Executable Grammars in Newspeak

Gilad Bracha
Distinguished Engineer
Cadence Design Systems

Parser Combinators

- Long history in Functional Programming
- Operators of BNF are functions from (grammar) productions to productions
- Productions are isomorphic to parsers
- Build parser by writing out BNF!

Parser Combinators

BNF

*id = letter (letter | digit) **

Parser Combinators

BNF

*id = letter (letter | digit) **

Newspeak

id = letter, (letter | digit) star.

Parser Combinators

BNF

*id = letter (letter | digit) **

Newspeak

id = letter, (letter | digit) star.

Javanese

id = letter().seq(letter().or(digit()).star());

How it Works

```
id = letter().seq(letter().or(digit()).star());
```

How it Works

```
id = letter().seq(letter().or(digit()).star());
```

How it Works

```
id = letter().seq(letter().or(digit()).star());
```

How it Works

```
id = letter().seq(letter().or(digit()).star());
```

How it Works

id = letter().seq(letter().or(digit()).star());

How it Works

id = letter().seq(letter().or(digit()).star());

How it Works

id = letter().seq(letter().or(digit()).star());

How it Works

id = letter().seq(letter().or(digit()).star());

How it Works

id = letter().seq(letter().or(digit()).star());

How it Works

id = letter().seq(letter().or(digit()).star());

How it Works

id = letter().seq(letter().or(digit()).star());

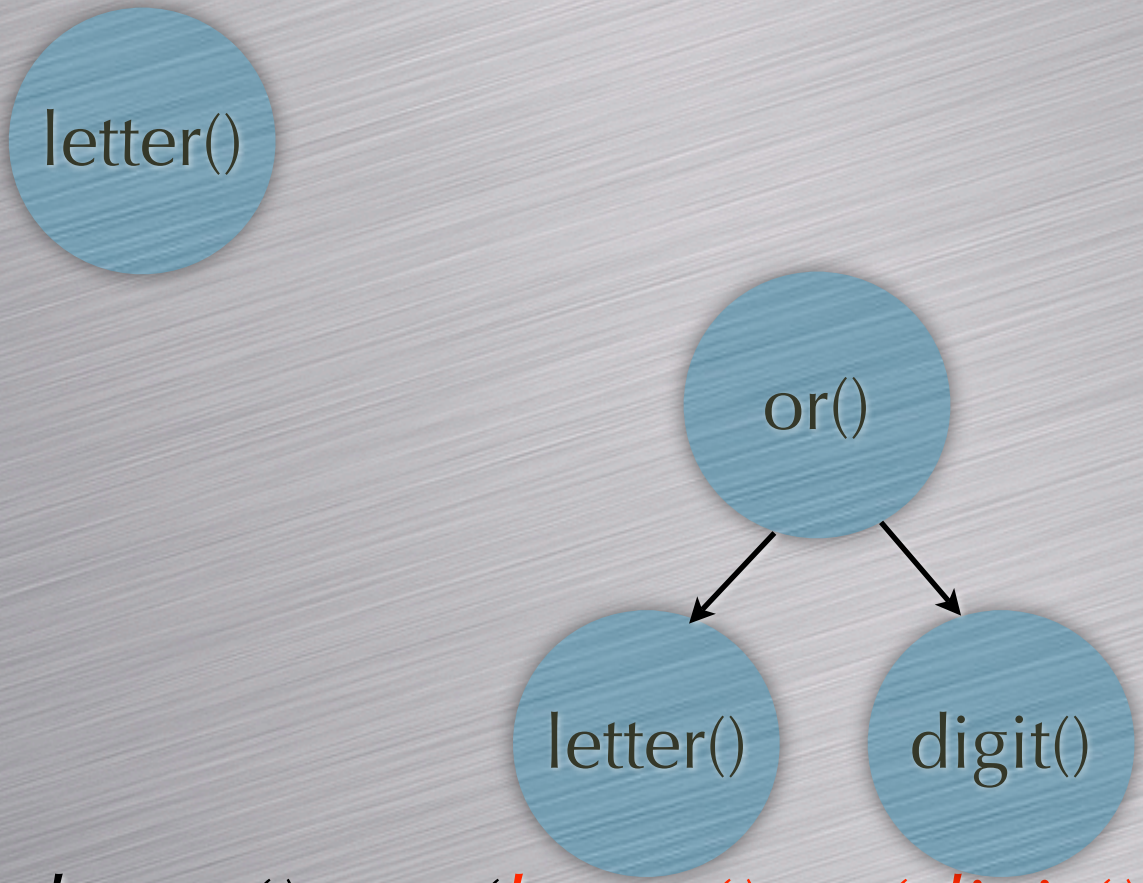
How it Works


```
id = letter().seq(letter().or(digit()).star());
```

How it Works

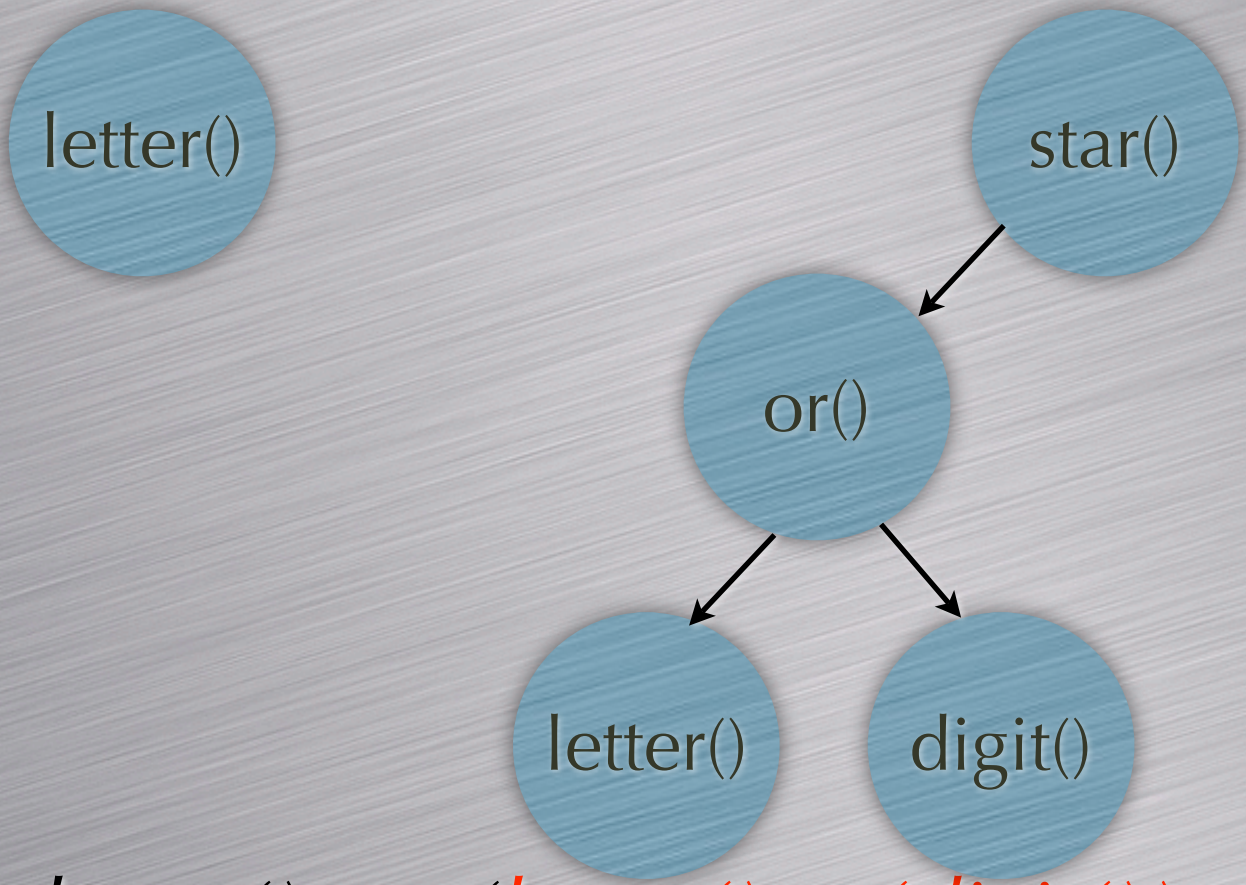

```
id = letter().seq(letter().or(digit()).star());
```

How it Works



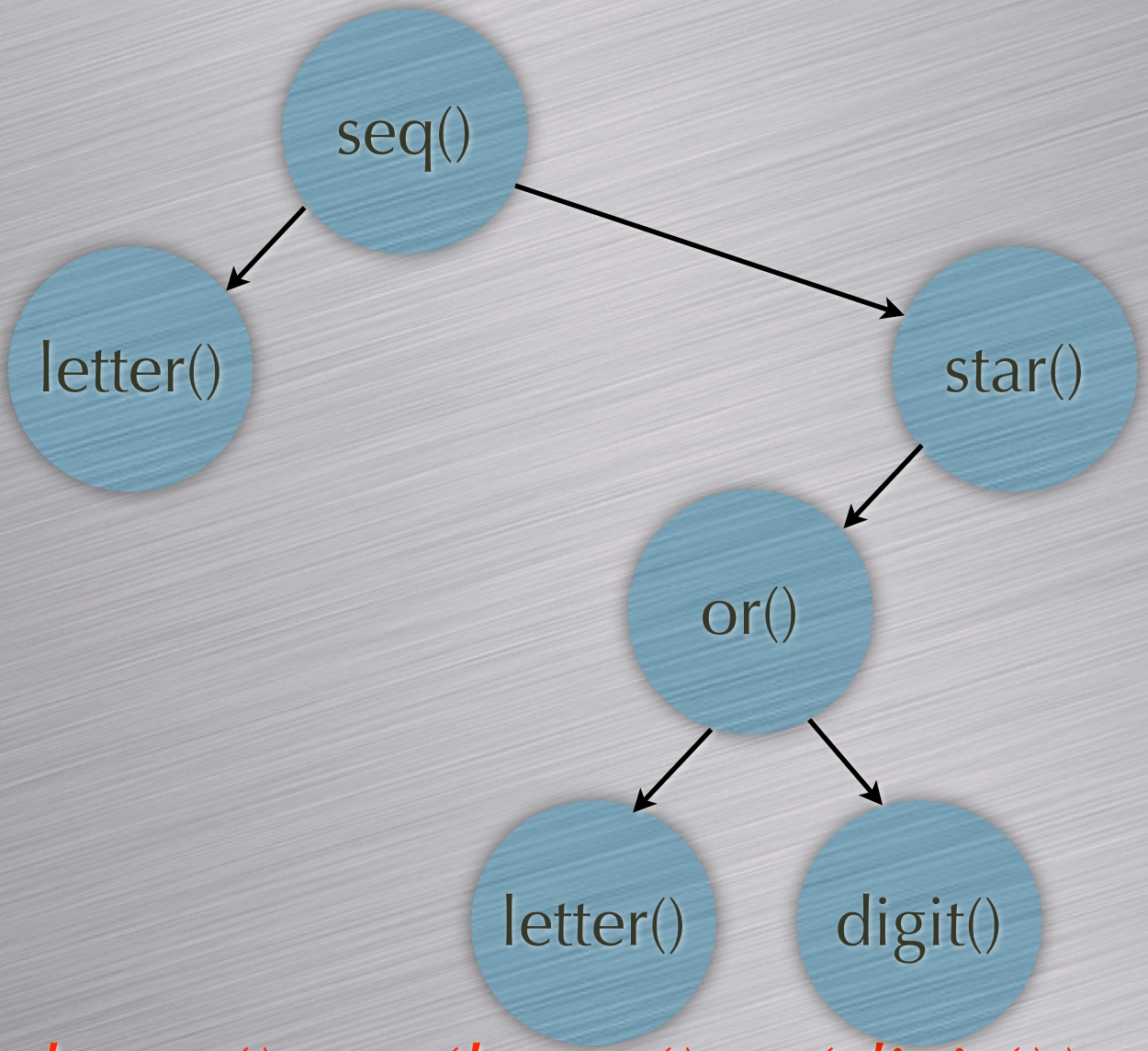
id = letter().seq(letter().or(digit()).star());

How it Works



id = letter().seq(letter().or(digit()).star());

How it Works



id = letter().seq(letter().or(digit()).star());

How it Works

id = letter, (letter | digit) star.

How it Works

id = letter, (letter | digit) star.

How it Works

id = letter, (letter | digit) star.

How it Works

id = letter, (letter | digit) star.

How it Works

id = letter, (letter | digit) star.

How it Works

id = letter, (letter | digit) star.

How it Works

id = letter, (letter | digit) star.

How it Works

id = letter, (letter | digit) star.

How it Works

id = letter, (letter | digit) star.

How it Works

id = letter, (letter | digit) star.

How it Works

id = letter, (letter | digit) star.

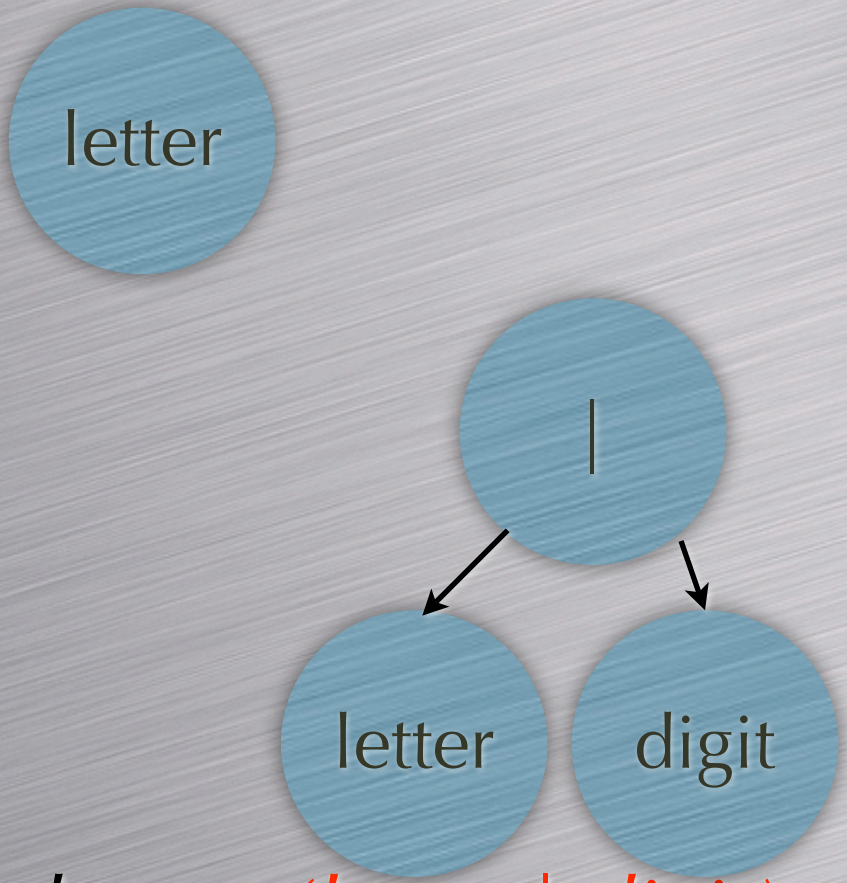
How it Works

id = letter, (letter | digit) star.

How it Works

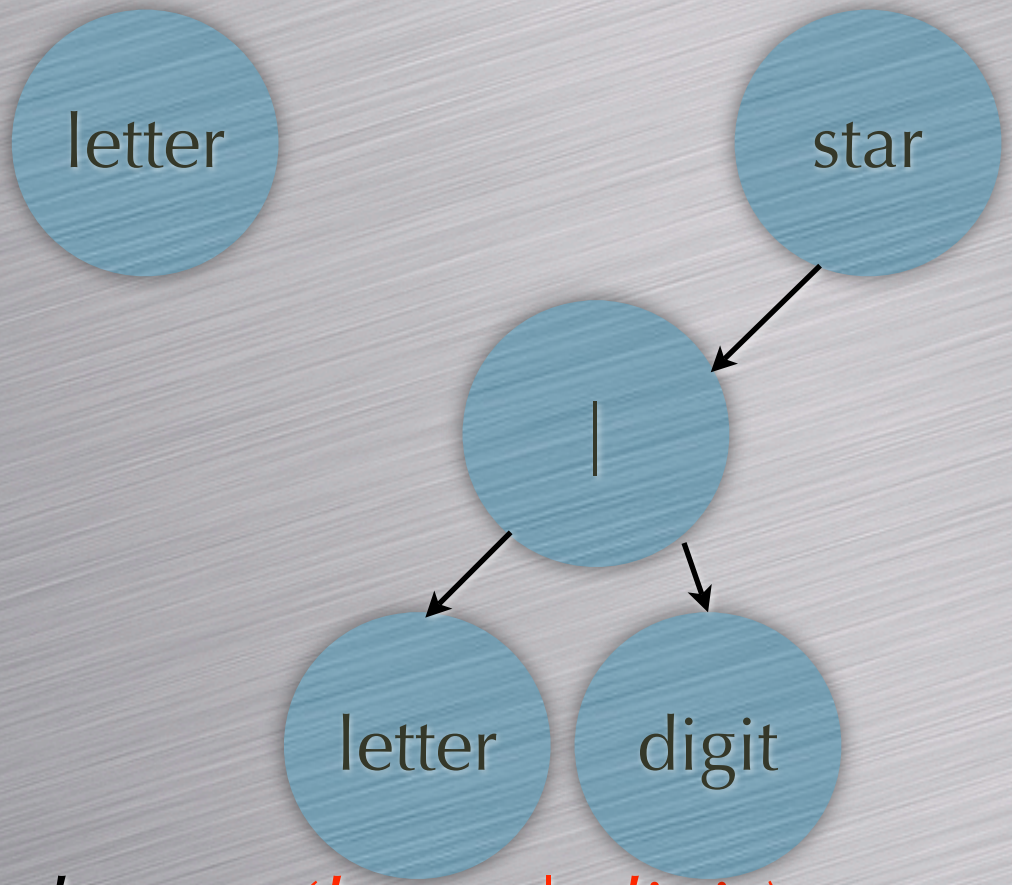
id = letter, (letter | digit) star.

How it Works



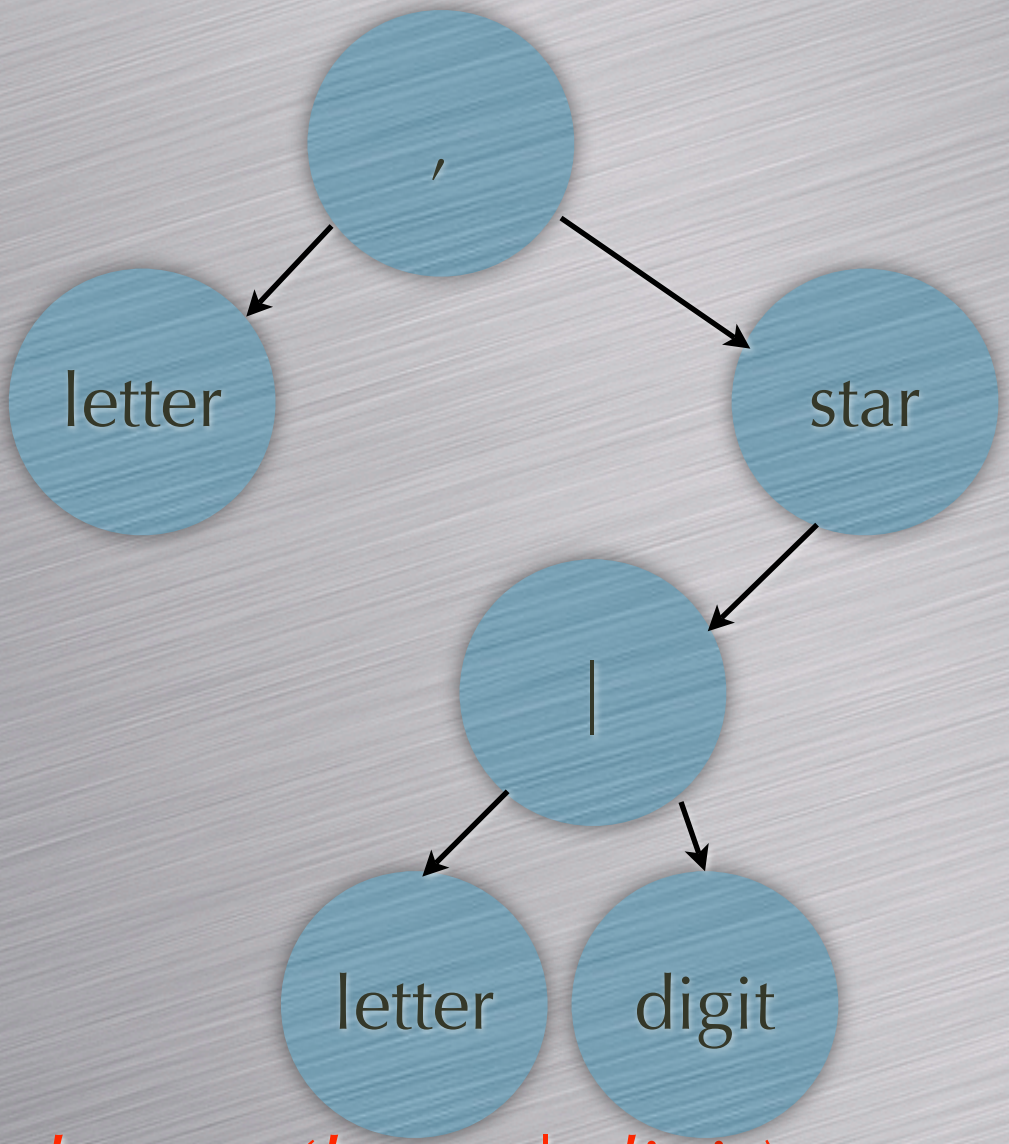
id = letter, (letter | digit) star.

How it Works



id = letter, (letter | digit) star.

How it Works



id = letter, (letter | digit) star.

Why is this Ugly?

```
id = letter().seq(letter().or(digit()).star());
```

Why is this Ugly?

id = letter().seq(letter().or(digit()).star());

vs.

*id = letter (letter | digit) **

Why is this Ugly?

```
id = letter().seq(letter().or(digit()).star());
```

vs.

```
id = letter (letter | digit) *
```

Why is this Ugly?

```
id = letter().seq(letter().or(digit()).star());
```

vs.

```
id = letter (letter | digit) *
```

vs.

```
id = letter, (letter | digit) star.
```

Why is it Ugly?

A programming language is low level when its programs require attention to the irrelevant

- Alan Perlis

A Complete Grammar

```
class ExampleGrammar1 = ExecutableGrammar (
|
digit = charBetween: $0 and: $9.
letter = (charBetween: $a and:$z) |
         (charBetween: $A and:$Z).
id = letter, (letter | digit) star.
identifier = tokenFor: id.
hat = tokenFromChar: $^.
expression = identifier.
returnStatement = hat, expression.
|
)()
```

Building an AST

```
returnStat = hat, expression
                    wrapper:[:caret : expr |
                                ReturnStat on: expr
                    ]
```

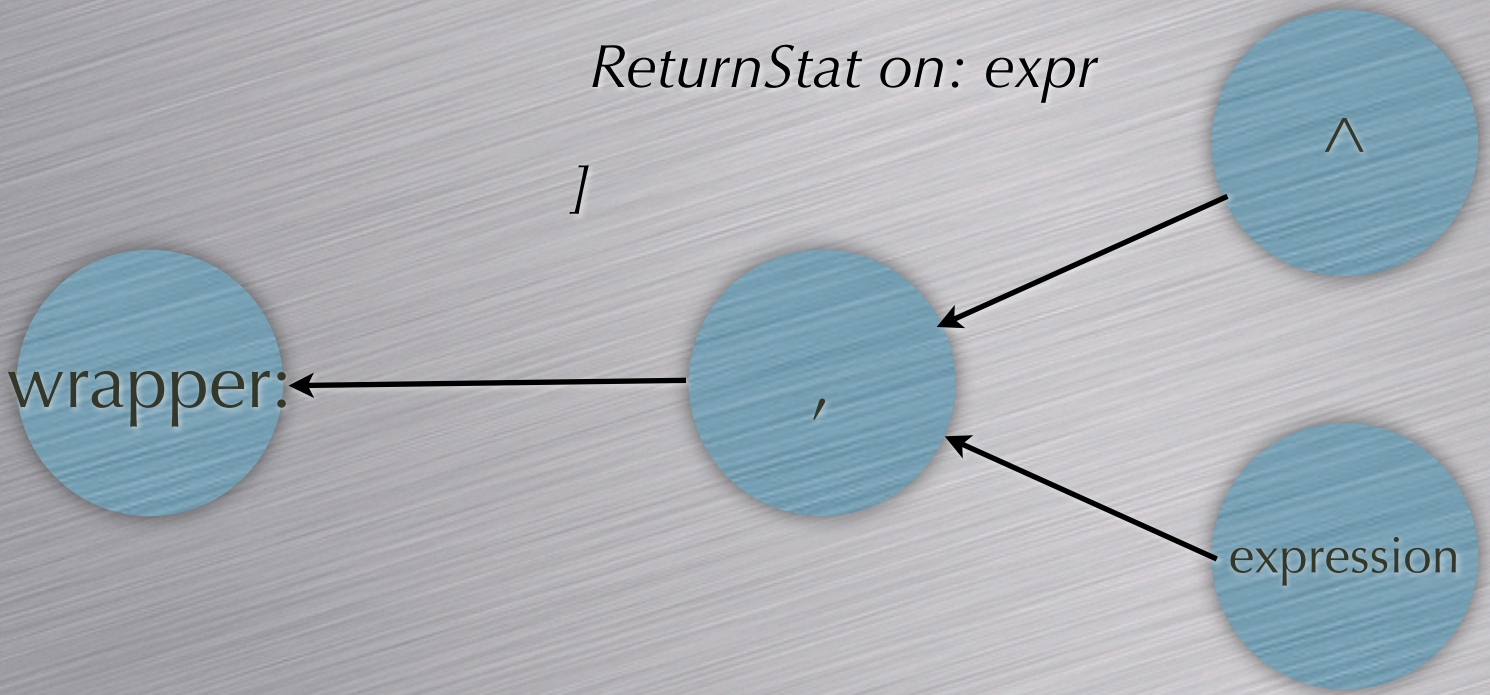
Building an AST

```
returnStat = hat, expression

        wrapper:[:caret : expr |

                ReturnStat on: expr

        ]
```



Grammar

```
returnStat = hat, expression

        wrapper:[:caret : expr |

                ReturnStat on: expr

        ]
```

AST construction

```
returnStat = hat, expression

                wrapper:[:caret : expr |

                        ReturnStat on: expr

                ]
```

Factor out Grammar

Superclass (pure grammar specification):

returnStat = hat, expression

Subclass (AST builder):

```
returnStat = (

        super returnStat

            wrapper:[:caret : expr |

                ReturnStat on: expr

            ] )
```

Modular Parser

```
class ExampleParser1= ExampleGrammar1 () (
  id = (
    ^super id
      wrapper:[:fst :snd | fst asString, (String withAll: snd)]
  )
 identifier = (
    ^super identifier
      wrapper:[:v |VariableAST new name: v token;
                                  start: v start; end: v end].
  )
  returnStatement = (
    ^super returnStatement
      wrapper:[:r :e | ReturnStatAST new expr:e;
                                  start: r start; end: e end].
)
```

Extending a Grammar

```
class ExampleGrammar2 = ExampleGrammar1 (
|
if = tokenFromSymbol:#if.
then = tokenFromSymbol:#then.
else = tokenFromSymbol:#else.
ifStatement = if, expression, then, statement, else, statement.
statement = ifStatement | returnStatement.
|
)()
```

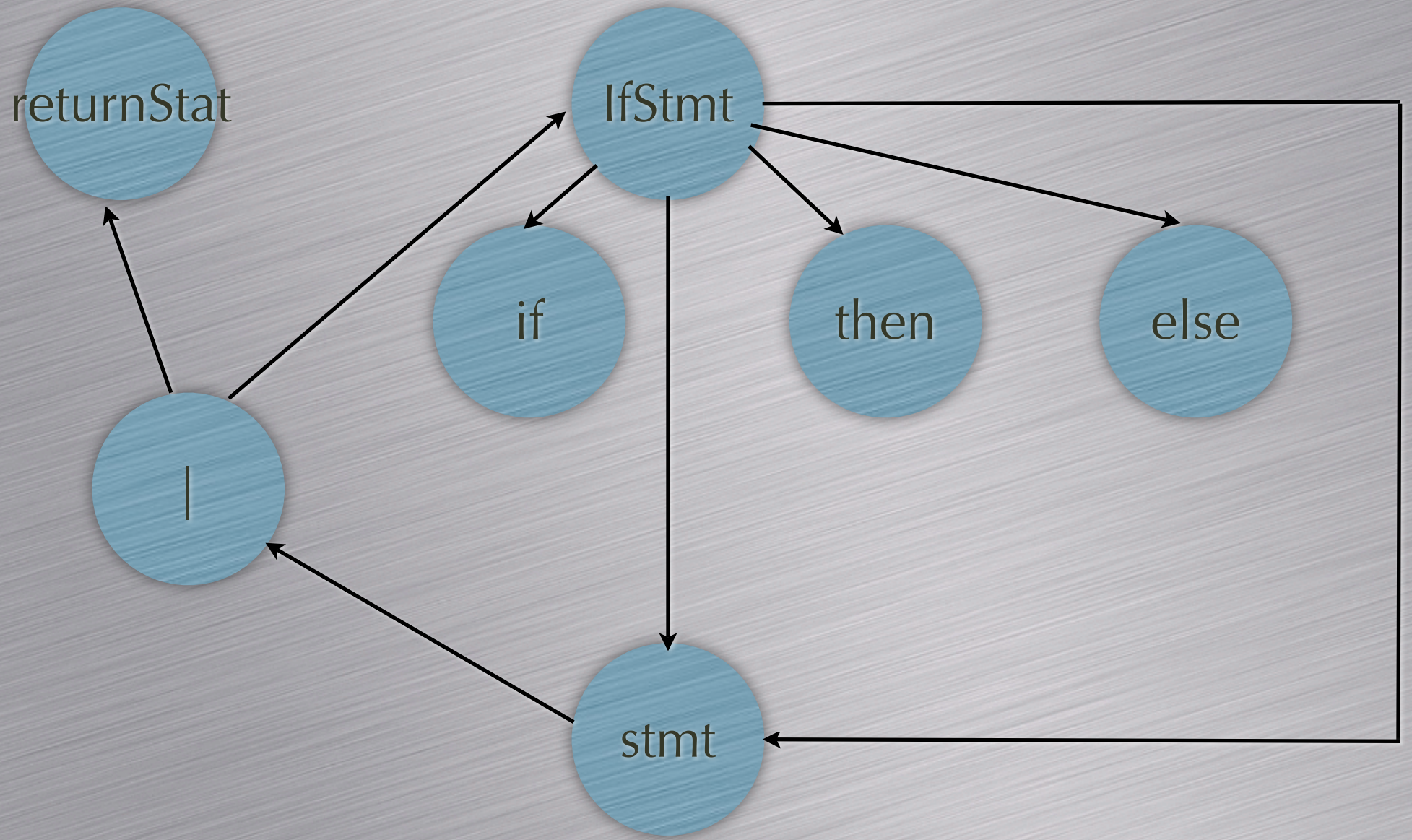
Mutual Recursion

ifStatement = if, expression, then, statement,

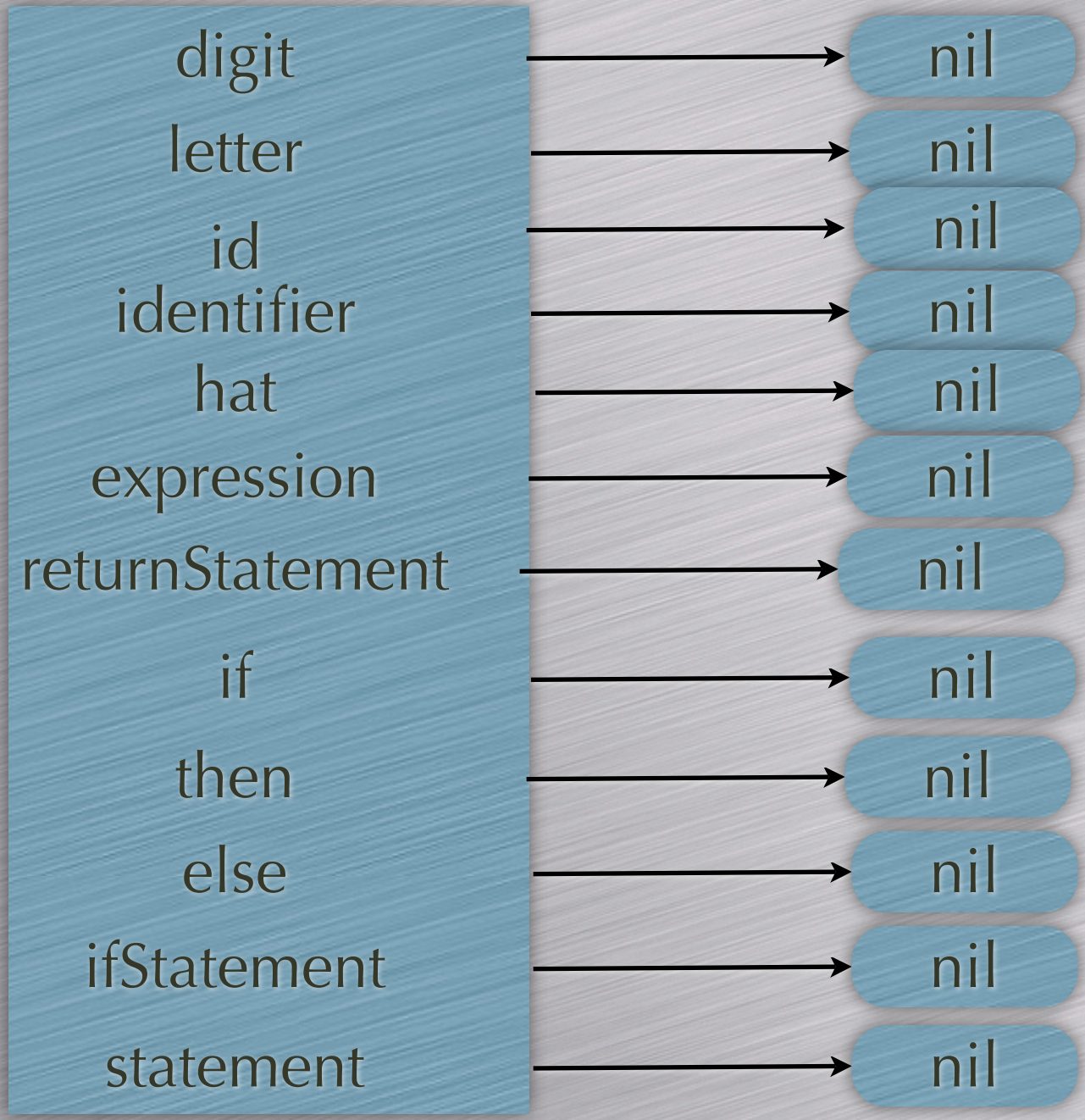
else, statement.

statement = ifStatement | returnStatement.

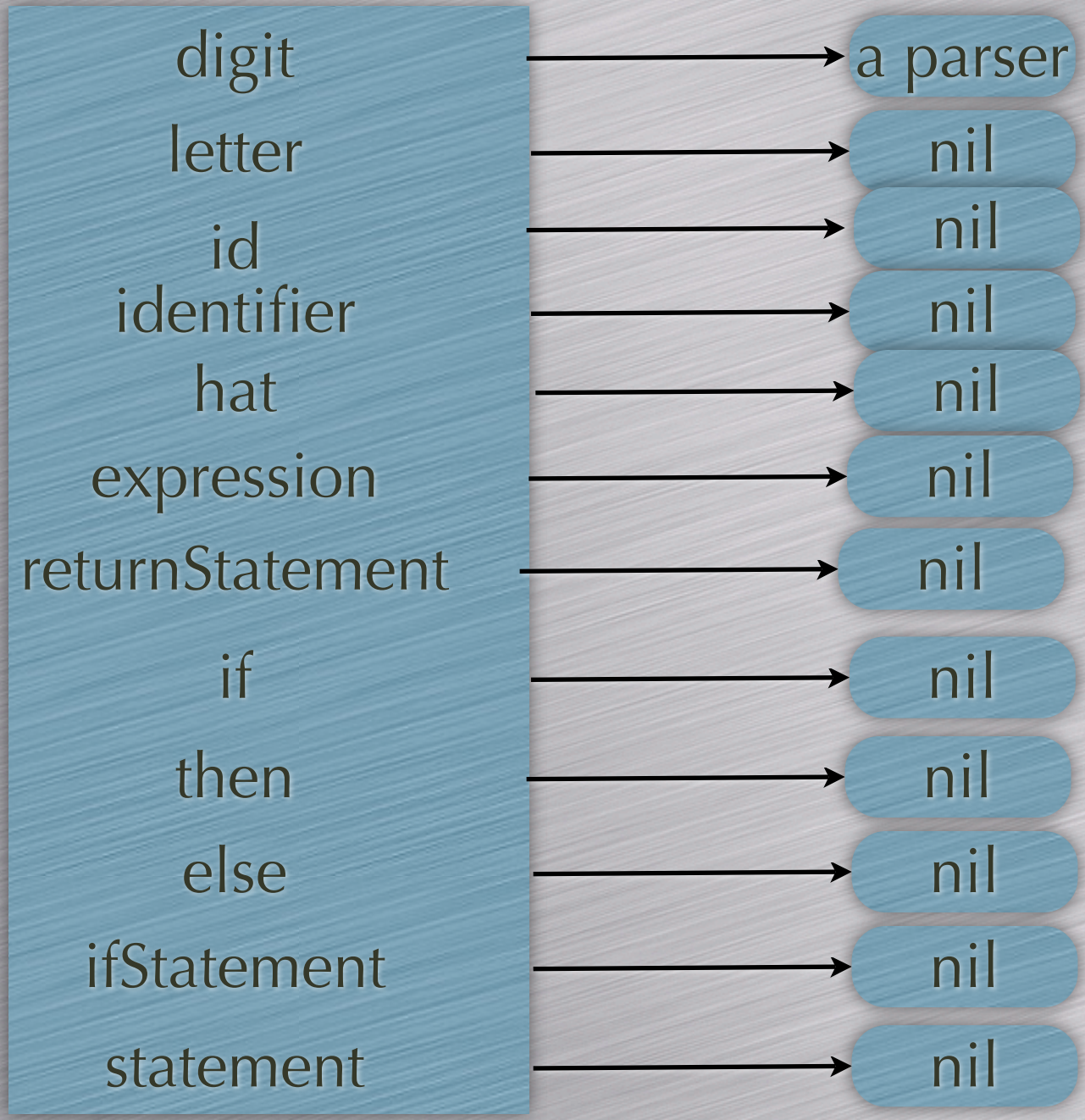
Mutual Recursion



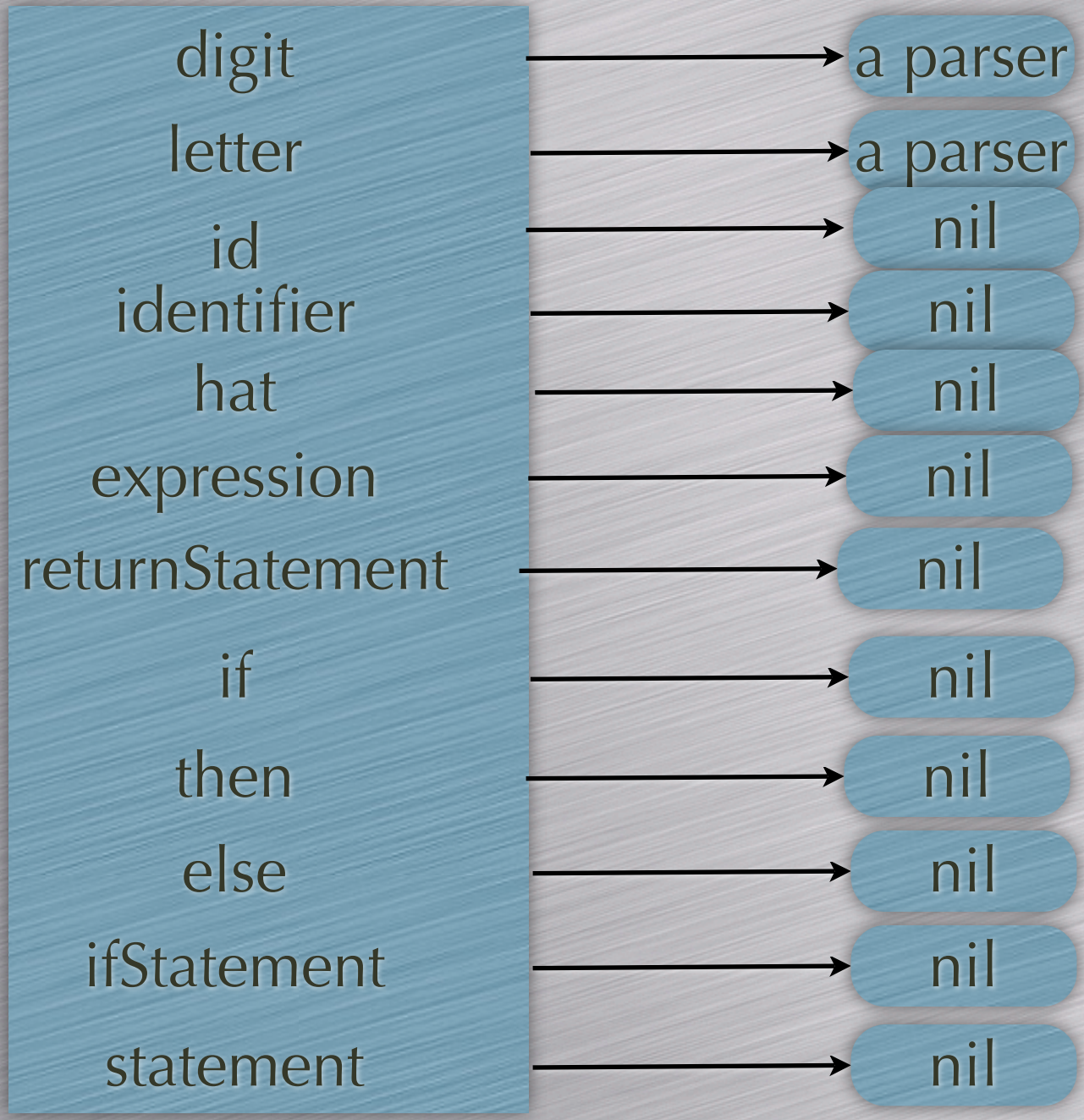
Mutual Recursion



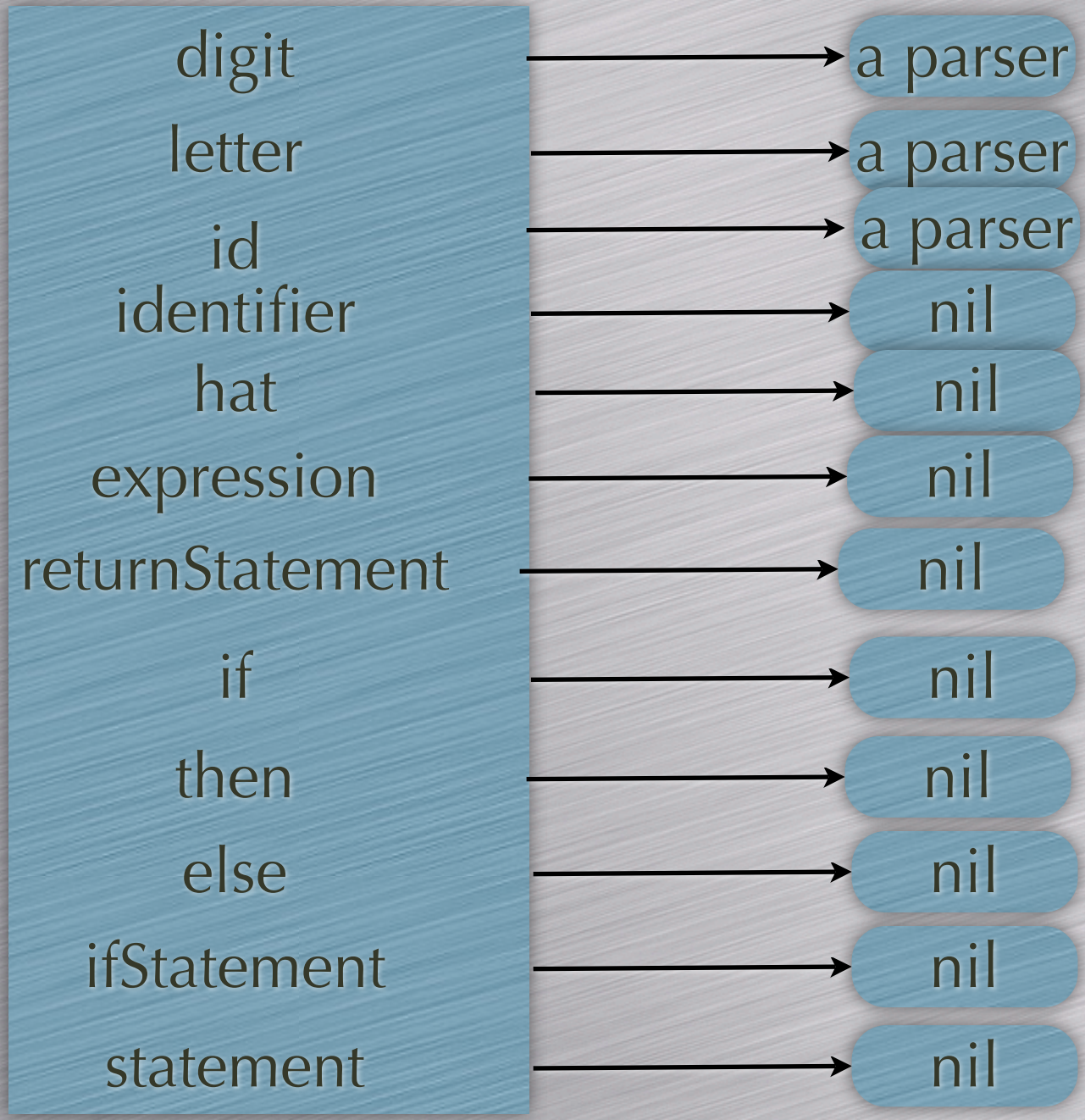
Mutual Recursion



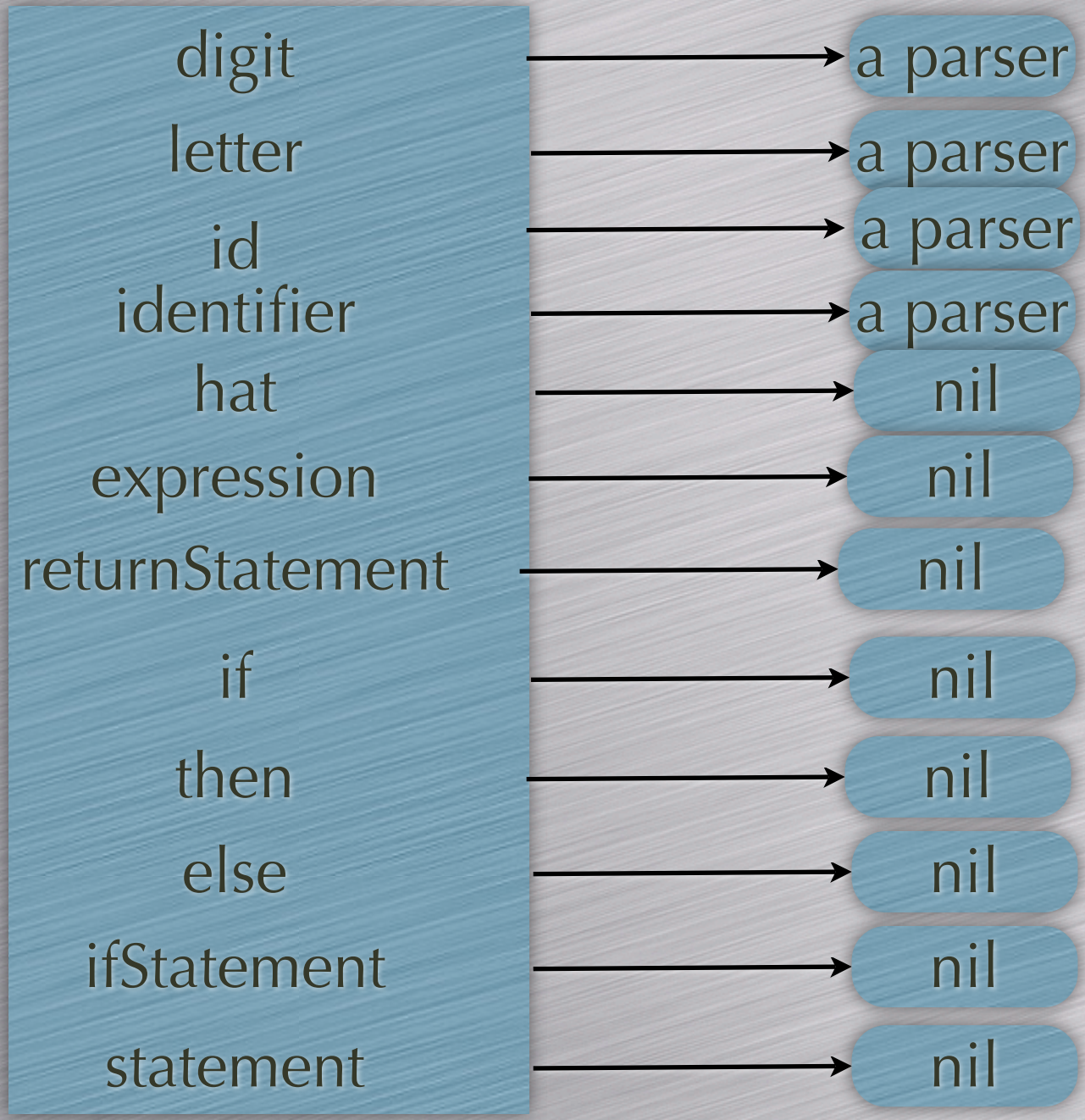
Mutual Recursion



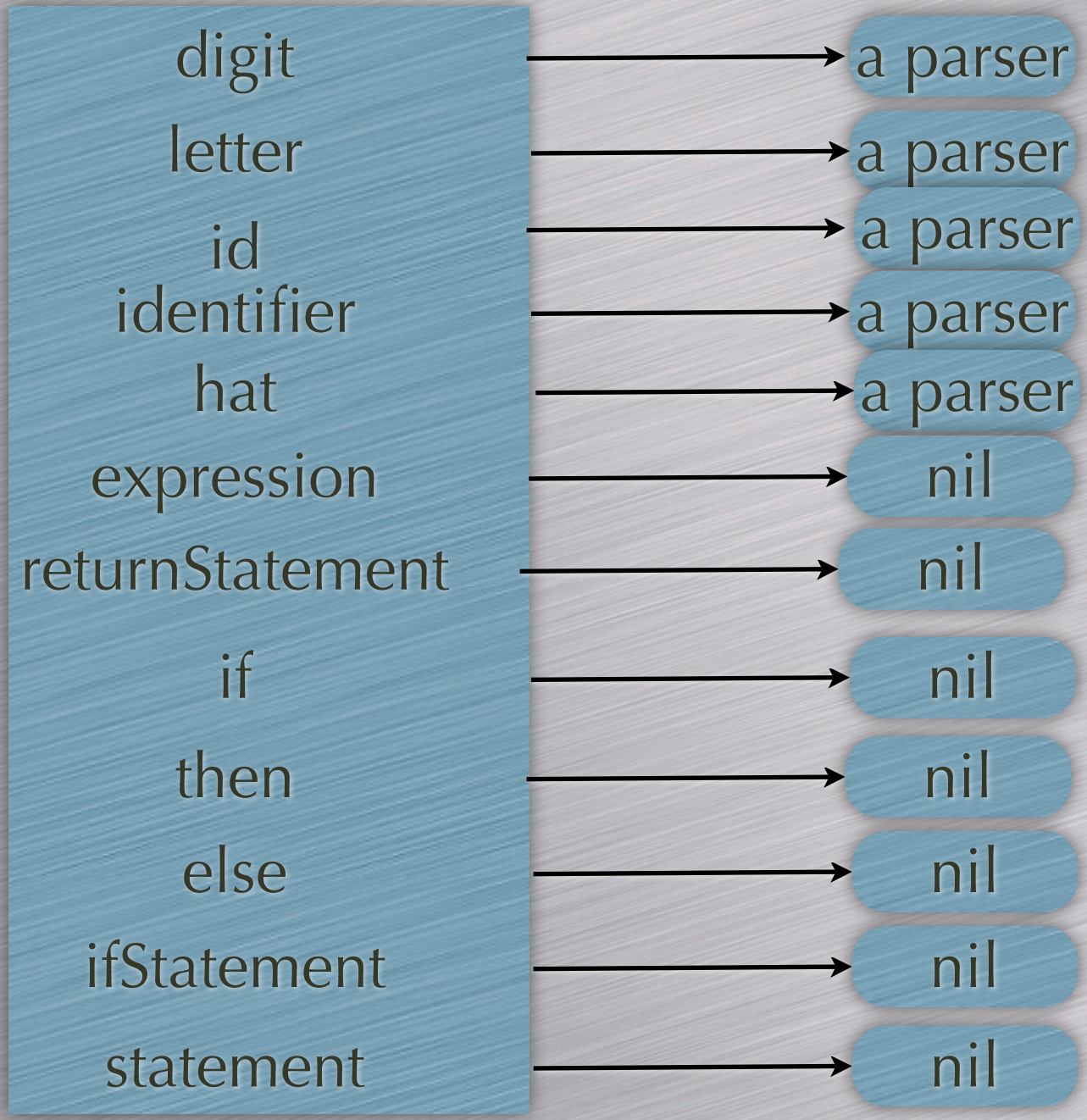
Mutual Recursion



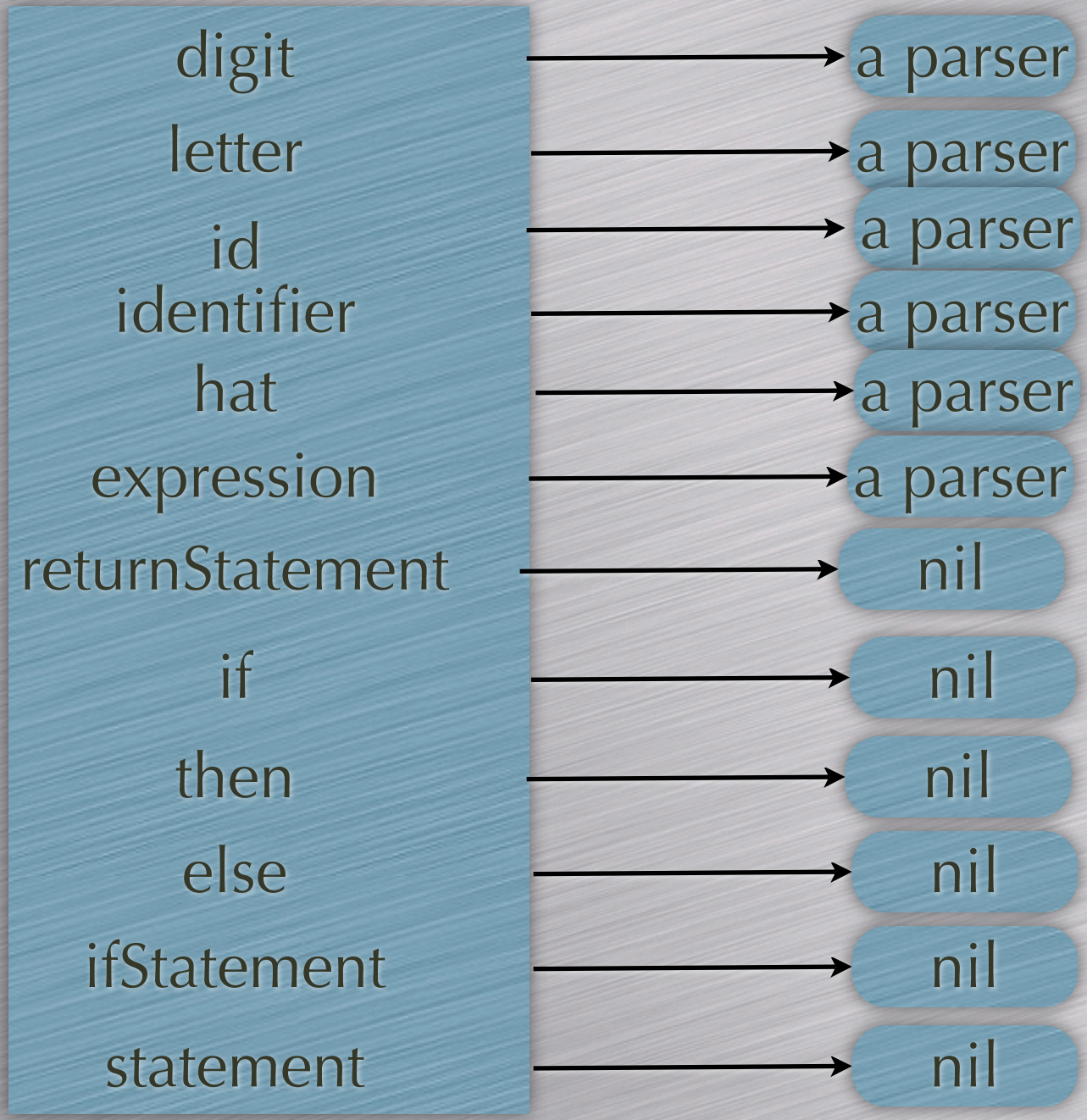
Mutual Recursion



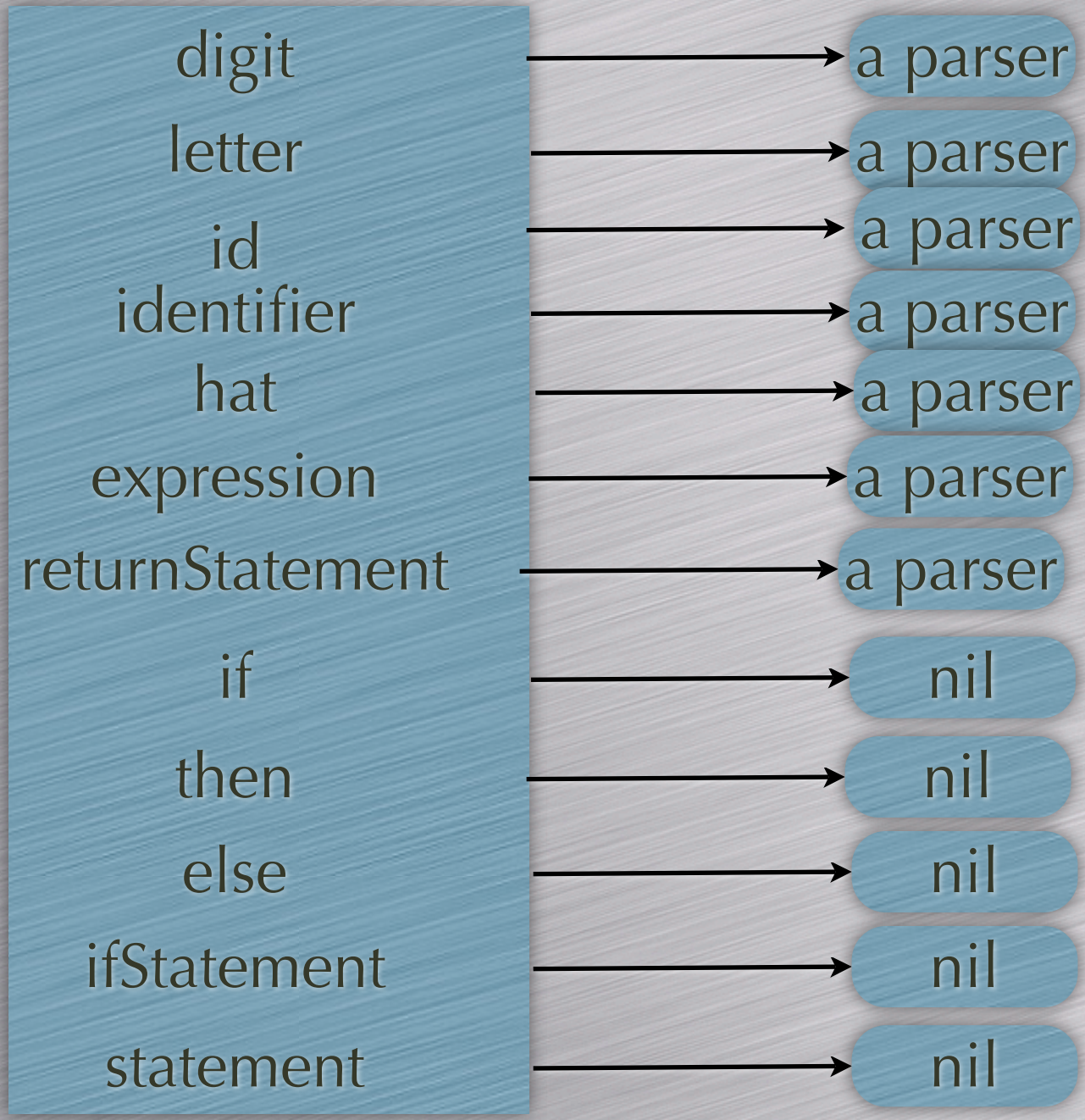
Mutual Recursion



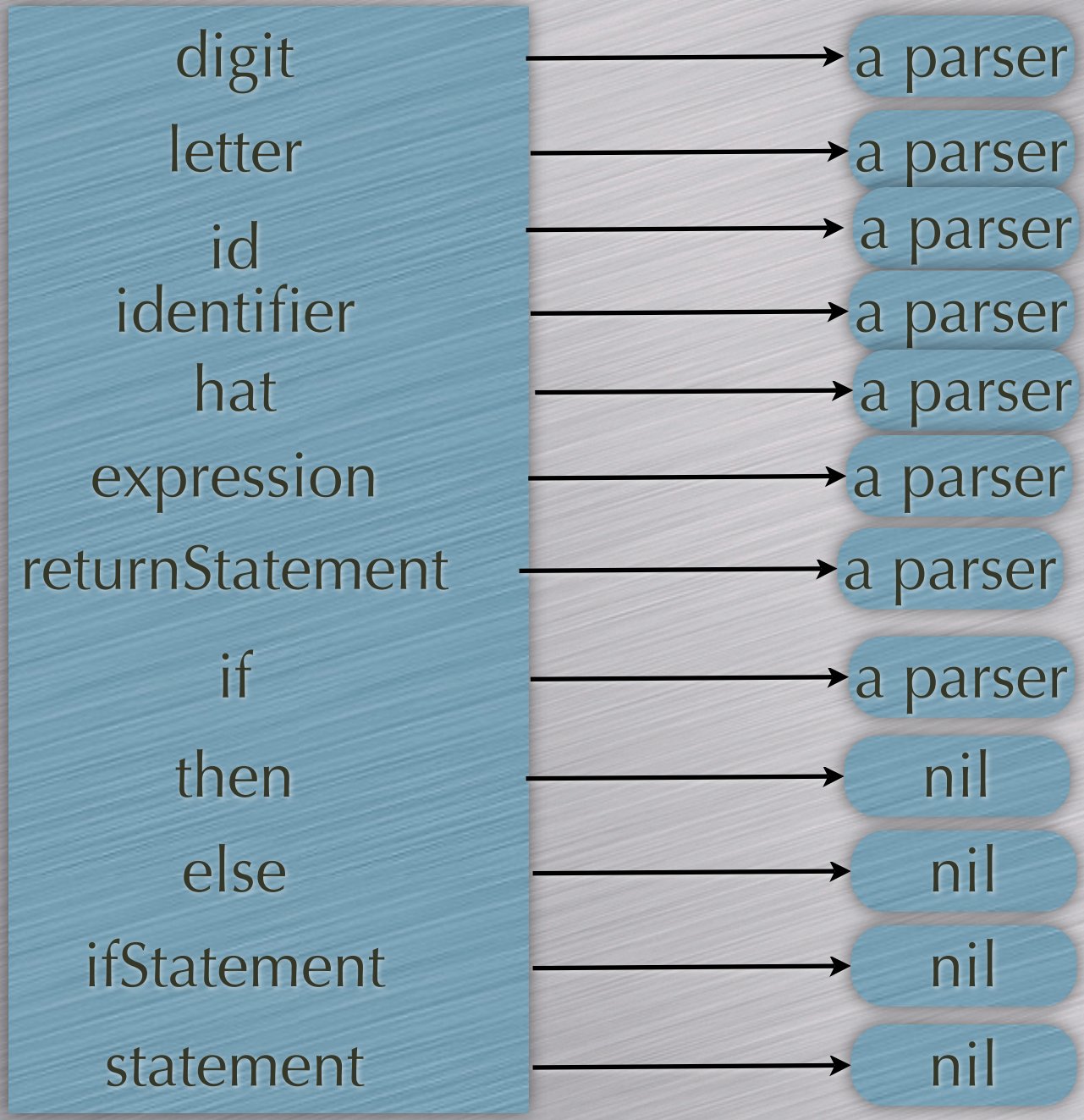
Mutual Recursion



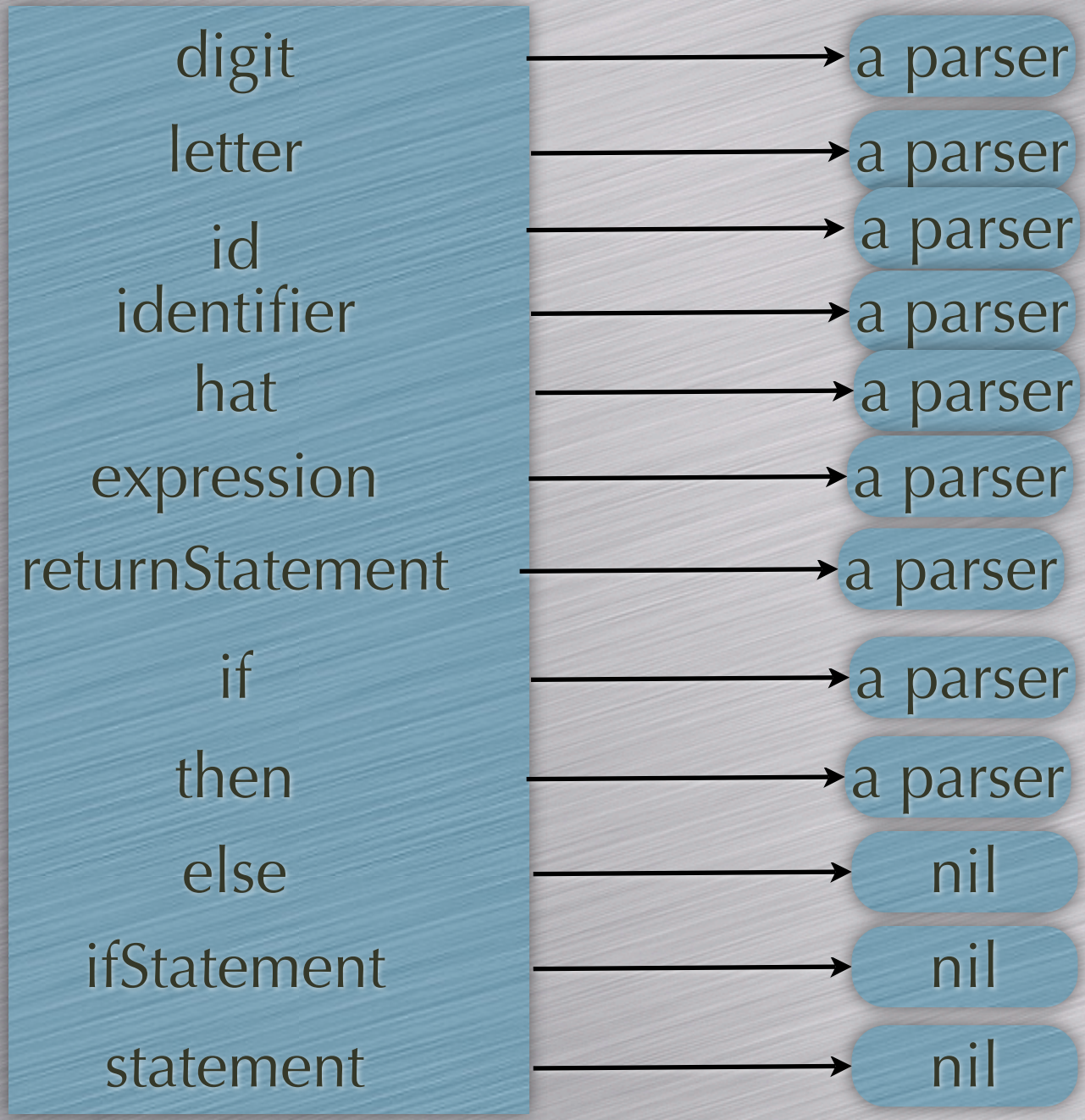
Mutual Recursion



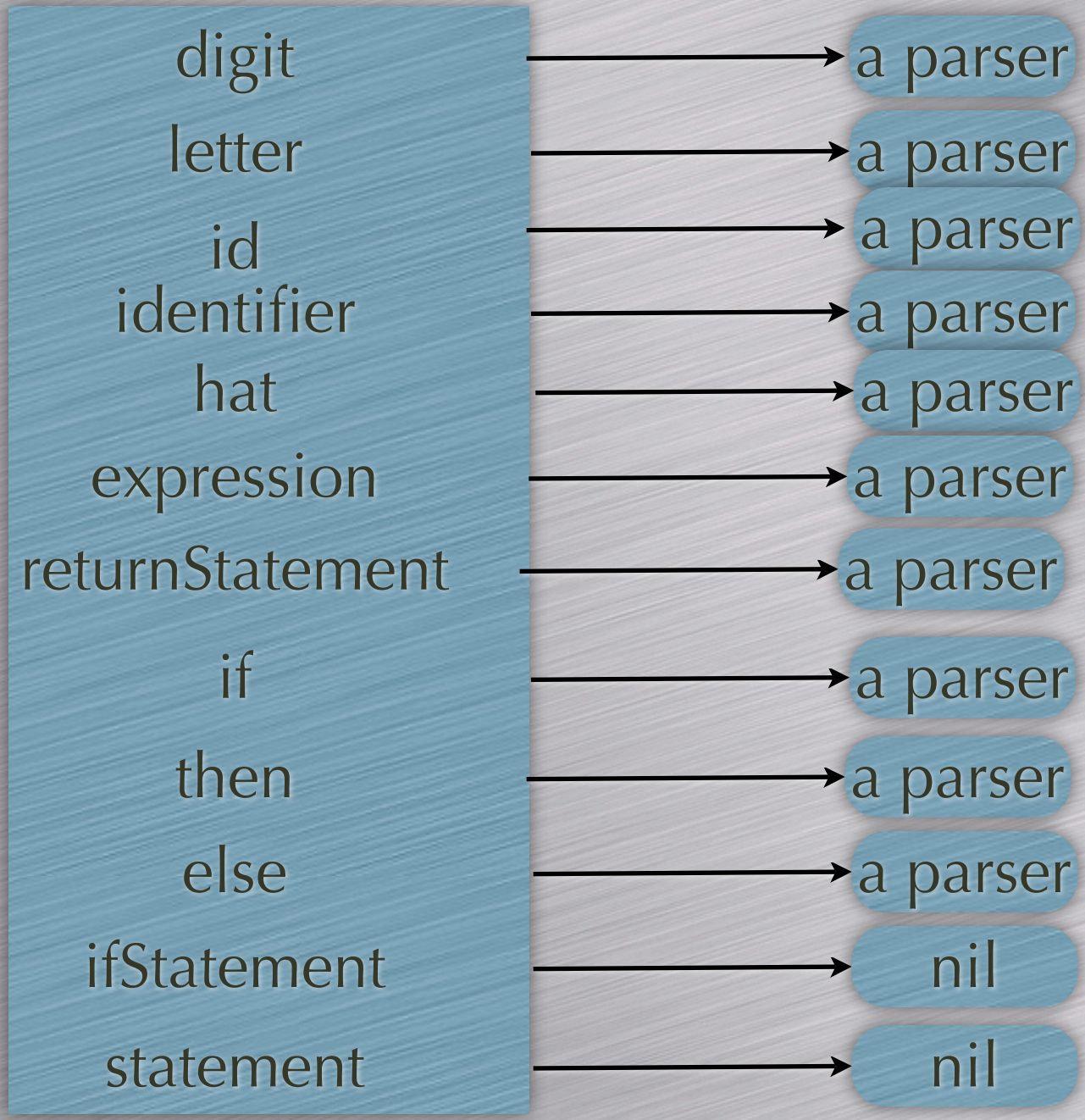
Mutual Recursion



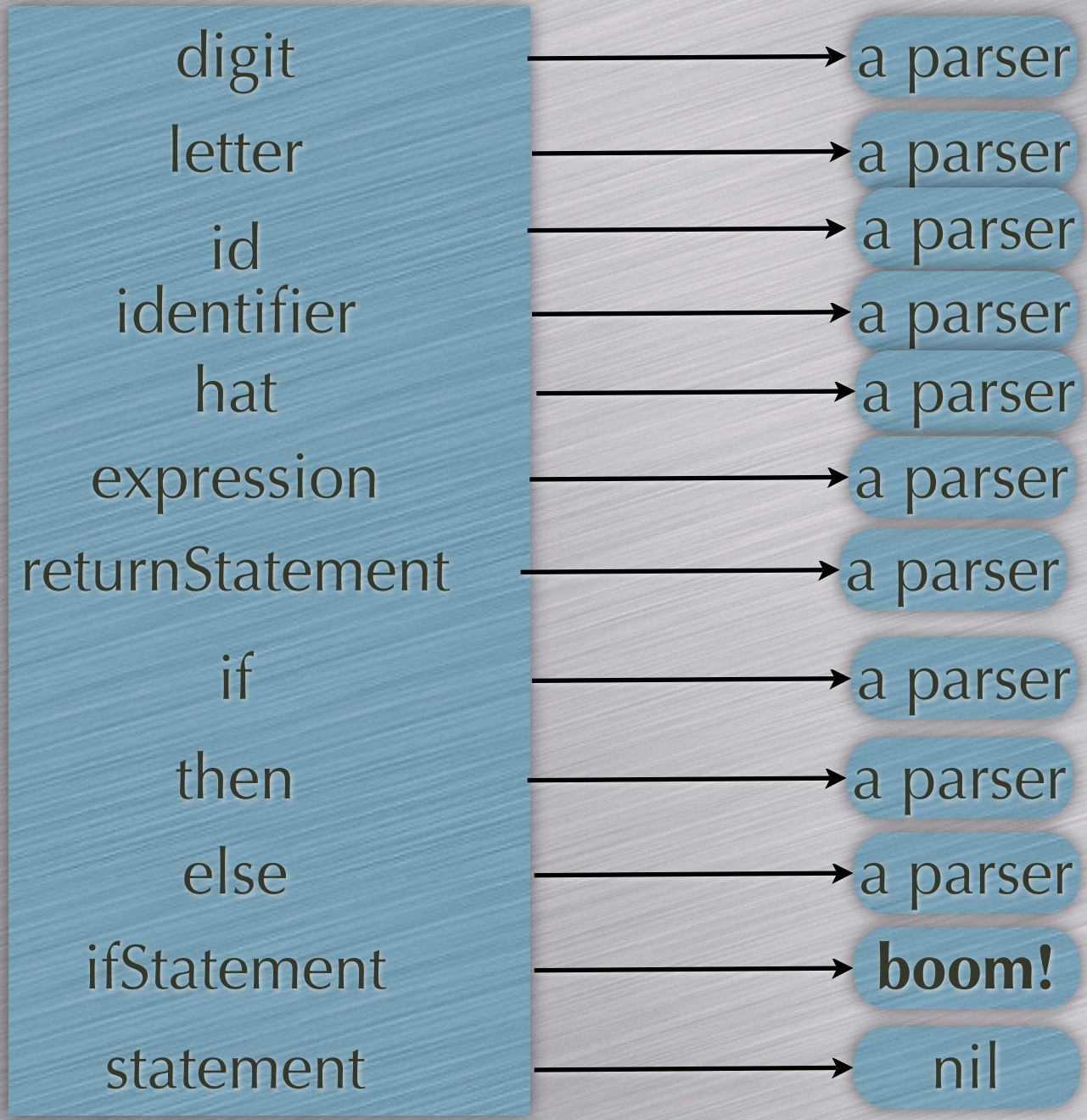
Mutual Recursion



Mutual Recursion



Mutual Recursion



Mutual Recursion

- Addressed with laziness in Haskell
- Can be addressed with closures

ifStatement = if, [expression], [then], [statement],

[else], [statement].

statement = ifStatement | [returnStatement].

Mutual Recursion

- Addressed with laziness in Haskell
- Can be addressed with closures

ifStatement = if, [expression], [then], [statement],

[else], [statement].

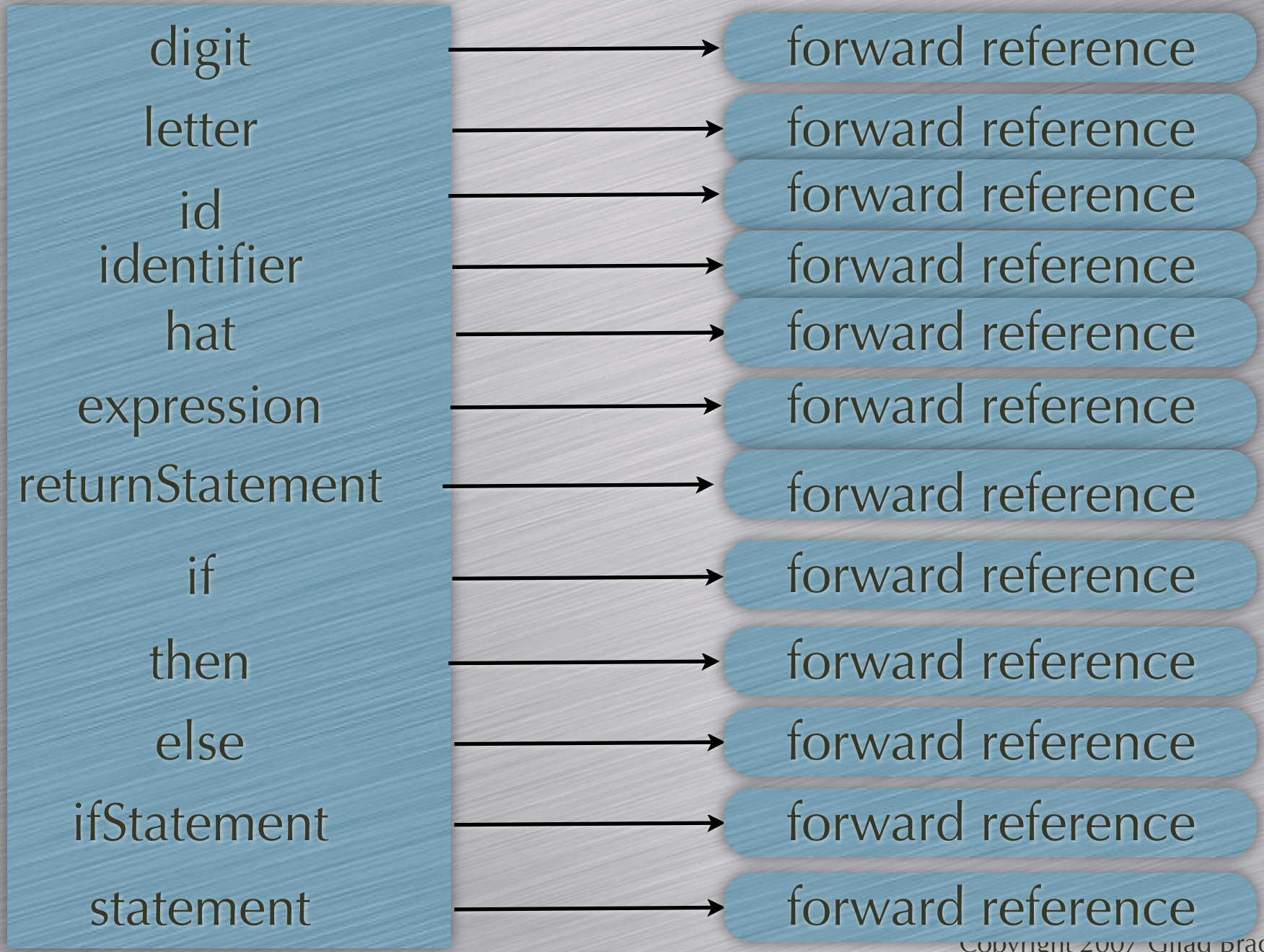
statement = ifStatement | [returnStatement].

Being Reflective rather than Lazy

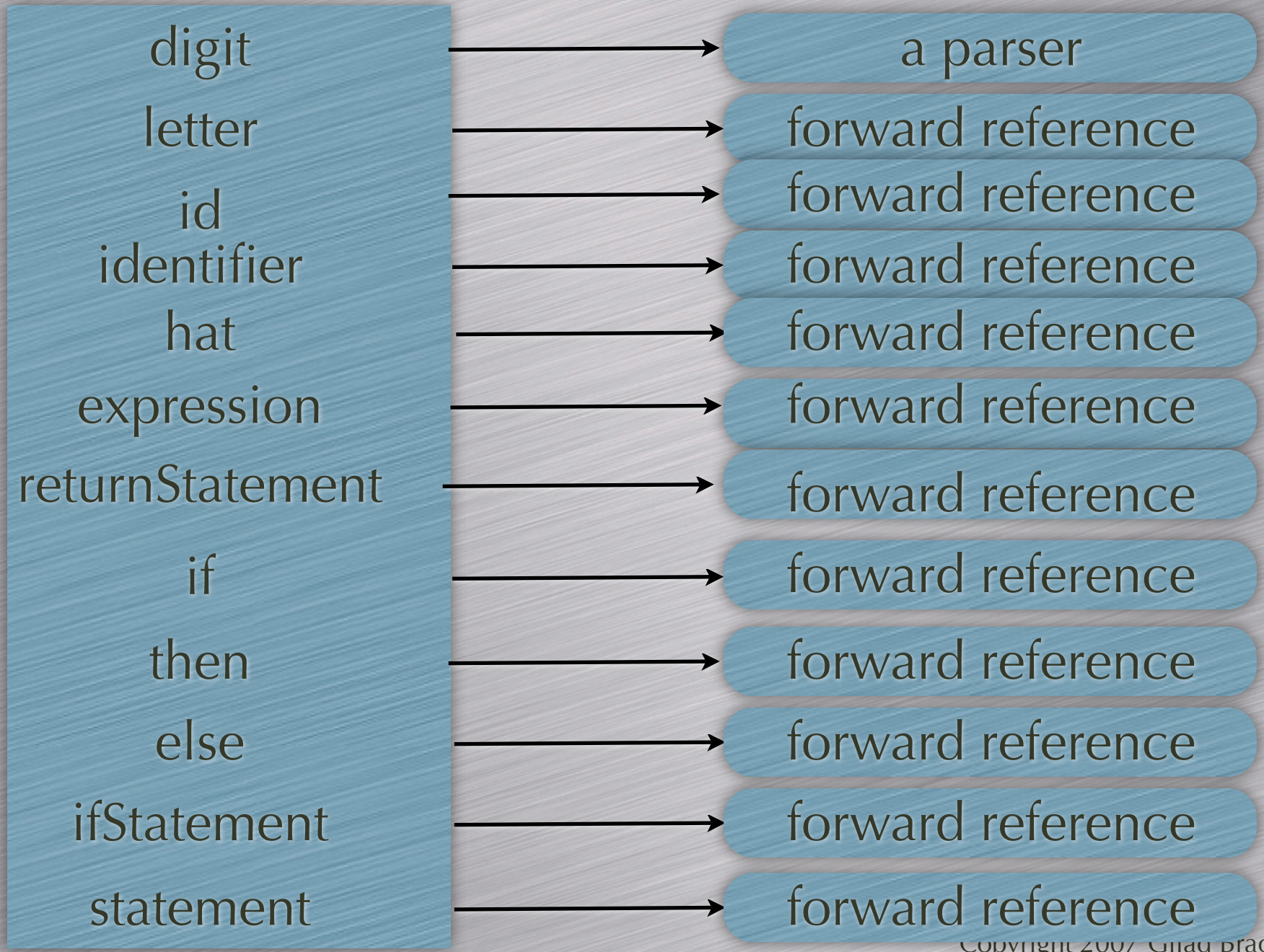
Use Reflection instead of Laziness.

Framework initializes all slots to “stand-in” parsers which act as forward references

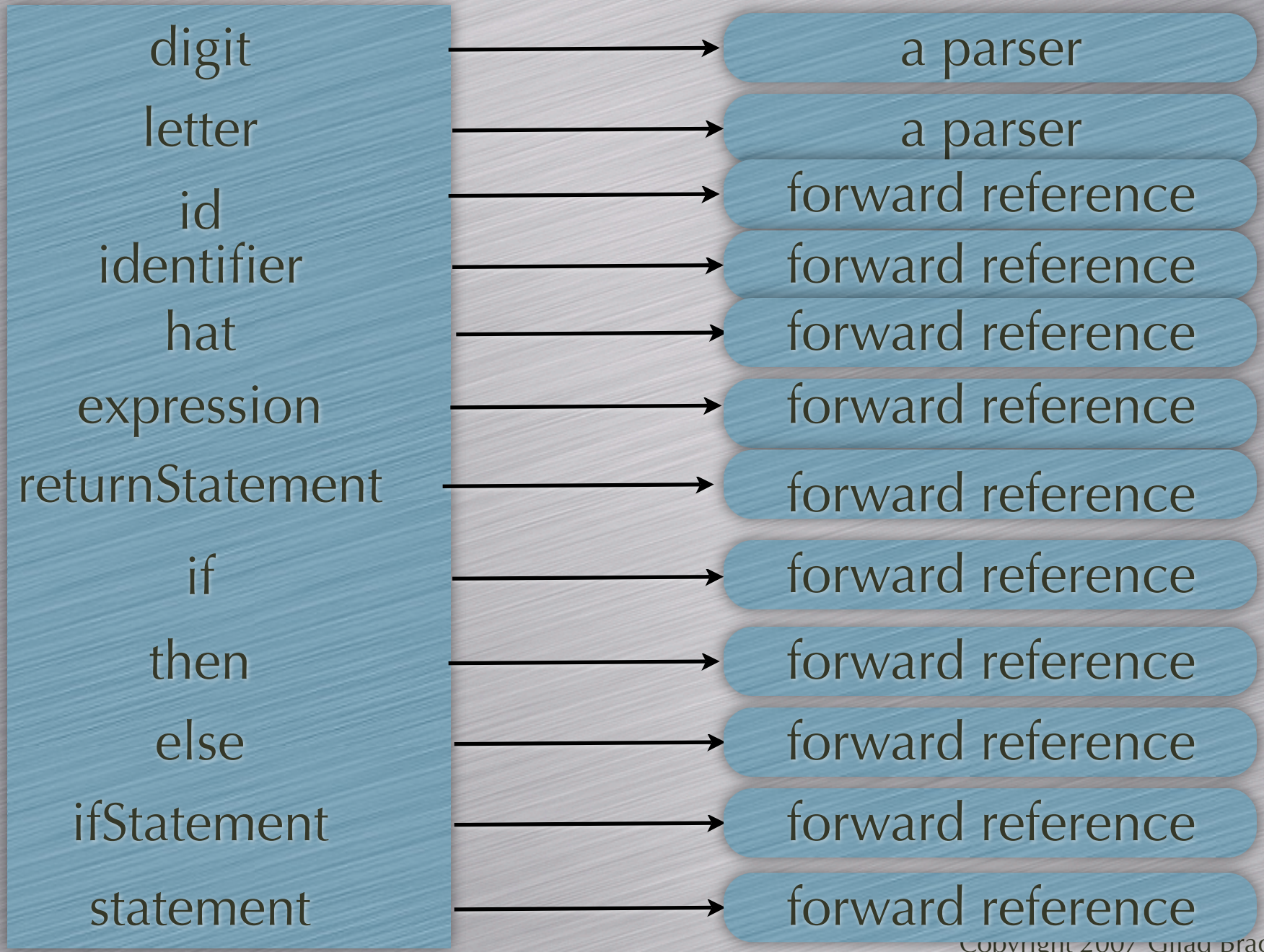
Mutual Recursion



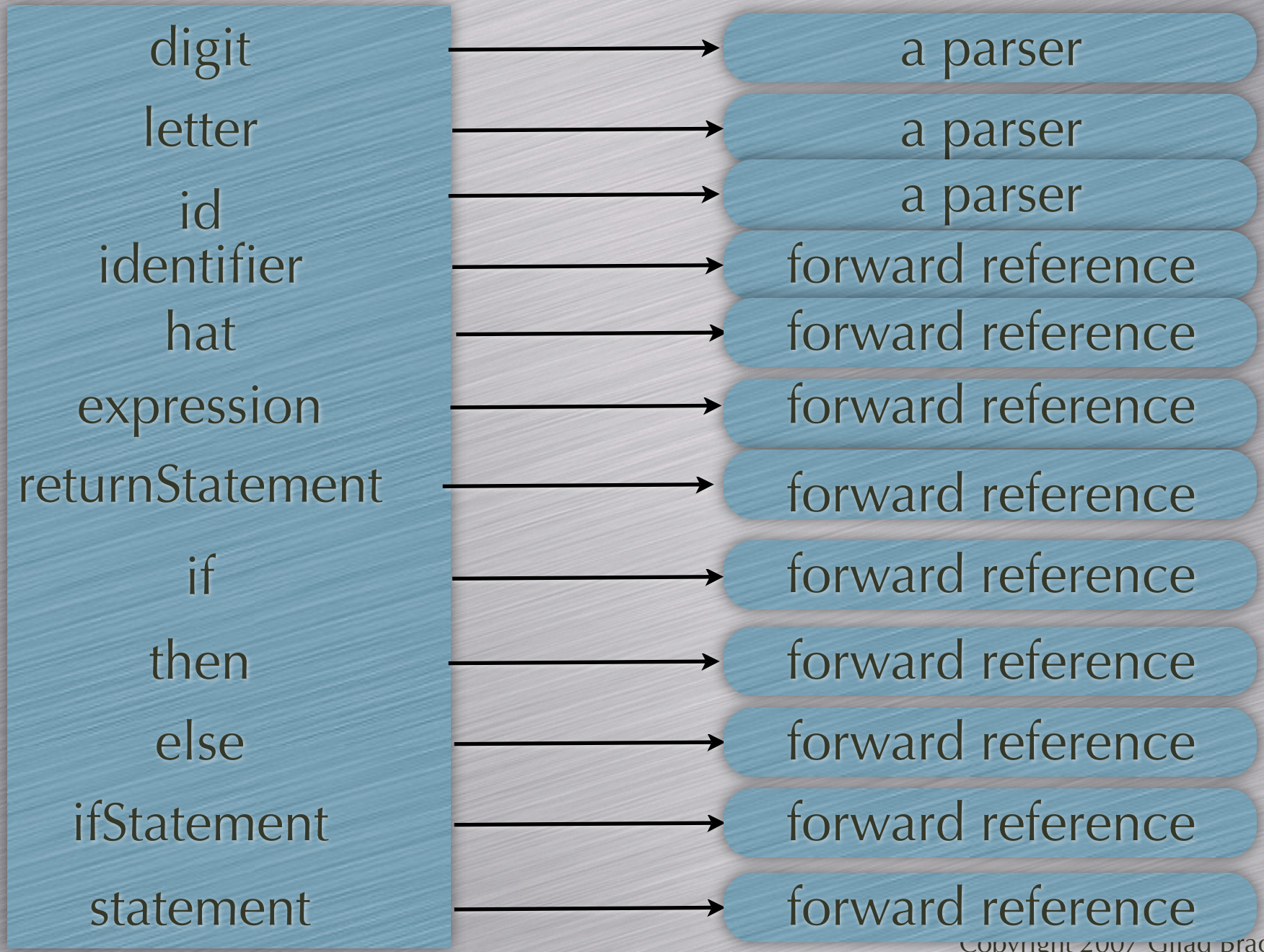
Mutual Recursion



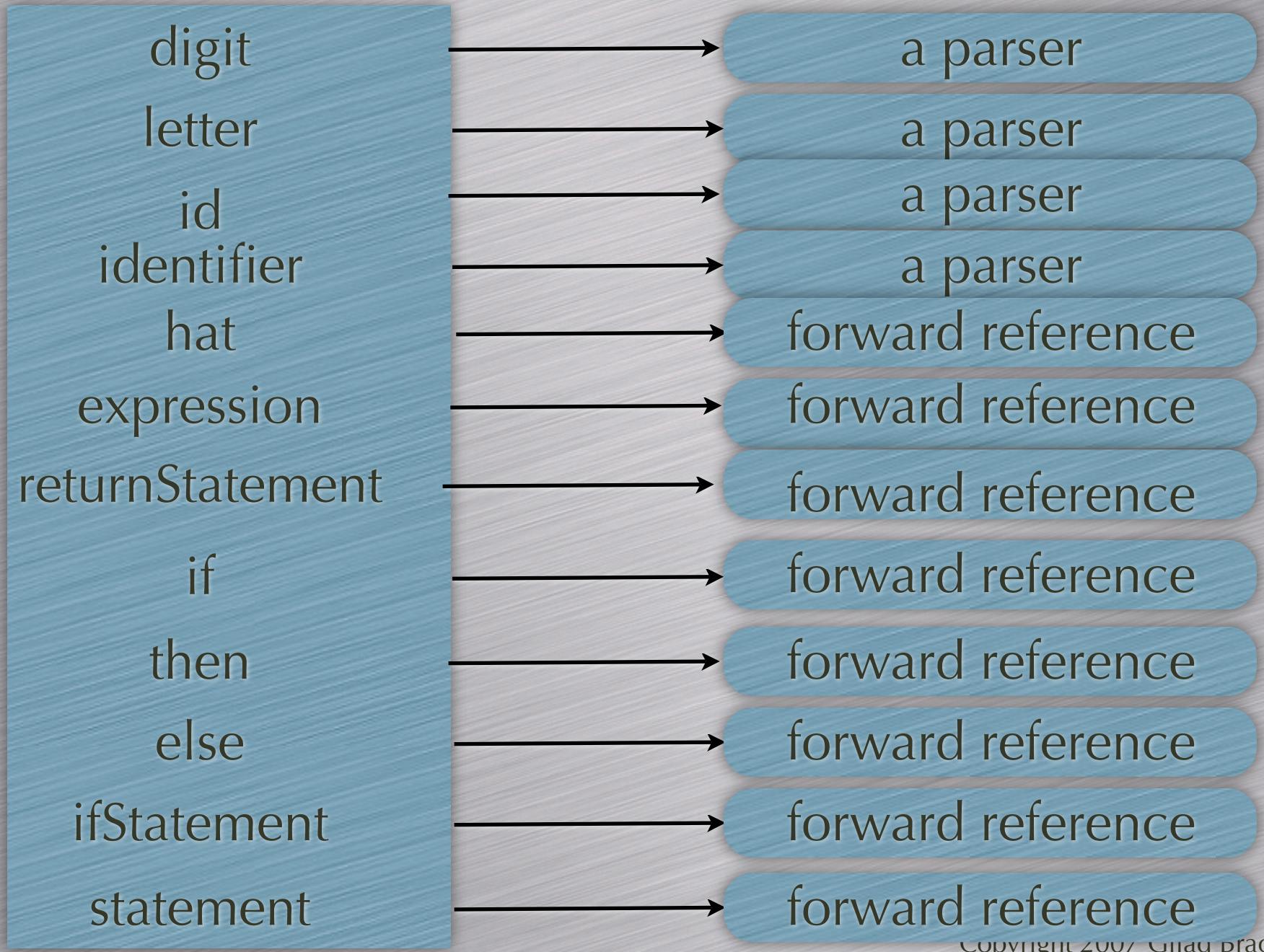
Mutual Recursion



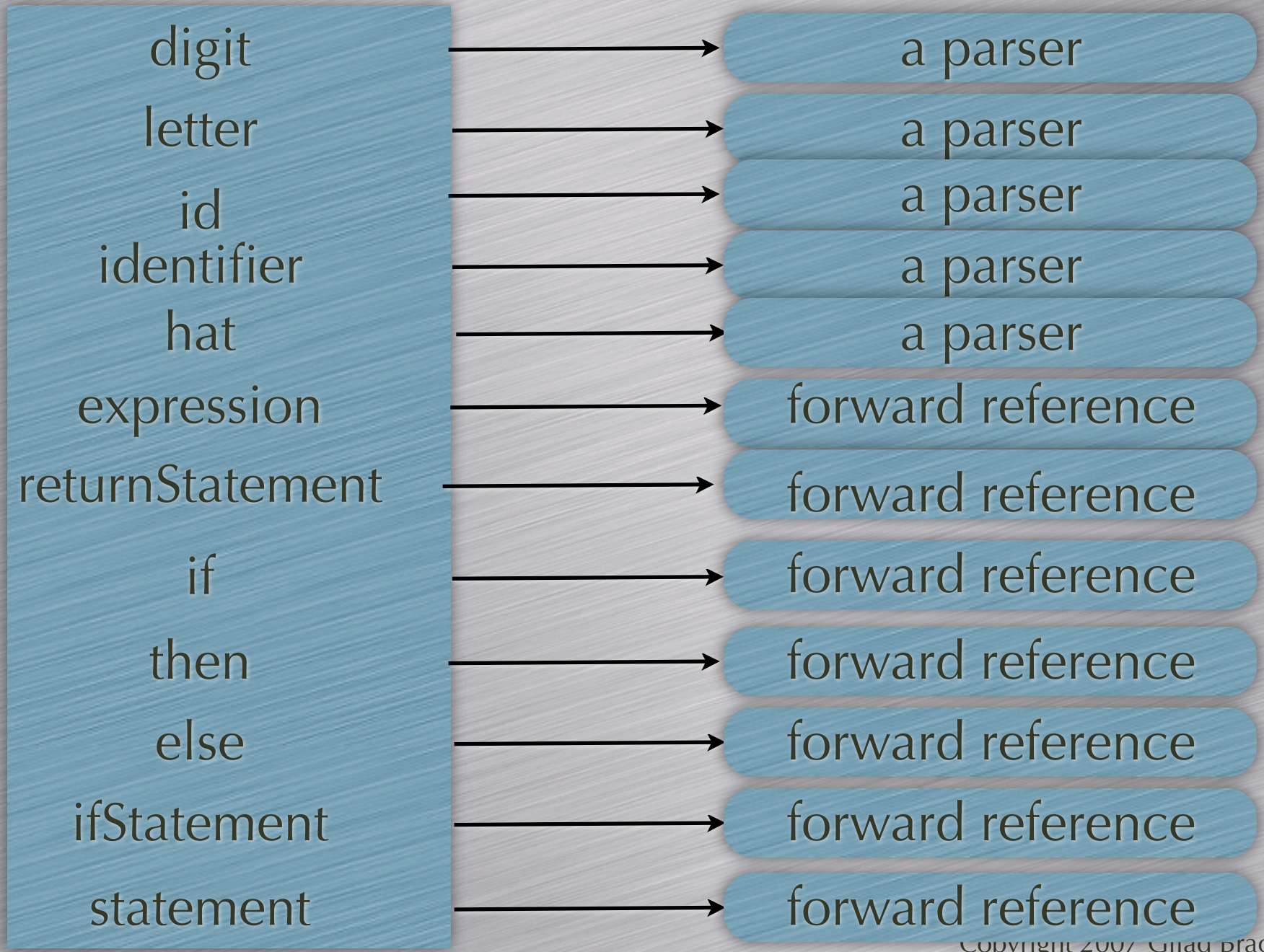
Mutual Recursion



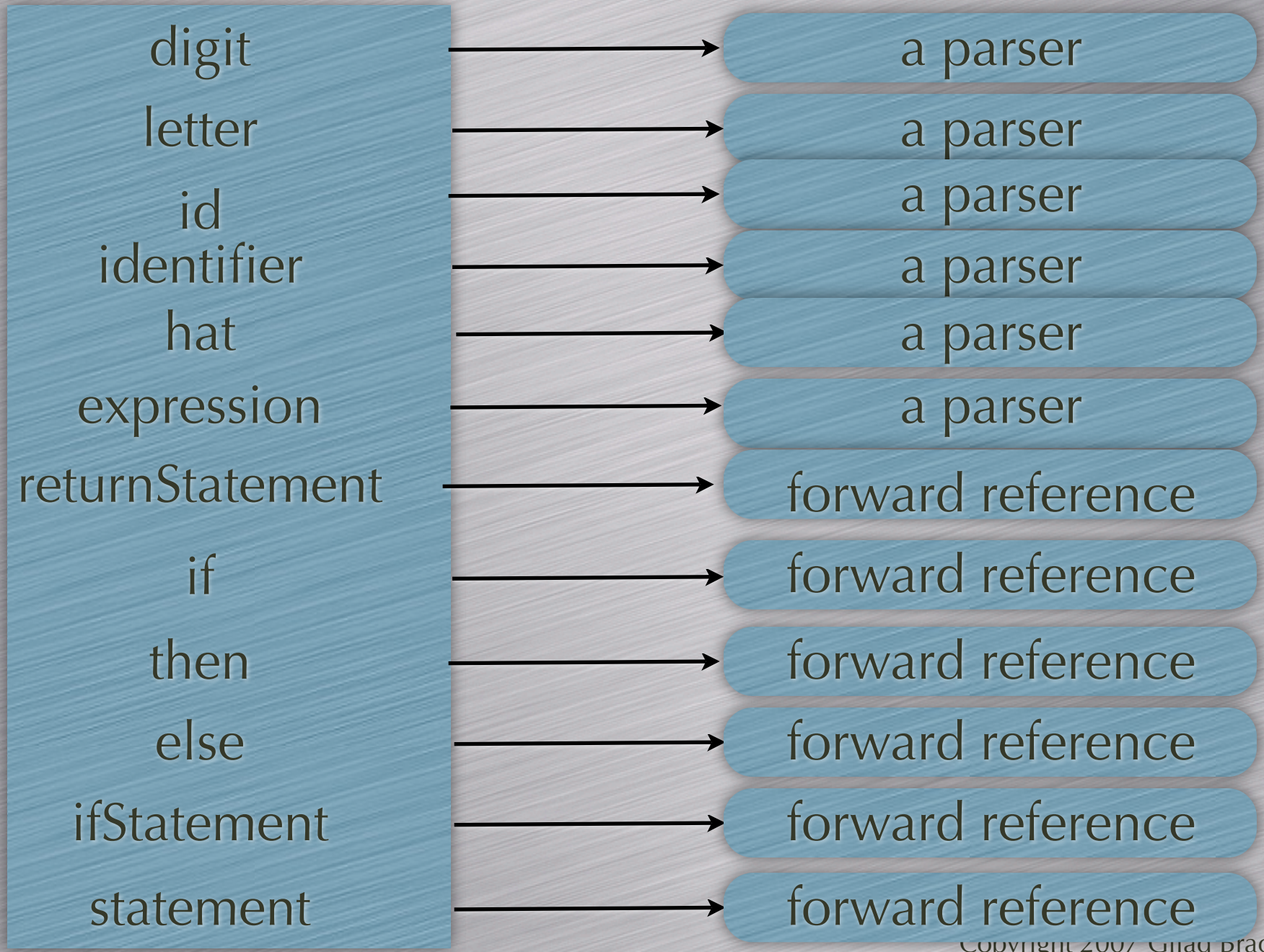
Mutual Recursion



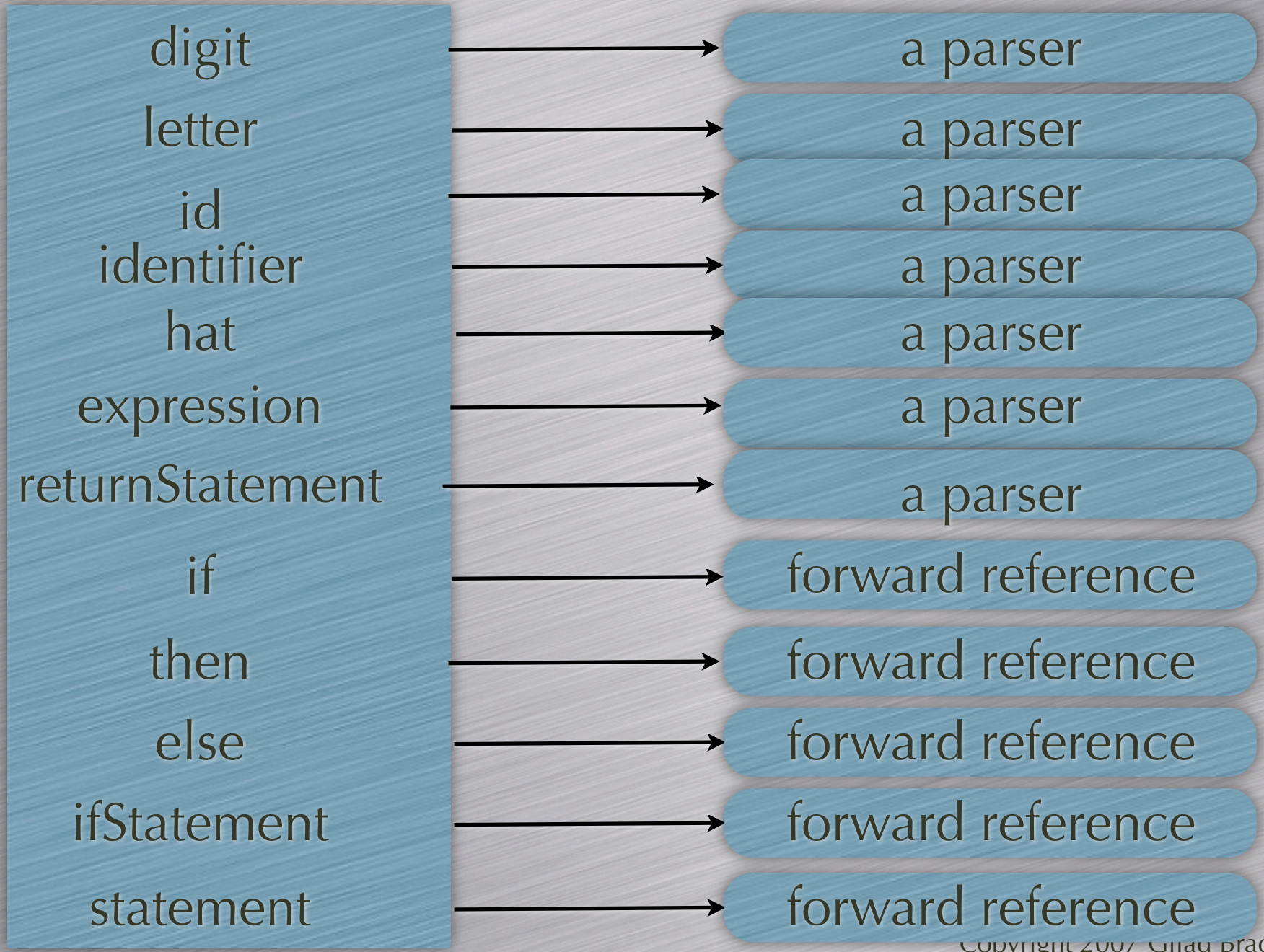
Mutual Recursion



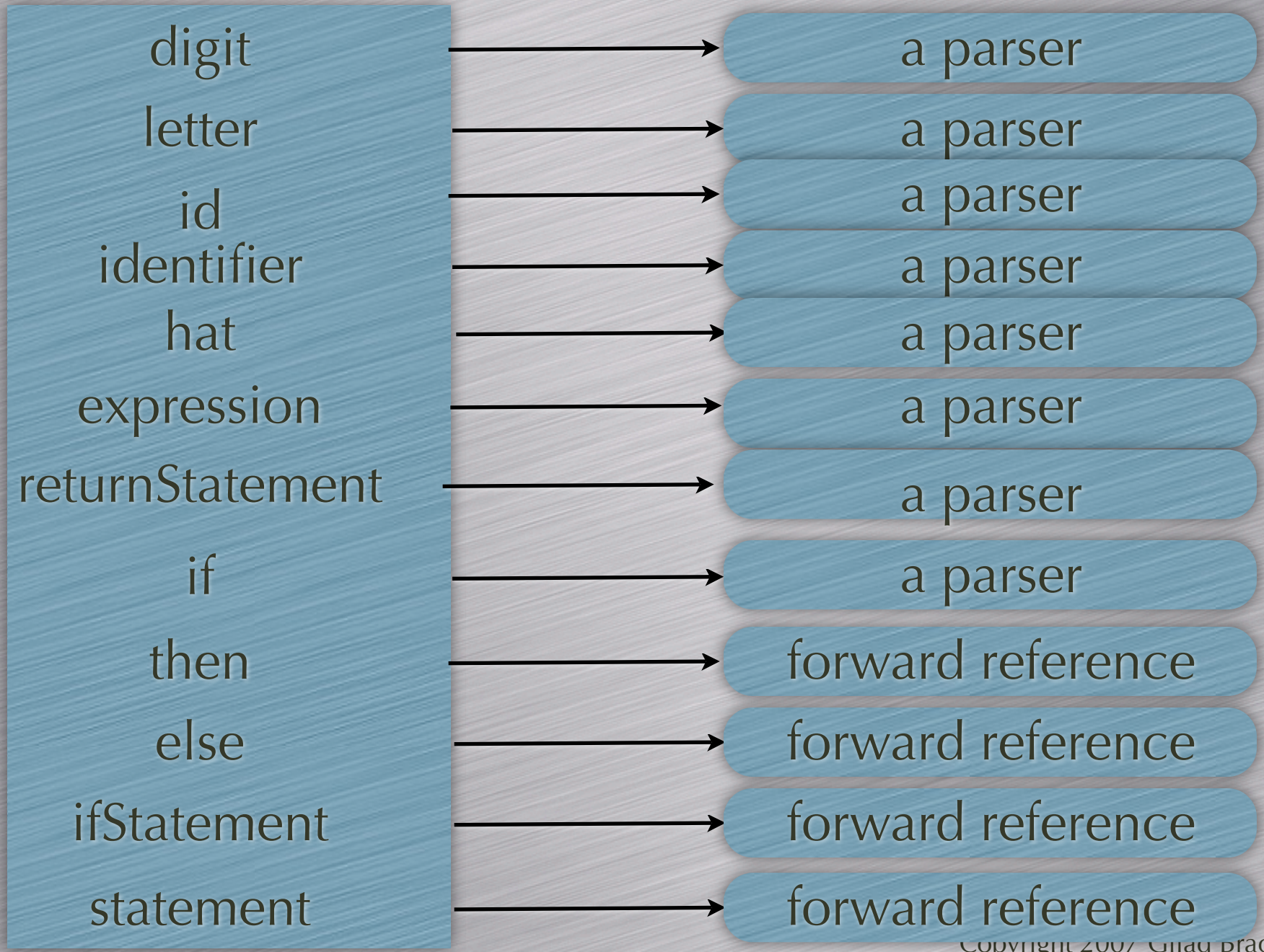
Mutual Recursion



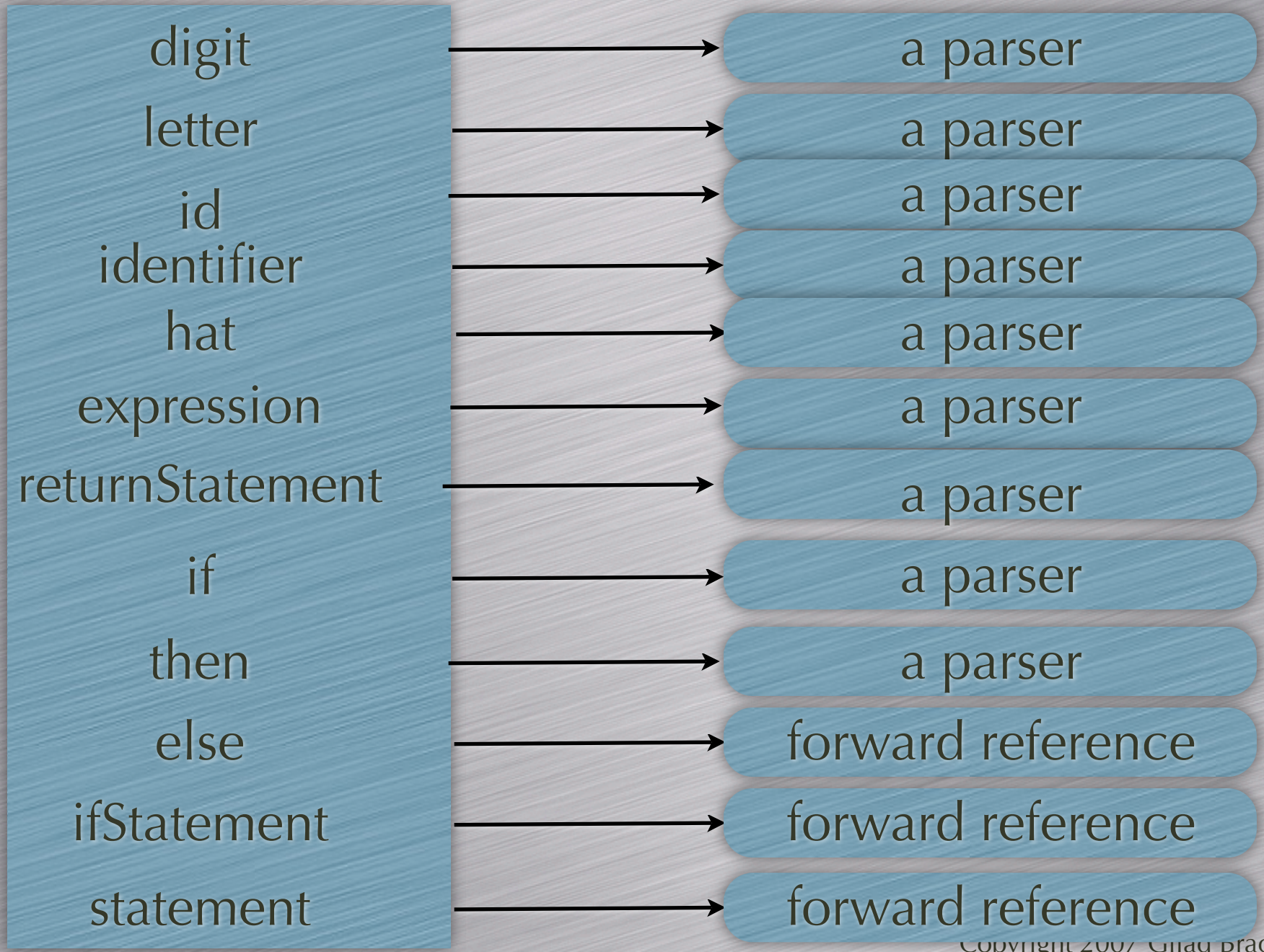
Mutual Recursion



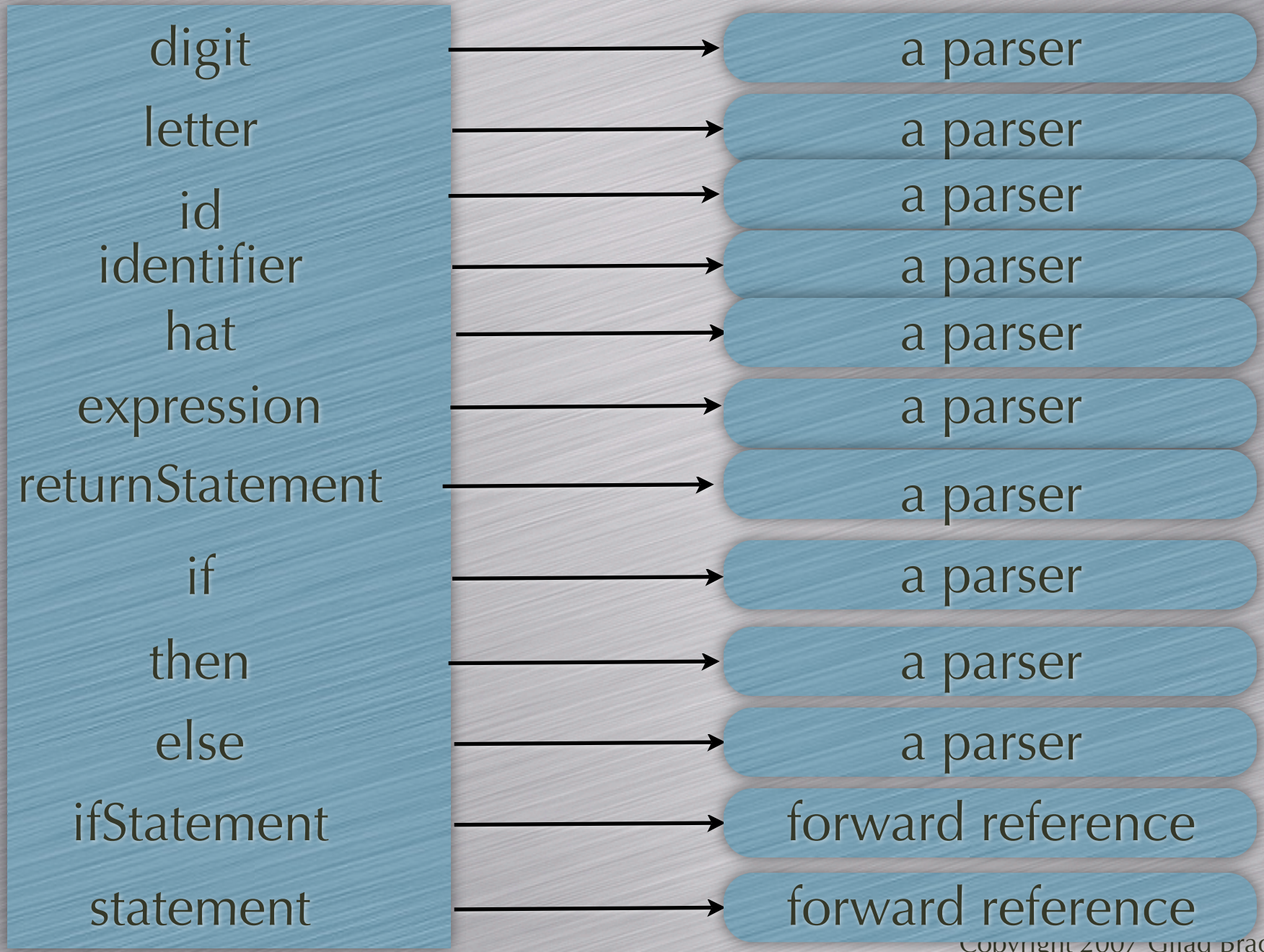
Mutual Recursion



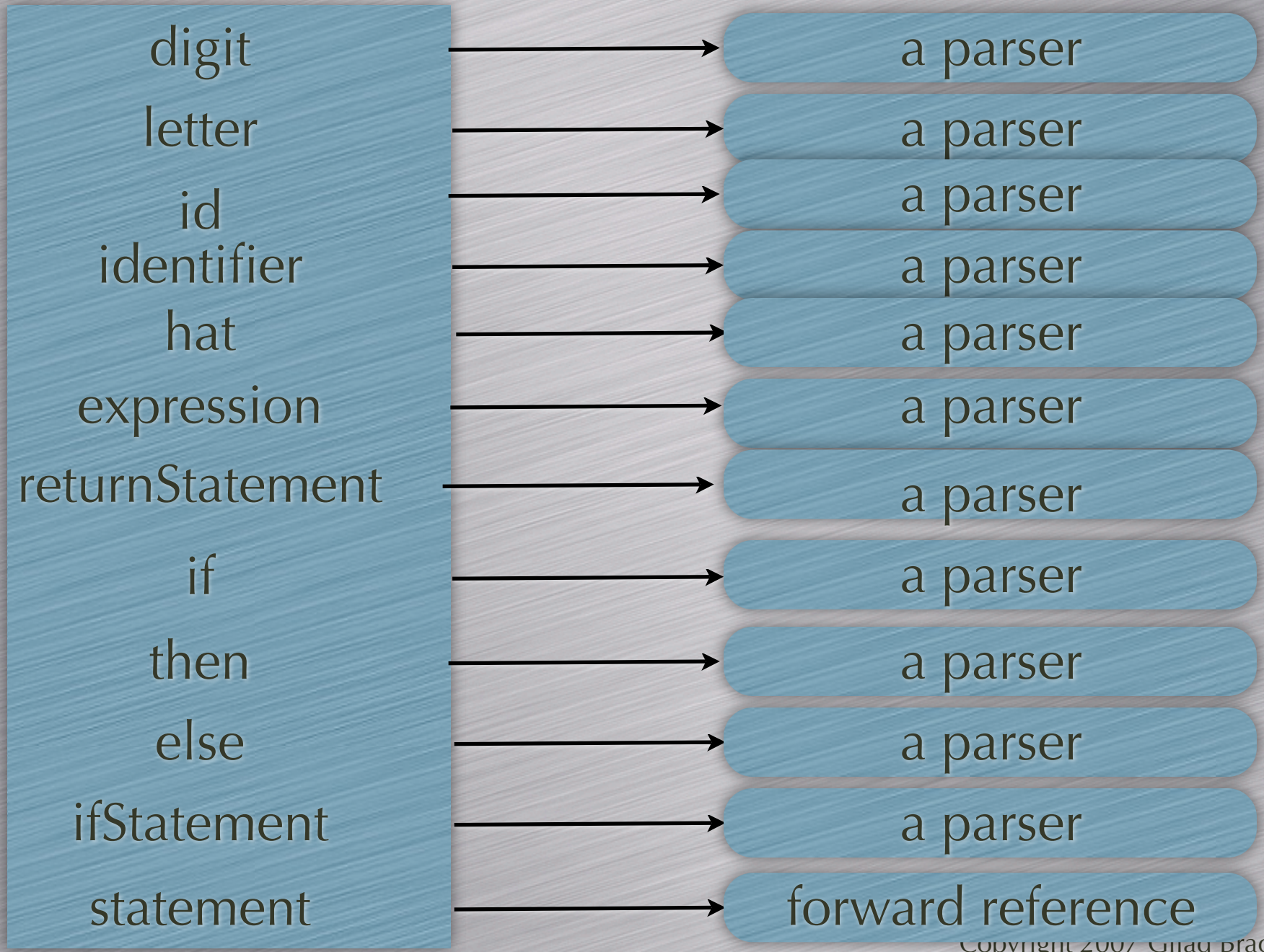
Mutual Recursion



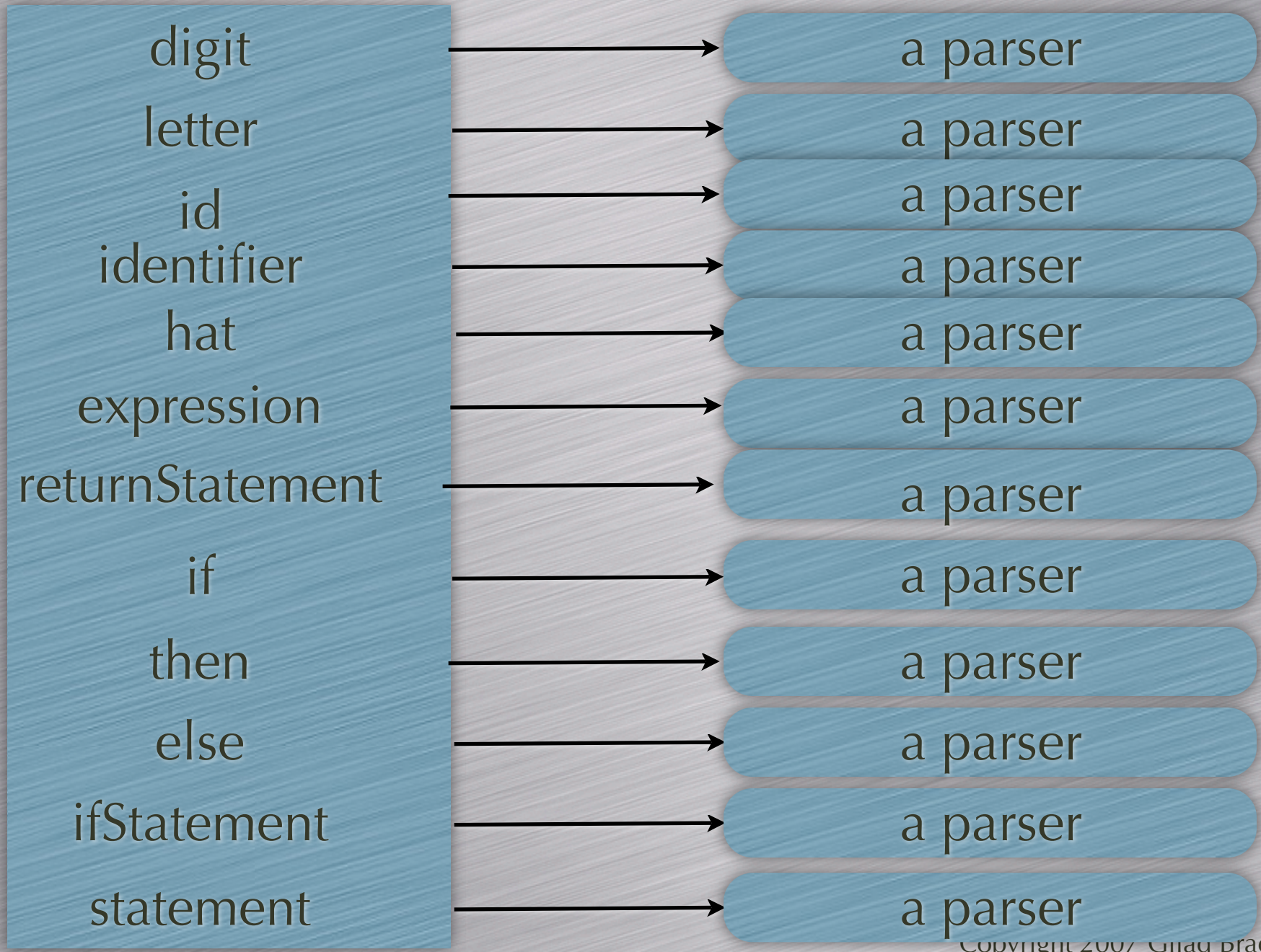
Mutual Recursion



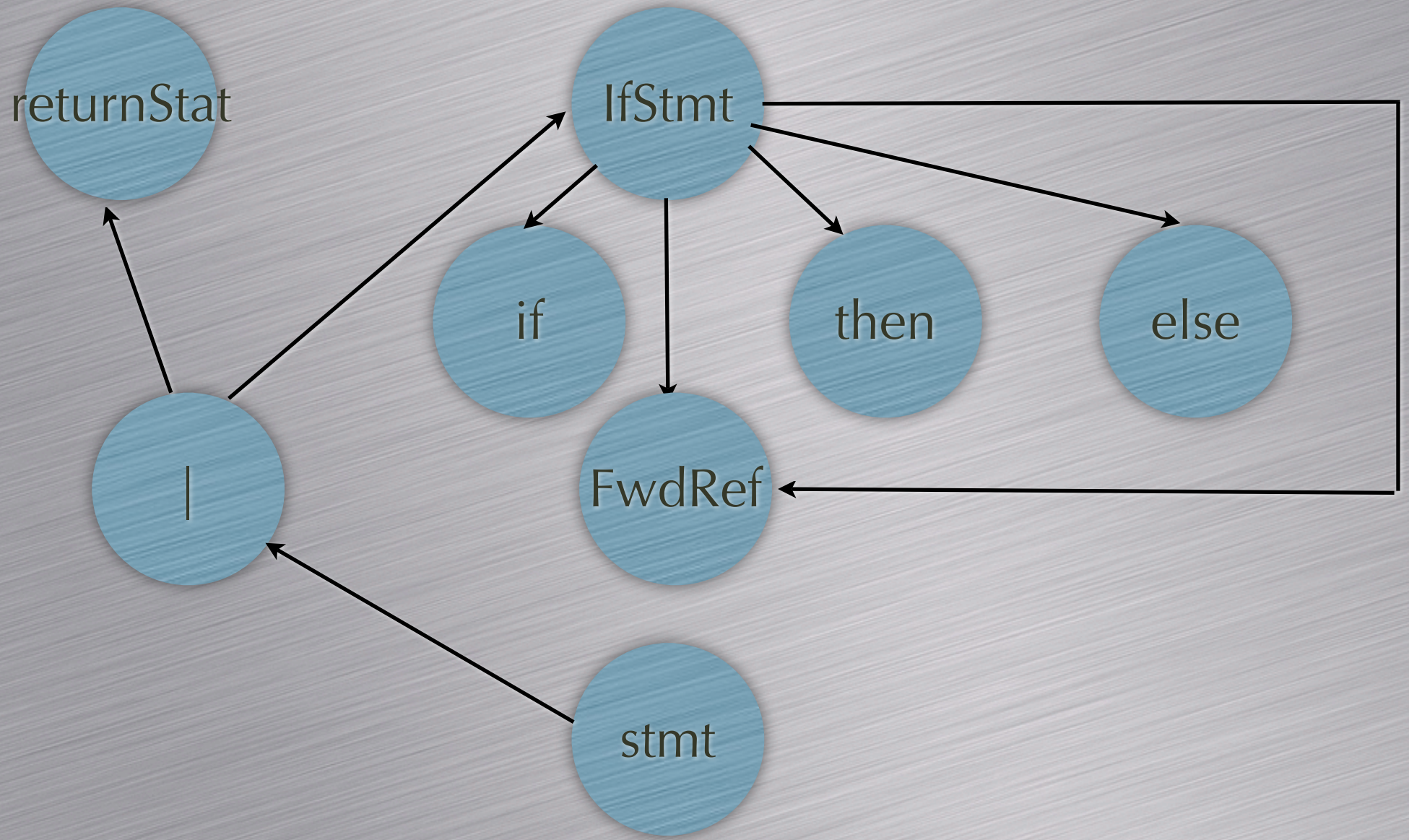
Mutual Recursion



Mutual Recursion



Mutual Recursion



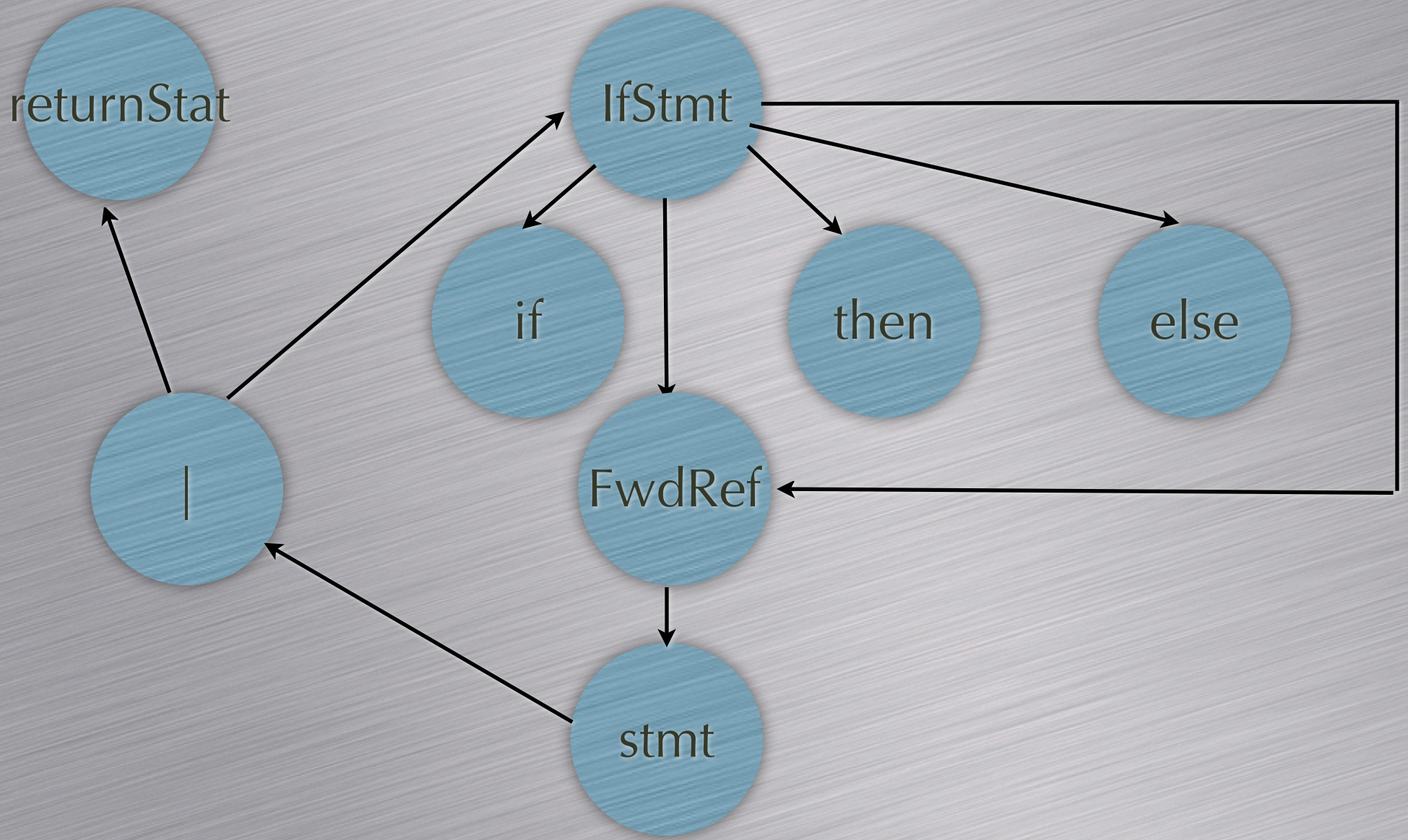
Being Reflective rather than Lazy

Use Reflection instead of Laziness.

Framework initializes all slots to “stand-in” parsers which act as forward references

When grammar is completely initialized, stand-ins are connected to originals and forward calls to them

Mutual Recursion



The Mirror is mightier than the Sloth

One can simulate a degree of laziness using reflection, but no amount of laziness will give us any reflection at all

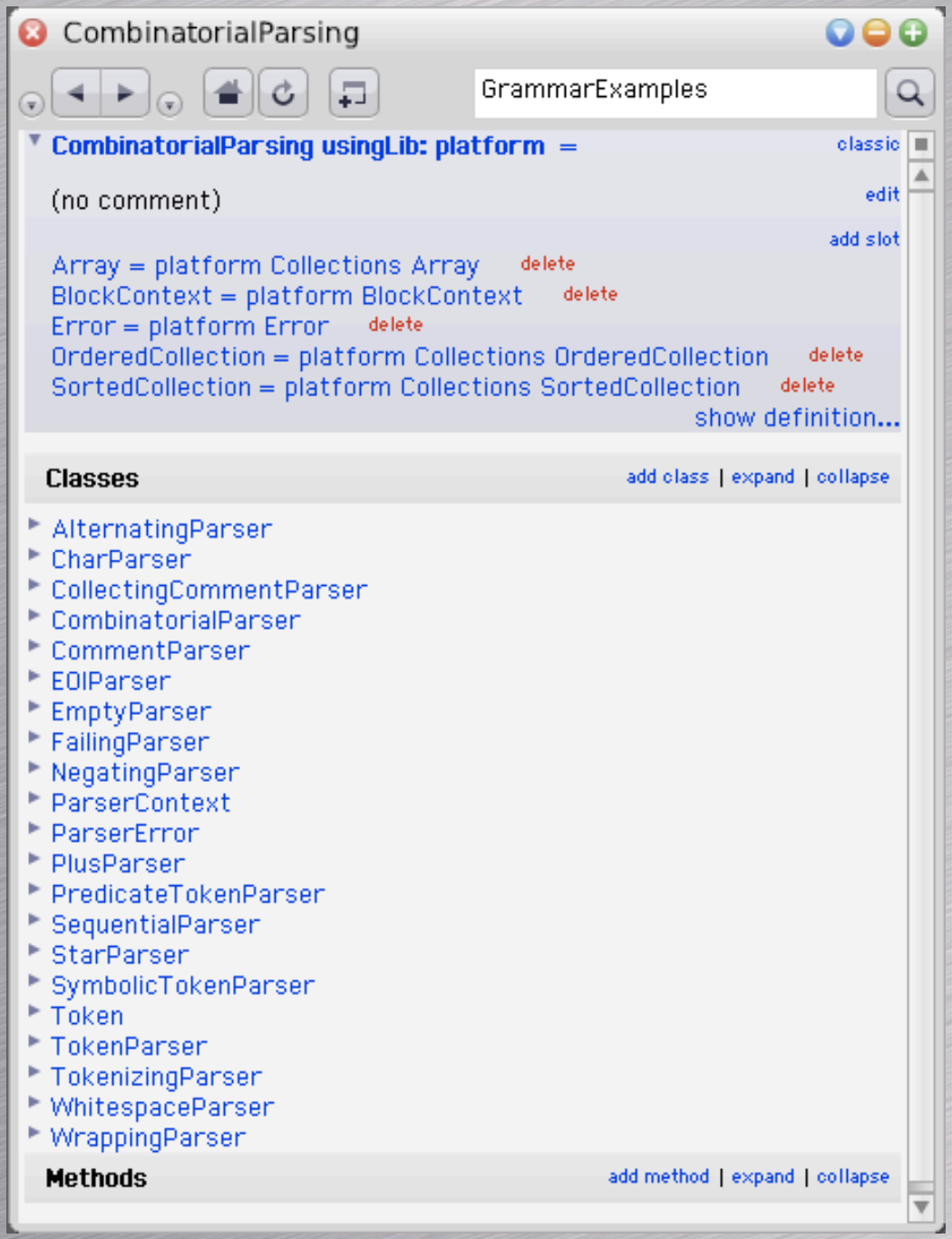
Composing Parsers

```
class ExampleParser2 = ExampleParser1 mixin |>
                                        ExampleGrammar2 ()
(
  ifStatement = (
    ^super ifStatement
      wrapper:[:ifKw :e :thenKw :s1 :elseKw :s2 |
                  IfStatAST new cond: e;
                                  trueBranch: s1;
                                  falseBranch: s2;
                                  start: ifKw start;
                                  end: s2 end
                ]
  )
)
```

Grammar is shared executable specification

- Shareable among several IDEs/ compilers that require distinct ASTs
- Shareable among other language tools: syntax colorizers, postprocessing tools

The Library

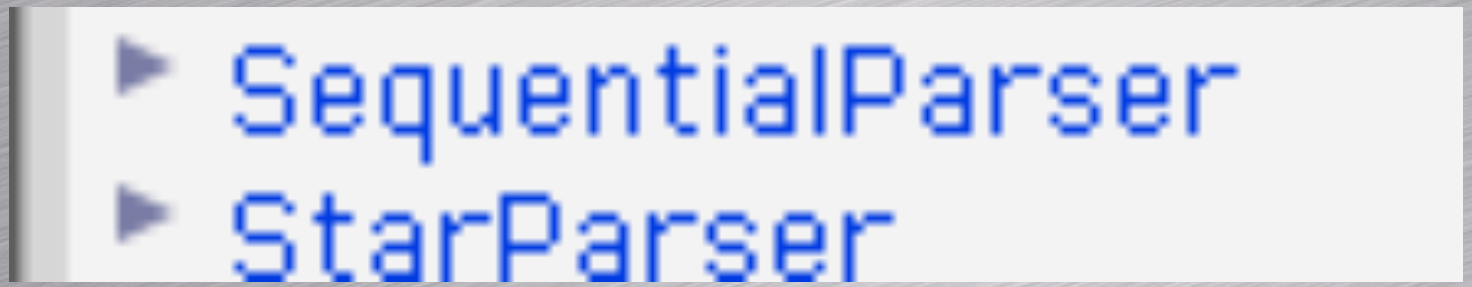


Nested Classes

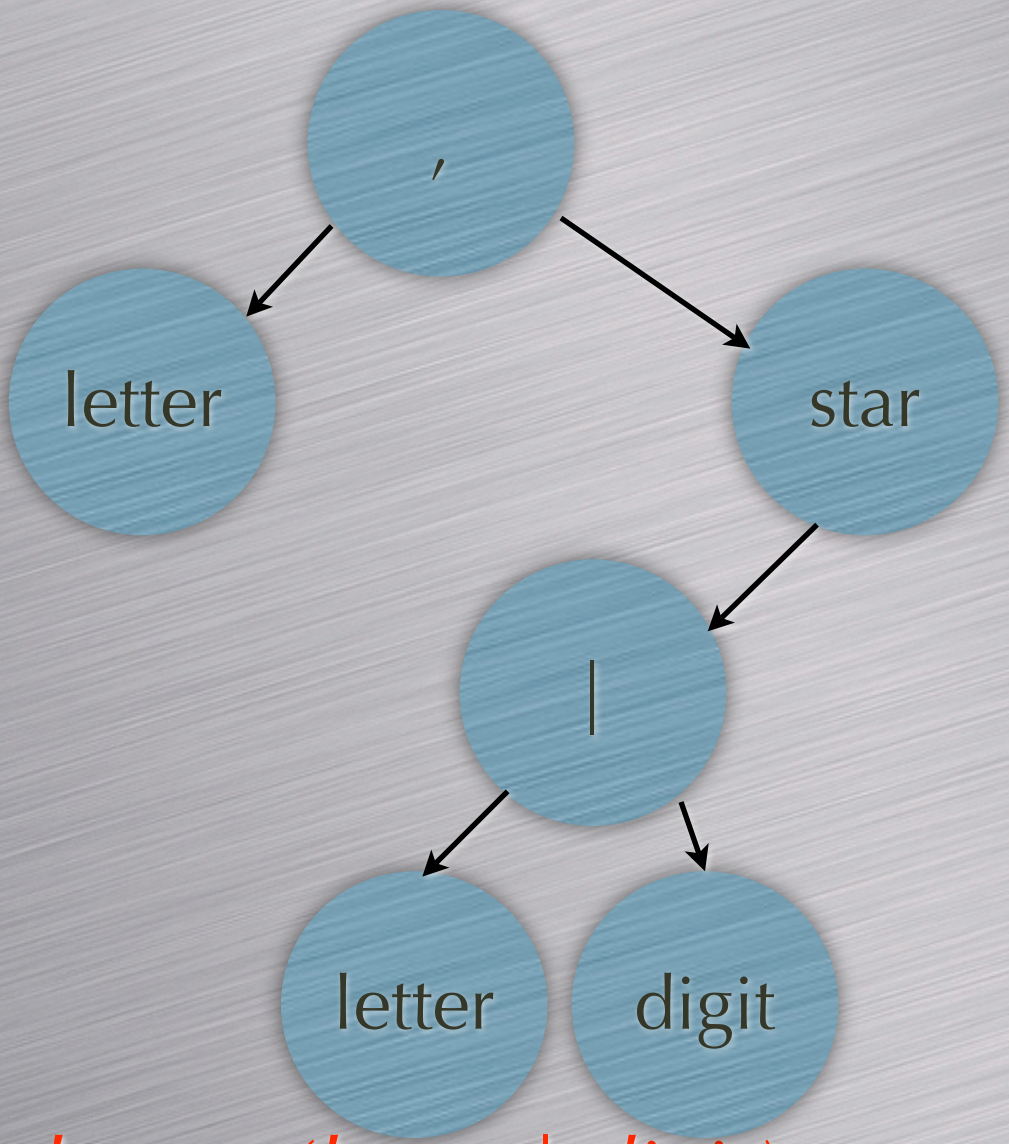
Classes add class | expand | collapse

- AlternatingParser
- CharParser
- CollectingCommentParser
- CombinatorialParser
- CommentParser
- EOIParser
- EmptyParser
- FailingParser
- NegatingParser
- ParserContext
- ParserError
- PlusParser
- PredicateTokenParser
- SequentialParser
- StarParser
- SymbolicTokenParser
- Token
- TokenParser
- TokenizingParser
- WhitespaceParser
- WrappingParser

Nested Classes In Detail



How it Works



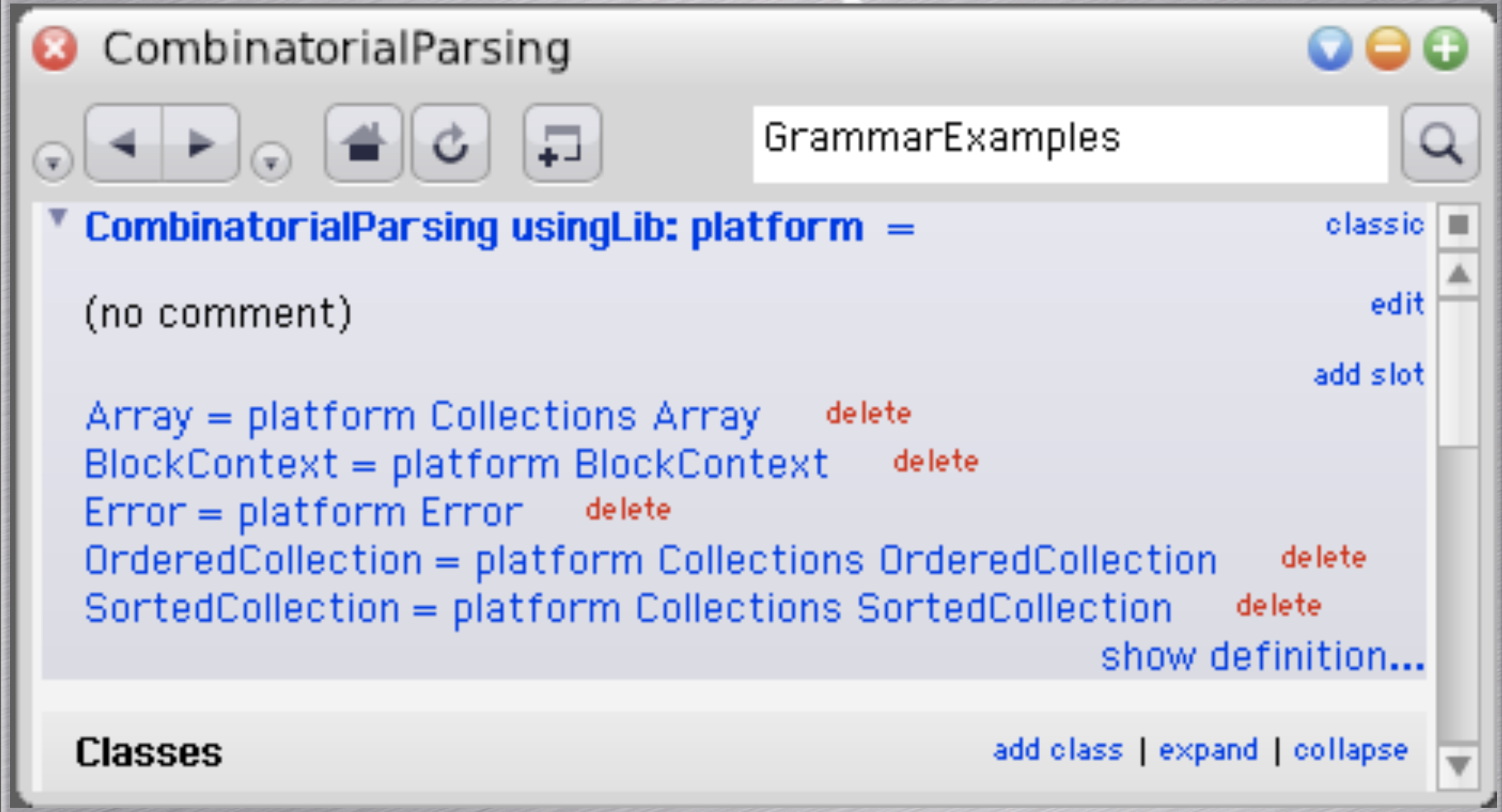
id = letter, (letter | digit) star.

Modules

```
class CombinatorialParsing usingLib: platform = (...)(
  class AlternatingParser = CombinatorialParser (...)(...)

  public class CombinatorialParser = ()( ...
     | p = (^AlternatingParser either: self or: p)
    ....
  )
  ....
)
```

Modules up Close



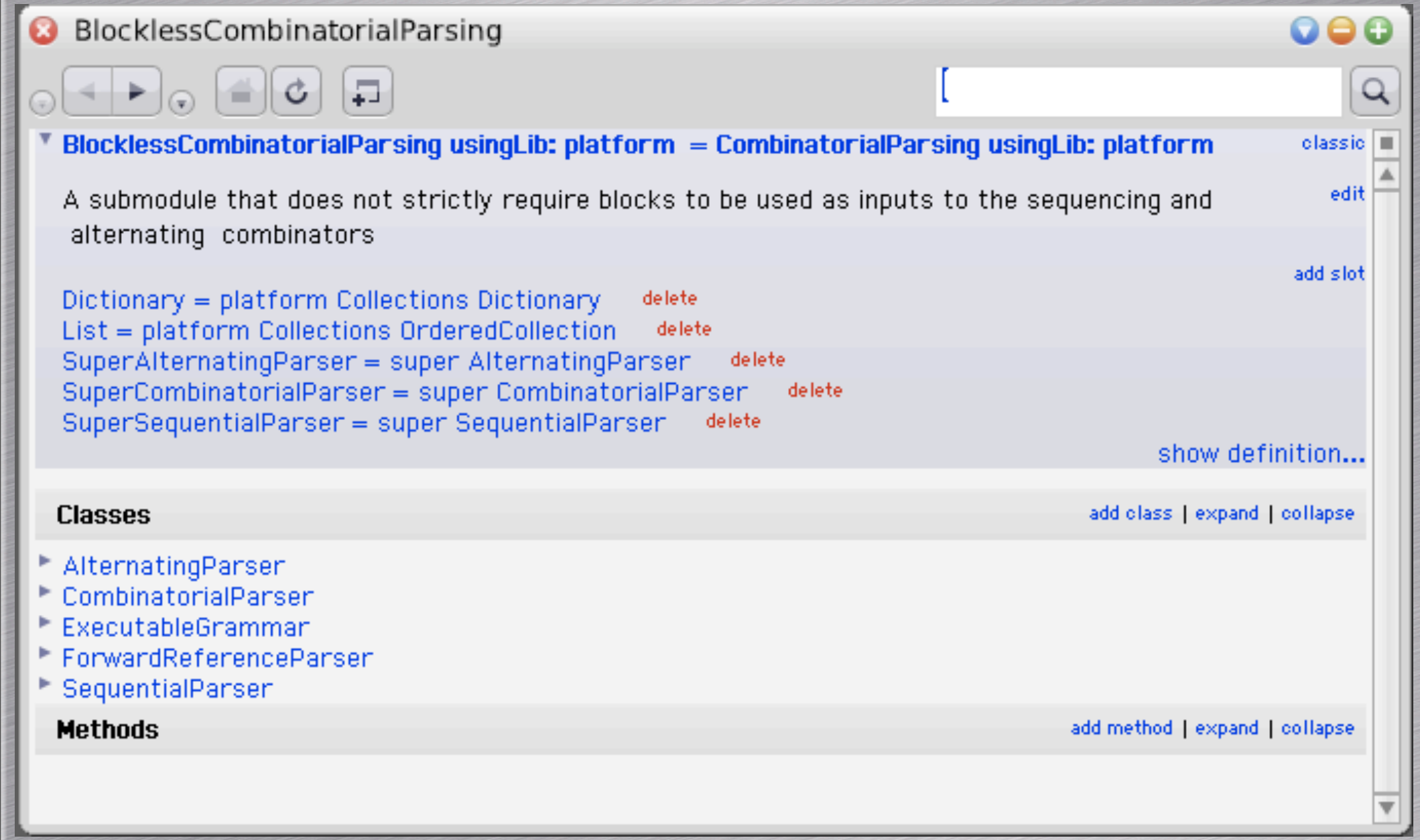
But, this version uses Closures

ifStatement = if, [expression], [then], [statement],

[else], [statement].

statement = ifStatement | [returnStatement].

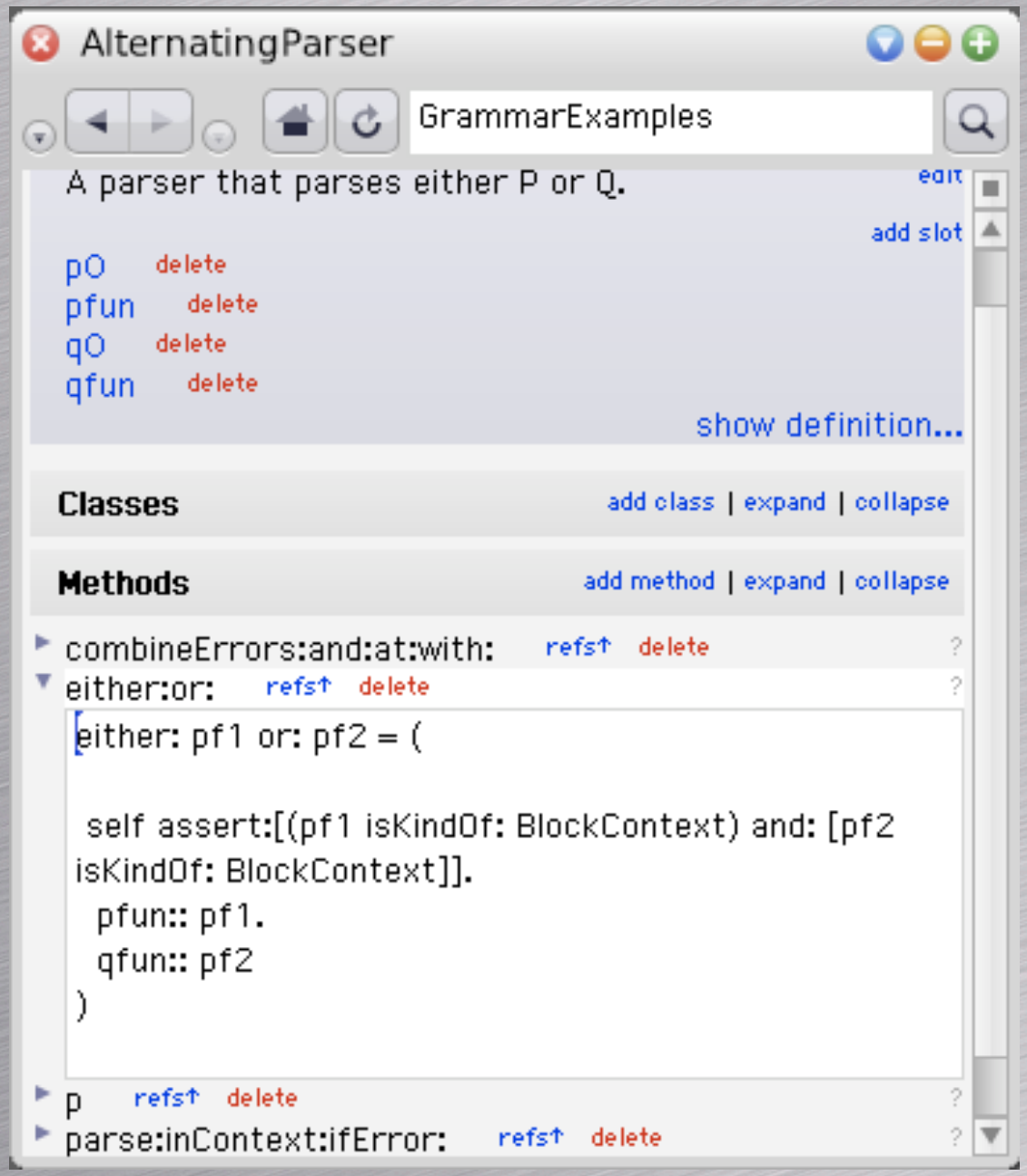
Closureless Version



Hierarchy Inheritance

```
class BlocklessCombinatorialParsing usingLib: platform =
   CombinatorialParsing usingLib: platform (
    | SuperCombinatorialParser = super CombinatorialParser.
      SuperAlternatingParser = super AlternatingParser.
    .....
    |
) (
   public CombinatorialParser = SuperCombinatorialParser ( ...
   )
   public AlternatingParser = SuperAlternatingParser( ... ) ( ... )
....
   public ExecutableGrammar =  CombinatorialParser (...) (...)
)
```

Closureful Alternation

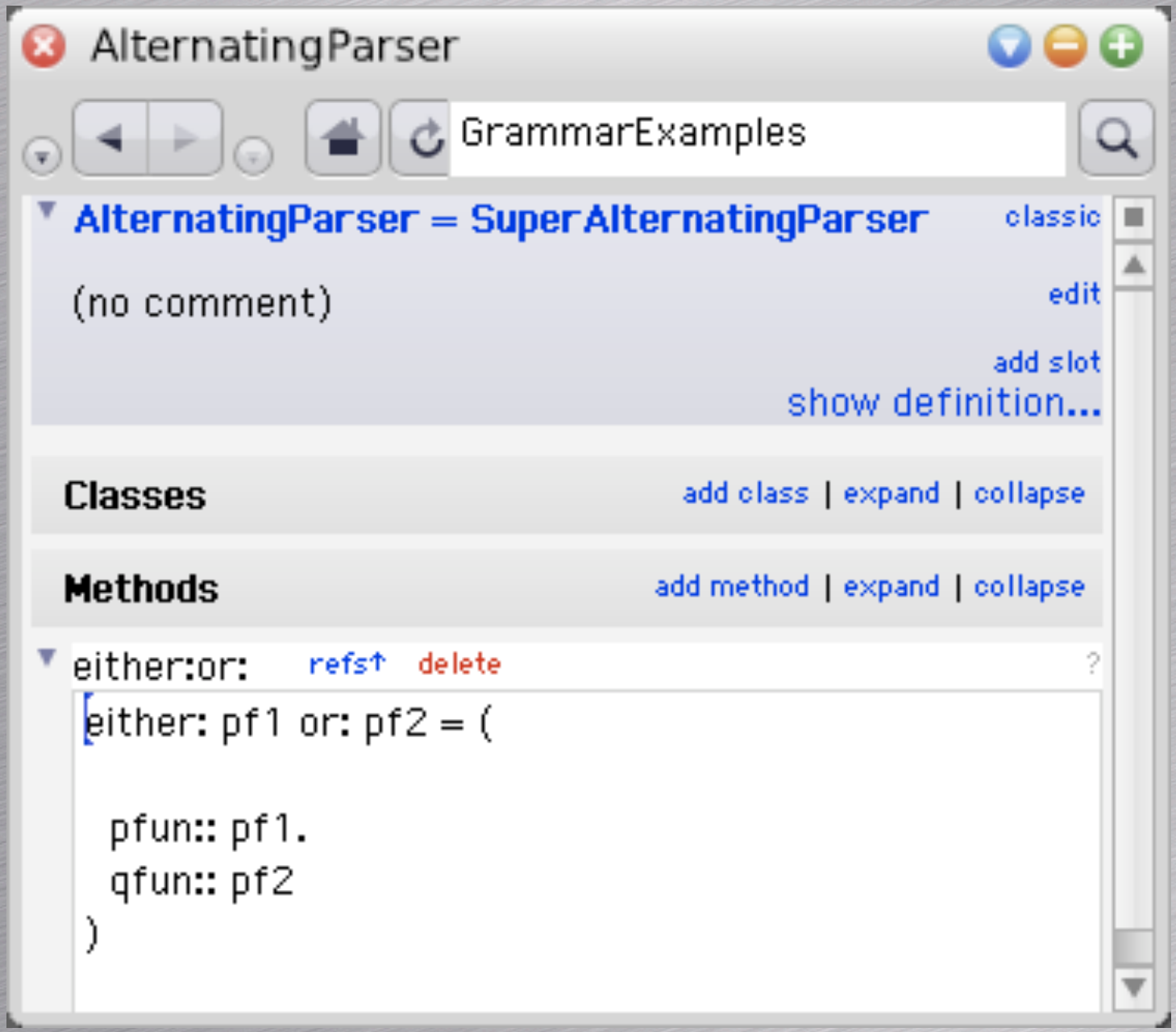


Closure Version

```
either: pf1 or: pf2 = (

 self assert:[(pf1 isKindOf: BlockContext) and: [pf2
isKindOf: BlockContext]].
  pfun:: pf1.
  qfun:: pf2
)
```

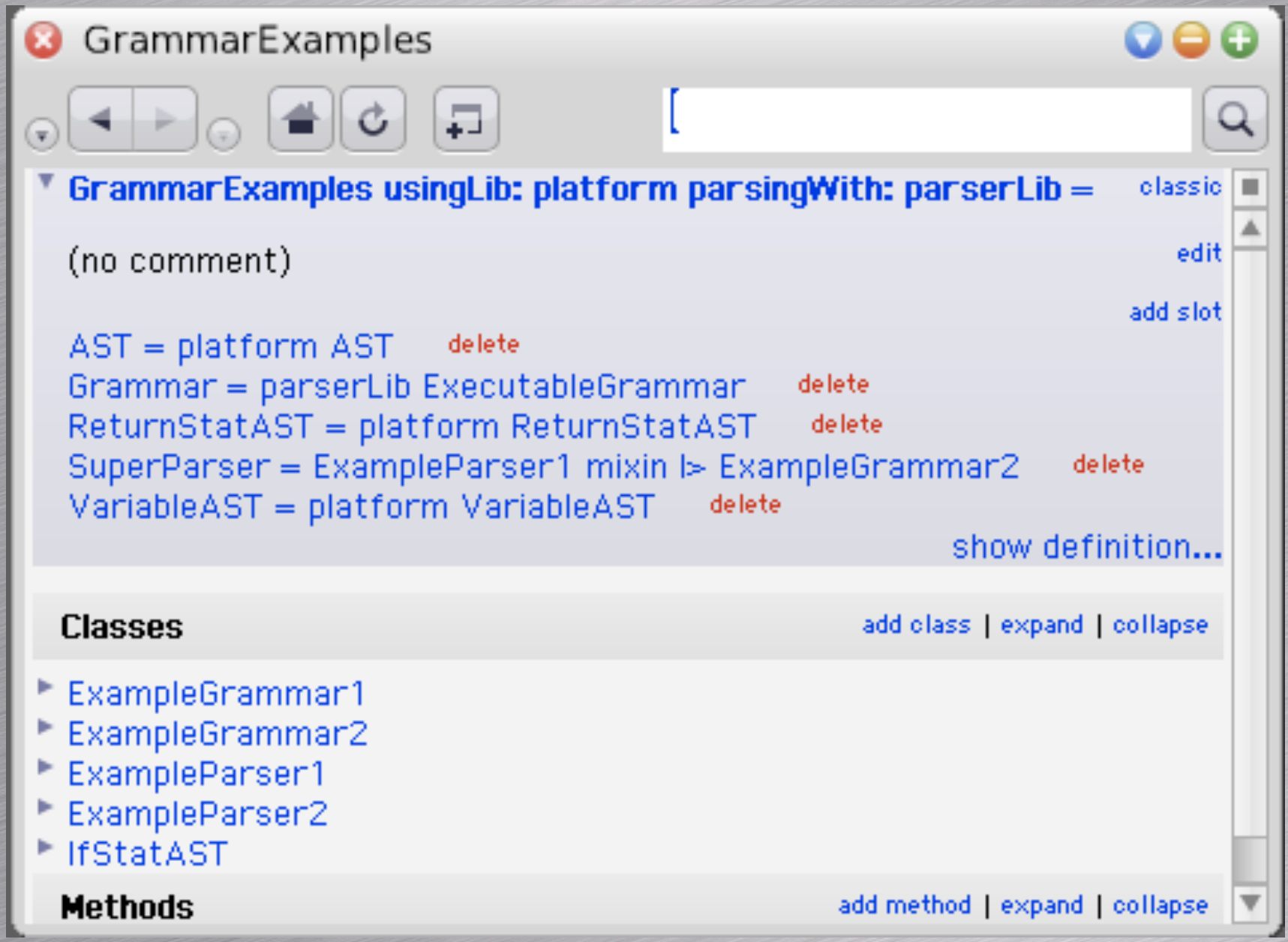
Closureless Alternation



Hierarchy Inheritance

```
class FullBNFParsing usingLib: platform =
   CombinatorialParsing usingLib: platform (
      | SuperParser = super CombinatorialParser. |
) (
   CombinatorialParser = SuperParser (
      | p =  ( ... )
   )
   BacktrackingParser = ( ... ) ( ... )
)
```

Modules Using Modules



Modules using Modules

```
class GrammarExamples
   using: platform parsingWith: parserLib =
 ( |
      Grammar = parserLib ExecutableGrammar.
      SuperParser = ExampleParser1 mixin |>
                                        ExampleGrammar2.
       List = platform Collections OrderedCollection.
    |
 )
(
   ExampleGrammar1 = Grammar ( ... ) (
   ....
    )
 )
```

Side by Side Modules

```
platform:: Platform new.

m1:: GrammarExamples
        using: platform
      parsingWith: (BlocklessCombinatorialParsing
                              usingLib: platform)

m2:: GrammarExamples
        using: platform
      parsingWith: (PackratParsing usingLib: platform)
```

Improving Readability

```
class ExampleGrammar1 = ExecutableGrammar (
|
digit = charBetween: $0 and: $9.
letter = (charBetween: $a and:$z) |
         (charBetween: $A and:$Z).
id = letter, (letter | digit) star.
identifier = tokenFor: id.
hat = tokenFromChar: $^.
expression = identifier.
returnStatement = hat, expression.
|
)()
```

A Modest Facelift

```
class ExampleGrammar3 = ExecutableGrammar (
|
digit = $0 - $9.
letter = ($a - $z) | ($A - $Z).
id = letter, (letter | digit) star.
identifier = tokenFor: id.
expression = identifier.
returnStatement = $^, expression.
|
)()
```

Before

```
class ExampleGrammar2 = ExampleGrammar1 (
|
if = tokenFromSymbol:#if.
then = tokenFromSymbol:#then.
else = tokenFromSymbol:#else.
ifStatement = if, expression, then, statement, else, statement.
statement = ifStatement | returnStatement.
|
)()
```

After

```
class ExampleGrammar3 = ExampleGrammar4 (
|
ifStatement = #if, expression, #then, statement, #else,
statement.
statement = ifStatement | returnStatement.
|
)()
```

Dirty Tricks?

```
class ExecutableGrammar usingLib: platform =
                CombinatorialParsing usingLib: platform (
|
PlatformCharacter = super Character.
Character = PlatformCharacter mixin |> CharacterParser.
Symbol = super Symbol mixin |> SymbolParser.
....
|
)()
```

Related Work

- Too much to survey; some highlights:
  - Parsec (Haskell parser combinator library)
  - Swierstra et al.
  - Sparsec (Scala parser combinator library)
  - PEGs (Ford, POPL04, ICFP)
  - Tedir (Plesner-Hansen's masters thesis)
  - Ometa (OOPLA 07 DLS)

Related Work

- Message-based programming: Smalltalk, Self
- Virtual Types: Beta, gBeta, Scala
- Modules: ML, Units
- Side-by-Side deployment: OSGi
- Hierarchy inheritance: Ossher & Harrisson ..

Conclusions

- Clean syntax and reflection allow effective embedding of BNF as a DSL in Newspeak
- Inheritance and dynamic typing make it easy to separate grammar from processing; huge SE benefits
- Message based programming supports strong component style modularity + class hierarchy inheritance