# 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:

| A = α. | A ∈ NTS, α non-empty sequence of TS and NTS<br>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***

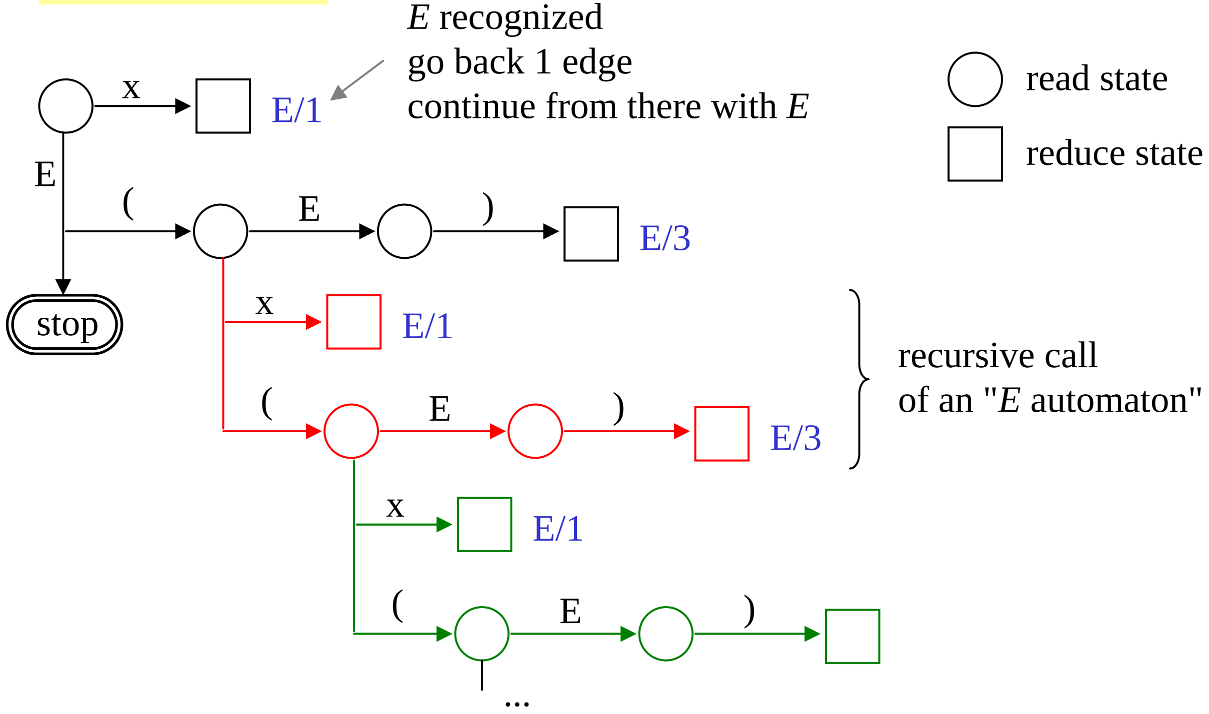
# *Push-Down Automaton (PDA)*

## Characteristics

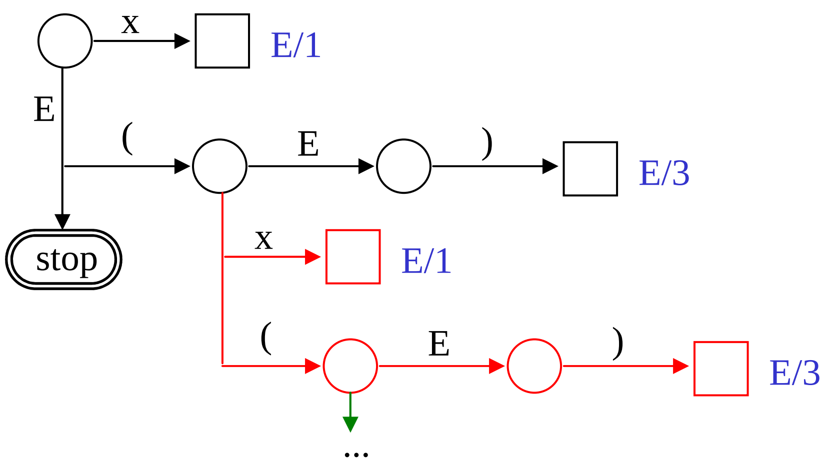
- Allows transitions with terminal symbols <u>and</u> nonterminal symbols
- Uses a stack to remember the visited states

## Example

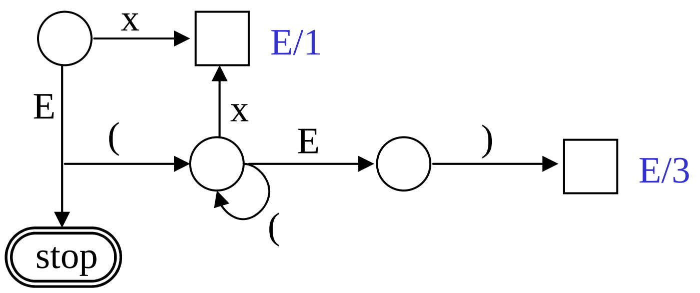
E = x | "(" E ")".



# *Push-Down Automaton (cont.)*



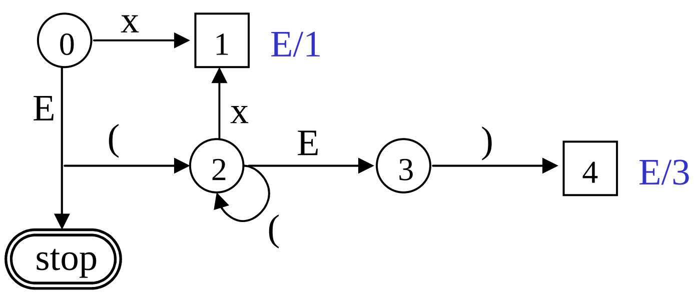

**Can be simplified to**



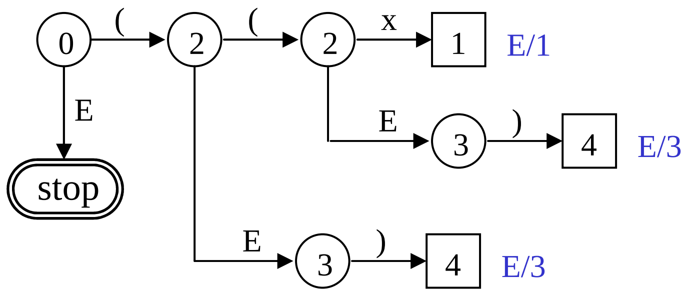

Needs a stack in order to find its way back through the visited states

# How a PDA Works

**Example:** input is ((x))




**Visited states are stored in a stack**



# *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 during <u>semantic analysis</u>***
  i.e. the syntax allows sentences for which the context conditions do not hold
  int 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 Z#

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*.

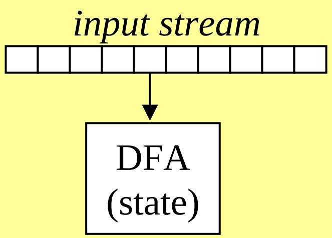
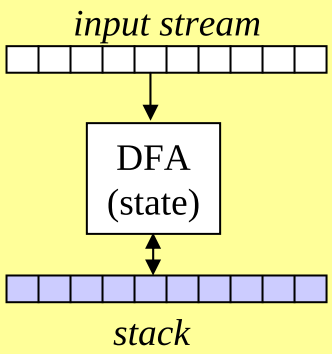
Factor = "new" ident "[" Expr "]".

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

$\text{Designator}_1$ = $\text{Designator}_2$ "[" Expr "]".

- The type of $\textit{Designator}_2$ must be an array type.
- The type of *Expr* must be *int*.

# *Regular versus Context-free Grammars*

| | **Regular Grammars** | **Context-free Grammars** |
|---|---|---|
| **Used for** | Scanning | Parsing |
| **Recogized by** | **DFA** (no stack)<br> | **PDA** (stack)<br> |
| **Productions** | A = a \| b C. | A = $\alpha$. |
| **Problems** | nested language constructs | context-sensitive constructs<br>(e.g. type checks, ...) |

# 3. Parsing

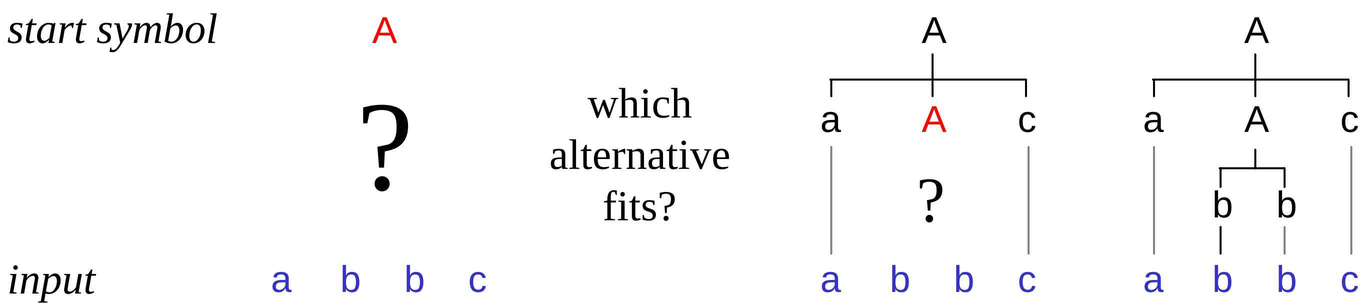
# *Recursive Descent Parsing*

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

**Example** *grammar* *input*

A = a A 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

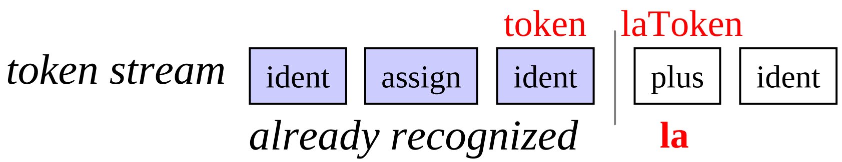
```
static int la;            // token number of the lookahead token
```

It remembers two input tokens (for semantic processing)

```
static Token token;       // most recently recognized token
static Token laToken;     // lookahead token (still unrecognized)
```

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

```
static void Scan () {
    token = laToken;
    laToken = Scanner.Next();
    la = laToken.kind;
}
```




*Scan()* is called at the beginning of parsing ⇒ first token is in *la*

# *How to Parse Terminal Symbols*

**Pattern**

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

**Needs the following auxiliary methods**

```
static void Check (int expected) {
    if (la == expected) Scan();  // recognized => read ahead
    else Error( Token.names[expected] + " expected"  );
}
```

```
public static void Error (string msg) {
    Console.WriteLine("– line {0}, col {1}: {2}", laToken.line, laToken.col, msg);
    throw new Exception("Panic Mode");  // for a better solution see later
}
```

in class *Token*:

```
public static string[] names = {"?", "identifier", "number", ..., "+", "-", ...};
```

ordered by token codes

The names of the terminal symbols are declared as constants in class *Token*

```
public const int  NONE = 0,
                  IDENT = 1, NUMBER = 2, ...,
                  PLUS = 4, MINUS = 5, ... ;
```

# *How to Parse Nonterminal Symbols*

**Pattern**

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

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

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

**Initialization of the Z# parser**

```
public static void Parse () {
    Scan();                // initializes token, laToken and la
    Program();             // calls the parsing method of the start symbol
    Check(Token.EOF);      // at the end the input must be empty
}
```

# *How to Parse Sequences*

## Pattern

production: A = a B c.

parsing method:

```
static void A () {
    // la contains a terminal start symbol of A
    Check(a);
    B();
    Check(c);
    // la contains a follower of A
}
```

## Simulation

A = a B c.
B = b b.

| | *remaining input* |
|---|---|
| static void **A** () { | a b b c |
| Check(a); | b b c |
| B(); | c |
| Check(c); | |
| } | |
| static void B() { | b b c |
| Check(b); | b c |
| Check(b); | c |
| } | |

# *How to Parse Alternatives*

**Pattern** $\alpha \mid \beta \mid \gamma$ — $\alpha$, $\beta$, $\gamma$ are arbitrary EBNF expressions

**Parsing action**

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

**Example**

```
A = a B | B b.
B = c | d.
```

First(aB) = {a}
First(Bb) = First(B) = {c, d}

```
static void A () {
    if (la == a) {
        Check(a);
        B();
    } else if (la == c || la == d) {
        B();
        Check(b);
    } else Error ("invalid start of A");
}
```

# *How to Parse EBNF Options*

**Pattern** $[\ \alpha\ ]$ $\alpha$ is an arbitrary EBNF expression

**Parsing action**

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

**Example**

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

```
static void A () {
    if (la == a) {
        Check(a);
        Check(b);
    }
    Check(c);
}
```

Example: parse a b c
parse c

# *How to Parse EBNF Iterations*

**Pattern** { α } α is an arbitrary EBNF expression

**Parsing action**

```
while (la ∈ First(α)) { ... parse α ... }
```

**Example**

```
A = a { B } b.
B = c | d.
```

```
static void A () {
   Check(a);
   while (la == c || la == d) B();
   Check(b);
}
```

Example: parse a c d c b
parse a b

alternatively ...

```
static void A () {
   Check(a);
   while (la != b && la != Token.EOF) B();
   check(b);
}
```

without EOF: danger of an infinite loop, if *b* is missing in the input

# *How to Deal with Large First Sets*

**If the set has more than 4 elements: use class *BitArray***

example: First(A) = {a, b, c, d, e}
First(B) = {f, g, h, i, j}

***The First sets are initialized at the beginning of the program***

```
using System.Collections;
static BitArray firstA = new BitArray(Token.names.Length);
firstA[a] = true; firstA[b] = true; firstA[c] = true; firstA[d] = true; firstA[e] = true;
static BitArray firstB = new BitArray(Token.names.Length);
firstB[f] = true; firstB[g] = true; firstB[h] = true; firstB[i] = true; firstB[j] = true;
```

***Set test***

```
C = A | B.
```

```
static void C () {
   if (firstA[la]) A();
   else if (firstB[la]) B();
   else Error("invalid C");
}
```

# *How to Deal with Large First Sets*

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

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

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

**If the set is an interval: use an interval test**

```
if (a <= la && la <= c) ...
```

Token codes can often be chosen so that frequently checked sets form intervals

**Example**

First(A) = { a, c, d }
First(B) = { a, d }
First(C) = { b, e }

```
const int
   a = 0,
   d = 1,
   c = 2,
   b = 3,
   e = 4,
```

First(B)
First(A)
First(C)

# *Optimizations*

## Avoiding multiple checks

```
A = a | b.
```

```
static void A () {
   if (la == a) Scan();  // no Check(a);
   else if (la == b) Scan();
   else Error("invalid A");
}
```

```
A = { a | B d }.
B = b | c.
```

```
static void A () {
   while (la == a || la == b || la == c) {
      if (la == a) Scan();
      else { // no check any more
         B(); Check(d);
      } // no error case
   }
}
```

## More efficient scheme for parsing alternatives in an iteration

```
A = { a | B d }.
```

```
static void A () {
   for (;;) {
      if (la == a) Scan();
      else if (la == b || la == c) { B(); Check(d); }
      else break;
   }
}
```

# Optimizations

## Frequent iteration pattern

$\alpha$ { separator $\alpha$ }

```
for (;;) {
    ... parse α ...
    if (la == separator) Scan(); else break;
}
```

*Example*

ident { "," ident }

```
for (;;) {
    Check(ident);
    if (la == Token.COMMA) Scan(); else break;
}
```

input e.g.: a , b , c :

# *Computing Terminal Start Symbols Correctly*

## Grammar

```
A = B a.
B = { b } c
  | [ d ]
  | e.
```

terminal start symbols

- `{ b } c` → *b* and *c*
- `[ d ]` → *d* and *a* (!)
- `e.` → *e*

```
C = D e
  | f.
D = { d }.
```

- `D e` → *d* and *e* (*D* is deletable!)
- `f.` → *f*

## Parsing methods

```
static void A () {
   B(); Check(a);
}
```

```
static void B () {
   if (la == b || la == c) {
      while (la == b) Scan();
      Check(c);
   } else if (la == d || la == a) {
      if (la == d) Scan();
   } else if (la == e) {
      Scan();
   } else Error("invalid B");
}
```

```
static void C () {
   if (la == d || la == e) {
      D(); Check(e);
   } else if (la == f) {
      Scan();
   } else Error("invalid C");
}
```

```
static void D () {
   while (la == d) Scan();
}
```

# *Recursive Descent and the Syntax Tree*

**The syntax tree is only built implicitly**

- it is denoted by the methods that are currently active
- i.e. by the productions that are currently under examination

**Example** A = a B c.
B = d e.

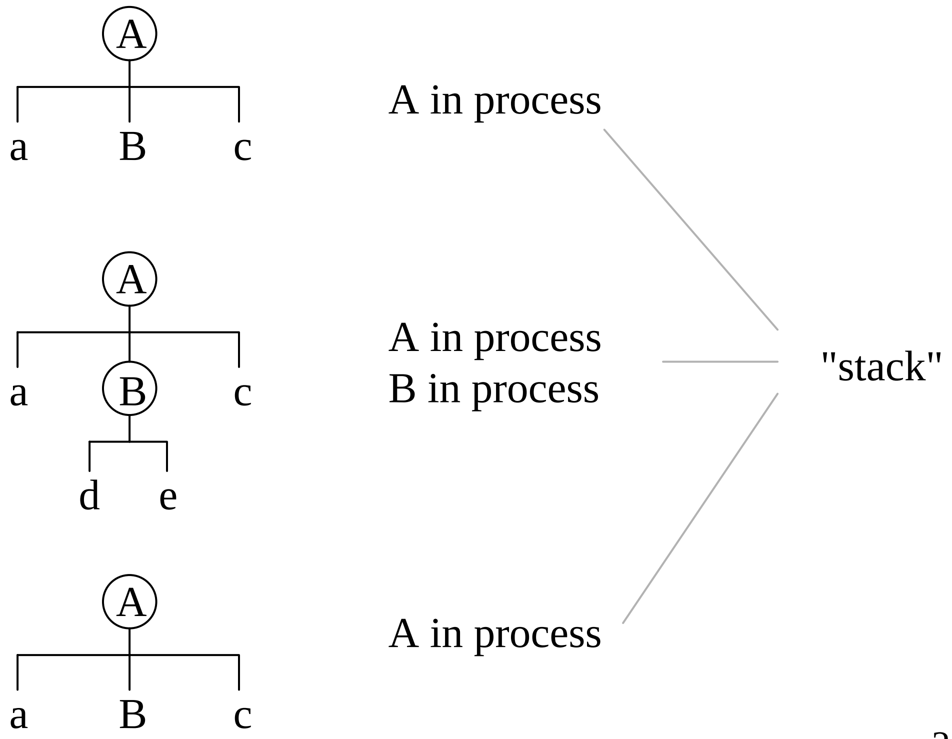
call *A()*

```
static void A () {
    Check(a); B(); Check(c);
}
```

recognize *a*
call *B()*

```
static void B () {
    Check(d); Check(e);
}
```

recognize *d* and *e*
return from *B()*



# 3. Parsing

# *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 alternative
   $\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$
   the following condition holds:
   $\text{First}(\alpha_i) \cap \text{First}(\alpha_j) = \{\}$ (for any *i* and *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**

A = [ α ] β. ⇔ A = α β | β. α and β are arbitrary EBNF expressions

**Rules**

| | |
|---|---|
| A = [ α ] β. | First(α) ∩ First(β) must be {} |
| A = { α } β. | First(α) ∩ First(β) must be {} |

| | |
|---|---|
| A = α [ β ]. | First(β) ∩ Follow(A) must be {} |
| A = α { β }. | First(β) ∩ Follow(A) must be {} |

| | |
|---|---|
| A = α \| . | First(α) ∩ Follow(A) 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*

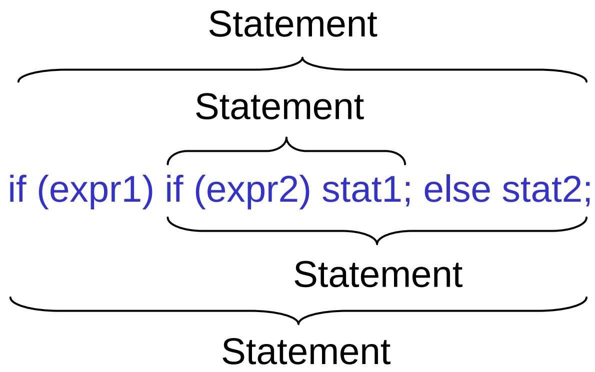
**If statement in C#**

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

**This is an LL(1) conflict!**

First("else" Statement) ∩ Follow(Statement) = {"else"}

**It is even an ambiguity which cannot be removed**




We can build 2 different syntax trees!

# *Can We Ignore LL(1) Conflicts?*

## An LL(1) conflict is only a warning

The parser selects the first matching alternative

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

← if the lookahead token is *a* the parser selects 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"

if (expr1) if (expr2) stat1; else stat2;

- `if (expr2) stat1; else stat2;` → Statement
- `if (expr1) if (expr2) stat1; else stat2;` → Statement

Luckily this is what we want here.

# *Other Requirements for a Grammar*

*(Preconditions for Parsing)*

## Completeness

For every NTS there must be a production

```
A = a B C .
B = b b .
```

error!
no production for *C*

## Derivability

Every NTS must be derivable (directly or indirectly) into a string of terminal symbols

```
A = a B | c .
B = b B .
```

error!
*B* cannot be derived into a string of terminal symbols

## Non-circularity

A NTS must not be derivable (directly or indirectly) into itself ($A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \ldots \Rightarrow A$)

```
A = a b | B .
B = b b | A .
```

error!
this grammar is circular because of $A \Rightarrow B \Rightarrow A$

# 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 "general anchors"
- Error handling with "special anchors"

# *Panic Mode*

**The parser gives up after the first error**

```
static void Error (string msg) {
    Console.WriteLine("-- line {0}, col {1}: {2}", laToken.line, laToken.col, msg);
    throw new Exception("Panic Mode - exiting after first error");

}
```

**Advantages**

- cheap
- sufficient for small command languages or for interpreters

**Disadvantages**

- not appropriate for production-quality compilers

# *Error Handling with "General Anchors"*

**Example**

expected input: a b c d ...
real input: a x y d ...

**Recovery** (synchronize the remaining input with the grammar)

1. **Find "anchor tokens" with which the parser can continue after the error.**

   What are the tokens which the parser can continue with in the above situation?
   *c* successor of *b* (which was expected at the error position)
   *d* successor of *b c*
   ...
   Anchors at this position are {*c*, *d*, ...}

2. **Skip input tokens until an achor is found.**
   *x* and *y* are skipped here, but *d* is an anchor; the parser can continue with it.

3. **Drive the parser to the position in the grammar from where it can continue.**

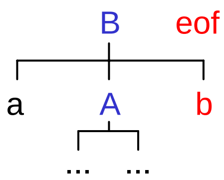
# *Computing Anchors*

**Every parsing method of a nonterminal *A* gets the current successors of *A* as parameters**

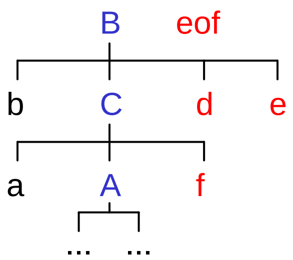
```
static void A (BitArray sux) {
   ...
}
```

*sux* ... successors of all NTS, that are currently in process

Depending on the current context $sux_A$ can denote different sets




$sux_A = \{b, eof\}$




$sux_A = \{f, d, e, eof\}$

*sux* always contains *eof* (the successor of the start symbol)

# *Handling Terminal Symbols*

*Grammar*

A = ... a $s_1$ $s_2$ ... $s_n$ .

$s_i \in$ TS $\cup$ NTS

*Parsing action*

check(a, $sux_A \cup First(s_1) \cup First(s_2) \cup ... \cup First(s_n)$);

$First(s_1) \cup ... \cup First(s_n)$: can be precomputed at compile time

$sux_A \cup ...$: must be computed at run time

```
static void Check (int expected, BitArray sux) {...}
```

**Example**

A = a b c.

```
static void A (BitArray sux) {
   Check(a, Add(sux, fs1));
   Check(b, Add(sux, fs2));
   Check(c, sux);
}
```

```
static BitArray fs1 = new BitArray();
fs1[b] = true; fs1[c] = true;
```

computed at the beginning of the program

```
static BitArray Add (BitArray a, BitArray b) {
   BitArray c = (BitArray) a.Clone();
   c[b] = true;
   return c;
}
```

# *Handling Nonterminal Symbols*

*Grammar*

A = ... B $s_1$ $s_2$ ... $s_n$ .

*Parsing action*

B($sux_A$ ∪ First($s_1$) ∪ First($s_2$) ∪ ... ∪ First($s_n$));

**Example**

A = a B c.
B = b b.

```
static void A (BitArray sux) {
   Check(a, Add(sux, fs3));     <-- fs3 = {b, c}
   B(Add(sux, fs4));            <-- fs4 = {c}
   Check(c, sux);
}
```

```
static void B (BitArray sux) {
   Check(b, Add(sux, fs5));     <-- fs5 = {b}
   Check(b, sux);
}
```

The parsing method for the start symbol *S* is called as *S(fs0);* where *fs0* = {*eof*}

# *Skipping Invalid Input Tokens*

**Errors are detected in *check()***

```
static void Check (int expected, BitArray sux) {
   if (la == expected) Scan();
   else Error(Token.names[expected] + " expected", sux);
}
```

**After printing an error message input tokens are skipped until an anchor occurs**

```
static void Error (string msg, BitArray sux) {
   Console.WriteLine("-- line {0}, col {1}: {2}", laToken.line, laToken.col, msg);
   errors++;
   while (!sux[la]) Scan();  // while (la ∉ sux) Scan();
   // la ∈ sux
}
```

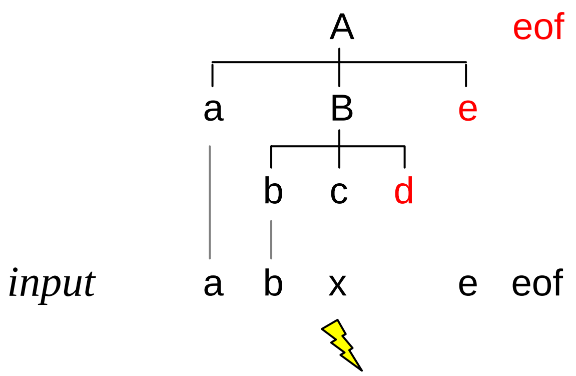
```
static int errors = 0;  // number of syntax errors detected
```

# *Synchronizing With the Grammar*

**Example**

```
A = a B e.
B = b c d.
```

```
static void A (BitArray sux) {
    Check(a, Add(sux, fs1));
    B(Add(sux, fs2));
    Check(e, sux);
}
static void B (BitArray sux) {
    Check(b, Add(sux, fs3));
    Check(c, Add(sux, fs4));
    Check(d, sux);
}
```




$sux_A = \{eof\}$

$sux_B = \{e, eof\}$

← the error is detected here; anchors = {*d*, *e*, *eof*}

1. *x* is skipped; *la* == *e* (∈ anchors)
2. parser continues: *Check(d, sux);*
3. detects an error again; anchors = {*e*, *eof*}
4. nothing is skipped, because *la* == *e* (∈ anchors)
5. parser returns from *B()* and does *Check(e, sux);*
6. recovery succeeded!

After the error the parser "jolts ahead" until it gets to a point in the grammar where the found anchor token is valid.

# *Suppressing Spurious Error Messages*

During error recovery the parser 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.

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

```
static void Scan () {
    ...
    errDist++;  // one more token recognized
}
```

```
public static void Error (string msg, BitArray sux) {
    if (errDist >= 3) {
        Console.WriteLine("-- line {0}, col {1}: {2}", laToken.line, laToken.col, msg);
        errors++;
    }
    while (!sux[la]) Scan();
    errDist = 0;  // counting is restarted
}
```

# *Handling Alternatives*

A = α | β | γ .

α, β, γ are arbitrary EBNF expressions

```
static void A (BitArray sux) {
   // the error check is already done here so that the parser can synchronize with
   // the starts of the alternatives in case of an error
   if (la ∉ (First(α) ∪ First(β) ∪ First(γ)))
      Error("invalid A", sux ∪ First(α) ∪ First(β) ∪ First(γ));
   // la matches one of the alternatives or is a legal successor of A
   if (la ∈ First(α)) ... parse α ...
   else if (la ∈ First(β)) ... parse β ...
   else ... parse γ ... // no error check here; any errors have already been reported
}
```

First(α) ∪ First(β) ∪ First(γ) can be precomputed at compile time
sux ∪ ... must be computed at run time

# *Handling EBNF Options and Iterations*

## Options

A = [ α ] β.

```
static void A (BitArray sux) {
   // error check already done here, so that the parser can
   // synchronize with the start of α in case of an error
   if (la ∉ (First(α) ∪ First(β)))
      Error("...", sux ∪ First(α) ∪ First(β));
   // la matches α or β or is a successor of A
   if (la ∈ First(α)) ... parse α ...;
   ... parse β ...
}
```

## Iterations

A = { α } β.

```
static void A (BitArray sux) {
   for (;;) {
      // the loop is entered even if la ∉ First(α)
      if (la ∈ First(α)) ... parse α ...;                // correct case 1
      else if (la ∈ First(β) ∪ sux) break;              // correct case 2
      else Error("...", sux ∪ First(α) ∪ First(β));     // error case
   }
   ... parse β ...
}
```

# *Example*

```
A = a B | b {c d}.
B = [b] d.
```

```
static void A (BitArray sux) {
   if (la != a && la != b)
      Error("invalid A", Add(sux, fs1));  // fs1 = {a, b}
   if (la == a) {
      Scan(); B(sux);
   } else if (la == b) {
      Scan();
      for (;;) {
         if (la == c) {
            Scan();
            Check(d, Add(sux, fs2));  // fs2 = {c}
         } else if (sux[la]) {
            break;
         } else {
            Error("c expected", Add(sux, fs2));
         }
      }
   }
}
```

```
static void B (BitArray sux) {
   if (la != b && la != d)
      Error("invalid B", Add(sux, fs3));  // fs3 = {b, d}
   if (la == b) Scan();
   Check(d, sux);
}
```

# *Assessment*

**Error handling with general anchors**

**Advantage**

\+ systematically applicable

**Disadvantages**

- slows down error-free parsing
- inflates the parser code
- complicated

# *Error Handling With Special Anchors*

**Error handling is only done at particularly "safe" positions**

i.e. at positions that start with keywords which do not occur at any other position in the grammar

For example

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

(anchor sets)

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

**Code that has to be inserted at the synchronization points**

```
...
if (la ∉ expectedSymbols) {          // expectedSymbols: anchor set at this synchronization point
    Error("..."); // no successor sets; no skipping of tokens in Error()
    while (la ∉ (expectedSymbols ∪ {eof})) Scan();   // {eof}: in order not to get into an endless loop
}
...
```

- No successor sets have to be passed to parsing methods
- Anchor sets can be computed at compile time
- After an error the parser "jolts ahead" to the next synchronization point

# *Example*

## Synchronization at the start of Statement

```
static void Statement () {
   if (!firstStat[la]) {
      Error("invalid start of statement");
      while (!firstStat[la] && la != Token.EOF) Scan();
   }
   if (la == Token.IF) { Scan();
      Check(Token.LPAR);
      Conditions();
      Check(Token.RPAR);
      Statement();
      if (la == Token.ELSE) { Scan(); Statement(); }
   } else if (la == Token.WHILE) {
      ...
}
```

```
static BitArray firstStat = new BitArray();
firstStat[Token.WHILE] = true;
firstStat[Token.IF] = true;
...
```

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

## No Synchronization in *Error()*

```
public static void Error (string msg) {
   if (errDist >= 3) {
      Console.WriteLine("-- line {0}, col {1}: {2}", laToken.line, laToken.col, msg);
      errors++;
   }
   errDist = 0;
}
```

heuristics with *errDist* can also be applied here (← `errDist = 0;`)

# *Example of a Recovery*

```
static void Statement () {
   if (!firstStat[la]) {
      Error("invalid start of statement");
      while (!firstStat[la] && la != Token.EOF) Scan();
   }
   if (la == Token.IF) { Scan();
      Check(Token.LPAR);
      Condition();
      Check(Token.RPAR);
      Statement();
      if (la == Token.ELSE) { Scan(); Statement(); }
   ...
}
```

```
static void Check (int expected) {
   if (la == expected) Scan();
   else Error(...);
}
```

```
public static void Error (string msg) {
   if (errDist >= 3) {
      Console.WriteLine(...);
      errors++;
   }
   errDist = 0;
}
```

erroneous input: if a > b then max = a;

| *la* | *action* | |
|---|---|---|
| IF | Scan(); | IF ∈ *firstStatement* ⇒ ok |
| a | Check(LPAR); | error message: ( expected |
| | Condition(); | recognizes a > b |
| THEN | check(RPAR); | error message: ) expected |
| | Statement(); | THEN does not match ⇒ error, but no error message |
| | | THEN is skipped; synchronization with *ident* (if in *firstStat*) |
| max | | |

# *Synchronization at the Start of an Iteration*

**For example**

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

**Standard pattern in this case**

```
static void Block () {
   Check(Token.LBRACE);
   while (firstStat[la])
      Statement();
   Check(Token.RBRACE);
}
```

Problem: If the next token does not match *Statement* the loop is not executed.
Synchronization point in *Statement* is never reached.

# *Synchronization at the Start of an Iteration*

**For example**

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

**Better to synchronize at the beginning of the iteration**

```
static void Block() {
   Check(Token.LBRACE);
   for (;;) {
      if (la ∈ First(Statement)) Statement();       // correct case 1
      else if (la ∈ {rbrace, eof}) break;            // correct case 2
      else {                                         // error case
         Error("invalid start of Statement");
         do Scan(); while (la ∈ (First(Statement) ∪ {rbrace, eof}));
      }
   }
   Check(Token.RBRACE);
}
```

**No synchronization in *Statement()* any more**

```
static void Statement () {
   if (la == Token.IF) { Scan(); ...
}
```

# *Assessment*

**Error handling with special anchors**

**Advantages**

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

**Disadvantage**

- needs experience and "tuning"