no structure → "**Words**" → just "strings"

non-terminal

OR

terminal

digit = "0" | ... | "9" .

dot!

informal

production

recursion

integer = digit | integer digit .

⇝ integer is any sequence of digit with at least one digit!

integer = digit { digit } .

repeat zero or more times

⇝ we do not need recursion!

letter = "A" | ... | "Z" | "a" | ... | "z" .

identifier = letter { letter | digit } .

⇝ identifier is any sequence of letter and digit starting with a letter

why?

⇝ Scanner knows if it is dealing with an identifier or an integer at first symbol!

Sentences

start symbol

optional

grouping

leftmost derivation

```
expression = [ "-" ] term { ( "+" | "-" ) term } .
term = factor { ( "*" | "/" ) factor } .
factor = identifier | integer | "(" expression ")" .
```

produce

stack (push)

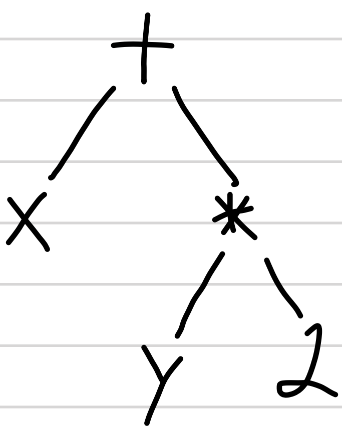
lookahead = 1

```
x+y*2 (I)     # expression
x+y*2 (P)     # ["-"] term { ( "+" | "-" ) term }
x+y*2 (M)     # term { ( "+" | "-" ) term }
x+y*2 (P)     # factor { ( "*" | "/" ) factor } ...
x+y*2 (P)     # identifier | integer | "(" expression ")" ... ...
+y*2 (M)      # { ( "*" | "/" ) factor } ...
+y*2 (M)      # { ( "+" | "-" ) term }
y*2 (M)       # term { ( "+" | "-" ) term }
...           ...
*2 (M)        # { ( "*" | "/" ) factor } ...
2 (M)         # factor { ( "*" | "/" ) factor } ...
...           ...
(M)           #
```

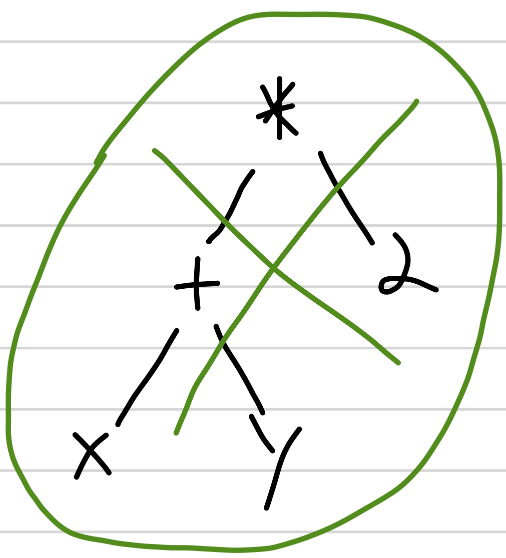
top-down parsing

match

syntax tree



not



→ each sentence must have exactly one syntax tree

• structure determines semantics!

languages

a language is defined by a grammar which consists of:

1. a set of terminal symbols written in quotes which cannot be substituted (also called vocabulary)
2. a set of non-terminal symbols which can be substituted
3. a set of syntactic equations (also called productions)
4. a start symbol (non-terminal)

the language is the set of sequences (sentences) of terminal symbols which, starting with the start symbol, can be generated by repeated application of syntactic equations (we only consider so-called leftmost derivations here)

- what is the fundamental difference between identifiers and and expressions?

⇝ the language of identifiers is <u>regular</u>!

a language is regular if it can be defined by a single production

⇝ the language of expressions is <u>not</u> regular (but <u>context-free</u>)! (productions apply in <u>any</u> context)

→ scanner scans identifiers, parser parses expressions, why?

⇝ makes parsing independent of (regular) vocabulary

⇝ focus on structure!

"recursive-descent" → # Parser Example

expression = ["-"] term { ("+" | "-") term } .
term = factor { ("*" | "/") factor } .
factor = identifier | integer | "(" expression ")" .

```
expression() {
  if (symbol == MINUS)
    getSymbol();
  term();
  while (symbol == PLUS || symbol == MINUS) {
    getSymbol();
    term();
  }
}
```

optimized

```
if (symbol == MINUS)
  getSymbol();
else
  error();
```

similarly optimized

```
term() {
  factor();
  while (symbol == TIMES || symbol == DIV) {
    getSymbol();
    factor();
  }
}
```

optimized

```
if (symbol == PLUS)
  getSymbol();
else if (symbol == MINUS)
  getSymbol();
else
  error();
```

"+" or "-" expected!

```
factor() {
  if (symbol == IDENTIFIER)
    getSymbol();
  else if (symbol == INTEGER)
    getSymbol();
  else if (symbol == LPAREN) {
    getSymbol();
    expression();
    if (symbol == RPAREN)
      getSymbol();
    else
      error();
  } else
    error();
}
```

")" expected!

ID..., INT..., "(" expected!

implements

| ← OR

How to Construct a Parser I

~> we need the syntax of grammars (defined using a grammar) :)

```
syntax = {production} .
production = nonterminal "=" expression "." .
nonterminal = identifier .
expression = term { "|" term } .
term = factor { factor } .
factor = terminal | nonterminal |
"[" expression "]" | "{" expression "}" | "(" expression ")" .
terminal = """ character { character } """ .
```

"printable character"

~> $\underbrace{\text{Extended}}_{\text{[] \{\} Wirth 1977}}$ $\underbrace{\text{Backus-Naur Form}}_{\text{recursion only (plus empty sequence } \emptyset\text{) Backus, Naur 1960}}$ (EBNF)

```
Parser<nonterminal = expression>:

nonterminal() {
  Parser<expression>
}
```

implement a procedure for each nonterminal!

```
Parser<terminal>:

if (symbol == Token<terminal>)
  getSymbol();
else
  error();
```

expected

pure structure

```
Parser<( expression )>:

Parser<expression>
```

How to Construct a Parser II

```
Parser<term_0 | ... | term_n>:

if (isIn(symbol, First<term_0>)) {
  Parser<term_0>
...
} else if (isIn(symbol, First<term_n>)) {
  Parser<term_n>
} else
  error();
```

the set of symbols with which $term_0$ may start (first set)!

$\bigcup_{0 \le i \le n}$ First<term_i> expected!

→ we require that

$\forall 0 \le i,j \le n.\ i \neq j \Rightarrow First\langle term_i \rangle \cap First\langle term_j \rangle = \{\}$

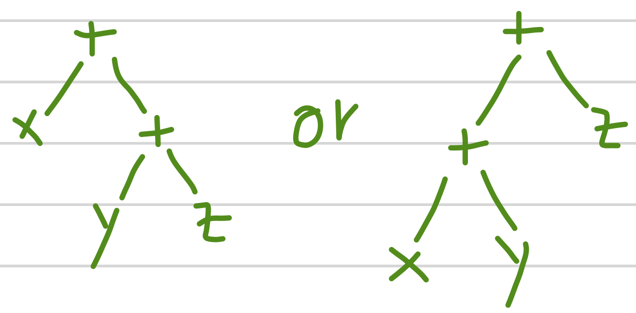
→ avoids ambiguity!

implement first set check efficiently through smart token encoding

e.g.: letter: symbol ≥ "A" & symbol ≤ "z"

expression = identifier | expression "+" expression .

x + y + z :



a grammar is called ambiguous if there is a sentence that can be generated in different ways by the grammar

a language is inherently ambiguous if it can only be generated by ambiguous grammars

→ no problem here because of associativity of +
(but what if we replace + by −?)
→ requires grammar engineering (later)

How to Construct a Parser III

Parser<factor_0 ... factor_n>:

Parser<factor_0>
...
Parser<factor_n> → we require that

$\forall 0 \le i < n.\ \{\} \in First\langle factor_i \rangle \Rightarrow First\langle factor_i \rangle \cap First\langle factor_{i+1} \rangle = \{\}$

Parser<[expression]>:

```
if (isIn(symbol, First<expression>)) {
  Parser<expression>
}
```

→ we require that

$First\langle expression \rangle \cap \underbrace{Follow\langle expression \rangle}_{\text{the set of symbols that may follow the expression (follow set)!}} = \{\}$

same here

Parser<{ expression }>:

```
while (isIn(symbol, First<expression>)) {
  Parser<expression>
}
```

Grammar Engineering

1. remove ambiguity through precedence
2. remove left recursion
3. left-factor

expression
term
factor
⋮

$$A = A\alpha \mid \beta.$$

subexpressions

$$A = \beta\{\alpha\}.$$

$$A = \alpha\beta.$$
$$A = \alpha\gamma.$$

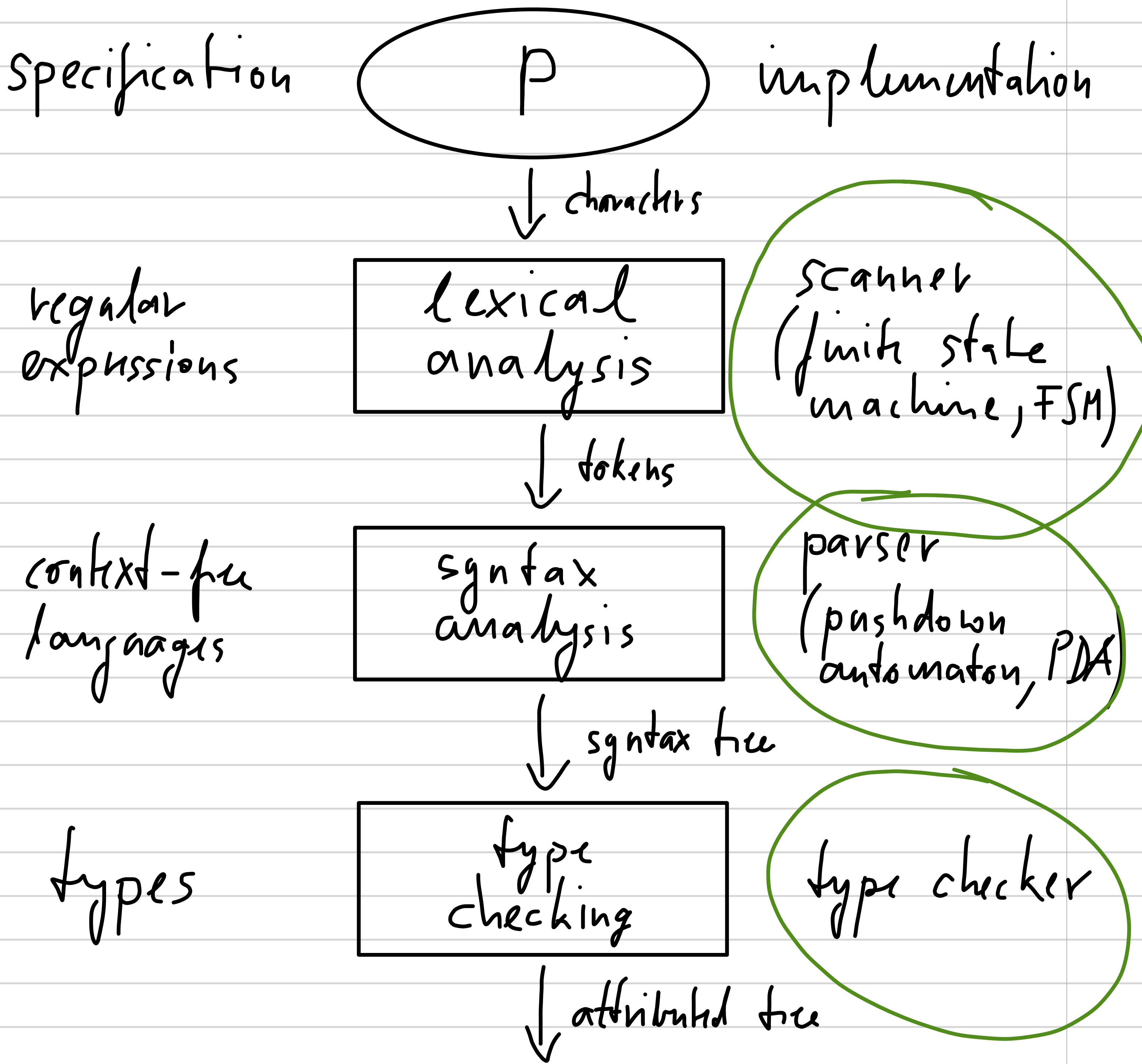
or $A = \alpha\beta \mid \alpha\gamma.$

$$A = \alpha B.$$
$$B = \beta \mid \gamma.$$

new nonterminal

such that $\mathrm{First}\langle\beta\rangle \cap \mathrm{First}\langle\gamma\rangle = \{\}$
(introduce new keywords if necessary)

FSM and PDA



- where is the FSM in the scanner?
  - → states are implicit in code location!
  - → code implements transitions!
- where is the PDA in the (recursive-descent) parser?
  - → same as scanner, and the stack?
  - → the stack is implemented by the procedure call stack!
- where is the type checker?
  - → will be in the arguments of the parser procedures!

Bottom-up Parsing

"start" symbol

expression = ["-"] term { ("+" | "-") term } .
term = factor { ("*" | "/") factor } .
factor = identifier | integer | "(" expression ")" .

init

shift ← stack (push)

reduce

rightmost derivation

lookahead = 1

(I) # x+y*2
(S) identifier # +y*2
(R) factor # +y*2
(R) term # +y*2
(S) term + # y*2
(S) term + identifier # *2
(R) term + factor # *2
(S) term + factor * # 2
(S) term + factor * integer #
(R) term + factor * factor #
(R) term + term #
(R) expression #

read left to right (input)

- bottom-up parsing: LR(k) ← lookahead

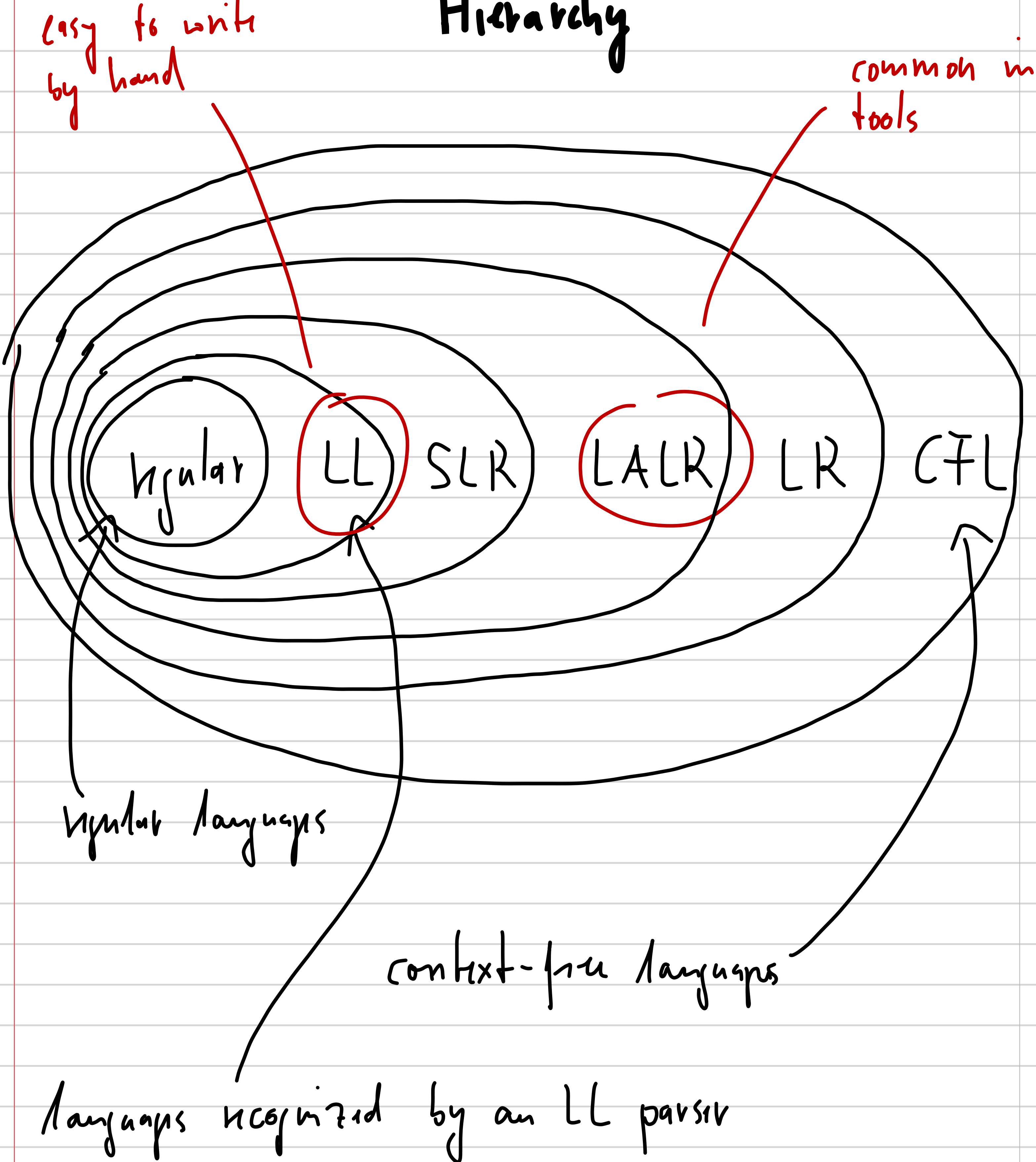
reduce at right end

produce at left end (of rules)

- top-down parsing: LL(k)

- $\underbrace{\text{simple LR (SLR), lookahead LR (LALR)}}_{\text{based on parsing tables generated by parser tools}}$ / represent FSMs

Hierarchy


easy to write by hand
common in tools
regular
LL
SLR
LALR
LR
CFL
regular languages
context-free languages
languages recognized by an LL parser

Attributed Grammars and Semantics

left version only for demo

expression(v0) = term(v1) .	v0 = v1
expression(v0) = expression(v1) "+" term(v2) .	v0 = v1 + v2
expression(v0) = expression(v1) "-" term(v2) .	v0 = v1 - v2
term(v0) = factor(v1) .	v0 = v1
term(v0) = factor(v1) "*" factor(v2) .	v0 = v1 * v2
term(v0) = factor(v1) "/" factor(v2) .	v0 = v1 / v2
factor(v0) = integer(v1) .	v0 = v1
factor(v0) = "(" expression(v1) ")" .	v0 = v1

syntactic   semantic

```
expression() {
 term();
 while (symbol == PLUS || symbol == MINUS) {
  getSymbol();
  term();
 }
}
```

attribute could also be expressed by call-by-reference parameter

attribute in return parameter

```
int expression() {
 int value;
 value = term();
 while (symbol == PLUS || symbol == MINUS)
  if (symbol == PLUS) {
   getSymbol();
   value = value + term();
  } else {
   getSymbol();
   value = value - term();
  }
 return value;
}
```

~> calculator for integer expressions!

Attributed Grammars and Types

```
expression(T0) = term(T1) .                            T0=T1
expression(T0) = expression(T1) "+" term(T2) .         T0=T1 T1==T2
expression(T0) = expression(T1) "-" term(T2) .         T0=T1 T1==T2
term(T0) = factor(T1) .                                T0=T1
term(T0) = factor(T1) "*" factor(T2) .                 T0=T1 T1==T2
term(T0) = factor(T1) "/" factor(T2) .                 T0=T1 T1==T2
factor(T0) = integer(T1) .                             T0=T1
factor(T0) = "(" expression(T1) ")" .                  T0=T1
```

type

constraint!

```
expression() {
  term();
  while (symbol == PLUS || symbol == MINUS) {
    getSymbol();
    term();
  }
}
```

to be defined

efficient!

```
type_t expression() {
  type_t type;
  type = term();
  while (symbol == PLUS || symbol == MINUS)
    if (symbol == PLUS) {
      getSymbol();
      type = resolveType(PLUS, type, term());
    } else {
      getSymbol();
      type = resolveType(MINUS, type, term());
    }
  return type;
}
```

attribute in return parameter

implements type system

- single rule for multiple types
- values vs. types: actual type unknown at compile time
- static typing approximates semantics without executing code!
- static typing single most successful compile-time debugging tool

Attributed Grammars and Code

expression = term .
expression = expression "+" term . emit(+)
expression = expression "-" term . emit(-)
term = factor .
term = factor "*" factor . emit(*)
term = factor "/" factor . emit(/)
factor = integer(value) . emit(value)
factor = "(" expression ")" .

target machine operator

from scanner

```
expression() {
 term();
 while (symbol == PLUS || symbol == MINUS) {
  getSymbol();
  term();
 }
}
```

infix to postfix!

```
expression() {
 term();
 while (symbol == PLUS || symbol == MINUS)
  if (symbol == PLUS) {
   getSymbol();
   term();
   emit(+)
  } else {
   getSymbol();
   term();
   emit(-);
  }
}
```

implicit attributes here: code!

→ code generation as soon as first sentential structure has been recognized: single-pass compiler!