16

Error Handling

Until now, we have discussed parsing techniques while largely ignoring what happens when the input contains errors. In practice, however, the input often contains errors, the most common being typing errors and misconceptions, but we could also be dealing with a grammar that only roughly, not precisely, describes the input, for example in pattern matching. So the question arises how to deal with errors. A considerable amount of research has been done on this subject, far too much to discuss in one chapter. We will therefore limit our discussion to some of the more well-known error handling methods, and not pretend to cover the field; see (Web)Section 18.2.7 for references to more in-depth information.

16.1 Detection versus Recovery versus Correction

Usually, the least that is required of a parser is that it detects the occurrence of one or more errors in the input, that is, we require *error detection*. The least informative version of this is that the parser announces: "input contains syntax error(s)". We say that the input contains a *syntax error* when the input is not a sentence of the language described by the grammar. All parsers discussed in the previous chapters (except operator-precedence) are capable of detecting this situation without extensive modification. However, there are few circumstances in which this behavior is acceptable: when we have just typed a long sentence, or a complete computer program, and the parser only tells us that there is a syntax error somewhere, we will not be pleased at all, not only about the syntax error, but also about the quality of the parser or lack thereof.

The question as to where the error occurs is much more difficult to answer; in fact it is almost impossible. Although some parsers have the "correct-prefix property", which means that they detect an error at the first symbol in the input that results in a prefix that cannot start a sentence of the language, we cannot be sure that this indeed is the place in which the error occurs. It could very well be that there is an error somewhere before this symbol but that this is not a syntax error at that point. Thus there is a difference in the perception of an error between the parser and the user. In

the rest of this chapter, when we talk about errors, we mean syntax errors, as detected by the parser.

So what happens when input containing errors is offered to a parser with a good error detection capability? The parser might say: "Look, there is a syntax error at position so-and-so in the input, so I give up". For some applications, especially highly interactive ones, this may be satisfactory. For many, though, it is not: often, one would like to know about all syntax errors in the input, not just about the first one. If the parser is to detect further syntax errors in the input, it must be able to continue parsing (or at least recognizing) after the first error. It is probably not good enough to just throw away the offending symbol and continue. Somehow, the internal state of the parser must be adapted so that the parser can process the rest of the input. This adaptation of the internal state is called *error recovery*.

The purpose of error recovery can be summarized as follows:

- an attempt must be made to detect all syntax errors in the input;
- equally important, an attempt must be made to avoid *spurious* error messages. These are messages about errors that are not real errors in the input, but result from the continuation of the parser after an error with improper adaptation of its internal state.

Usually, a parser with an error recovery method can no longer deliver a parse tree if the input contains errors. This is sometimes a source of considerable trouble. In the presence of errors, the adaptation of the internal state can cause semantic actions associated with grammar rules to be executed in an order that is impossible for syntactically correct input, which sometimes leads to unexpected results. A simple solution to this problem is to ignore semantic actions as soon as a syntax error is detected, but this is not optimal and may not be acceptable. A better option is the use of a particular kind of error recovery method, an *error correction* method.

Error correction methods modify the input as read by the parser so that it becomes syntactically correct, usually by deleting, inserting, or changing symbols. Error correction methods will not always change the input into the input actually intended by the user, and they do not pretend that they can. Therefore, some authors prefer to call these methods *error repair* methods rather than error correction methods. The main advantage of error correction over other types of error recovery is that the parser still can produce a parse tree and that the semantic actions associated with the grammar rules are executed in an order that could also occur for some syntactically correct input. In fact, the actions only see syntactically correct input, sometimes produced by the user and sometimes by the error corrector.

In summary, error detection, error recovery, and error correction require increasing levels of heuristics. Error detection itself requires no heuristics: a parser detects an error, or it does not. Determining the place where the error occurs may require heuristics, however. Error recovery requires heuristics to adapt the internal parser state so that it can continue, and error correction requires heuristics to repair the input.

With error handling comes the obligation to provide good error messages. Unfortunately there is little research on this subject, and most of the pertinent publications

are of a reflective nature, for example Horning [296], Dwyer [307] and Brown [312]. Explicit algorithmic support is rare (Kantorowitz and Laor [316]). The only attempt at automating the production of error messages we know of is Jeffery [328] who supplies the parser generator with a long list of erroneous constructs with desired error messages. The parser can then associate each error message with the state into which the erroneous construct brings the parser.

16.2 Parsing Techniques and Error Detection

Let us first examine how good the parsing techniques discussed in this book are at detecting an error. We will see that some parsing techniques have the correct-prefix property while other parsing techniques only detect that the input contains an error but give no indication where the error occurs.

16.2.1 Error Detection in Non-Directional Parsing Methods

In Section 4.1 we saw that Unger's parsing method tries to find a partition of the input sentence that matches one of the right-hand sides of the start symbol. The only thing that we can be sure of in the case of one or more syntax errors is that we will find no such partition. For example, suppose we have the grammar of Figure 4.1, repeated in Figure 16.1, and input **×+**. Fitting the first right-hand side of **Expr** with

```
Expr_S  -->  Expr + Term | Term
  Term  -->  Term × Factor | Factor
Factor  -->  ( Expr ) | i
```

Fig. 16.1. A grammar describing simple arithmetic expressions

the input will not work, because the input only has two symbols. We will have to try the second right-hand side of **Expr**. Likewise, we will have to try the second right-hand side of **Term**, and then we will find that we cannot find an applicable right-hand side of **Factor**, because the first one requires at least three symbols, and the second one only one. So we know that there are one or more errors, but we do not know how many errors there are, nor where they occur. In a way, Unger's method is too well prepared for dealing with failures, because it expects any partition to fail.

For the CYK parser, the situation is similar. We will find that if the input contains errors, the start symbol will not be a member of the top element of the recognition table.

So, the unmodified non-directional parsing methods behave poorly on errors in the input. A method to improve that behavior by using dynamic programming is shown in Section 16.4.

16.2.2 Error Detection in Finite-State Automata

Finite-state automata are very good at detecting errors. Consider for example the deterministic automaton of Figure 5.12, repeated in Figure 16.2.

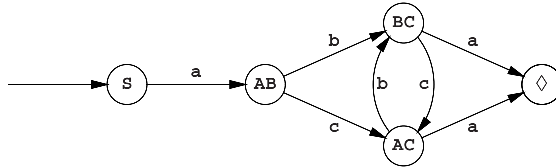


Fig. 16.2. Deterministic automaton for the grammar of Figure 5.6

When this automaton is offered the input **abcca**, it will detect an error when it is in state **AC**, on the second **c** in the input.

Finite-state automata have the correct-prefix property. In fact, they have the *immediate error detection property*, which we discussed in Chapter 8 and which means that an error is detected as soon as the erroneous symbol is first examined.

16.2.3 Error Detection in General Directional Top-Down Parsers

The breadth-first general directional top-down parser also has the correct-prefix property. It stops as soon as there are no predictions left to work with. Predictions are only dropped by failing match steps, and as long as there are predictions, the part of the input parsed so far is a prefix of some sentence of the language.

The depth-first general directional top-down parser does not have this property. It will backtrack until all right-hand sides of the start symbol have failed. However, it can easily be doctored so that it does have the correct-prefix property: the only thing that we must remember is the furthest point in the input that the parser has reached, a kind of high-water mark. The first error is found right after this point.

16.2.4 Error Detection in General Directional Bottom-Up Parsers

The picture is quite different for the general directional bottom-up parsers. They will just find that they cannot reduce the input to the start symbol. This is only to be expected because, in contrast to the top-down parsers, there is no test before an input symbol is shifted.

As soon as a top-down component is added, such as in Earley's parser, the parser regains the correct-prefix property. For example, if we use the Earley parser with the grammar from Figure 7.8 and input **a-+a**, we get the item sets of Figure 16.3 (compare this with Figure 7.11). We see that $itemset_3$ is empty, and the error is detected.

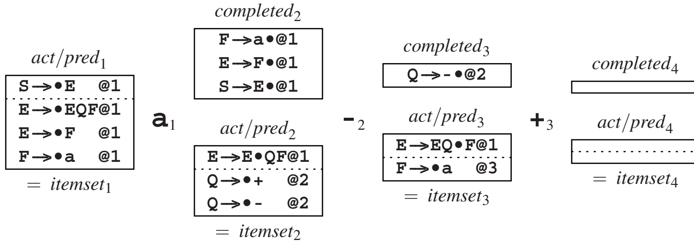


Fig. 16.3. Items set of the Earley parser working on `a-+a`

16.2.5 Error Detection in Deterministic Top-Down Parsers

In Sections 8.2.3 and 8.2.4 we have seen that strong-LL(1) parsers have the correct-prefix property but not the immediate error detection property, because in some circumstances they may make some ε-moves before detecting an error, and that full-LL(1) parsers have the immediate error detection property.

16.2.6 Error Detection in Deterministic Bottom-Up Parsers

Let us first examine the error detection capabilities of precedence parsers. We saw in Section 9.2.2 that operator-precedence parsers fail to detect some errors. When they do detect an error, it is because there is no precedence relation between the symbol on top of the parse stack and the next input symbol. This is called a *character-pair error*.

The other precedence parsers (simple, weak, extended, and bounded-right-context) have three error situations:

- there is no precedence relation between the symbol on top of the parse stack and the next input symbol (a *character-pair error*).
- the precedence relations indicate that a handle segment has been found and that a reduction must be applied, but there is no non-terminal with a right-hand side that matches the handle segment. This is called a *reduction error*.
- after a reduction has been made, there is no precedence relation between the symbol at the top of the stack (the symbol that was underneath the $\lessdot$) and the left-hand side to be pushed. This is called a *stackability error* or *stacking error*.

Reduction errors can be detected at an early stage by continuously checking that the symbols between the last $\lessdot$ and the top of the stack form the prefix of some right-hand side. Graham and Rhodes [295] show that this can be done quite efficiently.

In Section 9.6.3 we saw that an LR(1) parser has the immediate error detection property. LALR(1) and SLR(1) parsers do not have this property, but they do have the correct-prefix property. Error detection in GLR parsers depends on the underlying parsing technique.

16.3 Recovering from Errors

Error handling methods fall in different classes, depending on what level they approach the error. The general parsers usually apply an error handling method that considers the complete input. These methods use global context, and are therefore called *global error handling* methods. The Unger and CYK parsers need such a method, because they have no idea where the error occurred. These methods are very effective, but the penalty for this effectiveness is paid for in efficiency: they are very time consuming, requiring at least cubic time. As the general parsing methods already are time consuming anyway, this is usually deemed acceptable. We will discuss such a method in Section 16.4.

On the other hand, efficient parsers are used because they are efficient. For them, error handling methods are required that are less expensive. We will discuss the best known of these methods. They have the following information at their disposal:

- in the case of a bottom-up parser: the parse stack; in the case of a top-down parser: the prediction stack;
- the input string, and the point where the error was detected.

There are four classes of these methods: the *regional error handling* methods, which use some (regional) context around the point of error detection to determine how to proceed; the *local error handling* methods only use the parser state and the input symbol (local context) to determine what happens next; the *suffix methods*, which use zero context; and the *ad hoc* methods, which do not really form a class. Examples of these methods will be discussed in Sections 16.5, 16.6, 16.7 and 16.8.

In our discussions, we will use the terms *error detection point*, indicating the point where the parser detects the error, and *error symbol*, which indicates the input symbol on which the error is detected.

16.4 Global Error Handling

The most popular global error handling method is the *least-error correction* method. The purpose of this method is to derive a syntactically correct input from the supplied one using as few corrections as possible. Usually, a symbol deletion, a symbol insertion, and a symbol change all count as one correction (one edit operation).

It is important to realize that the number of corrections needed can easily be limited to a maximum: first, we compute the shortest sentence that can be generated from the grammar. Let us say it has length m. If the input has length n, we can change this input into the shortest sentence with a number of edit operations that is the maximum of m and n: change the first symbol of the input into the first symbol of the shortest sentence, etc. If the input is shorter than the shortest sentence, this results in a maximum of n changes, and we have to insert the last $m-n$ symbols of the shortest sentence. If the input is longer than the shortest sentence, we have to delete the last $n-m$ symbols of the input. This is not a very tight and useful maximum, but at least it shows the problem is finite. Also, when searching for a

least-error correction, if we already know that we can do it with, say, k corrections, we do not have to investigate possibilities known to require more.

With this in mind, let us see how such an error correction method works when incorporated in an Unger parser (Section 4.1). We will again use the grammar of Figure 16.1 as an example, again with input sentence `×+`. This is a very short sentence indeed, to limit the amount of work. The shortest sentence that can be generated from the grammar is **`i`**, of length one. The observation above limits the number of corrections needed to a maximum of two.

Now the first rule to be tried is **`Expr`**$\rightarrow$**`Expr+Term`**. This leads to the following partitions:

Expr					max:2	
Expr		+		**Term**		
	?		1	×+	?	
	?	×	1	+	?	
	?	×+	1		?	
×	?		1	+	?	
×	?	+	0		?	
×+	?		1		?	cut-off

When we compare this table to tables like Figure 4.2, we note that it includes the number of corrections needed for each part of a partition in the right of the column; a question mark indicates that the number of corrections is yet unknown. The total number of corrections needed for a certain partition is the sum of the number of corrections needed for each of the parts. The top of the table also contains the maximum number of corrections allowed for the rule. For the parts matching a terminal, we can decide how many corrections are needed, which results in the column below the `+`. Also notice that we have to consider empty parts, although the grammar does not have ε-rules. The empty parts stand for insertions. The cut-off comes from the Unger parser detecting that the same problem is already being examined.

Now that it has this list of partitions, the Unger parser concentrates on the first partition in it, which requires it to derive ε from **`Expr`**. The partition already requires one correction, so the maximum number of corrections allowed is now one. The rule **`Expr`**$\rightarrow$**`Expr+Term`** immediately results in a cut-off:

Expr					max:1	
Expr		+		**Term**		
	?		1		?	cut-off

So we will have to try the other rule for **`Expr`**: **`Expr`**$\rightarrow$**`Term`**. Likewise, **`Term`**$\rightarrow$**`Term×Factor`** will result in a cut-off, so we will have to use **`Term`**$\rightarrow$**`Factor`**. The rule **`Factor`**$\rightarrow$**`(Expr)`** will again result in a cut-off, so **`Factor`**$\rightarrow$**`i`** will be used:

`Expr`	max:1
`Term`	max:1
`Factor`	max:1
`i`	max:1
	1

So we find, not surprisingly, that input part ε can be corrected to `i`, requiring one correction (inserting `i`) to make it derivable from `Expr` (and `Term` and `Factor`).

To complete our work on the first partition of `×+` over the right-hand side `Expr+Term`, we have to examine if, and how, `Term` derives `×+`. We already need two corrections for this partition, so no more corrections are allowed because of the maximum of two. For the rule `Term→Term×Factor` we get the following partitions (in which we cheated a bit: we used some information computed earlier):

`Term` max:0						
`Term`		`×`		`Factor`		
	1		1	`×+`	?	too many corrections
	1	`×`	0	`+`	?	too many corrections
	1	`×+`	1		1	too many corrections
`×`	?		1	`+`	?	too many corrections
`×`	?	`+`	1		1	too many corrections
`×+`	?		1		1	cut-off

Each of these fails, so we try `Term→Factor`. The rule `Factor→(Expr)` then results in the following partitions:

`Term` max:0						
`Factor` max:0						
`(`		`Expr`		`)`		
	1		1	`×+`	2	too many corrections
	1	`×`	?	`+`	1	too many corrections
	1	`×+`	?		1	cut-off
`×`	1		1	`+`	1	too many corrections
`×`	1	`+`	?		1	too many corrections
`×+`	2		1		1	too many corrections

This does not work either. The rule `Factor→i` results in the following:

`Term`	max:0	
`Factor`	max:0	
`i`	max:0	
`×+`	2	too many corrections

So we get either a cut-off or too many corrections (or both). This means that the partition that we started with is the wrong one.

The other partitions are tried in a similar way, resulting in the following partition table, with completed error correction counts:

Expr					max:2	
Expr		+		Term		
	1		1	×+	>0	too many corrections
	1	×	1	+	1	too many corrections
	1	×+	1		1	too many corrections
×	1		1	+	1	too many corrections
×	1	+	0		1	
×+	?		1		1	cut-off

So, provided that we do not find better corrections later on, using the rule `Expr→Expr+Term` we find the corrected sentence `i+i`, by replacing the `×` with an `i`, and inserting an `i` at the end of the input.

Now the Unger parser proceeds by trying the rule `Expr→Term`. Continuing this process, we will find two more possibilities using two corrections: the input can be corrected to `i×i` by inserting an `i` in front of the input and replacing the `+` with another `i`, or the input can be corrected by replacing `×` with an `i` and deleting `+` (or deleting `×` and replacing `+` with an `i`).

This results in three possible corrections for the input, all three requiring two edit operations. Choosing between these corrections is up to the parser writer. If the parser is written to handle ambiguous input anyway, the parser might deliver three parse trees for the three different corrections. If the parser must deliver only one parse tree, it could just pick the first one found. Even in this case, however, the parser has to continue searching until it has exhausted all possibilities or it has found a correct parsing, because it is not until then that the parser knows if the input in fact did contain any errors.

As is probably clear by now, least-error correction does not come cheap, and it is therefore usually only applied in general parsers, because these do not come cheap anyway.

Lyon [294] has added least-error correction to the CYK parser and the Earley parser, although his CYK parser only handles replacement errors. In his version of the CYK parser, the non-terminals in the recognition table have an error count associated with it. In the bottom row, which is the one for the non-terminals deriving a single terminal symbol, all entries contain all non-terminals that derive a single terminal symbol. If the non-terminal derives the corresponding terminal symbol it has error count 0, otherwise it has error count 1 (a replacement). Now, when we find that a non-terminal A with rule $A \rightarrow BC$ is applicable, it is entered in the recognition table with an error count equal to the sum of that of B and C, but only if it is not already a member of the same recognition table entry, but with a lower error count.

Aho and Peterson [292] also added least-error correction to the Earley parser by extending the grammar with error productions, so that it produces any string of terminal symbols, with an error count. As in Lyon's method, the Earley items are extended with an error count indicating how many corrections were needed to create the item. An item is only added to an item set if it does not contain one like it which has a lower error count.

Tanaka and Fu [301] extended this method to context-sensitive parsers, in one of the few examples of error correction in systems stronger than context-free.

A completely different form of global error recovery is based on parsing by intersection and is treated in Section 13.5. It can give surprising results but there is hardly any research on it available yet.

16.5 Regional Error Handling

Regional error handling collects some context around the error detection point, consisting of a segment of the top of the stack and some prefix of the input, and reduces that part (including the error) to a left-hand side. Since it tries to collect a "phrase", which is a technical term for a terminal production of a non-terminal, this class of error handling methods is also often called *phrase level error handling*. Since the technique requires a reduction stack to participate in the desired reduction, it is applied exclusively to bottom-up parsers.

16.5.1 Backward/Forward Move Error Recovery

A well-known example of regional error handling in bottom-up parsers is the *backward/forward move* error recovery method, presented by Graham and Rhodes [295]. It consists of two stages: the first stage condenses the context around the error as much as possible. This is called the *condensation phase*. Then the second stage, the *correction phase*, changes the parsing stack and/or the input so that parsing can continue. The method is best applicable to simple precedence parsers, and we will use such a parser as an example.

Our example comes from the grammar and precedence table of Figure 9.9. Suppose that we have input **#n×+n#**. The simple precedence parser has the following parse stacks at the end of each step, up to the error detection point:

# ⋖	n × + n #	shift n
# ⋖ n ⋗	× + n #	reduce n
# ⋖ F ⋗	× + n #	reduce F
# ⋖ T ≐	× + n #	shift ×
# ⋖ T ≐ ×	+ n #	stuck

No precedence relation is found to exist between the × and the +, resulting in an error message that + is not expected.

Let us now examine the condensation phase in some detail. As said before, the purpose of this phase is to condense the context around the error as much as possible. The left-context is condensed by a so-called *backward move*: assuming a ⋗ relation between the top of the parse stack and the symbol on which the error is detected (that is, assuming that the parse stack built so far has the end of a handle as its top element), perform all possible reductions. In our example, no reductions are possible. Now assume a ≐ or a ⋖ between the top of the stack and the next symbol. This enables us to continue parsing a bit. This step is the so-called *forward move*: first we shift the next symbol, resulting in the following parse stack:

$\# \lessdot \mathtt{T} \doteq \times \doteq/\lessdot +$	$\mathtt{n}\ \#$	still stuck

Next, we disable the check that the top of the stack should represent a prefix of a right-hand side. Then, we continue parsing until either another error occurs or a reduction is called for that spans the error detection point. This gives us some right-context to work with, which can be condensed by a second backward move, if needed. For our example, this results in the following steps:

$\# \lessdot \mathtt{T} \doteq \times \doteq/\lessdot + \lessdot$	$\mathtt{n}\ \#$	shift **n**
$\# \lessdot \mathtt{T} \doteq \times \doteq/\lessdot + \lessdot \mathtt{n} \gtrdot$	$\#$	reduce **n**
$\# \lessdot \mathtt{T} \doteq \times \doteq/\lessdot + \lessdot \mathtt{F} \gtrdot$	$\#$	reduce **F**
$\# \lessdot \mathtt{T} \doteq \times \doteq/\lessdot + \lessdot \mathtt{T} \gtrdot$	$\#$	reduce **T**
$\# \lessdot \mathtt{T} \doteq \times \doteq/\lessdot + \doteq \mathtt{T'} \gtrdot$	$\#$	proposed reduction includes error point

So now we have the situation depicted in Figure 16.4. This is where the correction

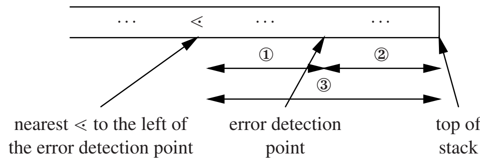
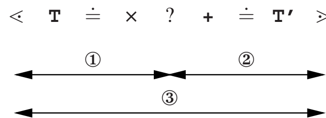


Fig. 16.4. Situation after the backward/forward moves

phase starts. The correction phase considers three parts of the stack for replacement with some right-hand side. These parts are indicated with ①, ② and ③ in Figure 16.4. Part ① is considered because the precedence at the error detection point could be $\gtrdot$, part ② is considered because the precedence at the error detection point could be $\lessdot$, and part ③ is considered because this precedence could be $\doteq$. Another option is to just delete one of these parts. This results in a fairly large number of possible changes, which now must be limited by making sure that the parser can continue after reducing the right-hand side to its corresponding left-hand side.

In the example, we have the following situation:



The left-hand sides that could replace part ① are: **E**, **T′**, **T**, and **F**. These are the non-terminals that have a precedence relation with the next symbol: the **+**. The only left-hand side that could replace part ② is **F**. Part ③ could be replaced by **E**, **T′**, **T**, and **F**. This still leaves a lot of choices, but some “corrections” are clearly better than others. Let us now see how we can discriminate between them.

Replacing part of the parse stack by a right-hand side can be seen as an edit operation on the stack. The cost of this edit operation can be assessed as follows. With every symbol, we can associate a certain insertion cost I and a certain deletion cost D. The cost for changing for example **T×** to **F** would then be $D(\mathtt{T})+D(\times)+I(\mathtt{F})$. These costs are determined by the parser writer. The cheapest parse stack correction is then chosen. If there is more than one with the same lowest cost, we just pick one.

Assigning identical costs to all edit operations, in our example, we end up with two possibilities, both replacing part ①: **T** (deleting the **×**), or **T×F** (inserting an **F**). Assigning higher costs to editing a non-terminal, which is not unreasonable, would only leave the first of these. Parsing then proceeds as follows:

$\# \lessdot \mathtt{T} \doteq \times \doteq/\lessdot + \doteq \mathtt{T'} \gtrdot$	# error situation
$\# \lessdot \mathtt{T} \doteq \times \doteq/\lessdot + \doteq \mathtt{T'} \gtrdot$	# correct error by deleting **×**
$\# \lessdot \mathtt{T} \gtrdot + \doteq \mathtt{T'} \gtrdot$	# reduce **T**
$\# \lessdot \mathtt{T'} \gtrdot + \doteq \mathtt{T'} \gtrdot$	# reduce **T'**
$\# \lessdot \mathtt{E} \doteq + \doteq \mathtt{T'} \gtrdot$	# reduce **E+T'**
$\# \lessdot \mathtt{E} \gtrdot$	# reduce **E**
$\# \lessdot \mathtt{E'} \gtrdot$	# reduce **E'**
$\# \lessdot \mathtt{S} \gtrdot$	# done

The principles of this method have also been applied in LR parsers. There, however, the backward move is omitted, because in an LR parser the state on top of the stack, together with the next input symbol, determine the reduction that can be applied. If the input symbol is erroneous, we have no way of knowing which reductions can be applied. For further details, see Pennello and DeRemer [300] and also Mickunas and Modry [299].

An interesting form of regional error handling is reported by Burke and Fisher [317]. Two parsers are used simultaneously, with one being several tokens ahead of the other; the input text between them is the region. This allows modifications to be made to the region when the first parser finds a syntax error. Several types of modifications can be applied, in such a way that the second parser never sees an error; see [317] for details. Charles [319] extends this technique with an impressive array of features, resulting in a robust error-correcting LALR parser.

16.5.2 Error Recovery with Bounded-Context Grammars

Error recovery, which is usually the most difficult part of error handling, is particularly easy when we use a bounded-context grammar (Section 9.3.1). The reason is that the bounded context allows the parser to get back on track quickly after an error, since little information is needed to start making correct decisions again.

A BRC parser using the grammar from Figure 9.2, the corresponding BC(2,1) parse table from Figure 9.10, and the input **#n×+n#**, performs the steps

$\# \mathtt{n} \gtrdot_{\mathtt{F}\to\mathtt{n}}$	$\times + \mathtt{n} \#$	
$\# \mathtt{F} \gtrdot_{\mathtt{T}\to\mathtt{F}}$	$\times + \mathtt{n} \#$	
$\# \mathtt{T} \lessdot$	$\times + \mathtt{n} \#$	
$\# \mathtt{T} \times$ Error	$+ \mathtt{n} \#$	stuck

and finds that there is an Error relation between **T×** and **+**. Now, rather than trying to repair the situation at the gap, the parser tries to find the next context in which it *can* take a decision. To this end it has to shift at least 2 tokens; in this case that is enough to continue parsing:

# T × Error + n $\gtrdot_{F \to n}$	#	
# T × Error + F $\gtrdot_{T \to F}$	#	
# T × Error + T $\gtrdot_{E \to E+T}$	#	stuck

The parser detects that it cannot perform the indicated reduction, because rather than a **E** it finds a **×** on the stack. So seen from left to right the **+** is the error symbol and seen from right to left the **×** is the error symbol. The parser can now either delete a token or insert a token. If it deletes the **×** we get the context (**#T**,**+**) which is defined. If it deletes the **+** we get the context (**#T**,**×**) which shifts the **×** which leads to the context (**T×**,**T**) which is not defined. If it inserts, it can insert an **n** or a **(**. The first leads to a correct recovery, the second to failure. Assuming that the parser deletes the **×**,

# T	+ T $\gtrdot_{E \to E+T}$ #	delete ×

it continues as follows:

# T $\gtrdot_{E \to T}$	+ T $\gtrdot_{E \to E+T}$ #
# E $\lessdot$	+ T $\gtrdot_{E \to E+T}$ #
# E + T $\gtrdot_{E \to E+T}$	#
# E $\gtrdot_{S \to E}$	#
# S Accept	#

Ruckert [322] describes the underlying algorithm; this integrated form of parsing and error recovery technique is called "robust parsing" in the paper, because the parser is not easily thrown off course. In [324] Ruckert shows that for the method to work the grammar must be a *continuous grammar*. A grammar is "continuous" if a small change in the input does not correspond to a discontinuous change in the parse tree, under a certain metric. It is shown that all BC grammars are continuous, but not vice versa, and that all continuous grammars are BCP but not vice versa. So we have for the grammars: BC $\subset$ continuous $\subset$ BCP.

16.6 Local Error Handling

All *local error recovery* techniques are so-called *acceptable-set error recovery* techniques. These techniques work as follows: when a parser detects an error, a certain set called the *acceptable-set* is computed from the parser state. Next, symbols from the input are skipped until a symbol is found that is a member of the acceptable-set. Then, the parser state is adapted so that the symbol that is not skipped becomes acceptable. There is a family of such techniques; the members of this family differ in the way they determine the acceptable-set, and in the way in which the parser state is adapted. We will now discuss several members of this family.

16.6.1 Panic Mode

Panic mode is probably the simplest error recovery method that is still somewhat effective. In this method, the acceptable-set is determined by the parser writer, and is fixed for the whole parsing process. The symbols in this set usually indicate the end of a syntactic construct, for example a statement in a programming language. For the programming language Pascal, this set could contain the symbols `;` and **`end`**. When an error is detected, symbols are skipped until a symbol is found that is a member of this set. Then, the parser must be brought into a state that makes this symbol acceptable. In an LL parser, this might require deleting the first few symbols of the prediction, in an LR parser this might involve popping states from the parse stack until a state is uncovered in which the symbol is acceptable.

The recovery capability of panic mode is often quite good, but many errors can go undetected, because sometimes large parts of the input are skipped. The method has the advantage that it is very easy to implement.

16.6.2 FOLLOW-Set Error Recovery

Another early acceptable-set recovery method is the *FOLLOW-set error recovery* method. The idea is applicable in an LL parser, and works as follows: when we are parsing a part of the input, and the top of the prediction stack results most recently from a prediction for the non-terminal A, and we detect an error, we skip symbols until we find a symbol that is a member of FOLLOW(A). Next, we remove the unprocessed part of the current right-hand side of A from the prediction, and continue parsing. As we cannot be sure that the current input symbol can follow A in the present context and is thus acceptable, this is not such a good idea. A better idea is to use that part of FOLLOW(A) that can follow A in this particular context, making sure that the symbol that is not skipped will be accepted, but this is not trivial to do.

The existence of this method is probably the reason that the family of acceptable-set error recovery methods is often called “FOLLOW-set error recovery”. However, for most members of this family this is a confusing name.

A variant of this method that has become very popular in recursive descent parsers is based on the observation that at any point during the parsing process, there are a number of active non-terminals (for which we are now trying to match a right-hand side), and in general this number is larger than one. Therefore, we should use the union of the FOLLOW sets of these non-terminals, rather than the FOLLOW set of just the most recent of them. A better variant uses the union of those parts of the FOLLOW sets that can follow the non-terminals in this particular context. An expansion of this idea is the following: suppose the parser is in the following state when it detects an error:

$\cdots$	$a \cdots$
$\cdots$	$X_1 \cdots X_n$ #

We can then have the acceptable-set contain the symbols in FIRST(X_1), FIRST(X_2), $\cdots$, and #, and recover by skipping symbols until we meet a symbol of this acceptable-set, and then removing symbols from the prediction until the input symbol becomes acceptable.

Many variations of this technique exist; see for example Pemberton [304] and Stirling [314].

16.6.3 Acceptable-Sets Derived from Continuations

A very interesting and effective member of the acceptable-set recovery method family is the one discussed by Röhrich [305]. The idea is as follows. Suppose that a parser with the correct-prefix property detects an error in the input after having processed a prefix u. Because of the correct-prefix property, we know that this prefix u is the start of some sentence in the language. Therefore, there must be a *continuation*, which is a terminal string w, such that uw is a sentence of the language. Now suppose we can compute such a continuation. We can then correct the error as follows:

- Determine a continuation w of u.
- For all prefixes w' of w, compute the set of terminal symbols that would be accepted by the parser after it has parsed w', and take the union of these sets. If a is a member of this set, $uw'a$ is a prefix of some sentence in the language. This set is our acceptable-set. Note that it includes all terminal symbols in w, including the end marker.
- Skip symbols from the input until we find a symbol that is a member of this set. Note that as a result of this, everything up to the end marker may be skipped.
- Insert the shortest prefix of w that makes this symbol acceptable in front of this symbol. If everything up to the end marker was skipped, insert w itself.
- Produce an error message telling the user which symbols were skipped and which symbols were inserted.
- Restart the parser in the state where the error was detected and continue parsing, starting with the inserted symbols. Now the error is corrected, and the parser continues as if nothing has happened.

16.6.3.1 Continuation Grammars

Deriving acceptable sets from continuations requires a solution for two problems: how to determine the continuation and how to compute the acceptable-set without going through all possible parsings. Let us regard a grammar as a generating device. Suppose we are generating a sentence from a grammar, and have obtained a certain sentential form. Now, we want to produce a sentence from it as quickly as possible, using the fewest possible production steps. We can do this if we know for each non-terminal which right-hand side is the quickest "exit", that is, which right-hand side leads to a terminal production in as few production steps as possible.

We can compute these "quickest" right-hand sides in advance. To this end, we compute for each symbol the minimum number of production steps needed to obtain

a terminal derivation from it. We call this number the step count. Terminal symbols have step count 0; non-terminal symbols have an as yet unknown step count, which we set to infinity. Next, we examine each right-hand side in turn. If we already have a step count for each of the members of a right-hand side, the right-hand side itself needs the sum of these step counts, and the left-hand side needs one more if it uses this right-hand side. If this is less than we had for this non-terminal, we update its step count. We repeat this process until none of the step counts changes, as in a transitive closure algorithm.

If we started from a proper grammar, all of the step counts will now be finite. Now all we have to do is for each left-hand side to mark the right-hand side with the lowest step count. The grammar rules thus obtained are called a *continuation grammar*.

Let us see how this works with an example. Consider the grammar of Figure 8.9, repeated in Figure 16.5 for reference. The first pass over the right-hand sides shows

```
Session_S  -->  Facts Question | ( Session ) Session
   Facts   -->  Fact Facts | ε
    Fact   -->  ! STRING
Question   -->  ? STRING
```

Fig. 16.5. An example grammar

us that **Facts**, **Fact**, and **Question** each have step count 1. In the next pass, we find that **Session** has step count 3: its first alternative has two members with step count 1 each, plus 1 for the rule itself. The resulting continuation grammar is presented in Figure 16.6.

```
Session_S  -->  Facts Question
   Facts   -->  ε
    Fact   -->  ! STRING
Question   -->  ? STRING
```

Fig. 16.6. The continuation grammar of the grammar of Figure 16.5

16.6.3.2 Continuation in an LL Parser

In an LL parser, it now is easy to compute a continuation when an error occurs. We take the prediction, and derive a terminal string from it using only rules from the continuation grammar, processing the prediction from left to right. Each terminal that we meet ends up in the acceptable-set; in addition, every time a non-terminal is replaced by its right-hand side from the continuation grammar, we add to the acceptable-set the terminal symbols from the FIRST set of the current sentential form starting with this non-terminal.

Let us demonstrate this with an example. Suppose that we have the input **(? STRING ? STRING** for the LL(1) parser of Figure 8.10. When the parser detects an error, it is in the following state:

(? STRING	? STRING #
···	) Session #

Now a continuation will be computed, starting with the sentential form **) Session #**, using the continuation grammar. During this computation, when the prediction starts with a non-terminal, the FIRST set of the prediction will be computed and the non-terminal will be replaced by its right-hand side in the continuation grammar. The FIRST set is shown in square brackets below the line:

) Session # →

) [(!?] **Facts Question #** →

) [(!?] [!?] ε **Question #** →

) [(!?] [!?] [?] **? STRING #**

Consequently, the continuation is **) ? STRING #** and the acceptable-set contains **(**, **)**, **!**, **?**, **STRING** and **#**. We see that we should keep the **?** and insert the first symbol of the continuation, **)**. So the parser is restarted in the following state:

(? STRING	) ? STRING #
···	) Session #

and proceeds as usual.

16.6.3.3 Continuation in an LR Parser

Unlike an LL parser, an LR parser does not feature a sentential form which represents the rest of the input. It is therefore more difficult to compute a continuation. Röhrich [305] demonstrates that an LR parser can be generated that has a terminal symbol associated with each state of the handle recognizer so that we can obtain a continuation by pretending that the parser has this symbol as input when it is in the corresponding state. The sequence of states that the parser goes through when these symbols are given as input then determines the continuation. The acceptable-set consists of the terminal symbols on which a shift or reduce can take place (i.e. which are acceptable) in any of these states.

16.6.4 Insertion-Only Error Correction

Fischer, Milton and Quiring [303] propose an error correction method for LL(1) parsers using only insertions. This method has become known as the *FMQ* error correction method. In this method, the acceptable-set is the set of all terminal symbols.

Fischer, Milton and Quiring argue that the advantage of using only insertions (and thus no deletions or replacements) is that a syntactically correct input is built around the input supplied by the user, so none of the symbols supplied by the user are deleted or changed.

Not all languages allow insertion-only error correction. If, for example, all strings start with the token **program** and that token cannot occur anywhere else in the input, then an input with two **program** tokens in it cannot be corrected by insertion only. However, many languages allow insertion-only, and other languages are easily modified so that they do.

Let us investigate which properties a language must have for every error to be correctable by insertions only. Suppose we have an input $xa\cdots$ such that the start symbol does derive a sentence starting with x, but not a sentence starting with xa; so x is a correct prefix, but xa is not. Now, if this error is to be corrected by an insertion y, xya must again be a correct prefix. This leads to the notion of *insert-correctable* grammars: a grammar is said to be insert-correctable if for every prefix x of a sentence and every symbol a in the language there is a continuation of x that includes a (so an insertion can always be found). Fischer, Milton and Quiring demonstrate that it is decidable whether an LL(1) grammar is insert-correctable.

So, the FMQ error correction method is applicable in an LL(1) parser built from an insert-correctable grammar. In addition, the LL(1) parser must have the immediate error detection property. As we have seen in Section 8.2.4, the usual (strong-)LL(1) parser does not have this property, but the full-LL(1) parser does. Fischer, Tai and Milton [302] show that for the class of LL(1) grammars in which every non-terminal that derives ε does so explicitly through an ε-rule, the immediate error detection property can be retained while using strong-LL(1) tables.

Now, how does the error corrector work? Suppose that an error is detected on input symbol a, and the current prediction is $X_1\cdots X_n$#. The state of the parser is then:

$\cdots$	$a\ \cdots$
$\cdots$	$X_1\ \cdots\ X_n$ #

As a is an error, we know that it is not a member of FIRST($X_1\cdots X_n$#). We also know that the grammar is insert-correctable, so $X_1\cdots X_n$# must derive a terminal string containing a. The error corrector now determines the cheapest insertion after which a is acceptable. Again, every symbol has associated with it a certain insertion cost, determined by the parser writer; the cost of an insertion is the sum of the costs of the symbols in the insertion.

To compute the cheapest insertion, the error corrector uses some tables that are precomputed for the grammar at hand (by the parser generator). First, there is a table that we will call **cheapest_derivation**, giving the cheapest terminal derivation for each symbol (for a terminal, this is of course the terminal itself). Second, there is a table that we will call **cheapest_insertion** giving for each symbol/terminal combination (X, a) the cheapest insertion y such that $X\overset{*}{\rightarrow}ya\cdots$, if it exists, or an

indication that it does not exist. Note that in any prediction $X_1 \cdots X_n$**#** there must be at least one symbol X such that the (X, a) entry of the **cheapest_insertion** table contains an insertion (or else the grammar was not insert-correctable).

Going back to our parser, we can now compute the cheapest insertion z such that a becomes acceptable. Consulting **cheapest_insertion**(X_1, a), we can distinguish two cases:

- **cheapest_insertion**(X_1, a) contains an insertion y_1; in this case, we have found an insertion.
- **cheapest_insertion**(X_1, a) does not contains an insertion. In this case, we use **cheapest_derivation**(X_1) as the first part of the insertion, and continue with X_2 in exactly the same way as we did with X_1. In the end, this will result in an insertion $y_1 \cdots y_i$, where $y_1, \ldots, y_{i-1}$ come from the **cheapest_derivation** table, and y_i comes from the **cheapest_insertion** table.

The most serious disadvantage of the FMQ error corrector is that it behaves rather poorly on those errors that are better corrected by a deletion. Advantages are that it always works, can be generated automatically, and is simple.

Anderson and Backhouse [310] present a significant improvement of the implementation described above, which is based on the observation that it is sufficient to only compute the first symbol of the insertion: if we detect an error symbol a after having read prefix u, and $w = w_1 w_2 \cdots w_n$ is a cheapest insertion, then $w_2 \cdots w_n$ is a cheapest insertion for the error a after having read uw_1. So the **cheapest_derivation** and **cheapest_insertion** tables are not needed. Instead, tables are needed that are indexed similarly, but only contain the first symbol of the insertion. Such tables are much smaller, and easier to compute.

16.6.5 Locally Least-Cost Error Recovery

Like the FMQ error correction method, *locally least-cost error recovery* (see Backhouse [153] and Anderson et al. [311]) is a technique for recovering from syntax errors by editing the input string at the error detection point. The FMQ method corrects the error by inserting terminal symbols; the locally least-cost method corrects the error by either deleting the error symbol, or inserting a sequence of terminal or non-terminal symbols after which the error symbol becomes correct, or changing the error symbol. Unlike the least-error analysis discussed in Section 16.4, which considers the complete input string in determining the corrections to be made, the locally least-cost method only considers the error symbol itself and the symbol after that. The correction is determined by its cost: every symbol has a certain insertion cost, every terminal symbol has a certain deletion cost, and every replacement also has a certain cost. All these costs are determined by the parser writer. When considering if the error symbol is to be deleted, the cost of an insertion that would make the next input symbol acceptable is taken into account. The cheapest correction is then chosen.

This principle does not rely on a particular parsing method, although the implementation does. The method has successfully been implemented in LL, LR, and Earley parsers; see Backhouse [153], Anderson and Backhouse [306], Anderson et al. [311], and Choe and Chang [315] for details.

McKenzie et al. [321] extend this method by doing a breadth-first search over an ever deepening set of combinations of insertions and deletions. The correcting combinations are then "validated" in the order of cost, by applying them provisionally to the input and running the parser. If the parser accepts a predetermined number of tokens the correction is accepted; otherwise the original input is restored and the system proceeds to the next proposed correction.

Cerecke [325] limits the breadth-first search by analysing the LR automaton. Kim and Choe [326] incorporate the search for validations in the LR parse table.

Corchuelo et al. [327] take a very systematic approach to the problem. The operators "insert", "delete" and "validate" (called "forward move" in the paper) are introduced in the LR parsing mechanism on an equal footing with the usual "shift" and "reduce", in such a way that the original LR parse tables still suffice. This allows very pliable error recovery and easy implementation in an existing parser.

16.7 Non-Correcting Error Recovery

Although the error correction and error recovery methods discussed above have their good and bad points, they all have the following problems in common:

- On an error, they change the input and/or the parser state, using heuristics to choose one of the many possibilities. We can, however, never be sure that we picked the right change.
- Selecting the wrong change can cause an avalanche of spurious error messages. Only the least-error analysis of Section 16.4 does not have this problem.

A quite different approach to error recovery is that of Richter [313]. He proposes a method that does not have the problems mentioned above, but has some problems of its own. The author argues that we should not try to repair an error, because we cannot be sure that we get it right. Neither should we try to change parser state and/or input. The only thing that we can assume is that the rest of the input is a suffix (tail) of a sentence of the language. This is an assumption made in several error recovery methods, but the difference is that most error recovery methods assume more than that, in that they use (some of) the parser state information built so far.

16.7.1 Detection and Recovery

The error recovery method now works as follows: parsing starts with a parser for the original language, preferably one with the correct-prefix property. When an error is detected, it is reported, and the present parsing effort is abandoned. To analyze the remaining suffix, a parser derived from the suffix grammar, a so-called *suffix parser*, is started on it. The detected error symbol is not discarded: it could very well be a

correct beginning of a suffix, for example when the only actual error is a missing symbol.

If during the suffix scan another syntax error is detected, it is again reported, and the suffix parser is reset to its starting state, ready to accept another suffix. This guarantees that each reported error is a genuine syntax error, since the situation is found to be incompatible with the input from the previous error onwards, regardless of what came before. It is also different from and not caused by the previous error, since all information from before the previous one has been discarded. For the same reason no error is reported more than once. This maintains a high level of user confidence in the error messages, which is a great advantage. A possible disadvantage is, that in the presence of errors the parser is unable to deliver a meaningful parse tree.

Since the method does not correct existing input, it is called a *non-correcting error recovery* method.

When the method was first invented in 1985, it was hard to apply it in real-world parsers. It was easy enough to construct the suffix grammar (see Section 12.1), but that grammar was not amenable to the usual LL or LR methods, and general CF methods were too expensive — or at least deemed to be so. That changed with the invention of efficient, and sometimes even linear-time suffix parsers (Chapter 12). Another way of solving the problem is to use an efficient GLR or GLL parser (Chapter 11) and have it generate the suffix grammar implicitly on the fly. An example of this technique is described by van Deudekom and Kooiman [170]. Given the complexity of writing a linear suffix parser or an efficient GLR or GLL parser, the simplicity of a general CF parser, and the speed of present-day processors, it might be easier to use a general CF parser to do the suffix analysis; that approach has additional advantages, as we shall see in the next section.

16.7.2 Locating the Error

Although non-correcting error recovery cannot give spurious error messages, it can miss errors, even arbitrarily many of them. If our input is described by the grammar `S-->(S)`, `S-->[S]`, `S-->`ε, which produces properly nested sequences of open and close parentheses and brackets, and the input is `((((]]]]`, the first `]` is detected as illegal and ends the correct prefix. But the rest, `]]]]` is a perfectly legal suffix, so only one error is reported. This is probably not a big disadvantage in an interactive environment.

When a directional parser finds a syntax error, the only thing we can say is that the error must have been somewhere in the input read so far. It can even be arbitrarily far back, at the start of the input. Suppose our input language consists of arithmetic expressions, and the input is *E*`)**3`, where *E* is a long, correct arithmetic expression and `**` is the exponentiation operator. The obvious error is that a left parenthesis was missing at the beginning: `(`*E*`)**3`.

In a non-correcting parser that uses a general CF parser for the suffix analysis we can do better. We can use the CF parser to scan the input text backwards, using the reverse grammar of the input language; that grammar probably has no redeeming

properties, but a general CF parser can handle it. If there is only one error, the backward scan will find an error at or to the left of the position of the forward error; the region between the two errors is called the *error interval*. In the example above the error is found at the start of the input, and the resulting error message

```
Syntax error:
unexpected ) at position N and
unexpected beginning of input at position 0
```

will give the user a good idea of where to look.

If there is more than one error, a similar scheme can be used to locate more errors, but care is required since error intervals may overlap. Richter [313] give the details.

16.8 Ad Hoc Methods

The *ad hoc error recovery* methods are called ad hoc because they cannot be automatically generated from the grammar. These methods are as good as the parser writer makes them, which in turn depends on how good the parser writer is in anticipating possible syntax errors. We will discuss three of these ad hoc methods: error productions, empty table slots and error tokens.

16.8.1 Error Productions

Error productions are grammar rules, added by the grammar writer so that anticipated syntax errors become part of the language (and thus are no longer syntax errors). These error productions usually have a semantic action associated with them that reports the error; this action is triggered when the error production is used. An example where an error production could be useful is the Pascal if-statement, which has the following syntax:

```
if-statement  -->  IF boolean-expression
                   THEN statement else-part
   else-part  -->  ELSE statement | ε
```

A common error is that an **if-statement** has an **else-part**, but the statement in front of the **else-part** is terminated by a semicolon. In Pascal, a semicolon is a statement separator rather than a statement terminator and is not allowed in front of an **ELSE**. This situation could be detected by changing the grammar rule for **else-part** into

```
else-part  -->  ELSE statement | ε | ; ELSE statement
```

where the last right-hand side is the error production.

The most important disadvantages of error productions are:

- only anticipated errors can be handled;
- the modified grammar might (no longer) be suitable for the parsing method used, because conflicts could be introduced by the added rules.

The advantage is that a very adequate error message can be given. Error productions can be used profitably in conjunction with another error handling method, to handle some frequent errors on which the other method does not perform well.

16.8.2 Empty Table Slots

In most of the efficient parsing methods, the parser consults one or more parse tables and bases its next parsing decision on the result. These parsing tables have error entries (represented as the empty slots), and if one of these is consulted, an error is detected. In this error handling method, the empty table slots are used to refer to error handling routines. Each empty slot has its own error handling routine, which is called when the corresponding slot is consulted. The error handling routines themselves are written by the parser writer. By very careful design of these error handling routines, very good results can be obtained; see for example Conway and Wilcox [293]. In order to achieve good results, however, the parser writer must invest considerable effort. Usually, this is not considered worth the gain, in particular because good error handling can be generated automatically.

16.8.3 Error Tokens

Another popular error recovery method uses error tokens. An *error token* is a special token that is inserted in front of the error detection point. The parser will pop states from the parse stack until this token becomes valid, and then skip symbols from the input until an acceptable symbol is found. The parser writer extends the grammar with rules using this error token. An example of this is the following grammar:

```
     input  -->  input input_line | ε
input_line  -->  ERROR_TOKEN NEWLINE | STRING NEWLINE
```

This kind of grammar is often seen in interactive applications, where the input is line by line. Here, **ERROR_TOKEN** denotes the error token, and **NEWLINE** denotes an end of line marker. When an error occurs, states are popped until **ERROR_TOKEN** becomes acceptable, and then symbols are skipped until a **NEWLINE** is encountered.

This method can be quite effective, provided that care is taken in designing the rules using the error token.

16.9 Conclusion

In principle error handling is a hopeless task, in that the goal of having a computer handle errors properly in any intuitive meaning of the word is out of reach. In practice the far less lofty goal of not looping and not crashing is often already difficult to achieve. The techniques described in this chapter walk a middle ground: they define a metric for the corrections, thus creating an objective goal, and insert a token only when it can be proven that no looping can occur.

The techniques are very parse-method specific but often involve a search for the “best” correction; sometimes the results of this search can be precomputed.

Problems

Problem 16.1: 1. Can the error message learning system of Jeffery [328] (Section 16.1) be implemented in an strong-LL(1) parser? 2. Implement it in your favorite parser generator.

Problem 16.2: *Project*: The globally least-error correction method explained in Section 16.4 can give spectacularly wrong results; for example, if the input is text in a computer programming language and contains more than 2 errors, the input can sometimes be "corrected" by putting comment symbols around the entire text. This suggests that "most-recognized correction", in which the number of accepted tokens is maximized, may be preferred over "least-error correction". Apply this idea to an Unger, CYK or Earley parser and investigate.

Problem 16.3: *Research project*: Research error handling by intersection parsing.

Problem 16.4: In Section 16.5 we claim that regional error handling is applicable to bottom-up parsers only. Why can we not just apply the top-down counterparts of its actions to a top-down parser: predict as much as we can, and then try to match with insertions and deletions?

Problem 16.5: Give an intuitive argument why BC and BRC grammars allow the method of Section 16.5.2 to be applied, and BCP grammars do not, as stated at the end of that section.

Problem 16.6: In Section 16.6.3.2 the implementation of acceptable-set error recovery with continuations is described using a scan over the prediction, but if we are dealing with a recursive descent parser there is no explicit prediction. When the conceptual prediction is $X_1 \cdots X_n \#$, we are in the routine for X_1 and the other elements of the prediction are hidden in the calling stack. Devise a way to obtain the acceptable-set when needed without explicitly constructing the prediction.

Problem 16.7: In Section 16.6.4 the implementation of insertion-only error correction is described using a scan over the prediction, but if we are dealing with a recursive descent parser there is no explicit prediction, as in Problem 16.6. Devise a way to obtain the cheapest insertion when needed without explicitly constructing the prediction.