3. Parsing

Context-Free Grammars

Problem

Regular Grammars cannot handle central recursion

```
E = x | "(" E ")".
```

For such cases we need context-free grammars

Definition

A grammar is called ***context-free*** (CFG) if all its productions have the following form:

$\boxed{X = \alpha.}$ $X \in$ NTS, α non-empty sequence of TS and NTS

In EBNF the right-hand side α can also contain the meta symbols |, (), [] and {}

Example

```
Expr = Term {("+" | "-") Term}.
Term = Factor {("*" | "/") Factor}.
Factor = id | "(" Expr ")".
```

indirect central recursion

Context-free grammars can be recognized by ***push-down automata***

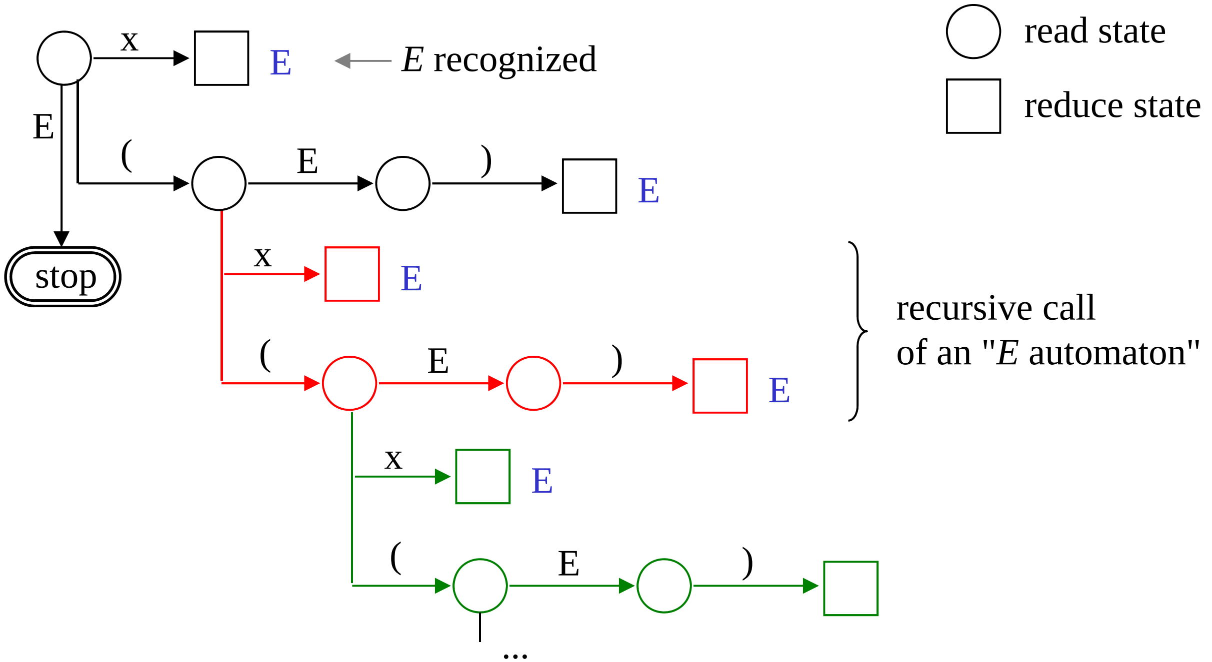
Push-Down Automaton (PDA)

Characteristics

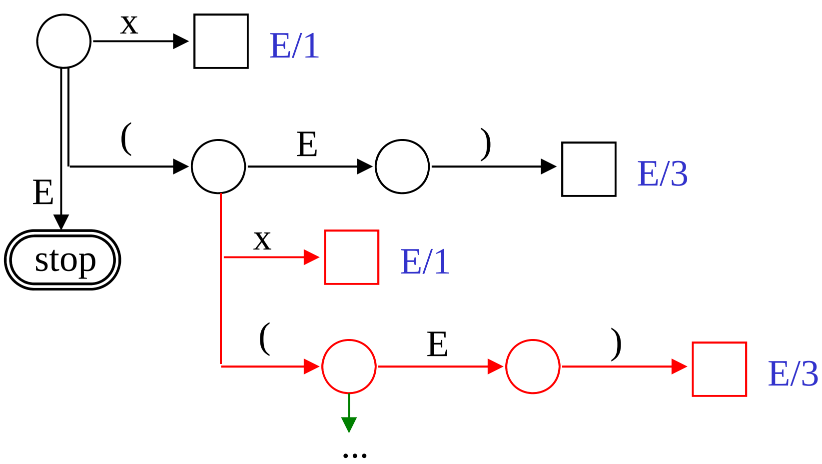
- Allows transitions with terminal symbols and nonterminal symbols
- Uses a stack to remember the visited states

Example

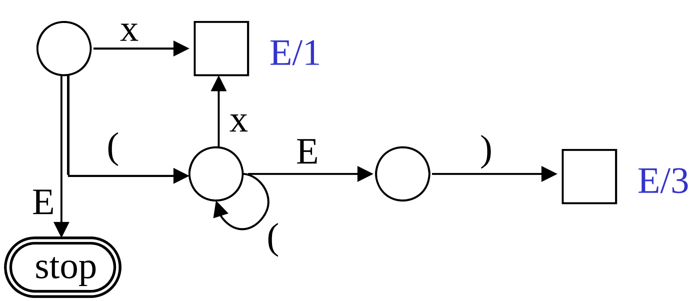
E = x | "(" E ")".



Push-Down Automaton (continued)



Can be simplified to …



Needs a stack to remember the way back from where it came

Limitations of Context-Free Grammars

CFGs cannot express *context conditions*

For example:

- ***Every name must be declared before it is used***
  The declaration belongs to the context of the use; the statement
    x = 3;
  may be right or wrong, depending on its context

- ***The operands of an expression must have compatible types***
  Types are specified in the declarations, which belong to the context of the use

Possible solutions

- ***Use context-sensitive grammars***
  too complicated

- ***Check context conditions later during <u>semantic analysis</u>***
  i.e. the syntax allows sentences for which the context conditions do not hold
    int x; … x = "three";   syntactically correct
                              semantically wrong
  The error is detected during semantic analysis (not during syntax analysis).

Context Conditions

Semantic constraints that are specified for every production

For example in MicroJava

Statement = Designator "=" Expr ";".

- *Designator* must be a variable, an array element or an object field.
- The type of *Expr* must be assignment compatible with the type of *Designator*.

Factor = "new" ident "[" Expr "]".

- *ident* must denote a type.
- The type of *Expr* must be *int*.

Designator_1 = Designator_2 "[" Expr "]".

- $Designator_2$ must be a variable, an array element or an object field.
- The type of $Designator_2$ must be an array type.
- The type of *Expr* must be *int*.

3. Parsing

3.1 Context-Free Grammars and Push-Down Automata

3.2 Recursive Descent Parsing

3.3 LL(1) Property

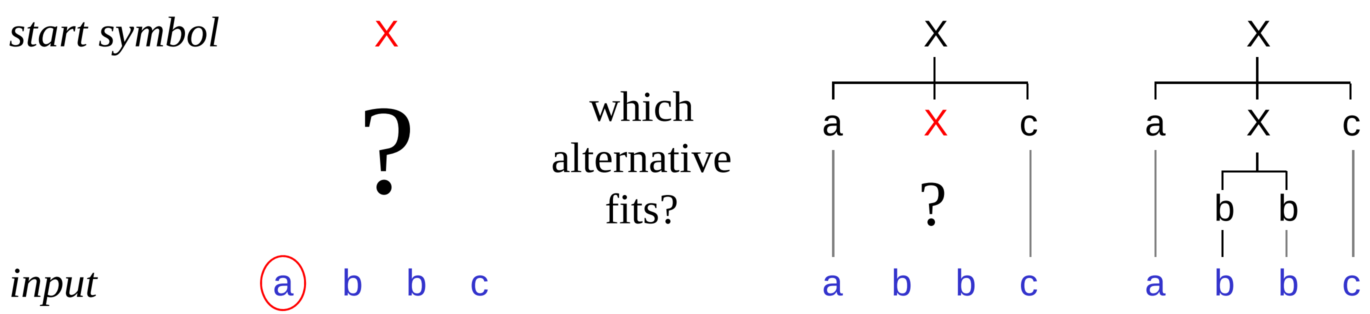
3.4 Error Handling

Recursive Descent Parsing

- Top-down parsing technique
- The syntax tree is build from the start symbol down to the sentence (top-down)

Example *grammar* *input*

`X = a X c | b b.` `a b b c`



The correct alternative is selected using ...

- the lookahead token from the input stream
- the terminal start symbols of the alternatives

Static Variables of the Parser

Lookahead token

At any moment the parser knows the next input token

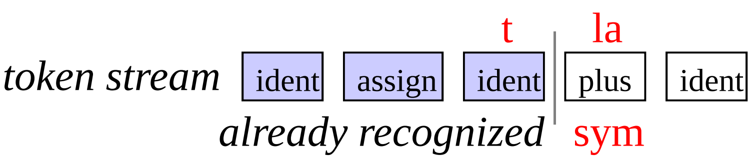
```
private static int sym;     // token number of the lookahead token
```

The parser remembers two input tokens (for semantic processing)

```
private static Token t;     // most recently recognized token
private static Token la;    // lookahead token (still unrecognized)
```

These variables are set in the method *scan()*

```
private static void scan() {
    t = la;
    la = Scanner.next();
    sym = la.kind;
}
```



scan() is called at the beginning of parsing ⇒ first token is in *sym*

How to Parse Terminal Symbols

Pattern

symbol to be parsed: a
parsing action: check(a);

Needs the following auxiliary methods

```
private static void check (int expected) {
    if (sym == expected) scan();  // recognized => read ahead
    else error( name[expected] + " expected" );
}
```

```
private static void error (String msg) {
    System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
    System.exit(1);  // for a better solution see later
}
```

```
private static String[] name = {"?", "identifier", "number", ..., "+", "-", ...};
```

ordered by token codes

The names of the terminal symbols are declared as constants

```
static final int
    none = 0,
    ident = 1,
    ... ;
```

How to Parse Nonterminal Symbols

Pattern

symbol to be parsed: X
parsing action: X(); // call of the parsing method X

Every nonterminal symbol is recognized by a parsing method with the same name

```
private static void X() {
    ... parsing actions for the right-hand side of X ...
}
```

Initialization of the MicroJava parser

```
public static void Parse() {
    scan();          // initializes t, la and sym
    MicroJava();     // calls the parsing method of the start symbol
    check(eof);      // at the end the input must be empty
}
```

How to Parse Sequences

Pattern

production: X = a Y c.

parsing method:

```
private static void X() {
    // sym contains a terminal start symbol of X
    check(a);
    Y();
    check(c);
    // sym contains a follower of X
}
```

Simulation

X = a Y c.
Y = b b.

	remaining input
private static void **X**() {	a b b c
check(a);	b b c
Y();	c
check(c);	
}	
private static void **Y**() {	b b c
check(b);	b c
check(b);	c
}	

How to Parse Alternatives

Pattern $\alpha \mid \beta \mid \gamma$ α, β, γ are arbitrary EBNF expressions

Parsing action

```
if (sym ∈ First(α)) { ... parse α ... }
else if (sym ∈ First(β)) { ... parse β ... }
else if (sym ∈ First(γ)) { ... parse γ ... }
else error("...");  // find a meaninful error message
```

Example

```
X = a Y | Y b.
Y = c | d.
```

First(aY) = {a}
First(Yb) = First(Y) = {c, d}

```
private static void X() {
    if (sym == a) {
        check(a);
        Y();
    } else if (sym == c || sym == d) {
        Y();
        check(b);
    } else error ("invalid start of X");
}
```

How to Parse EBNF Options

Pattern [α] α is an arbitrary EBNF expression

Parsing action

```
if (sym ∈ First(α)) { ... parse α ... }  // no error branch!
```

Example

```
X = [a b] c.
```

```
private static void X() {
    if (sym == a) {
        check(a);
        check(b);
    }
    check(c);
}
```

Example: parse a b c
parse c

How to Parse EBNF Iterations

Pattern $\{\alpha\}$ α is an arbitrary EBNF expression

Parsing action while (sym $\in$ First(α)) { ... *parse* α ... }

Example

```
X = a {Y} b.
Y = c | d.
```

```
private static void X() {
    check(a);
    while (sym == c || sym == d) Y();
    check(b);
}
```

alternatively ...

```
private static void X() {
    check(a);
    while (sym != b) Y();
    check(b);
}
```

Example: parse a c d c b
parse a b

... but there is the danger of an endless loop, if *b* is missing in the input

How to Deal with Large First Sets

If the set has 5 or more elements: use class *BitSet*

e.g.: First(X) = {a, b, c, d, e}
First(Y) = {f, g, h, i, j}

First sets are initialized at the beginning of the program

```
import java.util.BitSet;

private static BitSet firstX = new BitSet();
firstX.set(a); firstX.set(b); firstX.set(c); firstX.set(d); firstX.set(e);

private static BitSet firstY = new BitSet();
firstY.set(f); firstY.set(g); firstY.set(h); firstY.set(i); firstY.set(j);
```

Usage

Z = X | Y.

```
private static void Z() {
    if (firstX.get(sym)) X();
    else if (firstY.get(sym)) Y();
    else error("invalid Z");
}
```

If the set has less than 5 elements: use explicit checks (which is faster)

e.g.: First(X) = {a, b, c}

```
if (sym == a || sym == b || sym == c) ...
```

Optimizations

Avoiding multiple checks

X = a | b.

unoptimized

```
private static void X() {
    if (sym == a) check(a);
    else if (sym == b) check(b);
    else error("invalid X");
}
```

optimized

```
private static void X() {
    if (sym == a) scan();  // no check(a);
    else if (sym == b) scan();
    else error("invalid X");
}
```

X = {a | Y d}.
Y = b | c.

unoptimized

```
private static void X() {
    while (sym == a || sym == b || sym == c) {
        if (sym == a) check(a);
        else if (sym == b || sym == c) {
            Y(); check(d);
        } else error("invalid X");
    }
}
```

optimized

```
private static void X() {
    while (sym == a || sym == b || sym == c) {
        if (sym == a) scan();
        else { // no check any more
            Y(); check(d);
        } // no error case
    }
}
```

Optimizations

More efficient scheme for parsing <u>alternatives in an iteration</u>

X = {a | Y d}.

like before

```
private static void X() {
    while (sym == a || sym == b || sym == c) {
        if (sym == a) scan();
        else {
            Y(); check(d);
        }
    }
}
```

optimized

```
private static void X() {
    for (;;) {
        if (sym == a) scan();
        else if (sym == b || sym == c) {
            Y(); check(d);
        } else break;
    }
}
```

no multiple checks on a

Optimizations

Frequent iteration pattern

α {separator α}

so far

```
... parse α ...
while (sym == separator) {
    scan();
    ... parse α ...
}
```

shorter

```
for (;;) {
    ... parse α ...
    if (sym == separator) scan(); else break;
}
```

Example

ident {"," ident}

```
check(ident);
while (sym == comma) {
    scan();
    check(ident);
}
```

```
for (;;) {
    check(ident);
    if (sym == comma) scan(); else break;
}
```

input e.g.: a , b , c

Computing Terminal Start Symbols Correctly

Grammar

```
X = Y a.
Y = {b} c
  | [d]
  | e.
```

terminal start symbols

b and *c*

d and *a* (!)

e

```
Z = U e
  | f.
U = {d}.
```

d and *e* (*U* is deletable!)

f

Parsing methods

```
private static void X() {
    Y(); check(a);
}
```

```
private static void Y() {
    if (sym == b || sym == c) {
        while (sym == b) scan();
        check(c);
    } else if (sym == d || sym == a) {
        if (sym == d) scan();
    } else if (sym == e) {
        scan();
    } else error("invalid Y");
}
```

```
private static void Z() {
    if (sym == d || sym == e) {
        U(); check(e);
    } else if (sym == f) {
        scan();
    } else error("invalid Z");
}
```

```
private static void U() {
    while (sym == d) scan();
}
```

3. Parsing

3.1 Context-Free Grammars and Push-Down Automata

3.2 Recursive Descent Parsing

3.3 LL(1) Property

3.4 Error Handling

LL(1) Property

Precondition for recursive descent parsing

LL(1) ... can be analyzed from **L**eft to right
with **L**eft-canonical derivations (leftmost NTS is derived first)
and **1** lookahead symbol

Definition

1. A grammar is LL(1) if all its productions are LL(1).
2. A production is LL(1) if for all its alternatives
   $\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
   the following condition holds:
   $\mathrm{First}(\alpha_i) \cap \mathrm{First}(\alpha_j) = \{\}$ (for any $i \neq j$)

In other words

- The terminal start symbols of all alternatives of a production must be pairwise disjoint.
- The parser must always be able to select one of the alternatives by looking at the lookahead token.

How to Remove LL(1) Conflicts

Factorization

```
IfStatement =  "if" "(" Expr ")" Statement
            |  "if" "(" Expr ")" Statement "else" Statement.
```

Extract common start sequences

```
IfStatement =  "if" "(" Expr ")" Statement (
                                           | "else" Statement
                                           ).
```

... or in EBNF

```
IfStatement =  "if" "(" Expr ")" Statement ["else" Statement].
```

Sometimes nonterminal symbols must be inlined before factorization

```
Statement   =  Designator "=" Expr ";"
            |  ident "(" [ActualParameters] ")" ";".
Designator  =  ident {"." ident}.
```

Inline *Designator* in *Statement*

```
Statement   =  ident {"." ident} "=" Expr ";"
            |  ident "(" [ActualParameters] ")" ";".
```

then factorize

```
Statement   =  ident (   {"." ident} "=" Expr ";"
                     |   "(" [ActualParameters] ")" ";"
                     ).
```

How to Remove Left Recursion

Left recursion is always an LL(1) conflict

For example

IdentList = ident | IdentList "," ident.

generates the following phrases

ident
ident "," ident
ident "," ident "," ident
...

can always be replaced by iteration

IdentList = ident {"," ident}.

Hidden LL(1) Conflicts

EBNF options and iterations are hidden alternatives

X = [α] β. $\equiv$ X = α β | β. α and β are arbitrary EBNF expressions

Rules

X = [α] β.	First(α) $\cap$ First(β) must be {}
X = {α} β.	First(α) $\cap$ First(β) must be {}

X = α [β].	First(β) $\cap$ Follow(X) must be {}
X = α {β}.	First(β) $\cap$ Follow(X) must be {}

X = α | .	First(α) $\cap$ Follow(X) must be {}

Removing Hidden LL(1) Conflicts

```
Name = [ident "."] ident.
```

Where is the conflict and how can it be removed?

```
Prog = Declarations ";" Statements.
Declarations = D {";" D}.
```

Where is the conflict and how can it be removed?

Dangling Else

If statement in Java

```
Statement =  "if" "(" Expr ")" Statement ["else" Statement]
          |  ... .
```

This is an LL(1) conflict!

First("else" Statement) $\cap$ Follow(Statement) = {"else"}

It is even an ambiguity which cannot be removed

$$\overbrace{\underbrace{\text{if (expr1) }\underbrace{\overbrace{\text{if (expr2) stat1;}}^{\text{Statement}}\text{ else stat2;}}_{\text{Statement}}}_{\text{Statement}}}^{\text{Statement}}$$

We can build 2 different syntax trees!

LL(1) Conflicts are only warnings

What if we ignore them?

The parser will select the first matching alternative

```
X = a b c
  |  a d.
```

if the lookahead token is an *a* the parser will select this alternative

Example: Dangling Else

```
Statement =  "if" "(" Expr ")" Statement [ "else" Statement ]
          |   ... .
```

If the lookahead token is "else" here
the parser starts parsing the option;
i.e. the "else" belongs to the innermost "if"

$\underbrace{\text{if (expr1) } \underbrace{\text{if (expr2) stat1; else stat2;}}_{\text{Statement}}}_{\text{Statement}}$

Luckily this is what we want here.

3. Parsing

Goals of Syntax Error Handling

Requirements

1. The parser should detect as many errors as possible in a single compilation
2. The parser should never crash (even in the case of abstruse errors)
3. Error handling should not slow down error-free parsing
4. Error handling should not inflate the parser code

Error handling techniques for recursive descent parsing

- Error handling with "panic mode"
- Error handling with "dynamically computed recovery sets"
- Error handling with "synchronization points"

Panic Mode

The parser gives up after the first error

```
private static void error (String msg) {
    System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
    System.exit(1);
}
```

Advantages

- cheap
- sufficient for small command languages or for interpreters

Disadvantages

- inappropriate for production-quality compilers

Recovery At Synchronization Points

Error recovery is only done at particularly "safe" positions

i.e. at positions where keywords are expected which do not occur at other positions in the grammar

anchor sets

For example
- start of Statement: if, while, do, ...
- start of Declaration: public, static, void, ...

Problem: *ident* can occur at both positions!
ident is not a safe anchor $\Rightarrow$ omit it from the anchor set

Code that has to be inserted at the synchronization points

anchor set at this synchronization point

```
...
if (sym ∉ expectedSymbols) {
    error("...");
    while (sym ∉ (expectedSymbols ∪ {eof})) scan();
}
...
```

in order not to get into an endless loop

- Synchronization sets (i.e. *expectedSymbols*) can be computed at compile time
- After an error the parser "stumbles ahead" to the next synchronization point

Example

Synchronization at the start of Statement

```
private static void Statement() {
    if (!firstStat.get(sym)) {
        error("invalid start of statement");
        while (!syncStat.get(sym)) scan();
    }
    if (sym == if_) {
        scan();
        check(lpar); Expr(); check(rpar);
        Statement();
        if (sym == else_) { scan(); Statement(); }
    } else if (sym == while_) {
        ...
}
```

the rest of the parser remains unchanged
(as if there were no error handling)

```
public static int errors = 0;

public static void error (String msg) {
    System.out.println(...);
    errors++;
}
```

```
static BitSet firstStat = new BitSet();
firstStat.set(while_);
firstStat.set(if_);
...
static BitSet syncStat = ...; // firstStat without ident
    // but with eof
```

Suppressing Spurious Error Messages

While the parser moves from the error position to the next synchronization point it produces spurious error messages

Solved by a simple heuristics

If less than 3 tokens were recognized correctly since the last error, the parser assumes that the new error is a spurious error.
Spurious errors are not reported.

```
private static int errDist = 3;  // next error should be reported
```

```
private static void scan() {
    ...
    errDist++;  // another token was recognized
}
```

```
public static void error (String msg) {
    if (errDist >= 3) {
        System.out.println("line " + la.line + " col " + la.col + ": " + msg);
        errors++;
    }
    errDist = 0;  // counting is restarted
}
```

Example of a Recovery

```
private static void Statement() {
    if (!firstStat.get(sym)) {
        error("invalid start of statement");
        while (!syncStat.get(sym)) scan();
        errDist = 0;
    }
    if (sym == if_) {
        scan();
        check(lpar); Condition(); check(rpar);
        Statement();
        if (sym == else_) { scan(); Statement(); }
    ...
}
```

```
private static void check (int expected) {
    if (sym == expected) scan();
    else error(...);
}
```

```
private static void error (String msg) {
    if (errDist >= 3) {
        System.out.println(...);
        errors++;
    }
    errDist = 0;
}
```

erroneous input: if a > b , max = a; while ...

sym	*action*	
if	scan();	*if* ∈ *firstStat* ⇒ ok
ident_a	check(lpar);	error: (expected
	Condition();	parses a > b
comma	check(rpar);	error:) expected
	Statement();	*comma* does not match ⇒ error, but no error message
		skip ", *max = a;*" , synchronize with *while_*
while		synchronization successful!

Synchronization at the Start of an Iteration

For example

```
Block = "{" {Statement} "}".
```

Standard pattern in this case

```
private static void Block() {
   check(lbrace);
   while (sym ∈ First(Statement))
      Statement();
   check(rbrace);
}
```

If the token after *lbrace* does not match *Statement* the loop is not executed. Synchronization point in *Statement* is never reached.

Thus

```
private static void Block() {
   check(lbrace);
   while (sym ∉ {rbrace, eof})
      Statement();
   check(rbrace);
}
```

Improvement of the Synchronization

Consider ";" as an anchor (if it is not already in *First*(*Statement*) anyway)

x = ...; y = ...; if ......; while ......; z = ...;

synchronization points

```
private static void Statement() {
    if (!firstStat.get(sym)) {
        error("invalid start of statement");
        do scan(); while (sym ∉ (syncStat ∪ {rbrace, semicolon}));
        if (sym == semicolon) scan();
        errDist = 0;
    }
    if (sym == if_) {
        scan();
        check(lpar); Condition(); check(rpar);
        Statement();
        if (sym == else_) { scan(); Statement(); }
    ...
}
```

Assessment

Error handling at synchronization points

Advantages

+ does not slow down error-free parsing
+ does not inflate the parser code
+ simple

Disadvantage

- needs experience and "tuning"