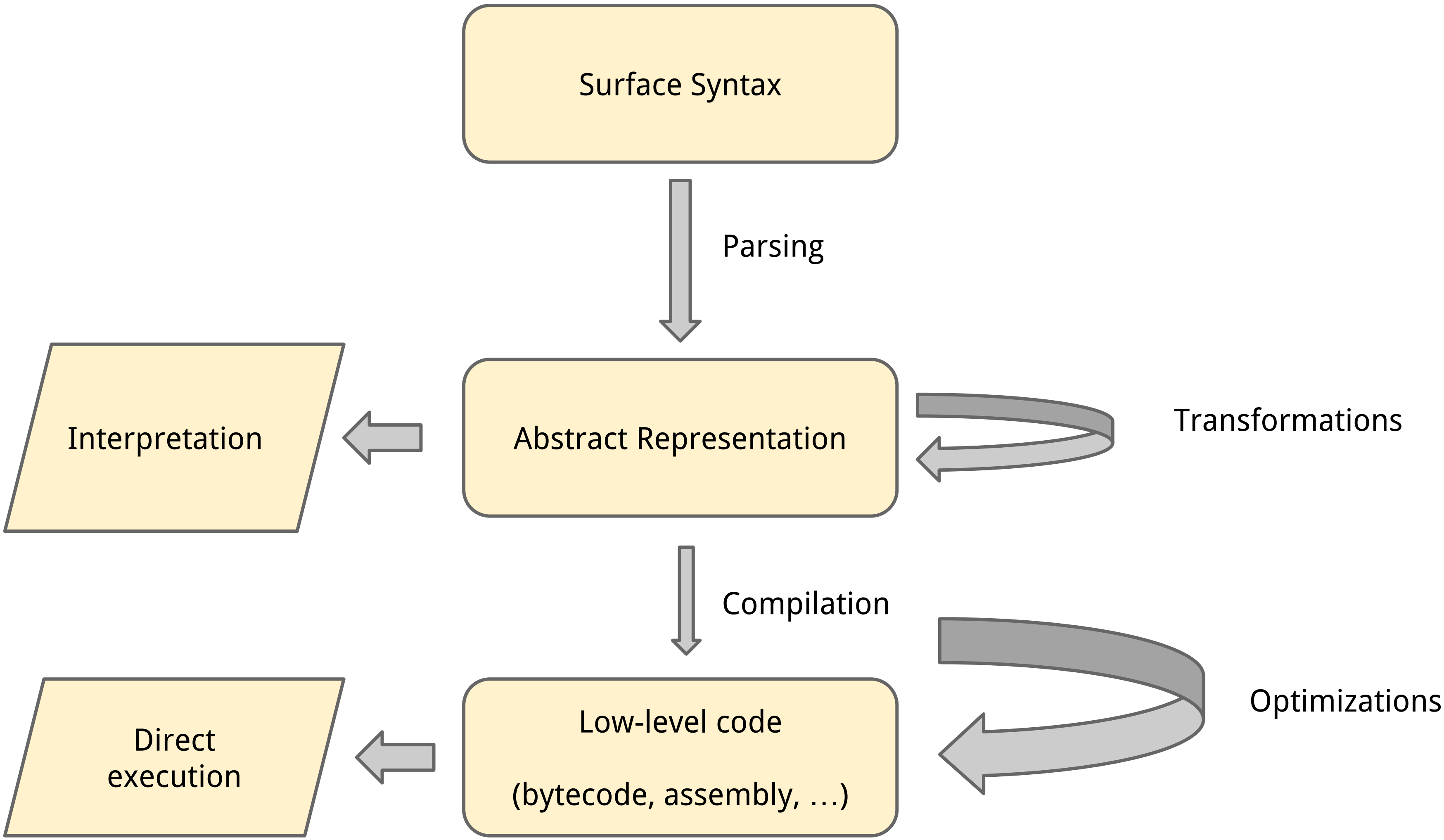
Surface Syntax and Parsing

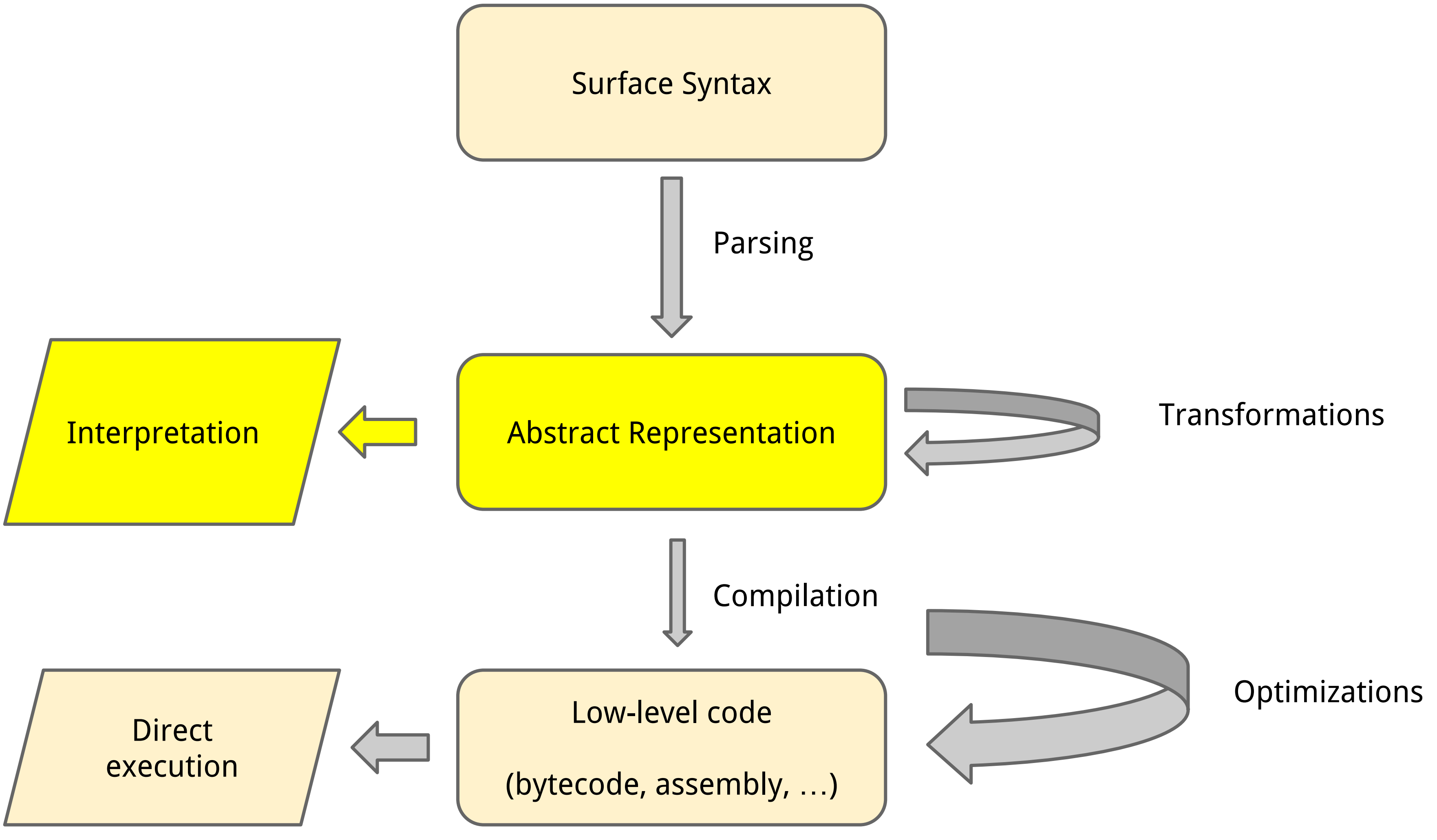
September 20, 2016

Riccardo Pucella

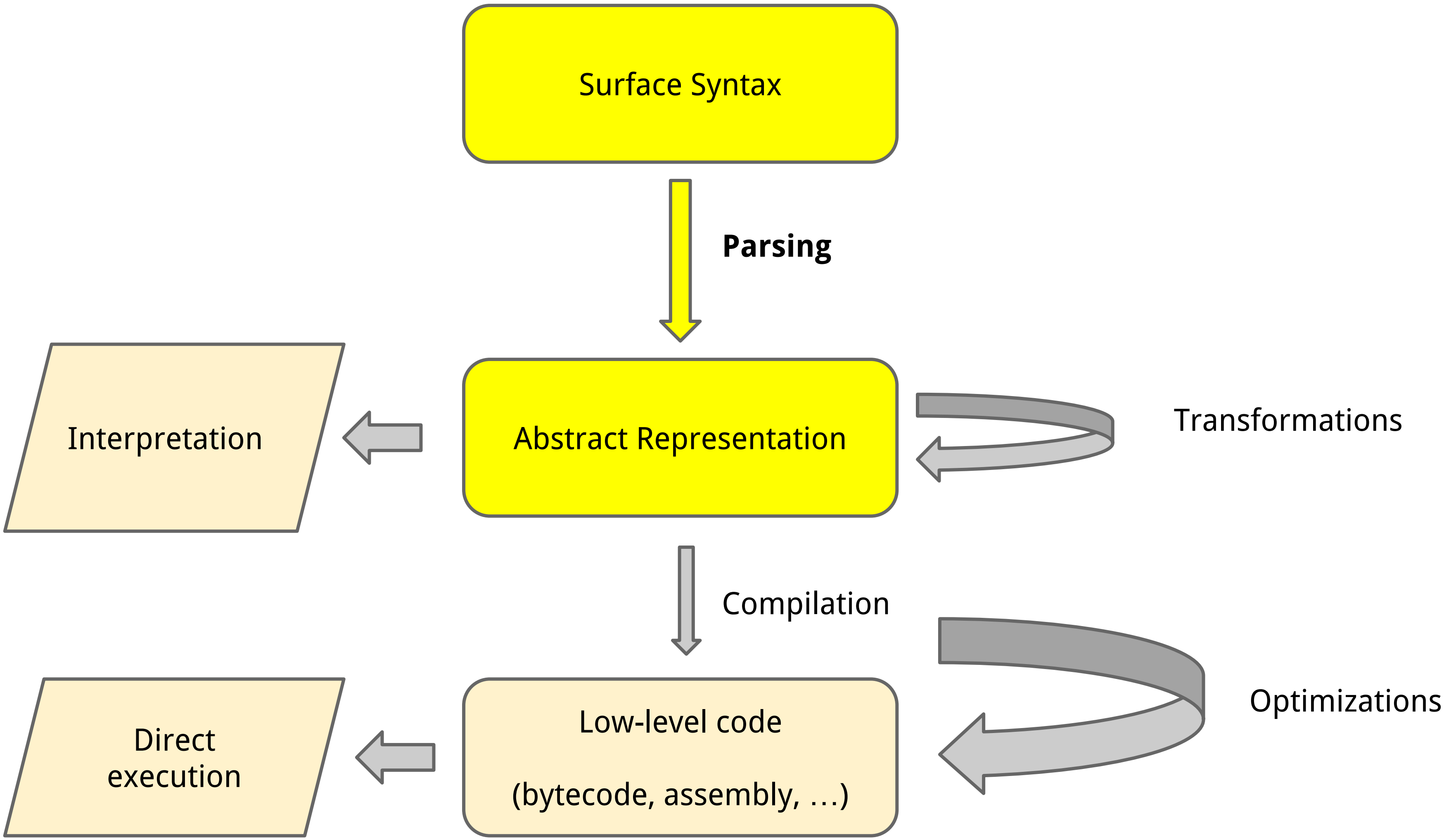
The structure of language execution



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
```

Surface syntax

Abstract representation

Generating abstract representation

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

Surface syntax

```
ELet ("x", EPlus (EInteger(10),
                  EInteger(20)),
      ETimes (EId("x"),
              EId("x")))
```

Abstract representation

The input

What is the input?

- files, input from interactive shells, ...
- abstraction: sequences of characters
- key operation:
  get next character from sequence

Two phases

Tokenization (or lexical analysis)

sequence of characters → sequence of tokens

Parsing

sequence of tokens → abstract representation

Same thing happens in natural languages

- phonemes into words into grammatical sentences

Example

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

Example

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

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

Example

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

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

```
SYM[let] LP SYM[x] EQUAL INT[10] PLUS INT[20] RP SYM[x]
   TIMES SYM[x]
```

Example

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

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

```
SYM[let] LP SYM[x] EQUAL INT[10] PLUS INT[20] RP SYM[x]
   TIMES SYM[x]
```

```
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
```

```
SYM[let] LP SYM[x] EQUAL INT[10] PLUS INT[20] RP SYM[x]
   TIMES SYM[x]
```

```
ELet (“x”, EPlus (EIn
                  EIn
     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	`/\".*\"/`
symbol	`/[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:

charact

Description of tokens: regular expressions

integer	`/[0-9]+/`
string	`/\".*\"/`
symbol	`/[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
```

Surface syntax

tokenization

Tokens

```
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"))
```

Surface syntax

tokenization

Tokens

parsing

Abstract representation

Parsing

- Identify valid token sequences
  - E.g. valid: `LET LP SYM[x] EQUAL INT[10] RP SYM[x]`
  - E.g. not: `LET LP SYM[x] SYM[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
           symbol
           true
           false

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

Example: `(let ((x (+ 10 20))) (* x x))`

Example: S-expressions

```
atomic ::= integer
           symbol
           true
           false

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

All **bolded terms** are tokens

Example: `(let ((x (+ 10 20))) (* x x))`

Creating parse results

```
atomic ::= integer_i
           symbol_s
           true
           false

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

Creating parse results

```
atomic ::= integer_i  ⟶  EInteger (i)
           symbol_s
           true
           false

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

Creating parse results

$$\begin{array}{lll} atomic ::= \textbf{\textit{integer}}_i & \longrightarrow & \texttt{EInteger}\ (i) \\ \qquad\qquad \textbf{\textit{symbol}}_s & \longrightarrow & \texttt{EId}\ (s) \\ \qquad\qquad \textbf{true} & \longrightarrow & \texttt{EBoolean (True)} \\ \qquad\qquad \textbf{false} & \longrightarrow & \texttt{EBoolean (False)} \end{array}$$

$$\begin{array}{l} expr ::= atomic \\ \qquad\quad \textbf{(+}\ expr\quad expr\ \textbf{)} \\ \qquad\quad \textbf{(*}\ expr\quad expr\ \textbf{)} \\ \qquad\quad \textbf{(if}\ expr\quad expr\quad expr\ \textbf{)} \\ \qquad\quad \textbf{(let ((}\ \textbf{\textit{symbol}}\quad expr\ \textbf{))}\ expr\ \textbf{)} \end{array}$$

Creating parse results

$$
\begin{array}{llll}
\textit{atomic} ::= & \textbf{\textit{integer}}_i & \longrightarrow & \texttt{EInteger}\ (i) \\
 & \textbf{\textit{symbol}}_s & \longrightarrow & \texttt{EId}\ (s) \\
 & \textbf{true} & \longrightarrow & \texttt{EBoolean}\ (\texttt{True}) \\
 & \textbf{false} & \longrightarrow & \texttt{EBoolean}\ (\texttt{False})
\end{array}
$$

$$
\begin{array}{llll}
\textit{expr} ::= & \textit{atomic}_r & \longrightarrow & r \\
 & (\ +\ \textit{expr}_{e1}\ \ \textit{expr}_{e2}) & \longrightarrow & \texttt{EPlus}\ (e1,e2) \\
 & (\ *\ \textit{expr}_{e1}\ \ \textit{expr}_{e2}) & \longrightarrow & \texttt{ETimes}\ (e1,e2) \\
 & (\ \textbf{if}\ \textit{expr}_{e1}\ \ \textit{expr}_{e2}\ \ \textit{expr}_{e3}) & \longrightarrow & \texttt{EIf}\ (e1,e2,e3) \\
 & (\ \textbf{let}\ (\ (\ \textbf{\textit{symbol}}_s\ \ \textit{expr}_{e1})\ )\ \textit{expr}_{e2}) & & \\
 & & \longrightarrow & \texttt{ELet}\ (s,e1,e2)
\end{array}
$$

These sort of grammars are often called *attribute grammars*

Two approaches to parsing

TOP-DOWN

- recursive descent
- coded by hand
- flexible but expensive

- good for simple grammars

BOTTOM-UP

- table-based
- generated by tools (yacc, bison, antlr)
- fast

- 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
           symbol
           true
           false

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

Predictive parser for S-expressions

```
def parse_atomic (tokens):
   if tokens[0] is an integer:
     return (EInteger(int(tokens[0])),tokens[1:])
   if tokens[0] is a symbol:
     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:])
```

(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

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

Left factorization

Sometimes we can transform a grammar into one that can be parsed by a predictive parser.

Left factorization:

- a kind of distributive law for grammars
- rules that start with common tokens...
  - can be replaced by a single rule with those tokens...
  - and a new nonterminal for the different leftovers

Example: S-expressions

```
atomic ::= integer
           symbol
           true
           false

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

Example: S-expressions

```
atomic ::= integer
           symbol
           true
           false

expr ::= atomic
         ( expr_seq )

expr_seq ::= + expr expr
             * expr expr
             if expr expr expr
             let ( ( symbol expr ) ) expr
```

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

Parser combinator libraries

Parser combinators

Idea:

- create small parsers
- combine parsers into more complex parsers

Example: **pyparsing** library in Python

Note: pyparsing does not in fact distinguish between tokenization and parsing.

Example: S-expressions

```
atomic ::= integer
           symbol
           true
           false

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

Combinators for S-expressions

```
idChars = alphas+"_+*-?!=<>"

pSYMBOL = Word(idChars,idChars+"0123456789")

pINTEGER = Word("-0123456789","0123456789")

pTRUE = Keyword("true")

pFALSE = Keyword("false")

pATOMIC = ( pINTEGER | pTRUE | pFALSE | pSYMBOL )
```

Combinators for S-expressions

```
pEXPR = Forward()

pIF = Keyword("(") + Keyword("if") + pEXPR + pEXPR +
       pEXPR + Keyword(")")

pLET = Keyword("(") + Keyword("let") + Keyword("(") +
        Keyword("(") + pSYMBOL + pEXPR + Keyword(")") +
        Keyword(")") + pEXPR + Keyword(")")

pPLUS = Keyword("(") + Keyword("+") + pEXPR + pEXPR +
        Keyword(")")
pTIMES = Keyword("(") + Keyword("*") + pEXPR + pEXPR +
        Keyword(")")

pEXPR << ( pATOMIC | pIF | pLET | pPLUS | pTIMES)
```

Demo

Example: Natural syntax

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

Example: Natural syntax

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

Grammar with *left recursion*

Bad for recursive-descent parsers — WHY?

Can eliminate left recursion by rewriting the grammar:

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

↓

```
expr ::= integer
         integer expr_rest

expr_rest ::= + expr
              * expr
```

expr

Grammar with *left recursion*

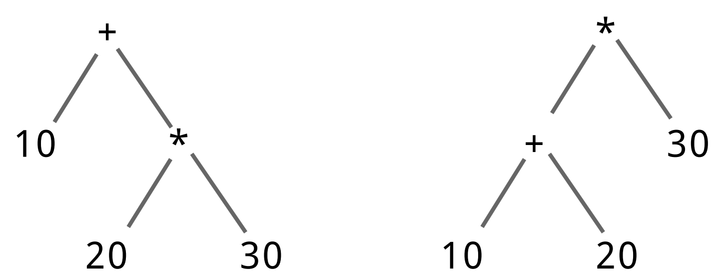
Bad for recursive-descent parsers — WHY?

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

Third homework

- extend the parser with function calls
- extend the parser with multi-binding lets
- implement an infix parser for expressions