Error Handling

- a compiler should not stop parsing if a syntactic error is encountered but instead report the error and then proceed with the analysis
- however, continuation after an error is only possible assuming certain hypothesis about the nature of the error
- choosing hypothesis is difficult
  - → missing punctuation symbols are a frequent mistake
  - → operator symbols are usually <u>not</u> omitted
  - → yet for a parser both kinds of symbols are the same

quality criteria for error handling:

- as many errors as possible should be detected in a single run
- as few additional assumptions as possible about the language should be made
- error handling should not slow down the compiler
- the parser should not grow in size

there are two cases of errors:

1. missing symbols: easy ← pretend as if symbols were present
2. wrong symbols: difficult

Missing Symbols

- the parser proceeds by omitting calls to the scanner

⇝ example:

- right parenthesis (RPAREN) at the end of a factor is missing

```
if (symbol == RPAREN)
  getSymbol();
else
  mark(") missing");
```

↑

report error, then continue (mention line, column)

note:

- only weak symbols are omitted in most cases (comma, semicolon, closing symbols)
- special case:
  equality sign "==" instead of assignment sign "="

⇝ can also be skipped!

Wrong Symbols

factor = (identifier selector) | integer | "(" expression ")" |
procedureCall | "!" factor | string .

First<factor> = {IDENTIFIER, INTEGER, LPAREN, NEG, STRING}

rarely misused! ~> strong symbols

~> serve as synchronization points where parsing can be resumed with high probability of success!

~> look at your grammar and identify synchronization points!

```
factor() {
  if (symbol == IDENTIFIER) {
    ...
  } else if (symbol == INTEGER) {
    ...
  } else if (symbol == LPAREN) {
    ...
  } else if (symbol == NEG) {
    ...
  } else if (symbol == STRING) {
    ...
  } else
    error();
}
```

symbol ∉ First<factor>?

skip symbols until a strong symbol is reached!

```
factor() {
  while (symbol != IDENTIFIER && symbol != INTEGER &&
         symbol != LPAREN && symbol != NEG &&
         symbol != STRING) {
    getSymbol();
    mark("identifier, integer, (, !, or string expected");
  }
  if (symbol == IDENTIFIER) {
    ...
  } else if (symbol == INTEGER) {
    ...
  } ...
}
```

skip symbols until symbol ∈ First<factor>

check for EOF!

token encoding in ranges enables constant-time check: symbol < IDENTIFIER && symbol > STRING

Sequences

statementSequence = { statement ";" } .

First<statementSequence>=First<statement>
First<statement>={IDENTIFIER, IF, WHILE, RETURN}
Follow<statementSequence>={END}

"}"

parser fragment:

symbol ∈ First<statementSequence>?

```
while (symbol == IDENTIFIER || symbol == IF ||
          symbol == WHILE || symbol == RETURN) {
  statement();
  if (symbol == SEMICOLON)
    getSymbol();
  else
    error();
}
```

";" expected!

instead we do:

symbol ∉ First<statementSequence>?

```
while (true) {
  while (symbol != IDENTIFIER && symbol != IF &&
            symbol != WHILE && symbol != RETURN) {
    getSymbol();
    mark("identifier, if, while, or return expected");
  }
  statement();
  if (symbol == SEMICOLON)
    getSymbol();
  else
    mark("; missing");
  if (symbol == END)
    return;
}
```

check for EOF!

skip symbols until symbol ∈ First<statement>

symbol ∈ Follow<statementSequence>?

→ handle other sequences similarly!

Termination, Robustness, Usefulness

- termination:
  - ⇝ read at least one symbol in each loop iteration
- robustness:
  - ⇝ no input leads to a crash
- usefulness:
  - ⇝ correct diagnosis of frequent errors with minimal additional error messages