always fit in space O(n^3).

The idea is to annotate at each item the sets that contain linked left- and right-complete items. In an example picture, this looks like (a). In this picture, only the sets containing right-complete items are annotated. The left-complete items can be annotated at each item in the same way.
When we make a new link, this data has to be updated. This can be done as follows: all items at the left side of the new link now also can reach the right-complete items of item B directly at the right of the link, so we pass the sets containing a right-complete item (these are annotated in B) to the left. The same holds for the items at the right side of the link, which can also reach the left-complete items annotated at the item directly at the left of the new link.

How can we derive a parse tree from this annotation? Take the picture of (a). If we want a parse tree for $S$ that covers all input symbols, we start looking for an item $S::=\alpha$ in set 0 that promises to reach set 3 (has 3 in its annotated sets). In our case, only $S::=\alpha S$ can be chosen, but if there are more possibilities, we have an ambiguity. So we choose $S::=\alpha S$. Now, we want to find a way to reach set 3. Therefore, we check which items at the other side of outgoing links still promise to reach set 3. In our case, the items $S::=S S$ in 1, 2 and 3 do so. We choose one of these, say the one in 2. This item still is not right-complete, so we again check which links can be followed. This time, only one link brings us to an item reaching set 3 (the $S::=S S\alpha$ in 3), so we have no real choice. We follow that link, and are in a right compete item, so we are ready. But with following the links, we did not yet reconstruct the subtrees, in a picture, we have found a parse tree like (b).

![Diagram of parse tree](image)

(b)

The ?? trees are unknown parts of the tree. But we can be sure that these parts exist, otherwise the $S::=\alpha S$ would not have these links. So the ?? trees can be reconstructed in the same way as the main tree. The left tree is reconstructed by looking for an $S::=\alpha$ in set 0 that promises to reach set 2, the right tree by looking for an $S::=\alpha$ in set 2 that promises to reach set 3.

With this idea, it gets harder to reconstruct a parse tree from the data structure. Producing a non-leaf node in the tree may take $O(n \log n)$ work, in very ambiguous grammars. But this seems much better than exponential space storage of the trees. Maybe the work for each non-leaf node can be lowered by not only annotating which sets can be reached, but also along which links they can be reached.

With this notation, we don't really need shift possibilities, since we can easily see what shift possibilities exist for $S$ by looking at $S::=\alpha$ items.
8 A more efficient implementation

Definitions

This chapter contains a specification and an implementation that realizes a more efficient way to store parse trees. The idea was described in the last section of the previous chapter.

For convenience, we define the $\Rightarrow_{direction}$ relation, similar to the $\Rightarrow$ relation:

- $\text{Item} \Rightarrow_{right} \text{Item}' \Leftrightarrow \text{Item} \Rightarrow \text{Item}'$
- $\text{Item} \Rightarrow_{left} \text{Item}' \Leftrightarrow \text{Item}' \Rightarrow \text{Item}$

In the brief description that we gave in chapter 7 for efficient parse tree storage, we already saw that each item directly maintains the sets containing uplinked left- and right-complete items. For a specification, it seems more convenient to make a kind of relation. Therefore, we define a relation $\Rightarrow_{direction}$:

- $\text{Item} \Rightarrow_{right} \text{Itemset} \Leftrightarrow \exists \text{Item}' \text{ with dotted rule } N=\alpha \bullet \text{ in Itemset: } \text{Item} \Rightarrow \star \text{ Item'}$
- $\text{Item} \Rightarrow_{left} \text{Itemset} \Leftrightarrow \exists \text{Item}' \text{ with dotted rule } N=\bullet \alpha \text{ in Itemset: } \text{Item'} \Rightarrow \star \text{ Item}$

Note that the $\Rightarrow_{left}$ relation is equivalent to the links Earley makes. The $\Rightarrow_{right}$ is the equivalent relation for right to left parsing. Making shift possibilities seems waste of time, since we know that there is a shift possibility for $N$ between $A$ and $B$ if and only if there is an Item with dotted rule $N=\bullet \alpha$ in $A$ with Item $\Rightarrow B$. Therefore, we remove the shift possibilities from the scenery.

With the $\Rightarrow$ relation, it is easy to reconstruct a parse tree from the parser situation itself, as shown in the example of chapter 7.

We don't make shift possibilities, and no alternative-objects, which was what we aimed for. Because of this, we also do not need to remember for each link how it was made. But then, a link would only consist of a left and a right side. In this case, we can also make the link directly, in stead of making a special link object.

There are at most $|G|$ items, and at most $n \Rightarrow$ links from each item, so there are at most $n|G|$ links to be checked when we look for a shift possibility. Since there are $O(n^2 |G|)$ shift possibilities in the old situation, this is an improvement.

A problem is caused by removing the shift possibilities: we can not notate a terminal shift possibility explicit. Therefore, we need to handle the shift of terminals implicit.

A Specification

With these relations, we can specify the new parser actions as follows. As in the previous parser, new items can only be added to Outermost.
new (Grammar)
  start in an empty situation
  Direction := 'right'
  G := Grammar  /* for later reference to G */
  L := make a context set
  Outermost := a new itemset at right side of L
  add all possible dotted rules of G to Outermost
  add_shift(L,T)

extend (X_1..X_n,Direction,TopNonterminal)
  set_directory(Direction)
  for 1 ≤ k ≤ n with k counting in Direction do
    U := Outermost
    Outermost := a new Direction-most 'normal' itemset
    add_shift(U,X_k)
    while an action is possible do action od
  od
  return decide_result(TopNonterminal)

The actions are as follows. They are mainly concerned in making new links/relations between items.

shifter /* U is an arbitrary set */
  if ∃ Item in set U with wants_to_shift(Item.dotted_rule)=S
    and ∃ UsefulItem in U
      with dotted rule 'S::=α' with • at Direction⁻¹
      and UsefulItem ⇒ Direction Outermost
    then shift(Item) fi

completer /* now only passes the ⇒ relation */
  if ∃ dir, Item, Item', Itemset
    with Item ⇒ dir Item' and Item' ⇒ dir ItemSet
  then Item ⇒ dir ItemSet fi

predictor
  if ∃ Item in Outermost with wants_to_shift(Item.dotted_rule)=N
    and N::=γ is production of G
  then add('N::=γ' with • at Direction⁻¹) fi

wants_to_shift(DottedRule) /* help function */
  return the symbol at the Direction side of •
  in DottedRule, or $ if there is no such symbol

shift(Item) /* shift an item to outermost */
  ShiftedItem :=
    add(Item.dotted_rule with • moved 1 symbol to Direction)
  Item ⇒ Direction ShiftedItem
add_shift(FromSet, S)
  for each Item in FromSet
    with wants_to_shift(Item.dotted_rule) == S
    do shift(Item) od
  while an action is possible do action od

add (N := α•β)
  Item := find an item with dotted rule N := α•β in Outermost
  if Item not found
  then add Item with dotted rule N := α•β to Outermost
    if α = ε then Item ⇒ left Outermost fi
    if β = ε then Item ⇒ right Outermost fi
  fi
  return Item

set_direction(Dir) /* set new Dir */
  if not Dir-most itemset is a normal set then
    /* watch out! the variable Dir≠Direction! */
    add a Direction-most context set if not yet present
    remove_context(Dir)
    Outermost := Direction-most itemset
    Direction := Dir
  fi

remove_context(Side) /* remove Side context */
  for each Item in Side context
    do retract all Item ⇒ Side⁻¹ Item' relations od
  /* This implies retraction of ⇒ relations! */
  remove Side context

decide_result(S)
  U₀ := leftmost normal set
  Uₙ := rightmost normal set
  if there is an Item in U₀ with dotted rule S := •α
    and Item ⇒ right Uₙ
  then return select_a_tree(Item)
  else /* no accept. Find tree that uses some context */
    make a direction-most context set
    L := left context set
    R := right context set
    if there is an Item in L
      with (L ⇒ right Uₙ or L ⇒ right R)
      or there is an Item in U₀ with Item ⇒ right R
    then return 'more-context'
    else return 'failure' fi
  fi

An implementation

Preventing double work gets harder because we dropped the shift possibilities. When
add_shift is called, we need to find out if the shift between FromSet and outermost for
S was made before, to prevent shift from adding a link twice, and to prevent
duplication of work. We can not find out about it by checking all the items in FromSet of the form S::=α•β (in the left-to-right case) whether they have a ⇒ link to Outermost. The reason is that the creation of such a link causes a call to new_shift. We also cannot find out by looking for items of the form N::=α•β in FromSet, whether they have a ⇒ link to some item in outermost. If this is the case, new_shift was called before, but it is possible that shift already has been called, but no such link has been made yet. This can be the case if add_shift has to add another item before it can make the link, and the add of that other item causes another call to shift. To solve the problem, we temporary store the shifts that are done to the context set. When a new outermost set is created, we can forget the old shifts that are done, because only actions on the outermost set are allowed. The shifts made since the last creation of an outermost set are stored in the 'created_shifts' field of ParserObject. In Chapter 9, a Lisp implementation is given.

object 'ParserObject':
  itemsets                            /* a list of our itemsets */
  direction                          /* the direction we're currently working in */
  outermost                          /* the direction-most itemset=the only changing set */
  G                                  /* the grammar */
  created_shifts                    /* list of tuples of the form <FromSet,ShiftedSymbol> */

object 'itemset':
  items                              /* a set of items */
  type                               /* 'normal' or 'context' */

object 'ends':
  left
  right

object 'item'
  dotted_rule
  links                               /* the links we have, an ends-object */
  completes                           /* ends-object: the sets with unlinked x-complete items */
  itemset                             /* the itemset this item is in */

We have the two following methods for parsing:

new (Grammar)
  Self := a new ParserObject, with G=Grammar, direction='right',
          itemsets=empty list
  L := add_context(Self)
  new_outermost_set(Self)
  add_dotted_rules(Self)
  add_shift(Self,L,T)
  return Self

extend (Self,X₁..Xₙ,Direction,Symbol)
  set_direction(Self,Direction)
  for 1≤k≤n with k counting in Direction do
    U := outermost
    new_outermost_set(Self)
add_shift(Self, U, Xk)

od
return decide_result(Self, Symbol)

Other functions we need are as follows. We use the same conventions as in the previous algorithms.

add_shift (Self, FromSet, S)
/* prevent double work */
if there is a <FromSet, Symbol> tuple in created_shifts
with Symbol=S then return fi
add <FromSet, S> to created_shifts
Items := set containing all Items in FromSet.items
with wants_to_shift(Item.dotted_rule, direction)=S
for each Item in Items do shift(Self, Item) od

shift (Self, Item)
/* shift does not check for existing links */
/* So only call when the link does not yet exist */
shiftedItem := add(Item.dotted_rule with * moved 1 symbol
to direction)
add shiftedItem to Item.links.direction
add Item to ShiftedItem.links.direction⁻¹
complete(Self, Item, ShiftedItem.completes.direction, direction⁻¹)
complete(Self, ShiftedItem, Item.completes.direction⁻¹, direction)

Complete passes completes-data to uplinked items. When a direction⁻¹-complete item gets ReallyNew complete sets, add_shift is called to process the new 'shift possibility'.

complete(Self, Item, NewCompleteSets, PassDirection)
/* ReallyNew := completes that are new for this item */
ReallyNew := NewCompleteSets - Item.completes.PassDirection⁻¹
if ReallyNew=∅ then return fi
add ReallyNew to Item.completes.PassDirection⁻¹
DottedRule := Item.dotted_rule
if wants_to_shift(DottedRule, direction⁻¹)=ε then
/* then we made a new shift possibility */
add_shift(Self, Item.itemset,
nonterminal produced by DottedRule)
else
for each LinkedItem in Item.links.PassDirection
do complete(Self, LinkedItem, ReallyNew, PassDirection) od
fi

add (Self, DottedRule)
Item := find an item with dotted_rule=DottedRule
in outermost.items
if Item not found then
Item := a new item-object, with dotted_rule=DottedRule,
links=new ends-object with left=right=∅,
completes=a new ends-object with left=right=∅,
itemset=outermost
add Item to outermost.items
if wants_to_shift(DottedRule, direction⁻¹)=ε then
Item.completes.direction⁻¹ := {outermost} fi
/* Now do possible actions */
S := wants_to_shift(DottedRule,direction)
if S#ε then
    if S is a nonterminal then
        /* terminals are handled implicit */
        /* check if Item can shift its nonterminal */
        if find_shift(outermost,S,
            outermost,direction)#ε
            then shift(Self,Item) fi
        /* do predict actions */
        for each S::=γ in G do
            do add(Self,S::=γ with * at direction⁻¹) od
        fi
    else
        complete(Self,Item,(outermost),direction⁻¹)
    fi
fi
return Item

set_direction (Self,Direction)
    if not Direction-most itemset of itemsets is 'normal' then
        make_context(Self)
        remove_context(Self,Direction)
        outermost := Direction-most itemset of itemsets
        direction := Direction
    fi

remove_context (Self,Side)
    Context := Side-most itemset of itemsets
    for each Item in Context.items do
        for each LinkedItem in Items.links.Side⁻¹ do
            uncomplete(LinkedItem,Context,Side)
            remove Item from LinkedItem.links.Side
        od
    remove Context from itemsets

uncomplete (Item,Context,Side) /* retract context annotations */
    if Context ∈ Item.completes.Side then
        remove Context from Item.completes.Side
        for each LinkedItem in Item.links.Side⁻¹ do
            uncomplete(LinkedItem,Context,Side) od
    fi

new_outermost_set (Self)
    outermost := a new itemset with items=Ø, type='normal'
    Self.created_shifts := Ø
    add outermost at direction side of itemsets
return outermost

add_dotted_rules(Self) /* add all dotted rules to outermost */
    for each possible DottedRule of G do add(Self,DottedRule) od
add_context(Self)
   Context := new_outermost_set(Self)
   Context.type := 'context'
   add_dotted_rules(Self)
   add_shift(Self, Context, '?')
   return Context

make_context (Self) /* make direction-most context if not present */
   if outermost.type = 'context'
      then return
   else Un := outermost
      add_context(Self)
      add_shift(Self, Un, '?')
   fi

decide_result(Self,S)
   U_0 := lefmost normal set in itemsets
   Un := rightmost normal set in itemsets
   Item := find_shift(U_0,S,Un,'right')
   if Item=nil
      then return select a tree(Self,S)
   else /* no accept. Find tree that uses some context */
      make_context(Self)
      L := left context set
      R := right context set
      if find_shift(L,?,Un,'right')#nil
         or find_shift(L,?,R,'right')#nil
         or find_shift(U_0,?,R,'right')#nil
      then return 'more-context'
         else return 'failure' fi
   fi

find_shift (StartSet,S,EndSet,Direction)
   /* find an S-producing item in StartSet that can reach */
   /* Destination in Direction, or nil if no such Item exists */
   if there is an Item in StartSet
      with dotted_rule 'N=S' with * at Direction^{-1} and N=S
      and EndSet ∈ Item.completes.Direction
      then return Item
      else return nil fi
9 An implementation in Lisp

LeLisp introduction

This chapter contains an implementation of the algorithms given in this theses. The algorithms are written in LeLisp [L90]. This Lisp dialect has a number of features that differ from normal Lisp. We will explain these differences first. A semicolon (;) indicates that the rest of the sentence is no Lisp code but a comment, e.g.

; this is an island parser.

(defvar #:sys-package:colon 'ParserObject)

causes all atoms starting with a colon (;) being prefixed by '#:ParserObject'. This way, it is easy to prevent name clashes with other modules.

(defstruct :ends left right)

This is a definition of a data structure, similar to a Pascal record. This example defines an ends-object containing a field named 'left' and a field named 'right'. These fields are typeless. We call an instantiation of such a data structure an object.

LeLisp has the following functions to manipulate objects. An object of type ends can be created with (omakeq :ends). It is possible to initialize the fields immediately. For example, we can put an ends-object in a variable named 'Link', with the left field initialized to 1, and the right field initialized to 2, in the following way:

(setq Link (omakeq :ends left 1 right 2))

A field is read from an object by send-ing the field name to the object, as follows:

(send 'left Link)

A field can be changed into new value by providing that value as third parameter:

(send 'left Link 8)

It is also possible to call a function with the send function. For example

(send 'shift AParseObject AnItem)

results in the following function call (AParseObject is of type #:ParserObject, which is prefixed to the 'shift' function):

(#:ParserObject:shift AParseObject AnItem)

If provided, the user-defined 'prin' functions are used for printing an object.
Some functions that might be LeLisp specific:

(mapcar Function List)

applies Function on each element in List, and returns the resulting list.

(any Function List)

does the following: if there is an Item in List for which Function applied to Item is not nil, that Item is returned. If such an item does not exist, nil is returned.

(progn s₁...sₙ)
evaluates the expressions s₁ .. sₙ in sequence, and returns the value of sₙ.

We have halved the formats of the code, to save paper. Island parser is the first version of the island parser. Improved island parser is the implementation of the version of chapter 8. Help functions and Grammar functions are used by both parsers.

Performance

In this section, we give some performance comparisons. The lisp code was not compiled. We used the Tomita parser of the ASF+SDF system [R92] for comparison. The code for this system is compiled, so the comparison is not very fair. But the difference will be a constant factor, so it gives some indication.

The first grammar is:

\[ E ::= T | E + T \]
\[ T ::= P | T \times P \]
\[ P ::= a \]

The performance with input string \( a(+a)^n a \) (n some natural) is (a):

![Graph showing performance comparison between Island parser, Improved parser, and Tomita parser.](image-url)
The next grammar is:
\[ A ::= x \mid A A \]
The input string is \( x^n \) with \( n \) odd.
The result is (b).

Take the grammar:
\[ A ::= x \mid A A \]
Again, the input string is \( x^n \).
The performance is picture (c). Note the sharp edge in the performance of the Tomita parser. We can not explain this.

Now, we take the grammar
\[ A ::= A A \mid x \mid \epsilon \]
With input string \( x^n \). We get picture (d):
The last grammar we take has a very high ambiguity:

\[ A ::= A A A A A A A A A A | x | \epsilon \]

With input string \( x^n \) we get picture (e):

Although these grammars are not practical, they give an impression of the strengths and weaknesses of our parsers. Our parser performs the worst in comparison with Tomita when simple grammars are used. The difference is about a factor 10. Maybe the compiler can improve the performance this much. The higher the ambiguity, the more efficient is our improved parser.
Island parser

; an implementation of an island parser...
; Wouter Passman, 2007

(devar #keys-packagecolon 'ParserObject)

;*****************************OBJECTS*****************************

(destruct ParserObject
  itemsets
  outermost
  shift_possibilities
  G
  direction
  id_count ; additional, for generating ids
)

(destruct itemset
  print_id ; an id to ease printing
  type
)

(destruct send
  left
  right
)

(destruct item
  dotted_rule
  links
  itemset
  mark
  ; for remove_useless
)

(destruct link
  arrow
  shift_possibility
)

(destruct shift_possibility
  symbol
  alternatives
  mark
  ; for remove_useless
)

(destruct alternative
  productions
  shifts
)

;*****************************PRINTING FUNCTIONS*****************************

(de ParserObject/print (self))
  ; print "Island Parser!"

  (mapcar 'print (send 'itemsets self))

  (print 'ShiftPossibilities)

  (mapcar 'print (send 'shift_possibilities self))

  (print (send 'G self))

  (print 'outermost)

  (print 'itemset (send 'outermost self))

  (print 'mark (send 'itemset self))

  (print 'direction (send 'direction self))

(de itemset/print (self))
  (print (send 'type self) "\nitemset " (send 'print_id self))

  (mapcar 'print (send 'items self))

(de send/print (self)
  (if (eq self nil) "nil" (send 'print_id self)))

(de shift_possibility/print (self)
  (print (yearself (send 'dotted_rule self))

  (if (send 'mark self) "*" "")

  (mapcar (lambda (Link)
           (send 'print_id (send 'itemset (send 'left (send 'arrow Link)))))
           (send 'right (send 'links self)))

  ("\n")
)

(de link/print (self)
  (print 'link from)

  (send 'dotted_rule (send 'left (send 'arrow self)))

  "in"

  (send 'print_id (send 'itemset (send 'left (send 'arrow self))))

  "to"

  (send 'print_id (send 'itemset (send 'right (send 'arrow self))))
)

(de PTTree (ShlfTree) (print_tree ShlfTree G) )
  ; PrintTree randomly selects a tree from the cyclic parse forest
  ; and prints it. If there is an infinite ambiguity, the PrintTree
  ; may not terminate...
  ; it is only an extra function, to attempt to view the parse forest

  (print_tree (ShlfTree Indentation))

  ; help function to process indentation

  (let (Alternatives Alternative)
    (print (makestring (* 2 Indentation) 32))
    (send symbol ShlfTree)

    (setq Alternatives (send 'alternatives ShlfTree))
    ; we are not interested in 7 shift possibilities
    (if (or (send 'symbol ShlfTree) T) ; progran ; select at random an alternative and print its shifts
      (setq Alternative (shl (random 0 (length Alternatives)) Alternatives))
      (if (eq (setq Alternative 'no_alternative) T)
        (mapcar (lambda (ShlfTree) (print_tree ShlfTree (+ 1 Indentation)))
            (send 'shifts Alternative))
        )))

;*****************************METHODS*****************************

(de new (Grammar)
  (let (L US Self)
    (setq Self
      (cons 'ParserObject
        (G Grammar direction 'right outermost US id_count 100 itemsets nil)))

    (setq L (add context Self))
    (setq US (new outermost set Self))
    (add dotted rules Self)

    (add shift possibility Self 'TT L 'no_alternative)

    (Self))
)

(de shextend (Self Test Direction Symbol)
  ; set_direction Self Direction)
  (let (Range US)
    (if (eq Direction 'right)
      (setq Range (range (length Test)))
      (setq Range (range (length Test))))

    (foreach Range)
      (lambda (k)
        (setq US (send 'extend Self))

        (new outermost set Self)

        (add shift possibility Self (shl (1-k) Test) 'no_alternative)

        (Self))

    (Self))
)

;*****************************OTHER FUNCTIONS*****************************

(de any (Direction) (if (eq Direction 'left) 'right Test))

(de mewid (self)
  (send id_count self (+ (send id_count self) 1)))

(de new outermost set (Self)
  ; return new Dis Most set
  (let (Newshat ItemSets)
    (setq Newshat (makenum items item set nil type 'normal
      print/id (concat 'new id wid))

    (setq ItemSets (send 'items Self))
    (send 'items Self))

(de items (self)
(de add (Self DottedRule)
  (let ((item-list (item-set Direction Self))
        (new-set NewItemSets))
    (if (eq item-list nil)
        (send DottedRule (send 'getter Direction Self))
      (de add_dotted_rules (Self)
        ; pull all dotted rules in outermost set
        ; (foreach (send dotted_rules (send 'getter Direction Self))
        (lambda (DottedRule) (add Self DottedRule))
      ))
    ))
  ))
  ))
  (de add-context (Self)
    ; add context adds a new outermost context set
    ; returns that set
    (let ((set (set (new outermost_set Self))
            (add_dotted_rules Self)
            (send 'getter (set context) (self))
            (add-shift-possibility Self 'set 'no-alternative)
            (set)))
      (de direction-most (item-set Direction)
        ; returns the direction-most set
        (if (eq Direction 'right) (last item-set) (car item-set)))
      ))
  (de find-item (item-list DottedRule)
    ; check if item with DottedRule is in item-list
    (any (lambda (item)
          (if (eq (send 'getter direction Most) (DottedRule))
            (item nil)
            nil))
      ItemList)
    ))
  (de append-at (side-list Eli)
    (if (eq side 'right)
        (append Eli side-list)
      (cons Eli side-list)))
  ))

***********************************HELP FUNCTIONS***********************************

(de set-direction (Self Direction)
  (let ((OtherItem (invDir (item-direction Direction)))
        (Set (set (direction-most (send 'getter sets Self) Direction))))
    (if (eq (send 'getter type Set) nil)
        (do nothing, Set is normal)
      (progn
        (make-context Self) ; make context if set not already present
        ; set up the wrong-sized context set
        (setq Set (direction-most (send 'getter sets Self) Direction))
        ; start with removing links to set
        (for each (send item-set (lambda (item)
                                   (foreach (send item-set (lambda (item-
                                     (foreach (send item-set (lambda (item-
                                       (remove-Link Link))
                                         ; remove shifts to set
                                         (send 'getter shift-possibilities Self)
                                         ; filter (send 'getter shift-possibilities Self)
                                         (item-set (shift) (iref (send 'getter direction (send 'getter sets Shift)) Set))
                                         ; remove the context set from the list of sets
                                         (item-set (item-set (send 'getter sets Self)))
                                         (item-set (item-set (send 'getter (send 'getter direction Self) Self)))
                                         (item-set (item-set (send 'getter direction Self) (item-set Outermost Self)))
                                         (item-set (item-set (send 'getter (direction-most (send 'getter sets Self) Self))))
                                         ))))
      )))
    ))
  ))

(de remove-link (Link)
  ; removes a link, that is the pointers that are to this link
  ; easy, since a link has pointers to the items that have a link to the link
  (let ((LeftItemLinks (send links (send left (send 'getter Link)))
         (RightItemLinks (send links (send right (send 'getter Link))))
         (send left LeftItemLinks (send link (send left 'getter Links)))))
    (send right RightItemLinks (send link (send right 'getter Links))
      ; now, no pointers exist to Link, it will be removed by the g.c.
    ))
  ))

(de predictor (Self N)
  (foreach (send 'predict_rules (send 'getter (send 'getter direction Self)) (N) (send DottedRule) (Add Self DottedRule))
    ))

(de add (Self DottedRule)
  (let ((item-list (send 'getter items Outermost Self))
        (Direction (send 'getter direction Self))
        (set (send 'getter shift-possibilities Self))
        (send 'getter shift-possibilities Self) ; make shift-possibilities Self
        (item-set (send 'getter items Outermost Self) (make-shift Self))
        (sh'tp nil)
        (item-set (send 'getter items Outermost Self))
        (item-set (send 'getter context Self))))
    (foreach (lambda (item)
                (if (eq item nil)
                    (do nothing)
                  (progn
                    (setq item (make-shift Self item (send 'getter items Outermost DottedRule))
                    (de add (self DottedRule)
                      (let ((item-list (send 'getter items Outermost Self))
                            (Direction (send 'getter direction Self))
                            (set (send 'getter shift-possibilities Self))
                            (item-set (send 'getter items Outermost Self))
                            (item-set (send 'getter context Self))))
                        )))
                  ))
    ))
  (de get-shift (Self Left Right)
    ; find one shift between Left and Right for 5
    (any (lambda (Shiftpass)
           (if (and (eq (send left (send 'getter Shiftpass)) Left)
                    (eq (send right (send 'getter Shiftpass)) Right)
                    (match (send 'getter symbol Shiftpass) 5))
               Shiftpass nil)
        ))
  ))

(de add-shift-possibility (Self 5 From-set Alternative)
  (let ((Ends (make-shift send))
        (Direction (send 'getter direction Self))
        (Shiftpass items)
        (send (send item (send 'getter From-set)) (send 'getter direction Self))
        (send 'getter shift-possibilities Self)
        (get-shift 5 (send 'getter items Left End))
        (if (eq Shiftpass nil)
            (send 'alternative Shiftpass)
          (cons Alternative (send 'alternatives Shiftpass))
        )
      )))
  )

(de make-shift-possibility (symbol 5 sets alternatives list Alternative)
  (let ((symbol 5)
        (send 'getter shift-possibilities Self)
        (cons Alternative (send 'getter shift-possibilities Self))
        (setq Shiftpass (send 'getter symbol Shiftpass))
        (foreach (send item (lambda (item)
                          (match (send 'getter items From-set) S))))
        (foreach (send item (lambda (item) (shift Shiftpass item)))))
      )))

(de shift (Self items Shiftpass)
  (let ((leftmost-to-complete ShiftedRule NewItem Production Symbol For Alternative)
        (Direction (send 'getter direction Self))
        (invDir (inv (send 'getter direction Self)))
        (setq Shiftpass (send 'getter symbol Shiftpass))
        (foreach (send 'getter items alternative ShiftedRule NewItem Production Symbol For Alternative)
          Direction (send 'getter direction Self)))
      )))
(Arrow (omake send))

(setq ShiftsRule (send 'shift (send 'dotted_rule item) Dir))
(setq NewItem (add ShiftsRule))
(send Dir NewItem)
(send lnDir NewItem ln)

(setq Lcit (omake link arrow Arrow shift possibility ShiftPoss))
(setq Lcit (send 'link (send 'left Arrow) ln))
(setq Rcit (send 'right Arrow (send 'right Link)))
(setq Lcit (send 'link (send 'right Arrow ln))
(send Lcit (send 'left Link ln))
(setq ToComplete (omake send))
(send Dir ToComplete)

(lshifts_to_complete (send Dir Arrow) Dir)

(if (eq (send Dir ToComplete nil) nil)
nil ; then return

(progn
  (send lnDir ToComplete)
  (lshifts_to_complete (send lnDir Arrow) ln))

(setq Production (send 'remove dot (send 'dotted_rule item) ln))
(setq Symbol (send 'nonterminal Productions))

(permute
  (lambda (LeftToComplete RightToComplete)
    (if (eq Direction 'right)
      (setq From (send 'unshift LeftToComplete))
      (setq From (send 'unshift RightToComplete)))
  )

(setq Alternative
  (omake alternative production Production shifts
    (append
      (send 'shiftlist LeftToComplete)
      (lst ShiftPoss)
      (send 'shiftlist RightToComplete)
    ))

(add shift possibility Self Symbol From Alternative)

(send LeftToComplete)
(send RightToComplete)
)
)

(destruct shift-to-complete
  shift_item)


de shifts_to_complete (item Direction)

(let
  (ShiftsToComplete ShiftsToComplete LinkShiftToCompleteA ShiftsToCompleteA
    lnDir (twa Directions))
  (if (eq (send wants_to_shift (send 'dotted_rule item) Direction) nil)
    (progn)
    (setq ShiftsToComplete
      (omake shift-to-complete shift_list nil
       (send (shift item) item)))
  )

(list ShiftsToComplete)

(progn
  (setq ShiftsToComplete nil)
  (foreach (send Direction (send 'links item))
    (lambda (Link)
      (setq ShiftsToCompleteA
        (add shift_list to_complete)
        (send 'shiftlist RightToComplete)
        (append at InvDir (send 'shiftlist to_completeA) LinkShift)
      )
      (setq ShiftsToComplete (append ShiftsToCompleteA ShiftsToComplete))
    )
  )

(ShiftsToComplete)
)

(de fixnormal (itemSets)
  ; returns first set if it is normal, else the second.
  (if (eq (send 'type itemSets) normal)
    (send 'unitem itemSets)
    (add itemSets)
  )
)

(de decide result (Self Symbol)
  (let (UO Un itemSets Shift)
    (setq itemSets (send 'itemSets itemSets))
    (setq UO (fixnormal itemSets))
    (setq Shift (get_shift self symbol Left Un))
    (if we found a Shift, then there is a shift pose we need)
      (if Shift
        ; no select a tree, just return the shift pose
        (progn
          ; check for failure or more-context
          (make_context Self)
          (setq itemSets (send 'itemSets itemSets))
          (setq L (car itemSets))
          (setq R (last itemSets))
          (if
            (get_shift L 'LR)
          (get_shift L 'UR)
          (get_shift L 'UR))
          (more-context failure) ;))))
    (de make_context (Self)
      (let (Un send 'context)
        (if (eq (send 'type Un) 'context)
          (progn
            (add connect Self)
            (add shift possibility Self ?T Un 'no_alternative)
            ; optional : (remove useless Self)
        )))

);**********functions to remove useless items**********

(de remove_useless (Self)
  (let
    (L R Un R reverseditemSets
      itemSets (send 'itemSets itemSets))
    (Shifts (send 'shift possibilities Self))
    (make context Self)
    (foreach itemSets
      (lambda itemSets)
      (if (eq (send 'type itemSets) normal)
        (foreach send 'items itemSets (send 'item (mark item nil))
      )
    )
    (foreach Shifts (lambda (Shift)
      (send 'shift possibilities Self)
      (filter Shifts
        (lambda (ShiftPoss)
          (if
            (let (Sets (send 'sets ShiftPoss))
              (and
                (let (Left (send 'left Sets))
                  (or (eq Left L) (eq Left U))
                )
                (let (Right (send 'right Sets))
                  (or (eq Right R) (eq Right U))
                )
              )
            )
            (useful ShiftPoss)
          )))
      )
    )
    (send 'shift possibilities Self)
    (send 'filter Shifts
      (lambda (ShiftPoss (send 'mark ShiftPoss))
        (foreach itemSets
          (lambda (item)
            (if (eq (send 'type item) normal)
              (foreach send 'items item)
              (lambda (item)
                (if
                  (let ((Links (send 'links item)))
                    (and
                      (eq (send 'mark item) nil)
                      (or
                        (eq (send 'left Links item) nil)
                        (eq (send 'right Links item) nil)
                      )
                    )
                  )
                )
              )
            )))
          )
        )
      )
    )))

(de useful (ShiftPoss)
  (if (send 'mark ShiftPoss)
    nil ; we're happy with returns in list
    (progn
      (send 'mark ShiftPoss) t
      (foreach send 'alternatives ShiftPoss)
      (lambda (Alternative)
        (if
          (eq Alternative 'no_alternative)
          (let (Production (send 'production Alternative))
            (foreach send 'shifts Alternative
              (lambda (ShiftPoss) (useful ShiftPoss))
            )
          )
        )
      )
    )
  )
)

(de send mark
  (find item (send 'items item) (send 'left (send 'sets ShiftPoss))
    (send 'put dot at start Production right)
  )
)

(de send mark
Improved island parser

(defn item (send item (send 'right (send 'sets ShiftPos)))
  (send 'put_first_at_start Production 'left))
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
(not (is_nonterminal Symbol))
    (and (eq (send .type .StartSet) context) (eq StartSet EndSet))
) nil; we print nothing more
(let (Alternatives Alternative)
    (else we choose a useful production rule (item)
        (setq Alternatives
            (filter (items .Startset)
                (lambda (item)
                    (let (DottedRule (send .dotted_rule item))
                        (and
                            (match (send .nonterminal DottedRule) Symbol)
                            (eq (send .wants_to_shift DottedRule left) nil)
                            (member EndSet (send .right (send .complete item))))))))
        (setq Alternative (nth (random 0 (length Alternatives)) Alternatives))
        (GetToEnd Alternative EndSet Depth)
))

(de GetToEnd (EndSet Depth)
    (let (ToShift UsefulLinkedItems ChosenItem
            DottedRule (send .dotted_rule item))
        (setq ToShift (send .wants_to_shift (send .dotted_rule item) right))
        (if we are at the end, stop it
            (if (eq ToShift nil)
                nil
                else follow links until we reach right-complete item in EndSet
                first find useful linked linked items that promise to reach EndSet
                setq UsefulLinkedItems
                filter (send .right (send .link items))
                (lambda (LinkedItems)
                    (member EndSet (send .right (send .complete LinkedItems))))
        )
        (choose one of these links
            (setq ChosenItem
                (nth (random 0 (length UsefulLinkedItems)) UsefulLinkedItems)
            )
            (and print how the link was made
                (PrintTree
                    (send items item) (send .to shift) (send .right (send .complete item))
                    (send .depth)
                    1)
                )
            (and print the rest of the links
                (GetToEnd ChosenItem EndSet Depth)
            )
        )
    )))

;***********************************************************************
(de new (Grammar)
    (let (Self L)
        (setq Self
            (cons #f #f)
            (G Grammar direction right id count 1 items sets nil)
        )
        (setq L (add context Self))
        (new outermost set self)
        (add dotted rules L)
        (add shift Self L 'T')
        Self
    )
)

(de extend (Self Symbols Direction Symbol)
    (when Symbols; check if there is really work to do
        (let (Range L)
            (set direction Self Direction)
            (if (eq Direction right)
                (setq Range (range 1 (length Symbols))
                    (set Range (range (length Symbols)))
                    (foreach Range
                        (lambda (L)
                            (setq L (send outermost Self))
                            (new outermost set Self)
                            (add shift Self L 'nil (1- t) Symbols)
                            (decide results Self Symbol)
                        ))
                )
            )
        )
    )))

;**************************************************************************
(de add_shift (Self Promiset S)
    (if
        (any
            (lambda (DoneBefore)
                (and (eq (send DonesBefore) Promiset) (match (send DoneBefore) S))
            )
            (send created_shifts Self)
        )
        nil; return done before
        (set Items (Direction (send .direction Self))
            (add elt created_shifts Self (cons Promiset S))
            (setq Items
                (filter (items Items Promiset)
                    (lambda (item)
                        (match S
                            (send wants_to_shift (send .dotted_rule item) Direction)
                        ))
                    (foreach Items Items (send .item Shift Item)
                        (add shift Self Items)
                    )
                )
            )
        )
    )
)

(de add_all (Lists Field Item)
    (add Item to the Object ListField)
    (send ListField Object (cons Item (send ListField Item)))
)

(de rem elt (Lists Field Item)
    remove Item from Object ListField
    (send ListField Item (send .remove Item (send ListField Object)))
)

(de shift (Self Item)
    (let (dir (send .dir direction) Self)
        (invariant Shift(ltem)
            (setq invariant // invariants)
            (setq ShiftItem Item)
            (add shift Self (send .shift (send .dotted_rule item) direction))
            (check for debug purposes
                (if (member ShiftItem (send direction (send .shift direction)))
                    (error 'shift 'shift twice Item)
                    )
                    (add elt direction (send .shift direction) ShiftItem)
                    (add elt invariant (send .shift direction) ShiftItem)
                    )
                    )
                    )
                    )
                    )
                    )
                    )

(de complete (Self Item NewCompleteSets PassDirection)
    (let (ReallyNew
            (inv PassDir (inv PassDirection))
            (complete (send .complete Items))
            (direction (send .direction Self))
            (DottedRule (send .dotted_rule item))
        )
        (setq ReallyNew (subtract (send inv PassDir) NewCompleteSets))
        (if (eq ReallyNew null)
            nil; return
            (send inv PassDir Completes
                (append ReallyNew (send inv PassDir Completes))
            )
        )
        (if (eq (send .wants_to_shift DottedRule (send direction Self)) nil)
            (add shift Self (send Items Item) (send .nonterminal DottedRule))
            (foreach (send PassDirection (send .link Items))
                (complete Self LinkedItems ReallyNew PassDirection))
        )
    )
)

(de add (Self DottedRule)
    (let
        (outermost set outermost Self)
        (direction (send direction Self))
        Item Items invariant
        (setq invariant (inv direction))
        (setq item (find item DottedRule outermost))
        (when (eq Item nil)
            (setq Item
                (outline item dotted_rule DottedRule)
                (link (consq epend send left nil right nil)
                    (complete (outline item send left nil right nil)
                        item outermost)
                    )
                (add_all_items outermost Item)
                (if (eq (send .wants_to_shift DottedRule invalid) nil)
                    (send invariant (send .complete Items) (list outermost))
                    (setq S i (send .wants_to_shift DottedRule direction))
                    (if (eq S nil)
                        (when (is_nonterminal S)
                            (if (find shift outermost S outermost direction)
                                (shift Item)
                                )
                            )
                            )
                            )
                    )
                    )
                    )
        )
        (complete Self Item (list outermost) invariant)
    )
)

(de find_item (DottedRule Itemset)
    (send Item (send .item Itemset))
)

(de find_澎 (DottedRule Itemset)}
any
  (lambda (item)
    (if (equal (send 'dotted_rule_item DottedRule) item
      nil
    )
    (send 'item itemset)
  ))

(de set_direction (Self Direction))
  (when
    (eq 'normal
      (send 'type 'direction-mst (send 'itemsets Self Direction))
  )
  (make-context Self)
  (remove-context Self Direction)
  (send 'outermost Self 'direction-mst (send 'itemsets Self Direction))
  (send 'direction Self Direction)
)

(de remove-context (Self Side)
  (let ((Context (direction-mst (send 'itemsets Self) Side)))
    (foreach (send 'items Context)
      (lambda (Item)
        (foreach (send (inv Side) (send 'links Item))
          (lambda (LinkedItem)
            (uncomplete LinkedItem Context Side)
            (rem_all Side (send 'links LinkedItem) Item)
          ))
        )))
  (rem_all 'itemsets Self Context)
)

(de uncomplete (Self Context Side)
  (let ((Context (direction-mst (send 'itemsets Self) Side)))
    (rem_all Side Complete Context)
    (foreach (send (inv Side) (send 'links Item))
      (lambda (LinkedItem)
        (uncomplete LinkedItem Context Side))
    ))
)

(de new_outermost-set (Self)
  (let ((outermost (make-itemset items nil 'normal id (newid Self))
    (send 'outermost Self outermost)
    (send 'created-shifts Self nil) ; no shifts made to it right now
    (send 'itemsets Self)
    (if (eq (send 'direction Self) 'right)
      (append (send 'itemsets Self) outermost)
      (cons outermost (send 'itemsets Self))
    ))
  ))

(de add-dotted-rules (Self)
  (foreach (send 'dotted-rules (send 'G Self))
    (lambda (DottedRule) (add 'DottedRule Self))
  ))

(de add-context (Self)
  (let ((Context (new-outermost-set Self)))
    (send 'type Context 'context)
    (add_dotted_rules Self)
    (add_shift Self Context 'T)
    Context)
)

(de make-context (Self)
  (let ((Uns (send 'outermost Self)))
    (if (eq (send 'type Un) 'context)
      nil
      (add_context Self)
      (add_shift Un 'T)
    ))
)

(de decide_result (Self S)
  (let
    (U0 Un item L R
      (ItemSets (send 'itemsets Self))
    )
    (seq U0 (firstnormal ItemSets))
    (seq Un (firstnormal (reverse ItemSets)))
    (seq item (find-shift U0 'W 'right))
    (if (seq item nil)
      (case S)
      (nil (select_a_tree
        (make-context Self)
        (seq L (case ItemSets)
          (seq R (last-item ItemSets)
            (if (or
              (find-shift L 'L 'right)
              (find-shift L 'R 'right)
            )
            (fail)
            (more-context failure))
          ))
      )))
)
Help functions

; common functions

( defvar "sys-package-colon" 'CommonFunctions)

; remove-duplicates
; removes duplicates from a list.
(de remove-duplicates (List)
  (if (eq list nil) nil
       (cons (car list) (remove-duplicates (delq (car list) (cdr list))))))

; foreach
; for each element x in List do Function(s)
(de foreach (List Function) (mapcar Function List))

; filter
; filter from a list only that elements e for which TestFunction(e)=true
(de filter (List TestFunction) (if (eq list nil) nil
                                     (if (apply TestFunction (car list))
                                         (cons (car list) (filter (cdr list) TestFunction))
                                         (filter (cdr list) TestFunction))))

; range
; same as member, but using eq in stead of equal
; for recursive structural
(de range (Begin End)
  (if (eq Begin End) nil
      (cons Begin (range (+ Begin 1) End))))

; memberp
; same as member, but using eq in stead of equal
; for recursive structural
(de memberp (x list)
  (cond ((atom x) list)
        ((eq x (car list)) t)
        ((memberp x (cdr list)) t)))

; to-one-list
; makes one list from list of lists
(de to-one-list 'List 'List)
  (apply append List))

; permute
; permute each member of first list with each of second.
; apply 1 to thmem
; ex: (permute '(1 2 3) '(4 5)) =
; '((1 2 3) (1 2 4) (2 3 1) (2 3 4) (3 1 2) (3 1 4))

; mapcar
; (lambda (x) lambda (x) (apply f (list x x)))

; lastq
; gives last element of a list.
(de lastq (List) (car (last List)))

; substract
; substract set1 from set2
; this is remove all items in set1 from set2
(de substract (Set1 Set2)
  (if (eq Set1 nil) Set2
      (substract (cdr Set1) (removeeq (car Set1) Set2))))
(filter
  (send rules Grammar)
  (lambda (Rule) (eq (send 'nonterminal Rule) Nonterminal))
)
)

;; put_dot_at_start
;; given a Rule, make a dotted rule with dot at start (opposite of Direction)
;; de put_dot_at_start (Rule Direction)
;; let ((dr (omake dotted_rule nonterminal (send 'nonterminal Rule)))
;; (if (eq Direction 'right')
;; (progn
;; (send 'reverse-of-before-dot dr dr nil)
;; (send 'after-dot dr (send 'production Rule)))
;; (progn
;; (send 'reverse-of-before-dot dr (reverse (send 'production Rule)))
;; (send 'after-dot dr nil))
;; dr)
;;
;; shift
;; shift the dot of the given dotted rule one in Direction
;; de shift (DottedRule Direction)
;; let
;; (dr (omake dotted_rule nonterminal (send 'nonterminal DottedRule)))
;; (after (send 'after-dot DottedRule))
;; (before (send 'reverse-of-before-dot DottedRule))
;; (if (eq Direction 'right')
;; ; shift dot right
;; (progn
;; (send 'reverse-of-before-dot dr (cons (car after) before))
;; (send 'after-dot dr (cdr after)))
;; ; shift dot left
;; (progn
;; (send 'reverse-of-before-dot dr (cdr before))
;; (send 'after-dot dr (cons (car before) after)))
;;
;; dr)
;;
;; wants_to_shift
;; given a dotted rule, look what symbol can be shift in Direction
;; if dot at outermost position for Direction, return nil
;; de wants_to_shift (DottedRule Direction)
;; (if (eq Direction 'left')
;; (car (send 'reverse-of-before-dot DottedRule))
;; (car (send 'after-dot DottedRule))
;;)
;;
;; remove_dot
;; converts dotted rule back into a rule
;; de remove_dot (DottedRule)
;; (omake_rule nonterminal (send 'nonterminal DottedRule)
;; production)
;; (append
;; (reverse (send 'reverse-of-before-dot DottedRule))
;; (send 'after-dot DottedRule))
;;)
;;
;; is_empty
;; checks if a DottedRule is empty (before dot and after dot both empty)
;; de is_empty (DottedRule)
;; (and
;; (eq (send 'reverse-of-before-dot DottedRule) nil)
;; (eq (send 'after-dot DottedRule) nil))
;;)
;;
;; match
;; checks if two symbols match
;; because we use ? and ?? symbols
;; de match (x y)
;; (and (eq x nil) (eq y nil) ; empty must NEVER match
;; or
;; (eq x y)
;; (eq x ?)
;; (eq y ?)
;; (and (eq x ??) (not (is_nonterminal y)))
;; (and (eq y ??) (not (is_nonterminal x)))
;;)}
## Index

- abstract 4
- accept 42
- actions 19, 27, 67
- alternative 45, 53, 62
- ambiguity 25
- ambiguous 5, 22, 62
- bidirectional recognising 19
- block 59, 60
- CheckForTree 10, 11, 12, 17
- CheckForTree' 12, 18
- children 5
- common parent 9, 10
- complete 20
- Completer 19, 20, 28, 44, 67
- concatenate 4
- Contents 2
- context 10, 33, 34, 35, 36, 62
- context-free 11, 16
- correctness 58
- decide_result 42
- derivation step 4
- differences with Earley 30
- dotted rule 18
- Earley 28, 66
- Earley's recogniser 18
- empty production 26
- expensive 13
- exponential space 45, 62, 64
- extend 43, 53
- failure 10, 12, 34, 35, 36, 42
- grammar 4
- Greek 4
- hard 7, 11
- hardware 88
- immediate neighbour 19
- implicit context 62
- incremental 4
- incremental parsing 7, 9
- insert 9
- invariant 47
- invent 10, 59
- island parser 7
- island recogniser 43
- item 18, 19, 20
- item set 18
- L(G) 4
- last edit 9, 10
- last made itemset 43
- leaves 9
- left context 36, 39
- left-complete 20
- left-to-right bias 19, 27, 43
- left-to-right parser 59
- LeLisp 73
- link 19, 20, 21, 31
- look-ahead 28, 30
- lowest Node 8
- mark 47, 49, 61
- membership 11
- more-context 10, 12, 34, 35, 42
- multiset 13
- necessary to invent 10
- new 43, 53
- node 5
- non-partial 12
- nonterminal 4
- object 73
- one context 36
- order of terminals 17
- outermost 43
- parent 5, 8, 10
- parse table 61
- parse tree 5, 45, 64
- parser 5, 66
- partial 12, 17
- performance 74
- Predictor 19, 23, 28, 44, 67
- preprocessing 61
- produces 4
- recognise 25, 27, 31
- recogniser 5, 18, 20
- regular expression 16
- remove_useless 48
- replace 8, 10
- reverse 31
- reversed grammar 59
- right context 36, 39
- right-complete 20
- right-to-left 31, 59
- root 8, 10
- rule 4
- Scanner 19, 28, 30, 44
- shift possibilities 30
shift possibility 20
shifter 67
sort 5
standard combinations 13
start node 9
substring 10, 12
substring parser 12, 18, 59, 60
successful 10
supervisors 88
swap the context 36
symbol 4
symmetric 30
syntax directed editor 5
terminal 4
Tomita 18, 59, 60, 61
turn the direction 33
unparsed 9, 10
useful 18, 47
Useless 45, 47
References


[J89] Mark Johnson, "The Computational Complexity of Tomita's Algorithm", in "International Parsing Workshop '89"

[K89] James R. Kipps, "Analysis of Tomita's algorithm for general context-free parsing", in "International Parsing Workshop '89"


[S90] Hiroaki Saito, "Bi-directional LR Parsing from an Anchor Word for Speech Recognition", in "COLING '90", vol. 3


Colofon

Technical information
Used hardware: Macintosh IIci for this thesis
Sun 3/60 and 3/140 for the LeLisp implementation

Programs used:
Microsoft Word Version 4.00C
MacDraw Version 1.9.5
Le-Lisp Version 15.24

Printer: Agfa P3400 PS laserprinter
Fonts: Palentino for normal text and Le-Lisp code
Symbol, Zapf Dingbats and Zapf Chancery for some special characters
Courier for pseudo pascal code

#Characters: 160208

Other information
Me: Wouter Pasman
Koedijk 15, HUIZEN, Holland

My supervisors:
Paul Klint for general remarks on this thesis
Jan Rekers for the parsers
Wilco Koorn for Chapter 2 and 3

Ideas worked out:
4th of February 1991 ... 7th of June 1991

Thesis written:
7th of June 1991 ... August 1991