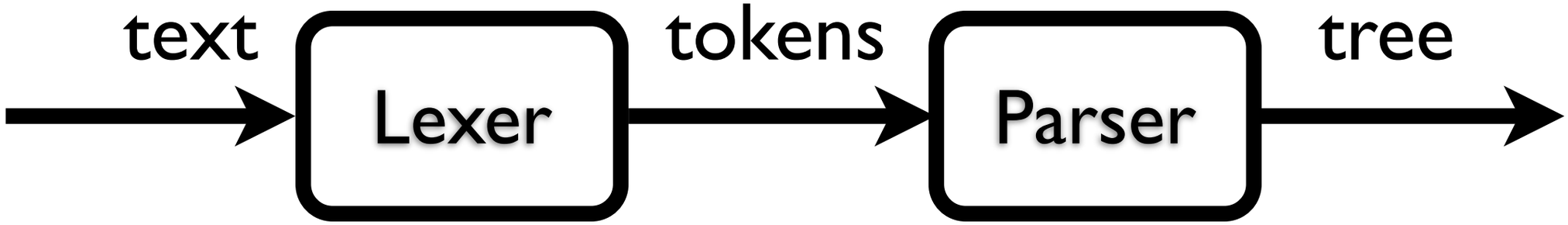
Parsing: Tokens to Trees

Matt Might
University of Utah
matt.might.net

Today

- Context-free languages
- Context-free grammars
- Notations for grammars
- Syntax trees
- Recursive-descent parsers
- Racket lexers


text
Lexer
tokens
Parser
tree

What is parsing?

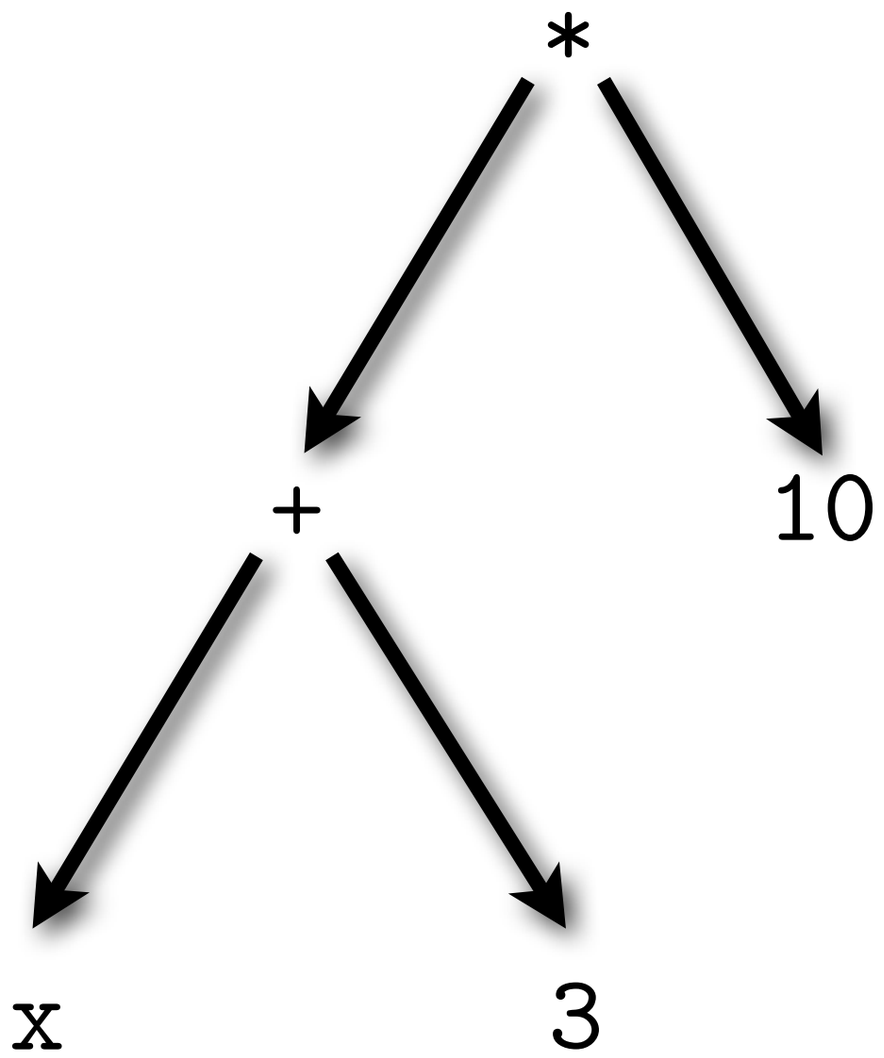
Unstructure to structure.

A transformation from a sequence to a tree.

$$(x + 3) * 10$$

```
(x + 3) * 10
```

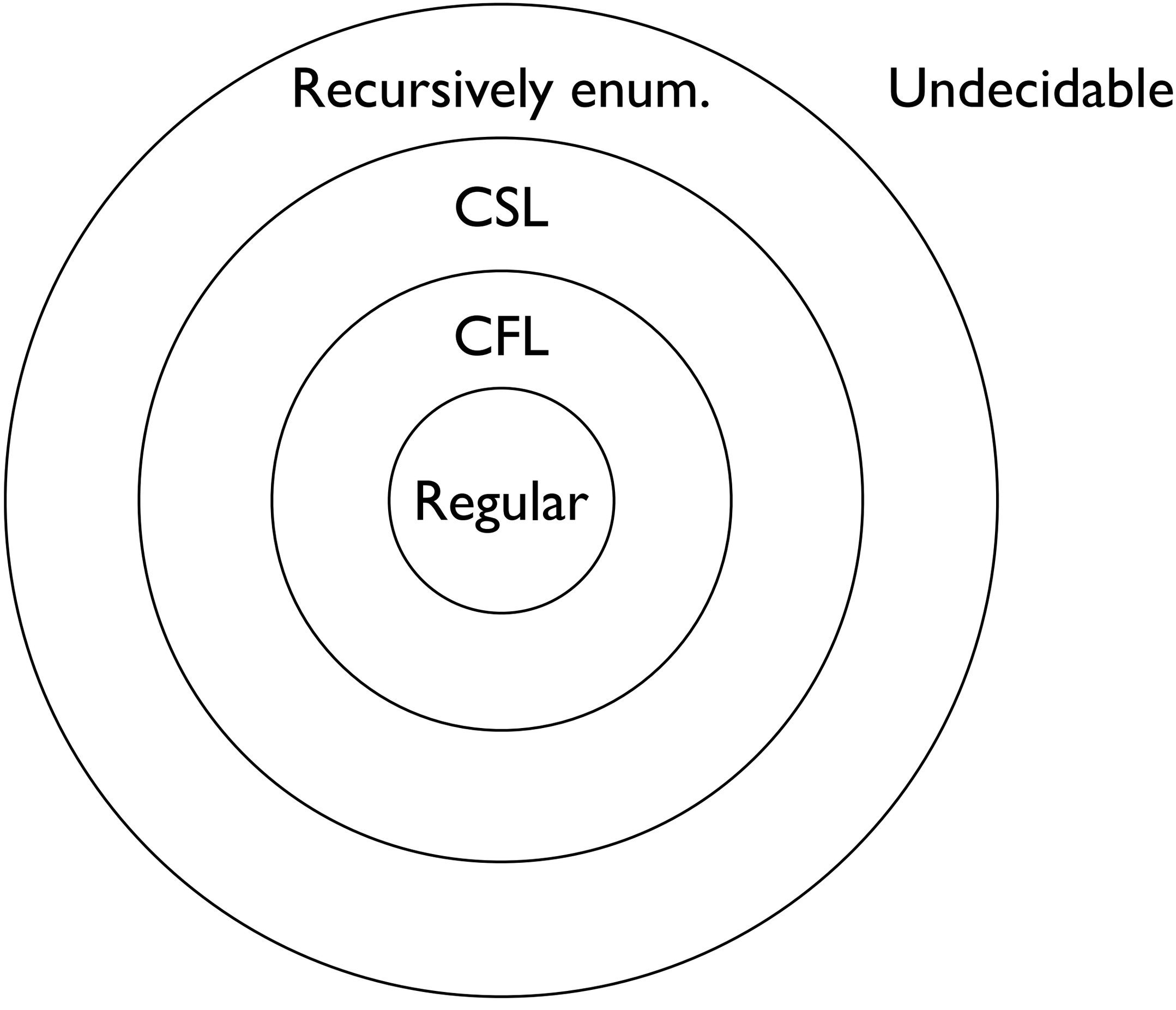
```
(x + 3) * 10
```

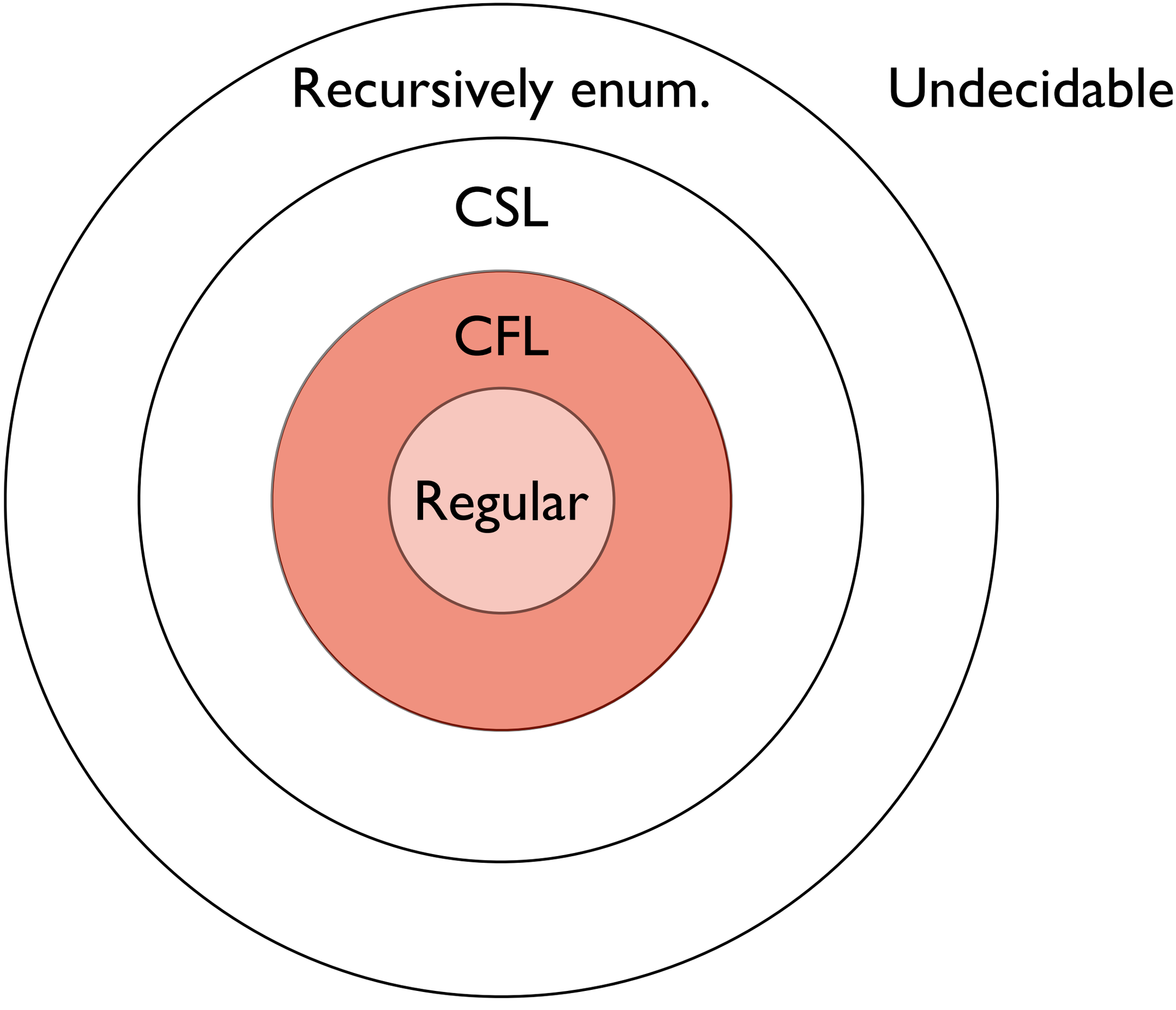

*
+
10
x
3

grammars :: tree structure

What's a context-free grammar?

Recursive regular expressions.

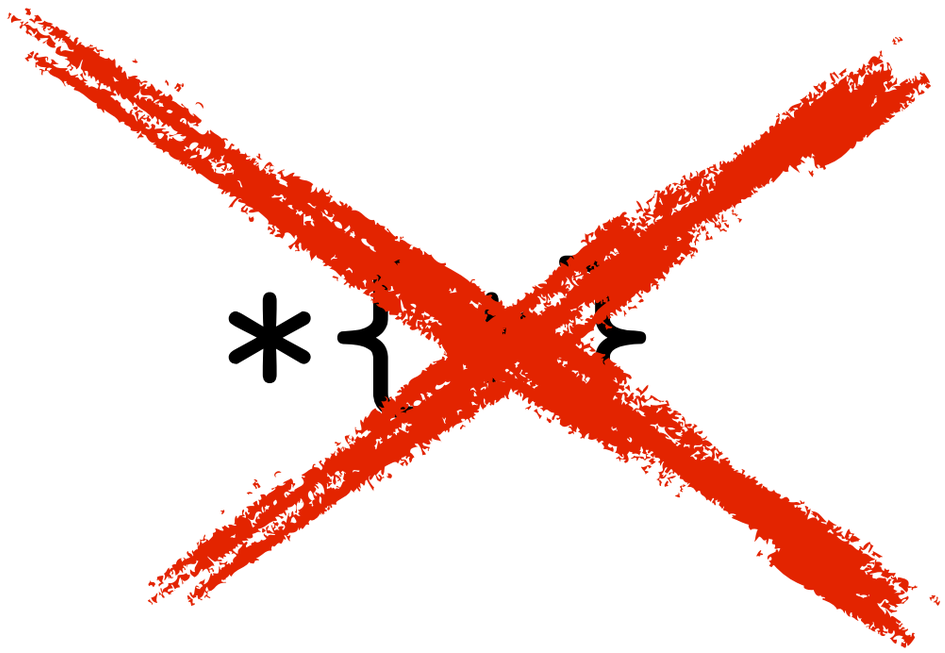

Recursively enum.
Undecidable
CSL
CFL
Regular


Recursively enum.
Undecidable
CSL
CFL
Regular

Regex for balanced braces?

|{}|{{}}|{{{}}}|...

{*}*



$$S ::= \{S\} | \varepsilon$$

CFG ingredients

- Concatenation
- Union
- Recursion

Kleene star?

$$\ldots A^{*}\ldots$$

$$\ldots B \ldots$$

$$B ::= AB$$

$$\mid \varepsilon$$

Context-free grammar

- **Terminals**: Set of atomic symbols
- **Nonterminals**: Phrase types
- **Productions**: Phrase structure rules
- **Start symbol**: Top-level phrase

Notations

- Backus-Naur Form (BNF)
- Extended Backus-Naur Form (EBNF)
- Regular extensions to BNF
- Augmented Backus-Naur Form (ABNF)
- Phrase structure rules
- Set-based language construction

Backus-Naur Form

<*nonterm*> ::= exp_1 ... exp_2

Backus-Naur Form

exp is <*nonterm*> or *term*

Backus-Naur Form

term is "*literal*"

EBNF

exp is *nonterm*

or *term*

or { *exp* }

or [*exp*]

or (*exp*)

or *exp* | *exp*

Regular EBNF

exp is *nonterm*
or *term*
or *exp** or { *exp* }
or *exp*+
or *exp*? or [*exp*]
or (*exp*)
or *exp* | *exp*

ABNF

exp is *nonterm*

or *term*

or *n***m exp*

or *n** *exp*

or * *exp*

or [*exp*]

or (*exp*)

or *exp* / *exp*

Phrase structure rules

$$NT \rightarrow S_1 \ldots S_n$$

Phrase structure rules

S is nonterminal or terminal

Set-based languages

$$L' = \epsilon \text{ or } \{\langle\rangle\}$$

$$L' = c \text{ or } \{c\}$$

$$L' = \emptyset \text{ or } \{\}$$

$$L' = L_1 \cup L_2$$

$$L' = L_1 \circ L_2$$

Examples

S-Expression

```
<S>  ::= "(" <S>* ")" | <A>

<A>  ::= SYMBOL
       |  NUMBER
```

Terminal symbols

```
<S>  ::= "(" <S>* ")" | <A>

<A>  ::= SYMBOL
       | NUMBER
```

Non-terminal symbols

```
<S> ::= "(" <S>* ")" | <A>

<A> ::= SYMBOL
      | NUMBER
```

Non-terminal symbols

```
<S> ::= "(" <S>* ")" | <A>

<A> ::= SYMBOL
      | NUMBER
```

Productions

```
<S>  ::= "(" <S>* ")" | <A>
```

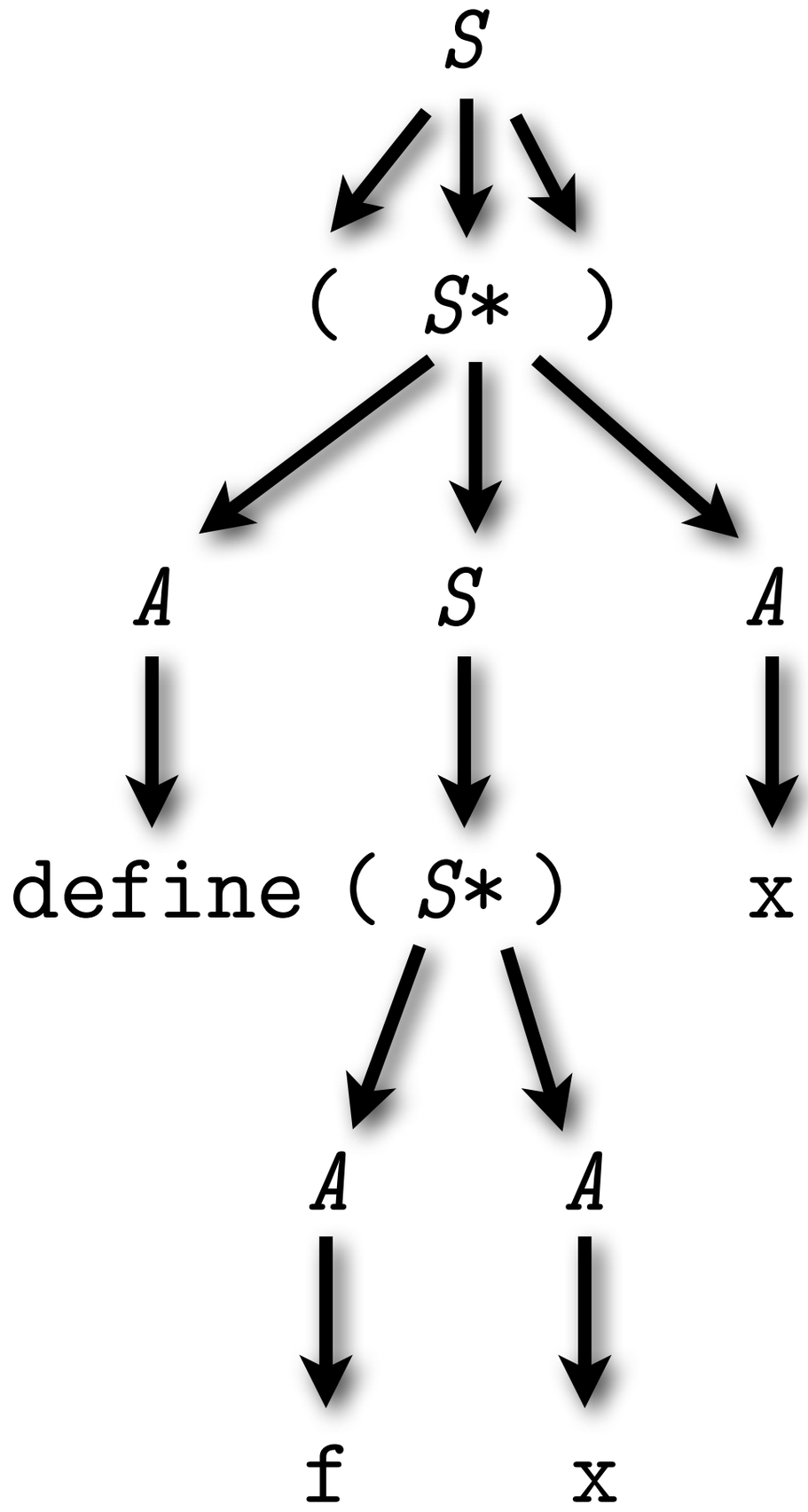
```
<A>  ::= SYMBOL
       |  NUMBER
```

Start symbol

```
<S> ::= "(" <S>* ")" | <A>

<A> ::= SYMBOL
      |  NUMBER
```

```
(define (id x) x)
```


S
(S*)
A
S
A
define (S*)
x
A
A
f
x

Terms

$$E \to T + E$$

$$E \to T$$

$$T \to F \times T$$

$$T \to F$$

$$F \to (\ E \ )$$

$$F \to n$$

Terminal symbols

$$E \rightarrow T + E$$

$$E \rightarrow T$$

$$T \rightarrow F \times T$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow n$$

Nonterminal symbols

$$E \rightarrow T + E$$

$$E \rightarrow T$$

$$T \rightarrow F \times T$$

$$T \rightarrow F$$

$$F \rightarrow (\ E \ )$$

$$F \rightarrow n$$

Nonterminal symbols

$$E \to T + E$$
$$E \to T$$
$$T \to F \times T$$
$$T \to F$$
$$F \to (E)$$
$$F \to n$$

Productions

$$E \to T + E$$

$$E \to T$$

$$T \to F \times T$$

$$T \to F$$

$$F \to (\ E \ )$$

$$F \to n$$

Start symbol

$$\boxed{E} \rightarrow T + E$$

$$\boxed{E} \rightarrow T$$

$$T \rightarrow F \times T$$

$$T \rightarrow F$$

$$F \rightarrow (\ E \ )$$

$$F \rightarrow n$$

Balanced parens

$$P = (\circ P \circ) \circ P \cup \epsilon$$

Terminals

$$P = \boxed{(} \circ P \circ \boxed{)} \circ P \cup \epsilon$$

Nonterminals

$$P = (\circ P \circ) \circ P \cup \epsilon$$

Productions

$$P = (\circ P \circ) \circ P \cup \epsilon$$

Start symbol

$$P = (\circ P \circ) \circ P \cup \epsilon$$

Syntax trees

Syntax trees encode structure

Expressions

$$E \to T + E$$

$$E \to T$$

$$T \to F \times T$$

$$T \to F$$

$$F \to (\ E \ )$$

$$F \to n$$

```
abstract class Exp {}

class Sum extends Exp {
 Exp left ;
 Exp right ;
}

class Prod extends Exp {
 Exp left ;
 Exp right ;
}

class Int extends Exp {
 int value ;
}
```

```
(struct sum     {left right})
(struct product {left right})
(struct int     {value})
```

How to parse (the hard way)

Recursive descent

- One function per nonterminal
- Allowed to look one token ahead

Recursive descent

- **peek()** : preview next token
- **next()** : consume & return next token
- **eat(token)** : match next token

Recursive descent example

S-Expressions

```
<S>  ::= "(" <SL> ")"
       |  <A>
<SL> ::= <S> <SL>
       |
<A>  ::= SYMBOL
       |  INTEGER
       |  #t
       |  #f
```

S-Expressions

S

SL

A

S-Expressions

```
parse S()

parse SL()

parse A()
```