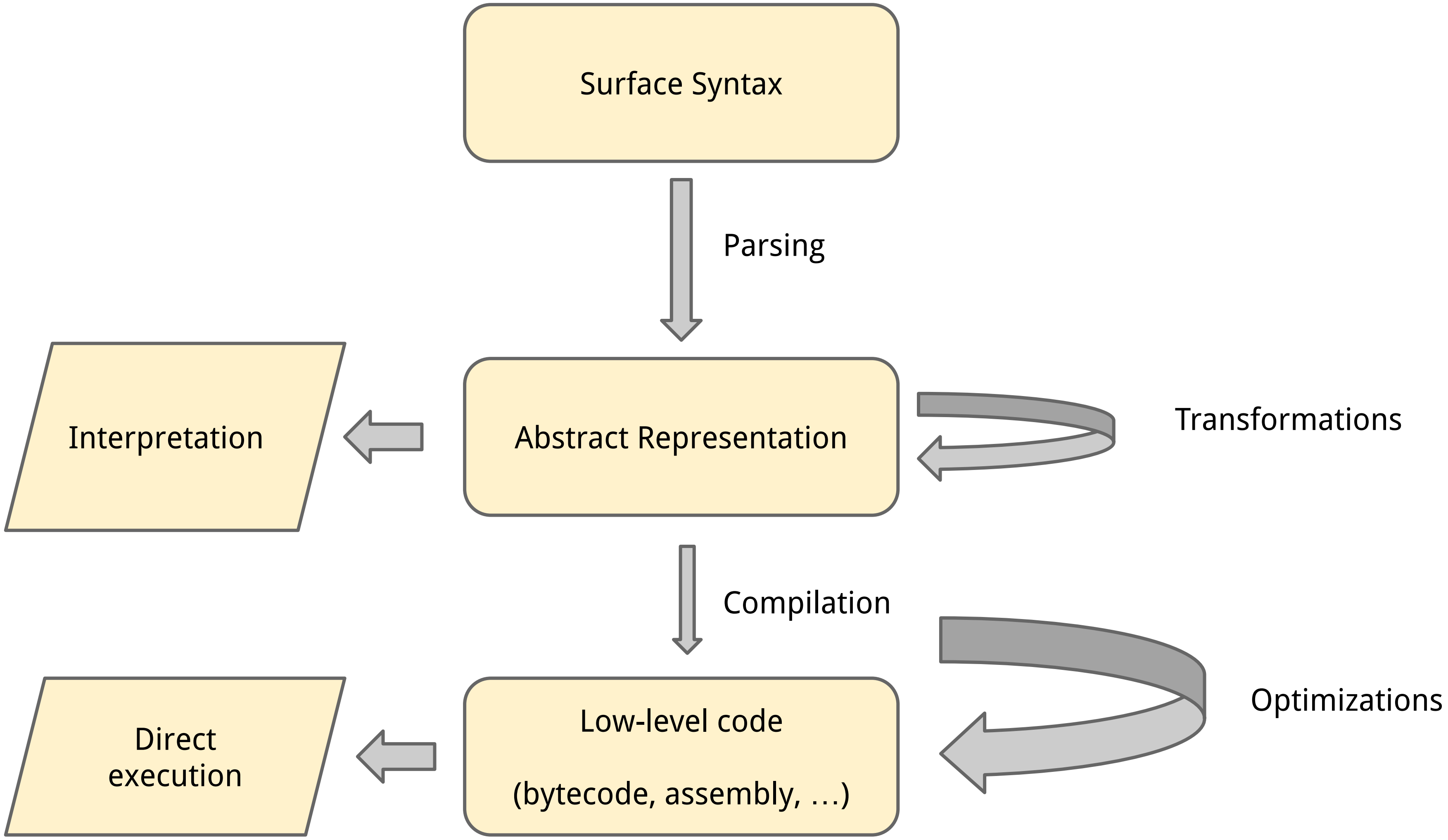
Surface Syntax and Parsing

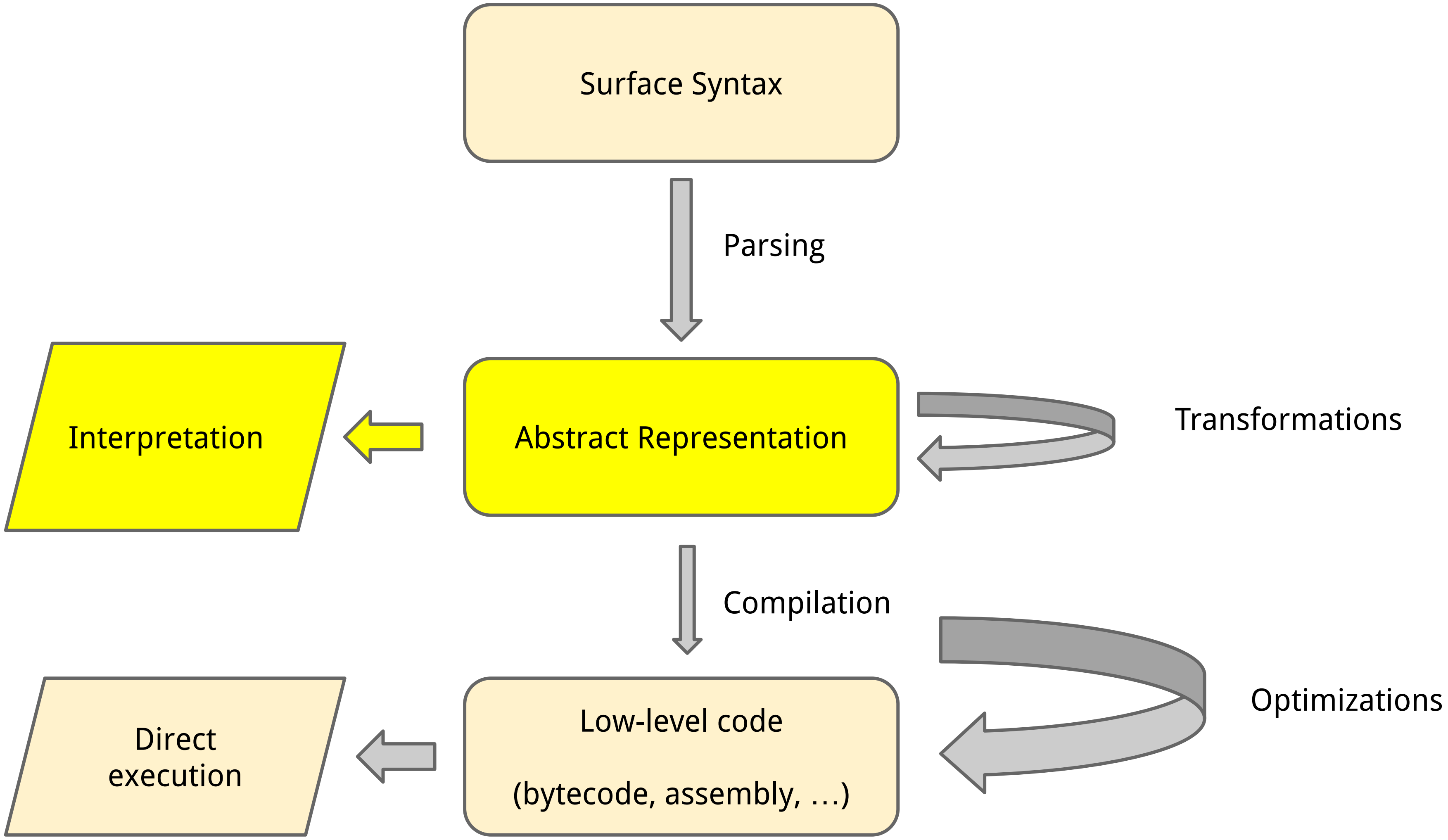
February 13, 2020

Riccardo Pucella

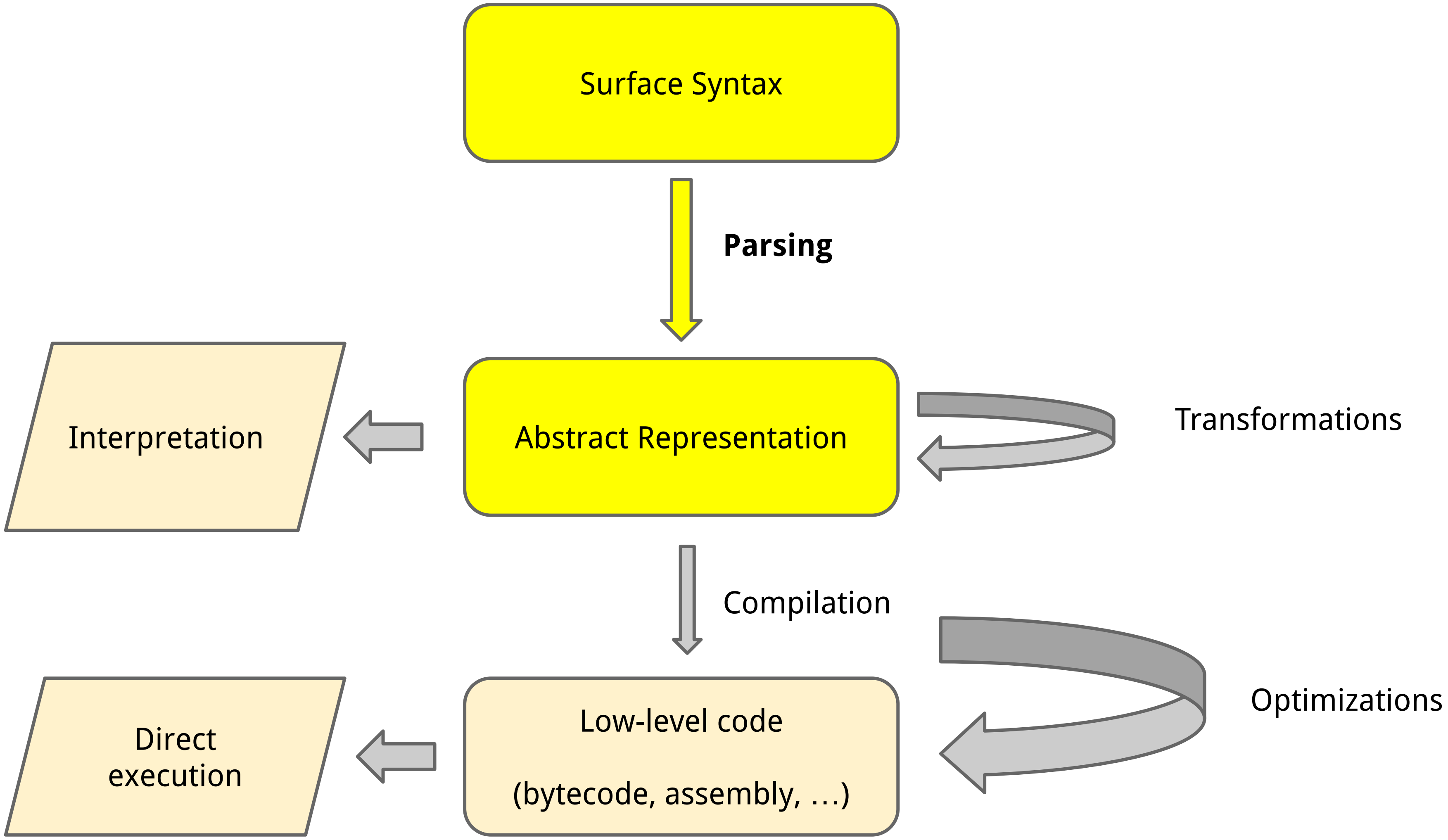
The structure of language execution



The structure of language execution



The structure of language execution



Why surface syntax?

Abstract representation:

good for computers

Surface syntax:

good for ~~humans~~ programmers

Generating abstract representation

```
let (x = 10 + 20)
  x * x
```

$$\begin{tikzcd}\text{Surface syntax} \arrow[d] \\ \text{Abstract representation}\end{tikzcd}$$

Generating abstract representation

```
let (x = 10 + 20)
  x * x
```

$$\Big\downarrow$$

```
ELet[“x”,EPlus[EInteger[10],
               EInteger[20]],
     ETimes [EId[“x”],
             EId[“x”]]]
```

Surface syntax

$$\Big\downarrow$$

Abstract representation

Sources of surface syntax

Where does surface syntax come from?

- files, input from interactive shells, ...
- abstraction: sequences of characters

Distinguishing two phases

Tokenization (aka lexical analysis)

sequence of characters $\rightarrow$ sequence of tokens

Parsing

sequence of tokens $\rightarrow$ abstract representation

Same thing happens in natural languages

- phonemes (units of elocution) merged into
  words (units of meaning) merged into sentences

Example

```
let (x = 10 + 20)
  x * x
```

Example

```
let (x = 10 + 20)
  x * x
```

$\downarrow$

```
l e t ␣ ( x ␣ = ␣ 1 0 ␣ + ␣ 2 0 ) ◆ ␣ x ␣ * ␣ x
```

Example

```
let (x = 10 + 20)
  x * x
```

$\downarrow$

```
l e t ␣ ( x ␣ = ␣ 1 0 ␣ + ␣ 2 0 ) ◆  ␣ x ␣ * ␣ x
```

$\downarrow$

```
KW(let) LP ID(x) EQUAL INT(10) PLUS INT(20) RP ID(x)
   TIMES ID(x)
```

Example

```
let (x = 10 + 20)
  x * x
```

$\downarrow$

```
l e t ␣ ( x ␣ = ␣ 1 0 ␣ + ␣ 2 0 ) ◆  ␣ x ␣ * ␣ x
```

$\downarrow$

```
KW(let) LP ID(x) EQUAL INT(10) PLUS INT(20) RP ID(x)
   TIMES ID(x)
```

$\downarrow$

```
ELet[“x”,EPlus[EInteger[10],
               EInteger[20]],
     ETimes [EId[“x”],
             EId[“x”]]]
```

Example

```
let (x = 10 + 20)
  x * x
```

```
l e t ␣ ( x ␣ = ␣ 1 0 ␣ + ␣ 2 0 ) ◆  ␣ x ␣ * ␣ x
```

```
KW(let) LP ID(x) EQUAL INT(10) PLUS INT(20) RP ID(x)
   TIMES ID(x)
```

```
ELet[“x”,EPlus[EInteg
               EInteg
     ETimes [EId[“x”]
            EId[“x”]
```

Many choices for the kind of tokens to use — practical trade-offs

Tokens

Unit of meaning

- sentences are made up of words
- programs are made up of tokens

Typical tokens:

- integers, floating point numbers, identifiers
- operation symbols + - * =
- punctuation ( ) , .

characters → token : local decision

Tokenization

Lexer:

character sequence → token sequence

Description of tokens: regular expressions

integer	`/[0-9]+/`
string	`/\“.*\”/`
identifier	`/[a-zA-Z][a-zA-Z0-9]*/`
keyword	`/let/` (e.g.)

Compact representation for families of strings

Efficient ways to check if a string is in the family

Lexer:

charac

Description of tokens: regular expressions

integer	`/[0-9]+/`
string	`/\".*\"/`
identifier	`/[a-zA-Z][a-zA-Z0-9]*/`
keyword	`/let/` (e.g.)

A naive lexer

```
While characters remain:
   For each possible token:
      Does token’s regexp match a prefix
      of the characters?
         Yes → output token
               tokenize remaining characters
```

- Works reasonably well for small programs
- Relies on regexp matching

A more clever lexer algorithm

- Matching a regular expression can be done with a deterministic finite automaton (DFA)

- Lexer algorithm:
  - Compile all token regexps into single large DFA
  - Tag final states with token recognized
  - Run the DFA with character sequence
  - When you hit a final state:
    - output token
    - restart DFA with remaining characters

- Usually implemented via a tool (`lex` family)

Generating abstract representation

```
let x = 10 + 20
  x * x
```

$$\begin{tikzcd}\text{Surface syntax} \arrow[d, "tokenization"] \\ \text{Tokens}\end{tikzcd}$$

```
ELet["x",EPlus[EInteger[10],
               EInteger[20]],
     ETimes [EId["x"],
             EId["x"]]]
```

Generating abstract representation

```
let x = 10 + 20
  x * x
```

```
ELet[“x”,EPlus[EInteger[10],
               EInteger[20]],
     ETimes [EId[“x”],
             EId[“x”]]]
```

$$\begin{tikzcd}
\text{Surface syntax} \arrow[d, "\textit{tokenization}"] \\
\text{Tokens} \arrow[d, "\textit{parsing}"] \\
\text{Abstract representation}
\end{tikzcd}$$

Parsing

- Identify valid token sequences
  - E.g. valid: `KW(let) LP ID(x) EQUAL INT[10] RP ID(x)`
  - E.g. not: `KW(let) LP ID(x) ID(y) EQUAL INT(10) RP ...`
- Map valid token sequences to elements of the internal representation
  - E.g. `ELet [...]`
- Anything that does that is a parser
- How do we describe valid token sequences?

Parsing

This is a HUGE field

A lot of work in programming languages, linguistics, natural language processing, and computational theory

This is merely to give you a taste

- Anything that does that is a parser

- How do we describe valid token sequences?

Grammars

A grammar is a description of valid sequences of tokens expressed as *production rules*

A production rule expands variables (known as nonterminals) into tokens and other variables

Think of English:

- A *sentence* is a *noun phrase* followed by a *verb phrase*
- A *noun phrase* is …

Example: S-expressions

```
atomic ::= integer
           identifier
           true
           false

expr ::= atomic
         ( + expr expr )
         ( * expr expr )
         ( if expr expr expr )
         ( let ( ( identifier expr ) ) expr )
```

Examples:

```
(+ 3 5)
(* (+ 3 5) (+ x 2))
(let ((x (+ 10 20))) (* x x))
```

Example: S-expressions

atomic ::= ***integer***
　　　　identifier
　　　　true
　　　　false

All **bolded terms** are tokens

expr ::= *atomic*
　　　(**+** *expr* *expr* **)**
　　　(***** *expr* *expr* **)**
　　　(**if** *expr* *expr* *expr* **)**
　　　(**let** **(** **(** ***identifier*** *expr* **)** **)** *expr* **)**

Examples:

```
(+ 3 5)
(* (+ 3 5) (+ x 2))
(let ((x (+ 10 20))) (* x x))
```

Creating parse results

```
atomic ::= integer(i)
           identifier(s)
           true
           false

expr ::= atomic
         ( + expr      expr      )
         ( * expr      expr      )
         ( if expr      expr      expr      )
         ( let ( ( identifier      expr     ) ) expr     )
```

Creating parse results

```
atomic ::= integer(i)          →   EInteger(i)
           identifier(s)
           true
           false

expr ::= atomic
         ( + expr     expr     )
         ( * expr     expr     )
         ( if expr     expr     expr     )
         ( let ( ( identifier     expr     ) ) expr     )
```

Creating parse results

```
atomic ::= integer(i)         ⟶   EInteger(i)
           identifier(s)      ⟶   EId(s)
           true               ⟶   EBoolean(true)
           false              ⟶   EBoolean(false)

expr ::= atomic
         ( + expr      expr      )
         ( * expr      expr      )
         ( if expr      expr      expr      )
         ( let ( ( identifier      expr      ) ) expr      )
```

Creating parse results

```
atomic ::= integer(i)                    →   EInteger(i)
           identifier(s)                 →   EId(s)
           true                          →   EBoolean(true)
           false                         →   EBoolean(false)

expr ::= atomic(r)                 →   r
         ( + expr(e1) expr(e2) )       →   EPlus(e1,e2)
         ( * expr(e1) expr(e2) )       →   ETimes(e1,e2)
         ( if expr(e1) expr(e2) expr(e3) )  →  EIf(e1,e2,e3)
         ( let ( ( identifier(s)  expr(e) ) ) expr(b) )

                                         →   ELet(s,e,b)
```

These sort of grammars are often called *attribute grammars*

Two approaches to parsing

TOP-DOWN	BOTTOM-UP
recursive descent	table-based
coded by hand	generated by tools
flexible but slow	fast
good for simple grammars	production systems

Recursive-descent parsers

For every nonterminal NT:

- define a function *parse_NT* that can match a sequence of tokens via any of the rules for NT

Predictive parsers are a simple class of recursive-descent parsers

- Applicable when *k* tokens uniquely identify which rule applies for each nonterminal

Predictive parser for S-expressions

```
atomic ::= integer
           identifier
           true
          false

expr ::= atomic
         ( + expr expr )
         ( * expr expr )
         ( if expr expr expr )
         ( let ( ( identifier expr ) ) expr )
```

Predictive parser for S-expressions

```
parse_atomic (tokens) =
   if tokens[0] is an integer token
     return (EInteger(tokens[0]),tokens[1:])
   if tokens[0] is an identifier token
     return (EId(tokens[0]),tokens[1:])
   if tokens[0] is token "true"
     return (EBoolean(True),tokens[1:])
   if tokens[0] is token "false"
     return (EBoolean(False),tokens[1:])
    fail
```

(Each function `parse_`*`NT`* returns a pair of the abstract representation of the parsed tokens, and the rest of the tokens not yet parsed)

Predictive parser for S-expressions

```
parse_expr (tokens) =
   if tokens[0] is token "("
     if tokens[1] is token "+"
       (e1,rest) = parse_expr(tokens[2:])
       (e2,rest) = parse_expr(rest)
       if rest[0] is token ")"
         return (EPlus(e1,e2),rest[1:])
       else fail
     if tokens[1] is token "*" … (similar)
     if tokens[1] is token "if" … (similar)
     if tokens[1] is token "let" … (similar)
     fail
   else
     return parse_atomic(tokens)
```

Parsing with backtracking

Some grammars cannot be parsed with a predictive parser

General recursive descent parsers:

1. Attempt to parse
2. if it succeeds, done
3. if it fails, backtrack and try a different rule — that's why it can be slow

Parser combinator libraries

Parser combinators

Idea:

- create small parsers
- combine parsers into more complex parsers

Example:

pyparsing library in Python

scala.util.parsing.combinator in Scala

Note: most parser combinator libraries do not distinguish between tokenization and parsing

What about a more natural syntax?

expr ::= ***integer***
identifier
(*expr* **)**
let (***identifier*** **=** *expr* **)** *expr*
if *expr* **then** *expr* **else** *expr*
expr **+** *expr*
expr ***** *expr*

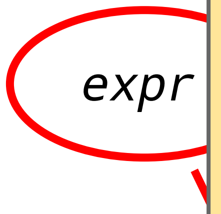
What about a more natural syntax?

```
expr ::=  integer
          identifier
          ( expr )
          let ( identifier = expr ) expr
          if expr then expr else expr
          expr + expr
          expr * expr
```

Grammar with *left recursion*

Bad for recursive-descent parsers — WHY?

What a



Need to eliminate left recursion by the rewriting grammar:

```
expr ::= integer
         expr + expr
         expr * expr
      ↓

expr ::= integer
         integer expr_rest

expr_rest ::= + expr
              * expr
```

Grammar with *left recursion*

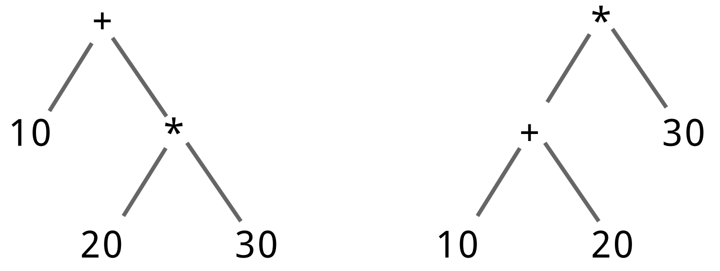
Bad for recursive-descent parsers — WHY?

Ambiguities

Ambiguities are another problem with the grammar…

A grammar is ambiguous if some sequences of tokens can parse in more than one way

10 + 20 * 30



To solve: impose operator precedence

Ambiguities

Ambig
gram

A gra   of
token

10 +

Another classic ambiguity

```
expr ::= ...
         if expr then expr else expr
         if expr then expr
```

Consider `if a then if b then c else d`

`if a then (if b then c) else d` ?

`if a then (if b then c else d)` ?

Ambiguities

Ambig
gr

A of
t

Ambiguities are *nasty*

There is no generic way of dealing with them

You have to understand your grammar well, and know which of the possible parses is the one you want

For a recursive-descent parser: you want to modify the grammar so that the parse you want is the first one found