

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());

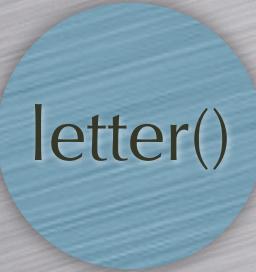
How it Works

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

How it Works

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

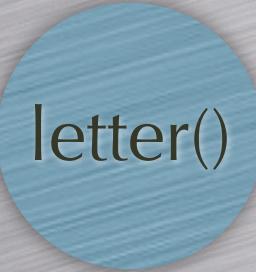
How it Works



letter()

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

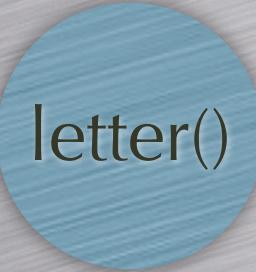
How it Works



letter()

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

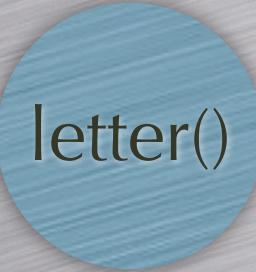
How it Works



letter()

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

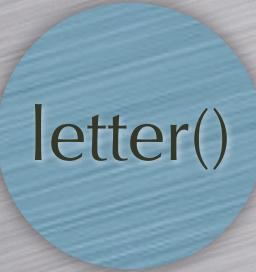
How it Works



letter()

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

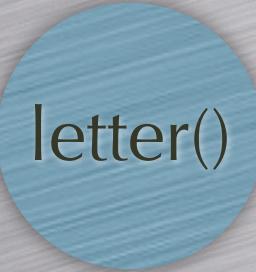
How it Works



letter()

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

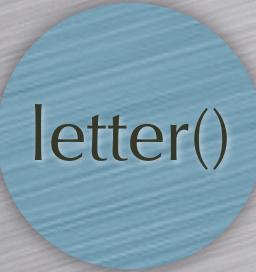
How it Works



letter()

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

How it Works



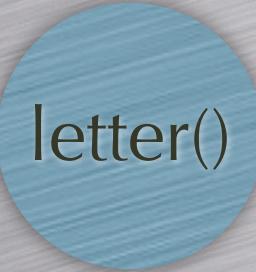
letter()



letter()

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

How it Works



letter()



letter()

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

How it Works

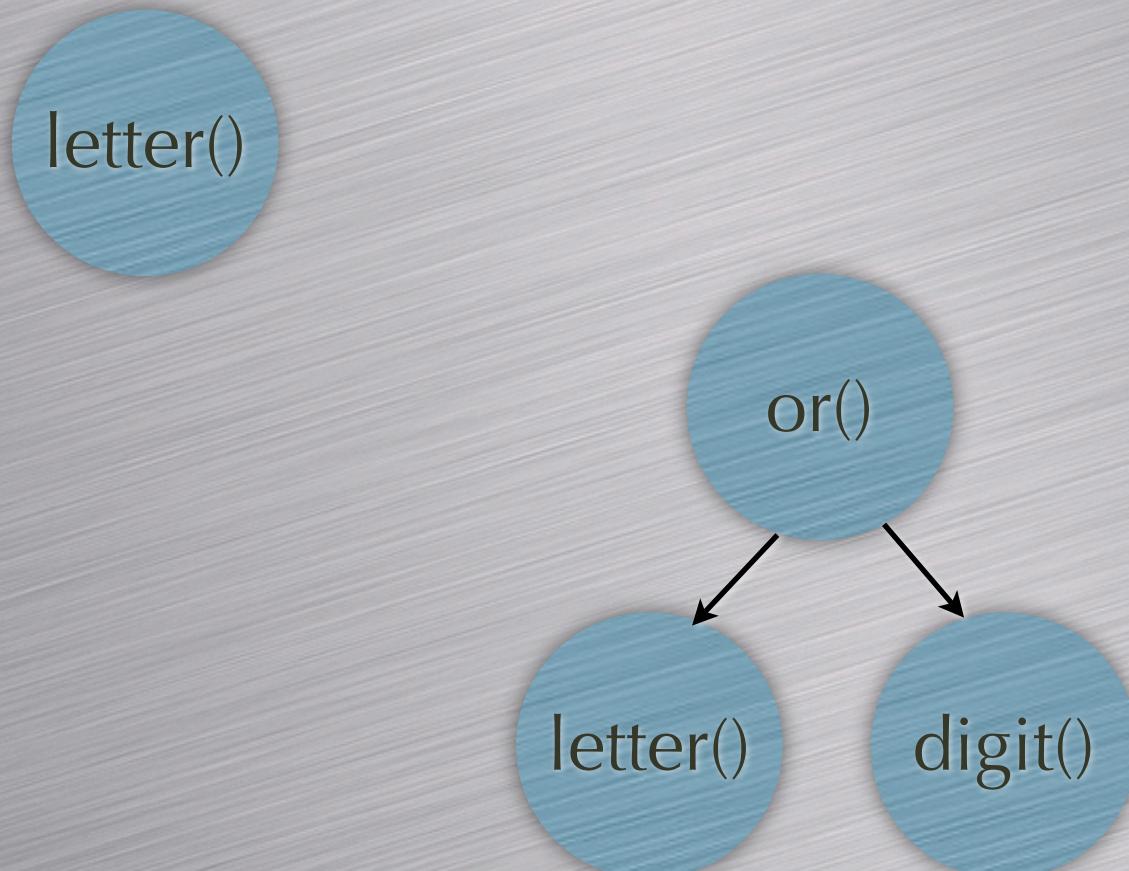
letter()

letter()

digit()

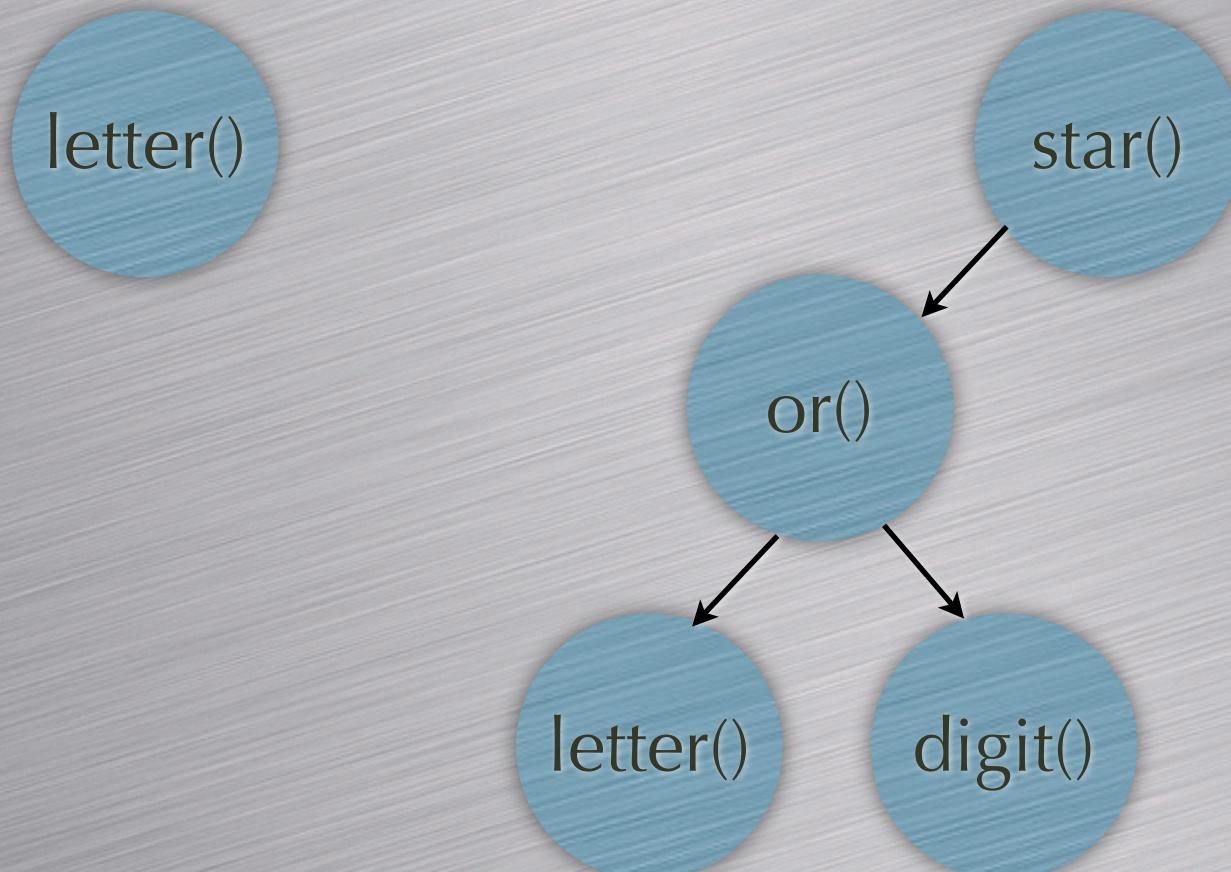
id = letter().seq(letter().or(digit()).star());

How it Works



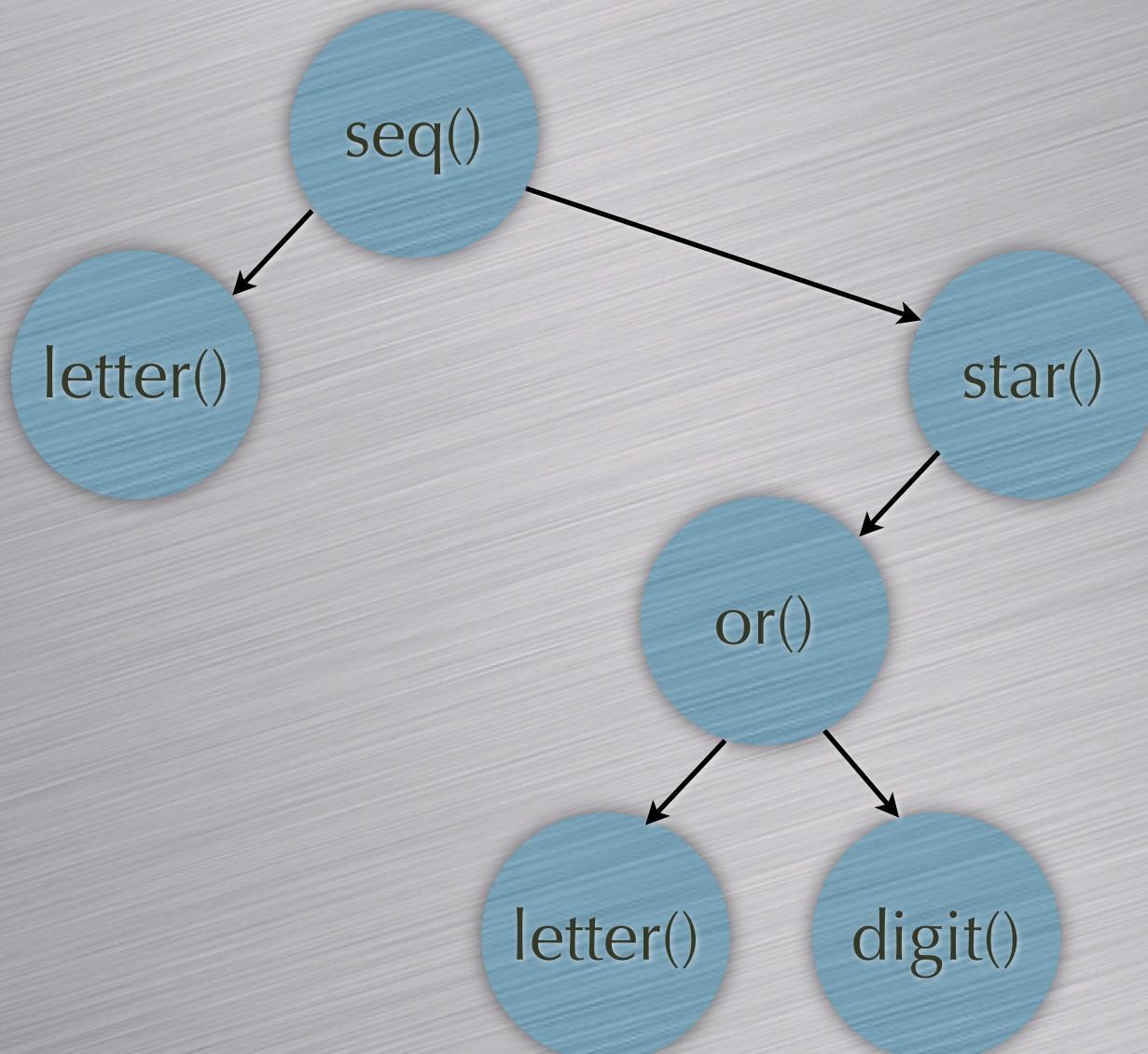
id = letter().seq(letter().or(digit())).star());

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

$id = \text{letter}, (\text{letter} \mid \text{digit})^*$.

How it Works



letter

$id = \textcolor{red}{letter}, (\textit{letter} \mid \textit{digit})^*$.

How it Works



letter

id = letter, (letter | digit) star.

How it Works



letter

$id = letter, (letter \mid digit)^*.$

How it Works



letter

id = letter, (letter | digit) star.

How it Works



letter

id = letter, (letter | digit) star.

How it Works



letter

$id = letter, (letter \mid digit)^*$

How it Works



letter



letter

$id = \text{letter}, (\text{letter} \mid \text{digit})^*$.

How it Works



letter



letter

$id = \text{letter}, (\text{letter} \mid \text{digit})^*$.

How it Works

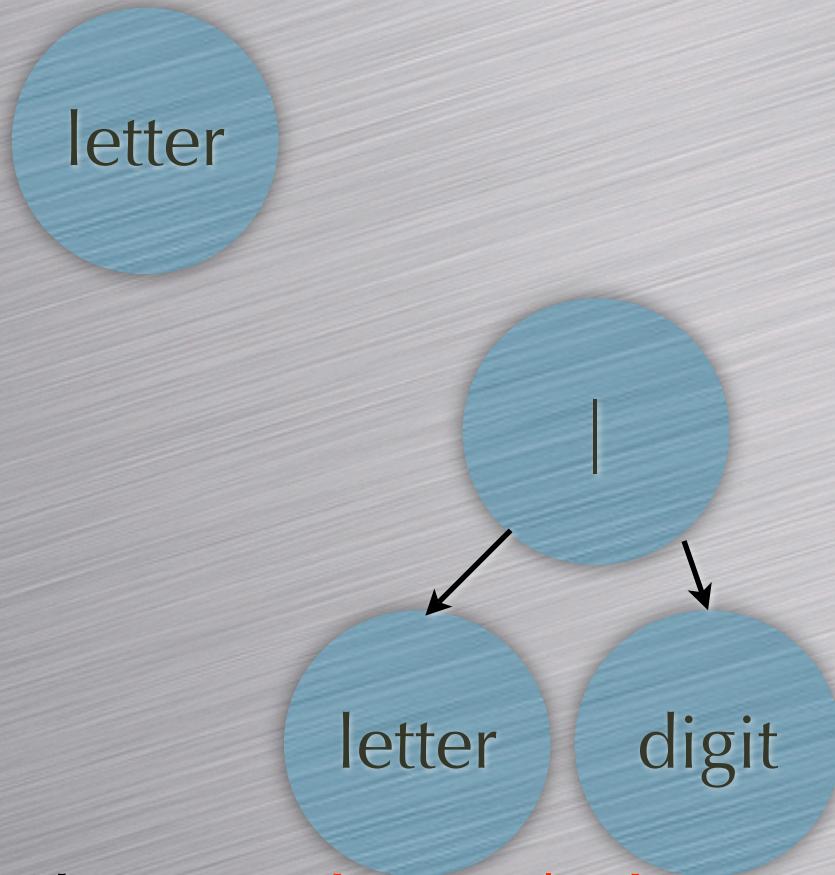
letter

letter

digit

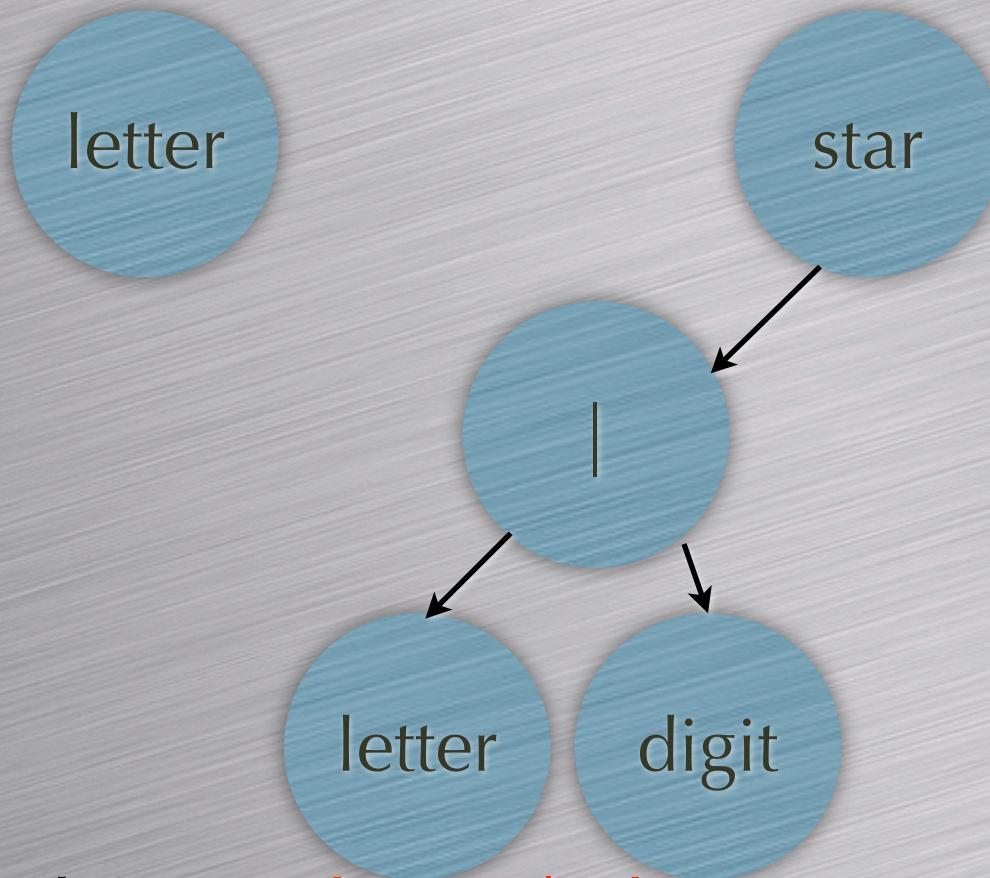
$id = letter, (letter \mid digit)^*$

How it Works



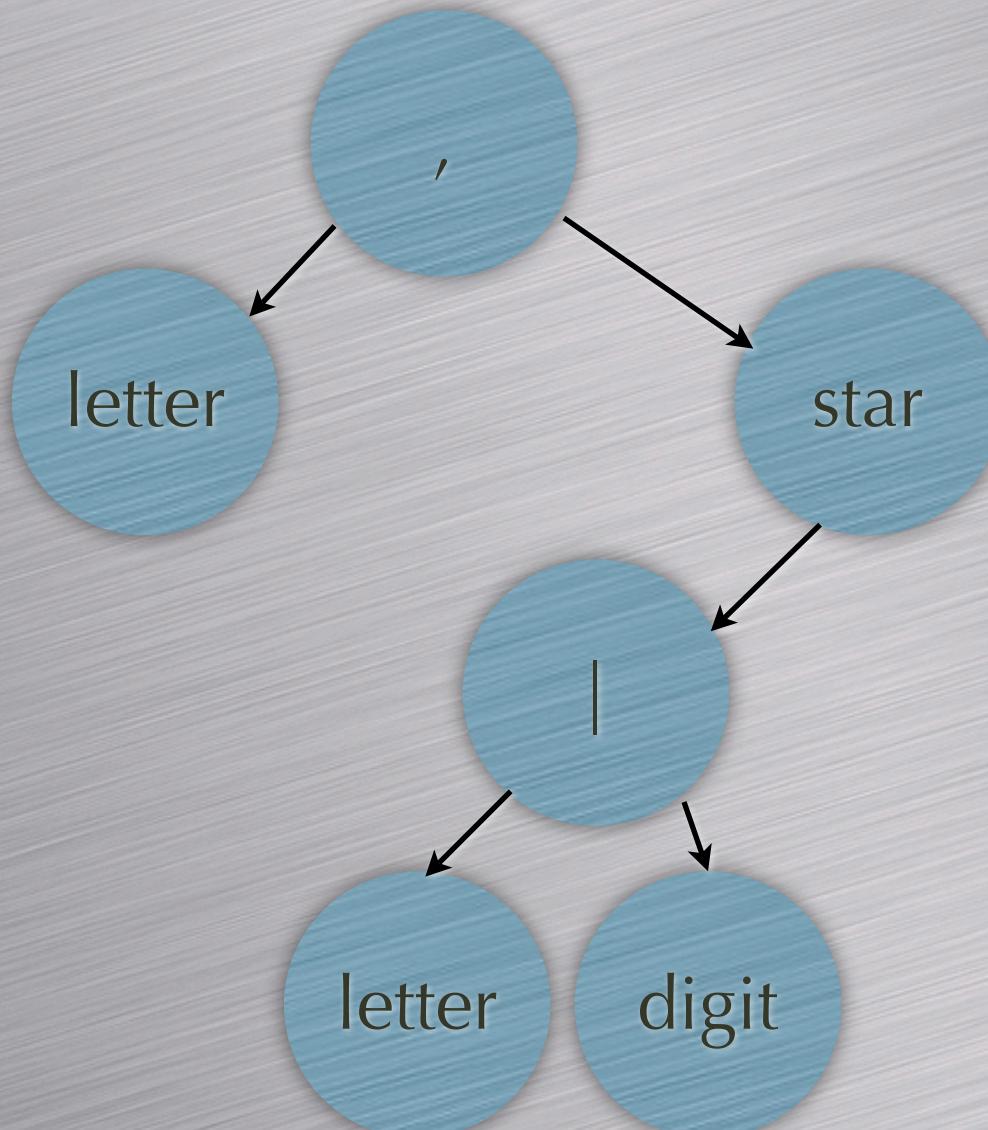
id = letter, (letter | digit) star.

How it Works



$id = \text{letter}, (\text{letter} \mid \text{digit})^*$.

How it Works



$id = \text{letter}, (\text{letter} \mid \text{digit})^*$.

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 |

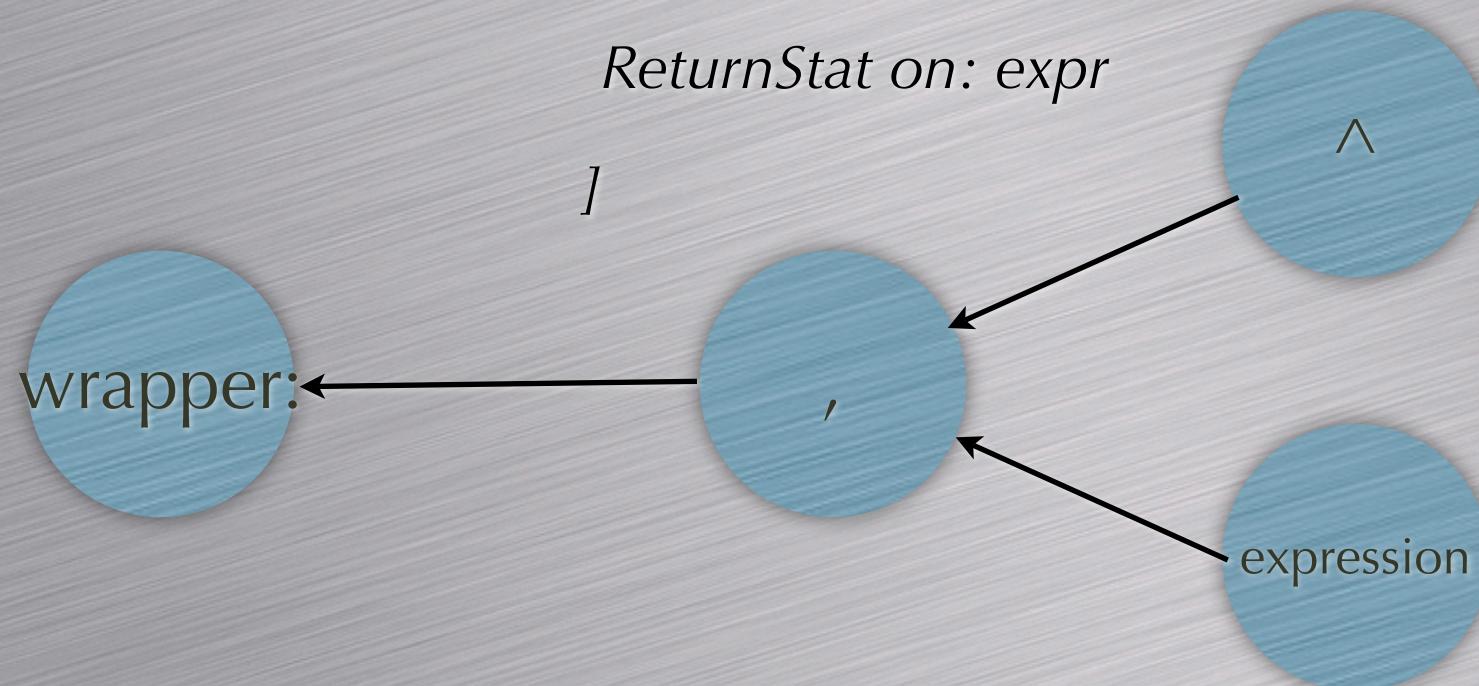
ReturnStat on: expr

]

Building an AST

returnStat = hat, expression

wrapper[:caret : expr |



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

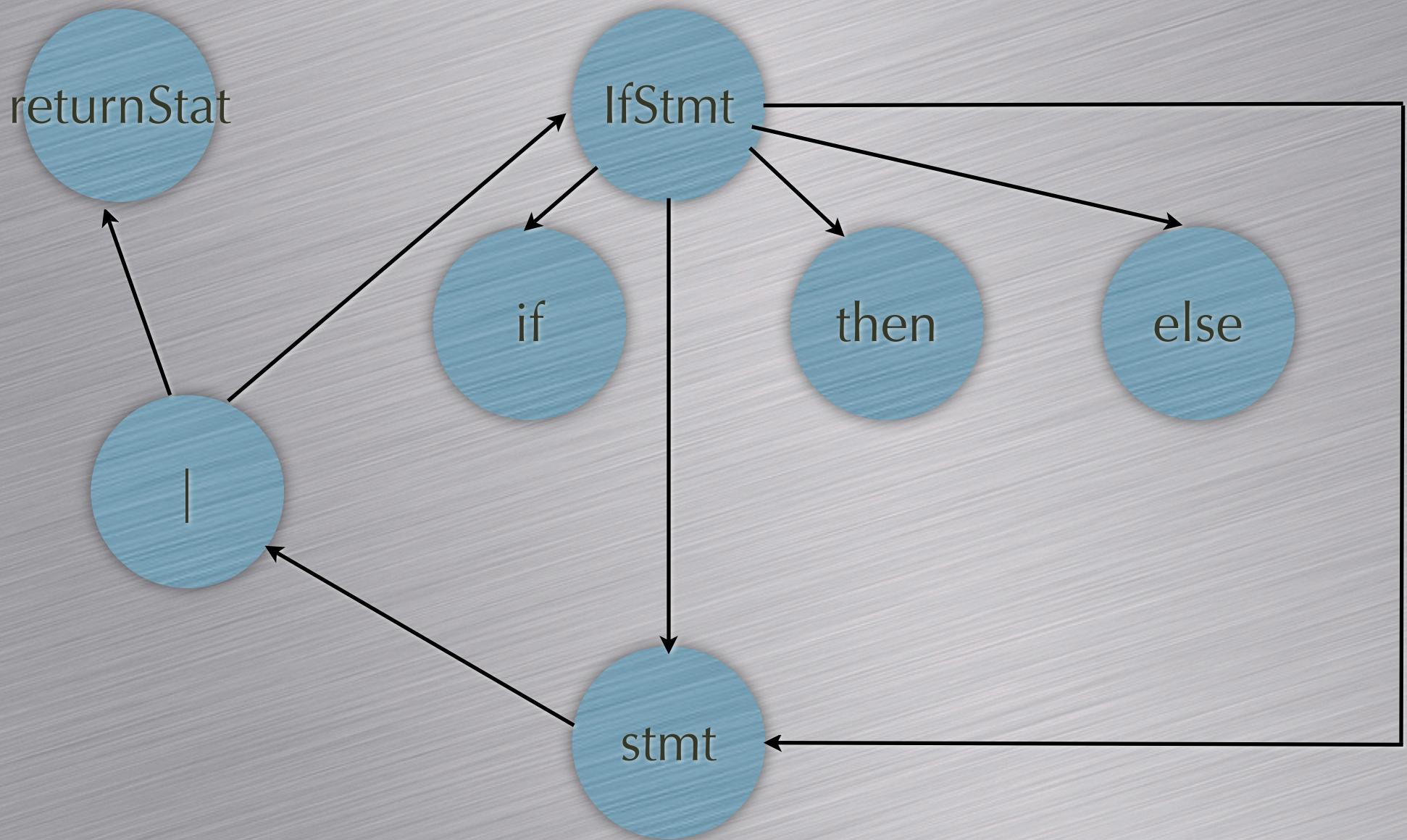
```
class ExampleGrammar2 = ExampleGrammar1 (
|
if = tokenFromSymbol:#if.
then = tokenFromSymbol:#then.
else = tokenFromSymbol:#else.
ifStatement = if, expression, then, statement, else, statement.
statement = ifStatement | returnStatement.
|
)()
```

Mutual Recursion

ifStatement = *if*, *expression*, *then*, *statement*,
else, *statement*.

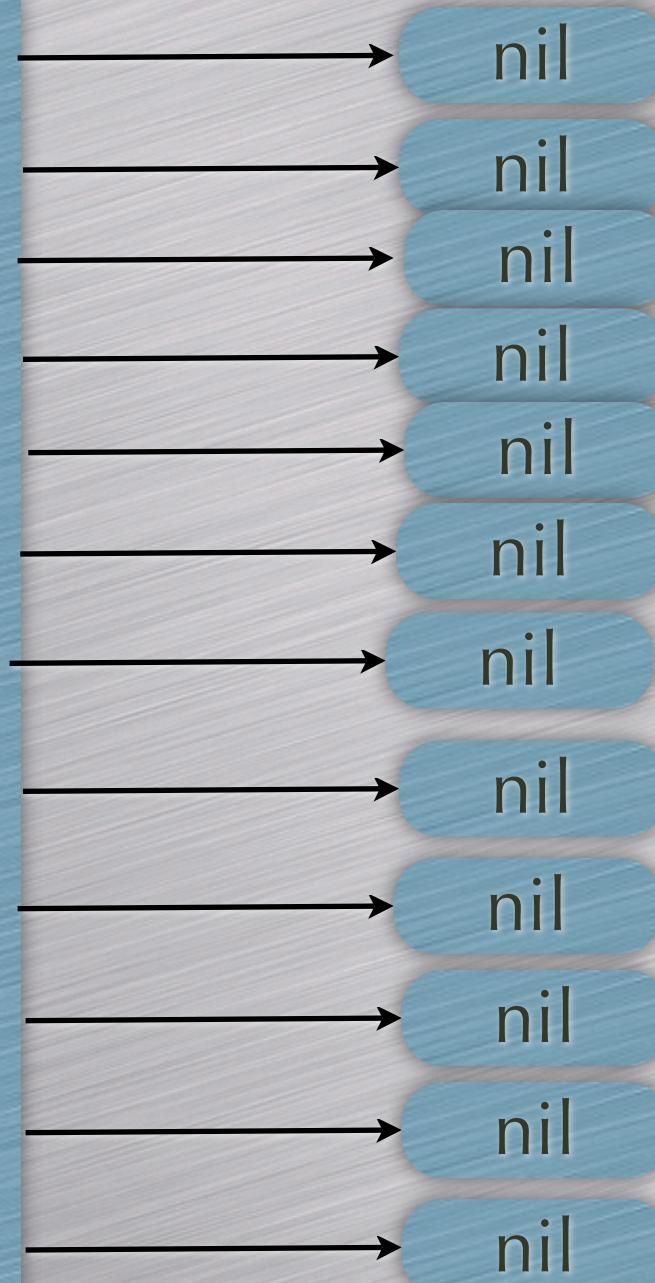
statement = *ifStatement* | *returnStatement*.

Mutual Recursion

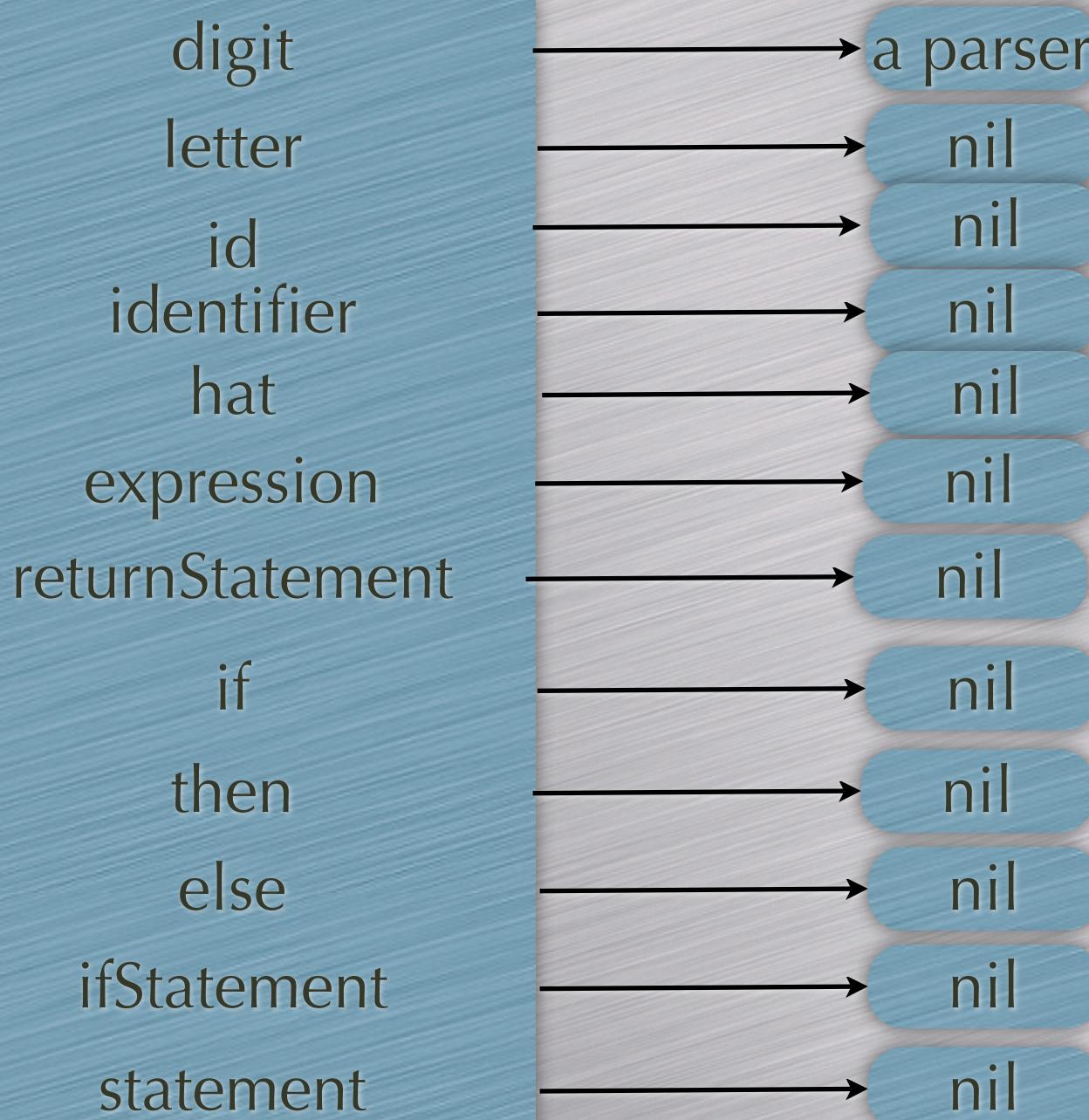


Mutual Recursion

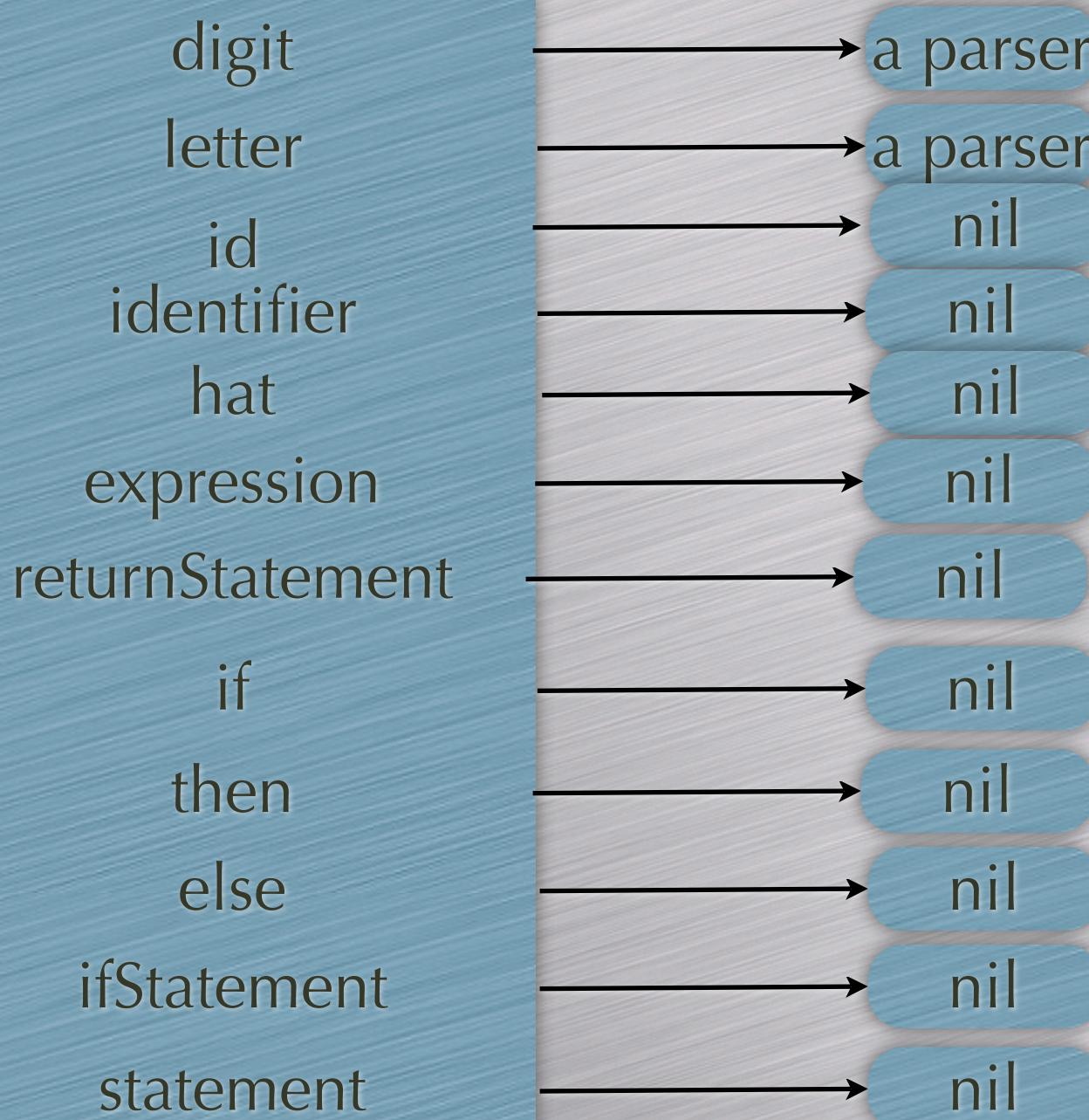
digit
letter
id
identifier
hat
expression
returnStatement
if
then
else
ifStatement
statement



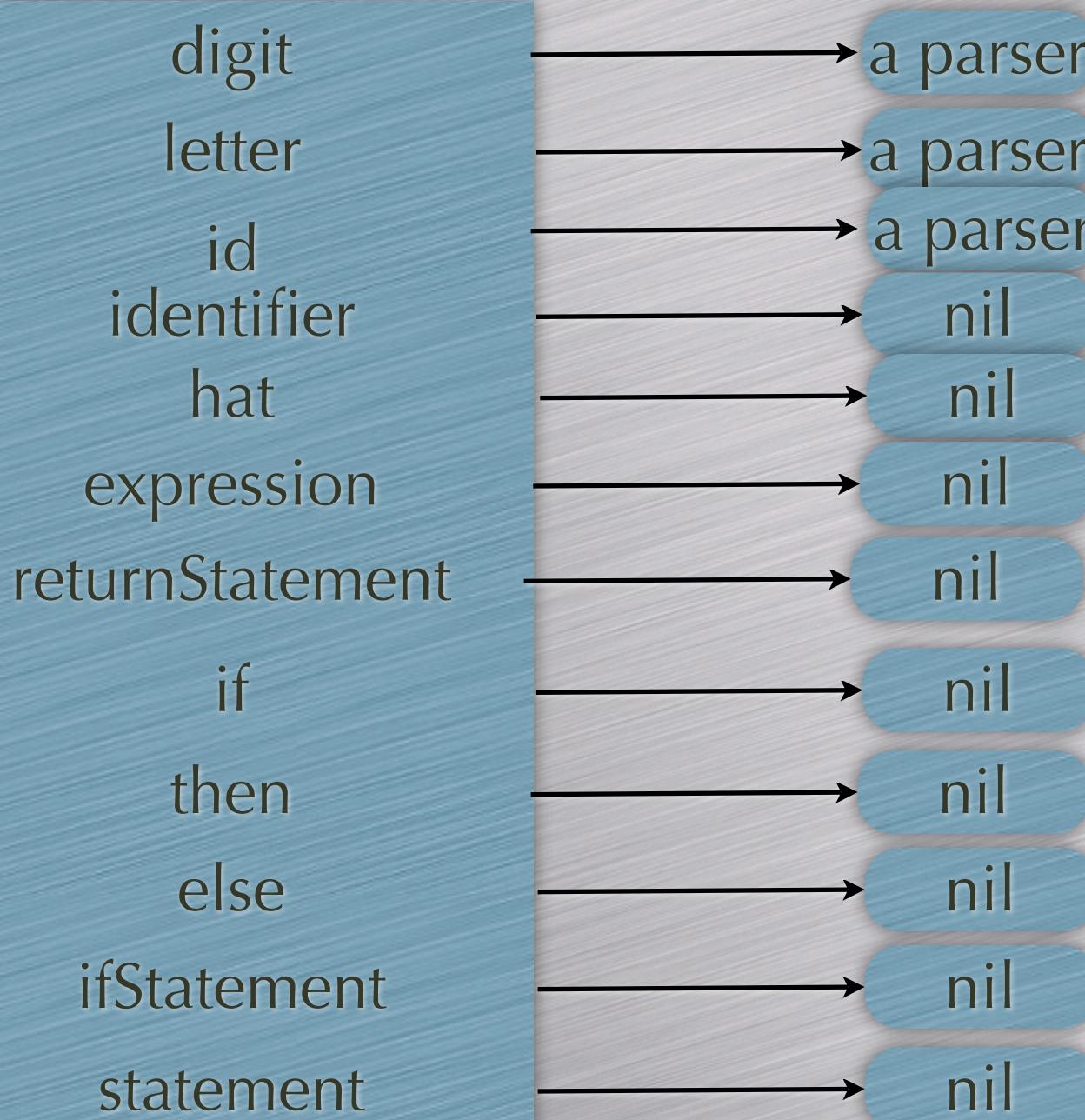
Mutual Recursion



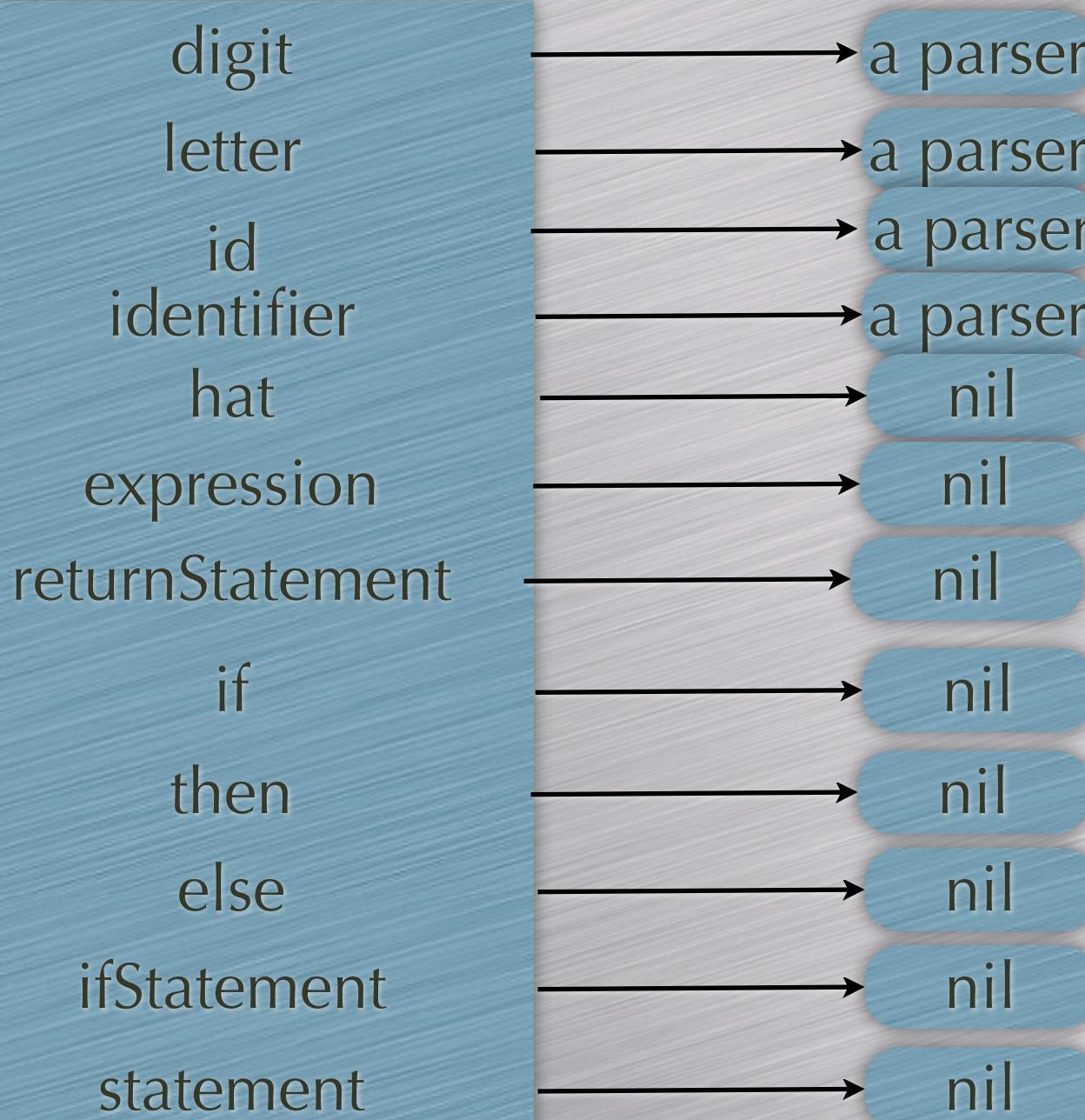
Mutual Recursion



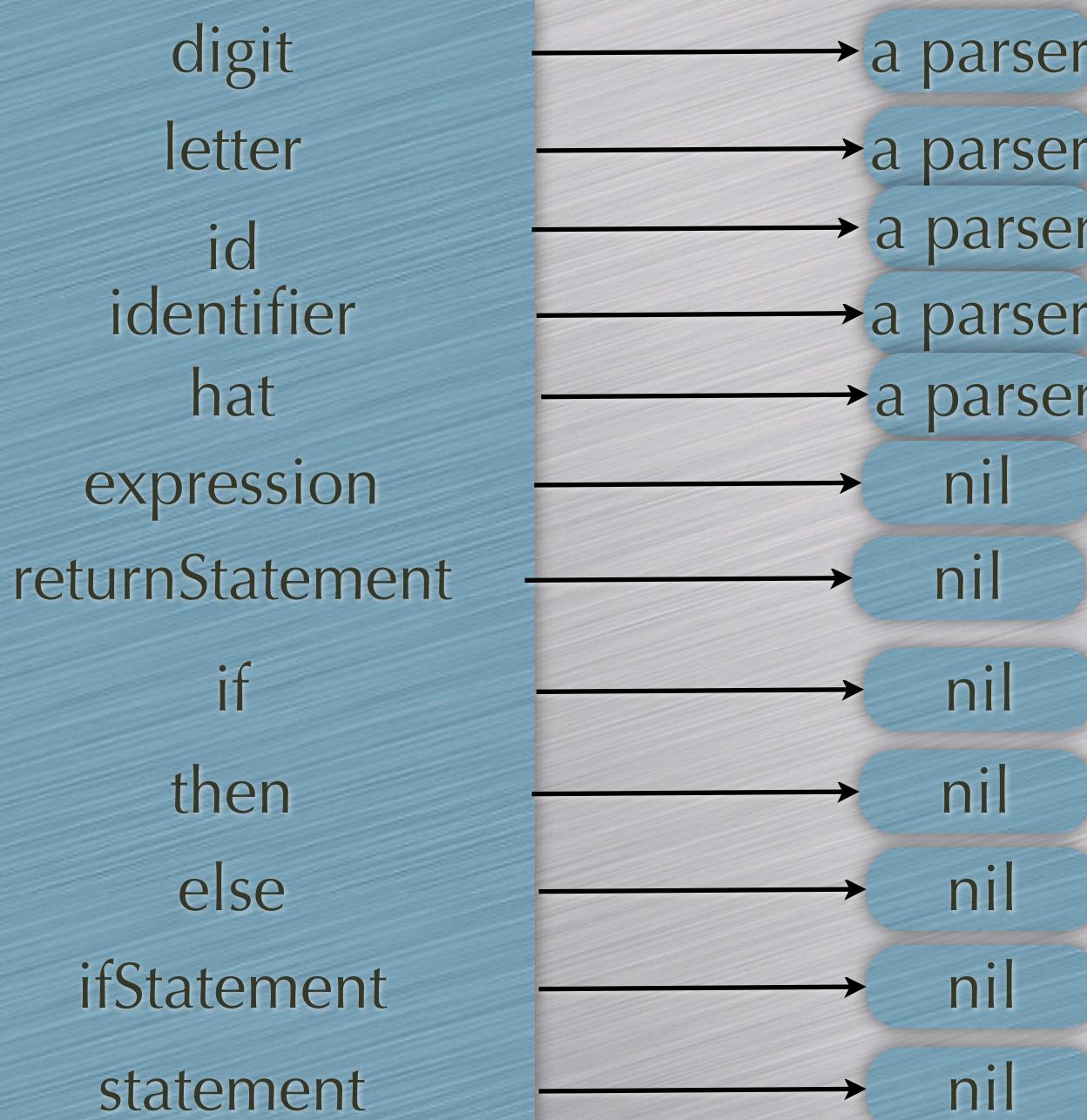
Mutual Recursion



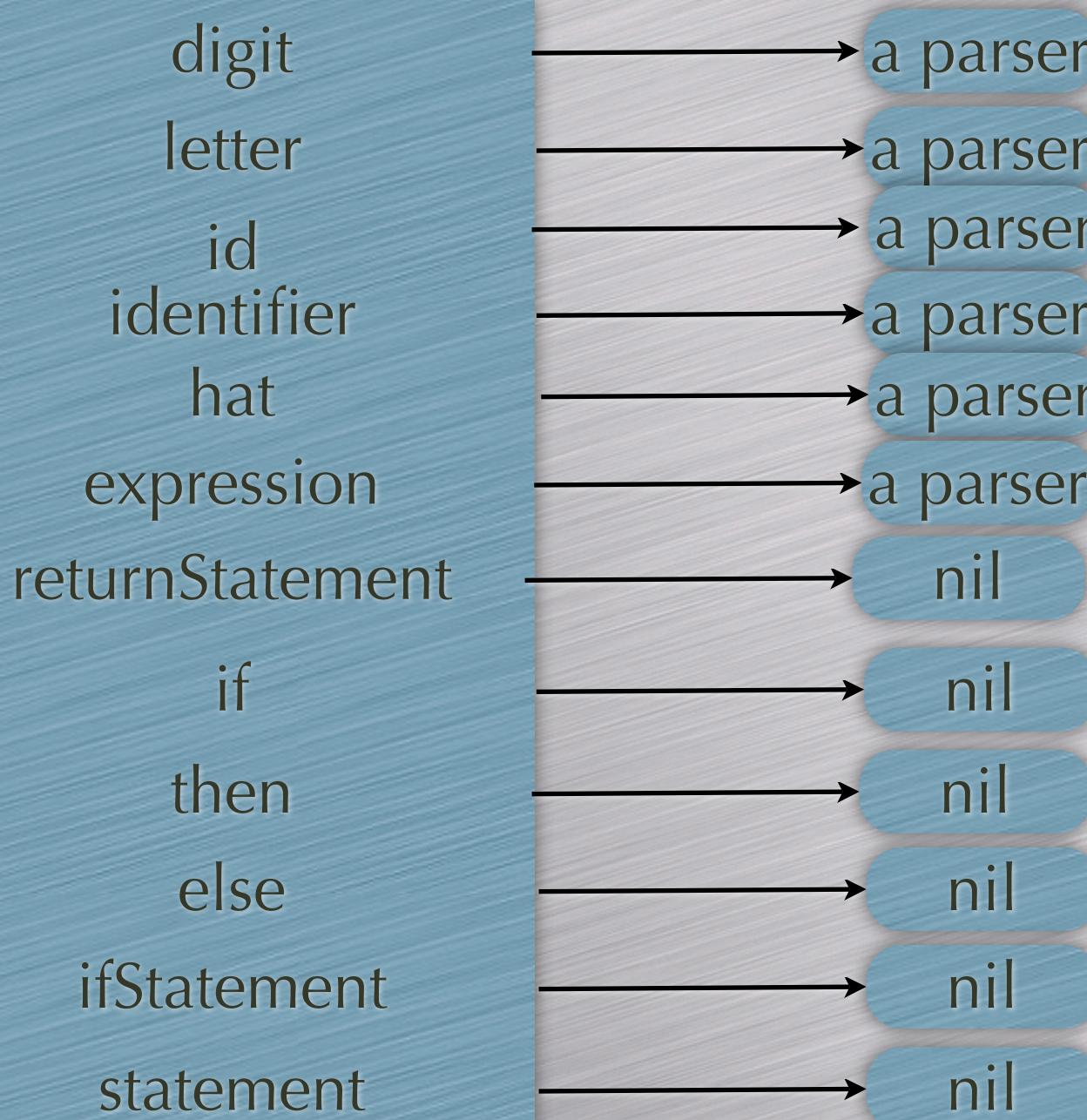
Mutual Recursion



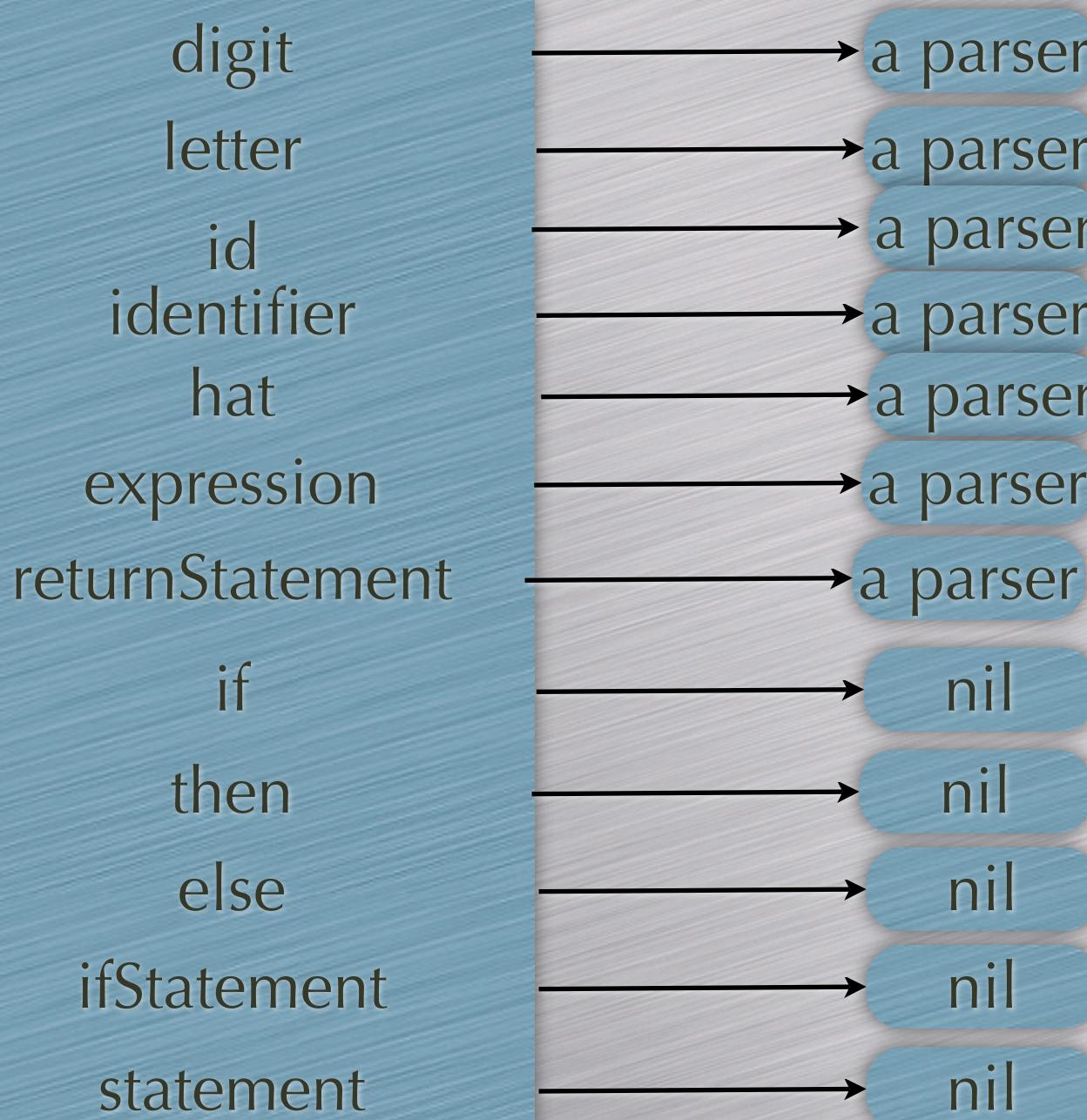
Mutual Recursion



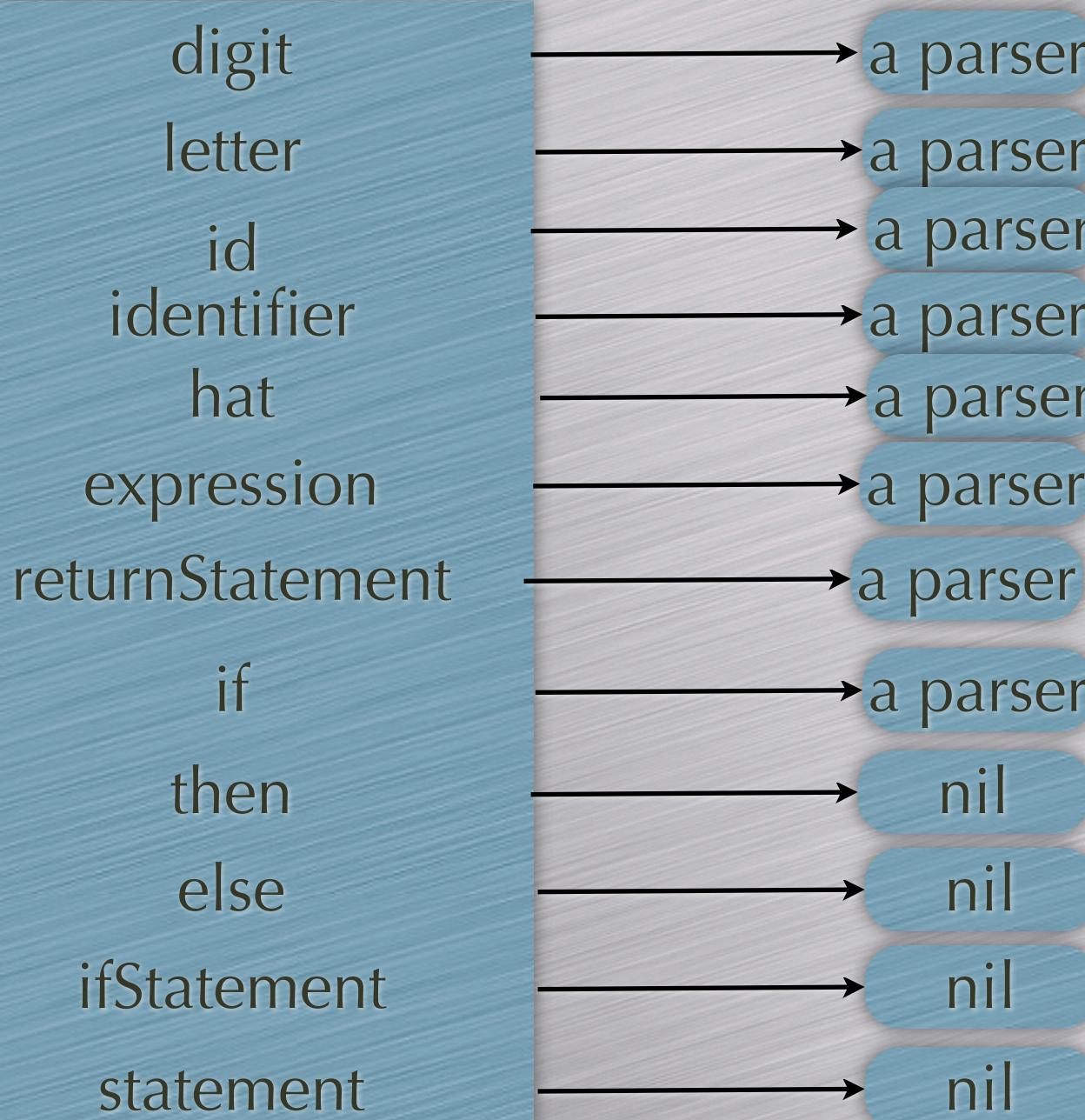
Mutual Recursion



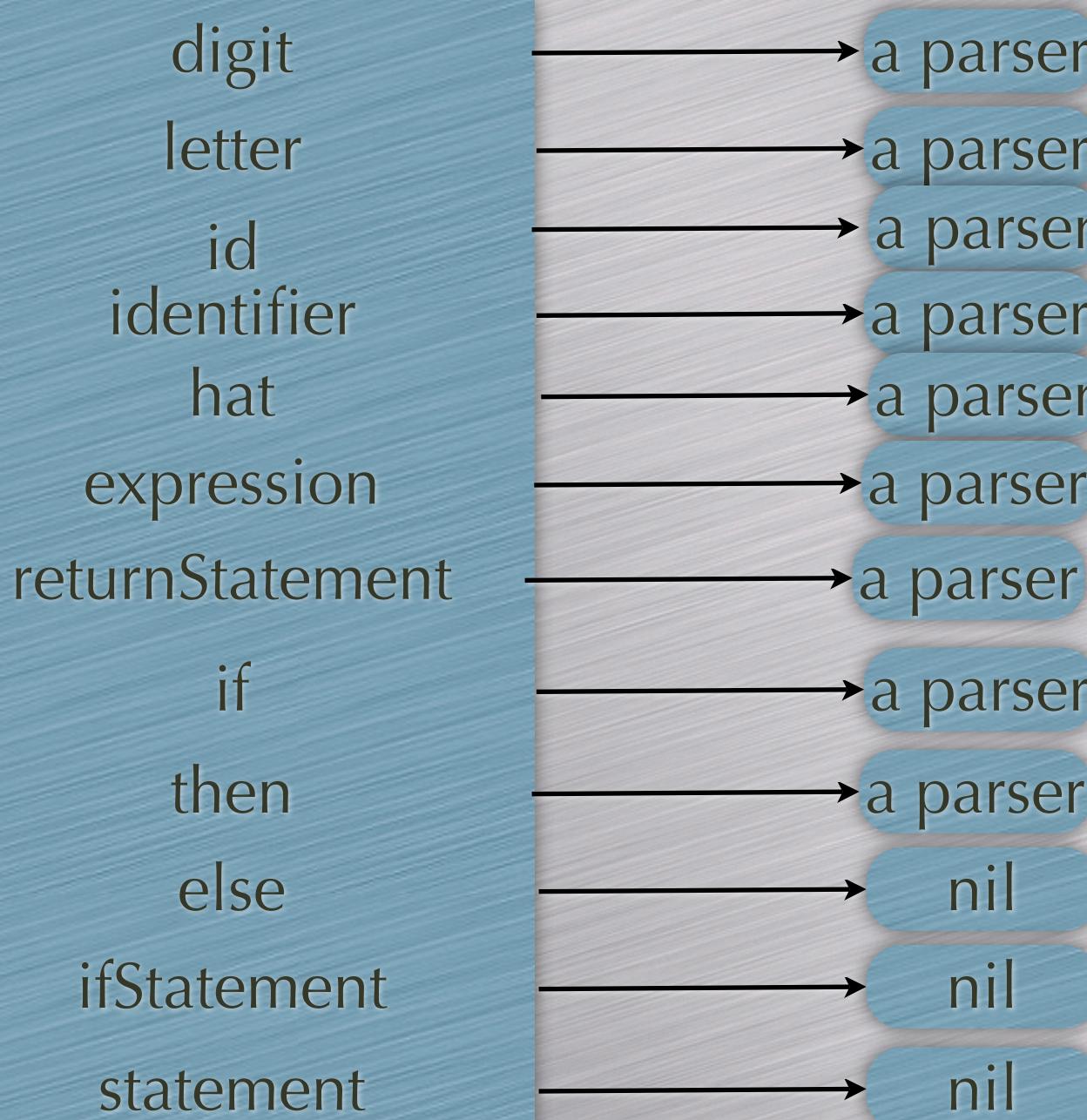
Mutual Recursion



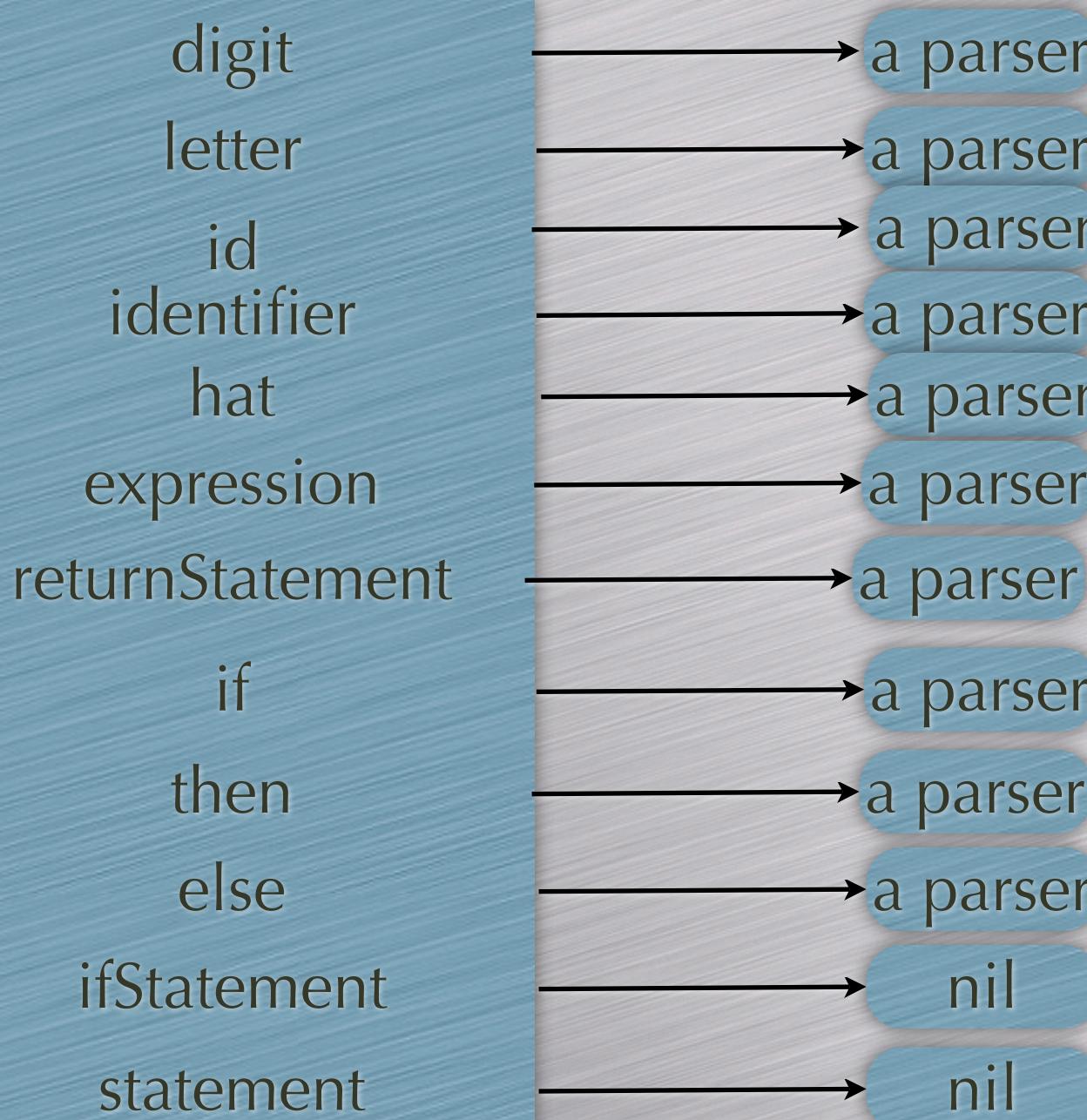
Mutual Recursion



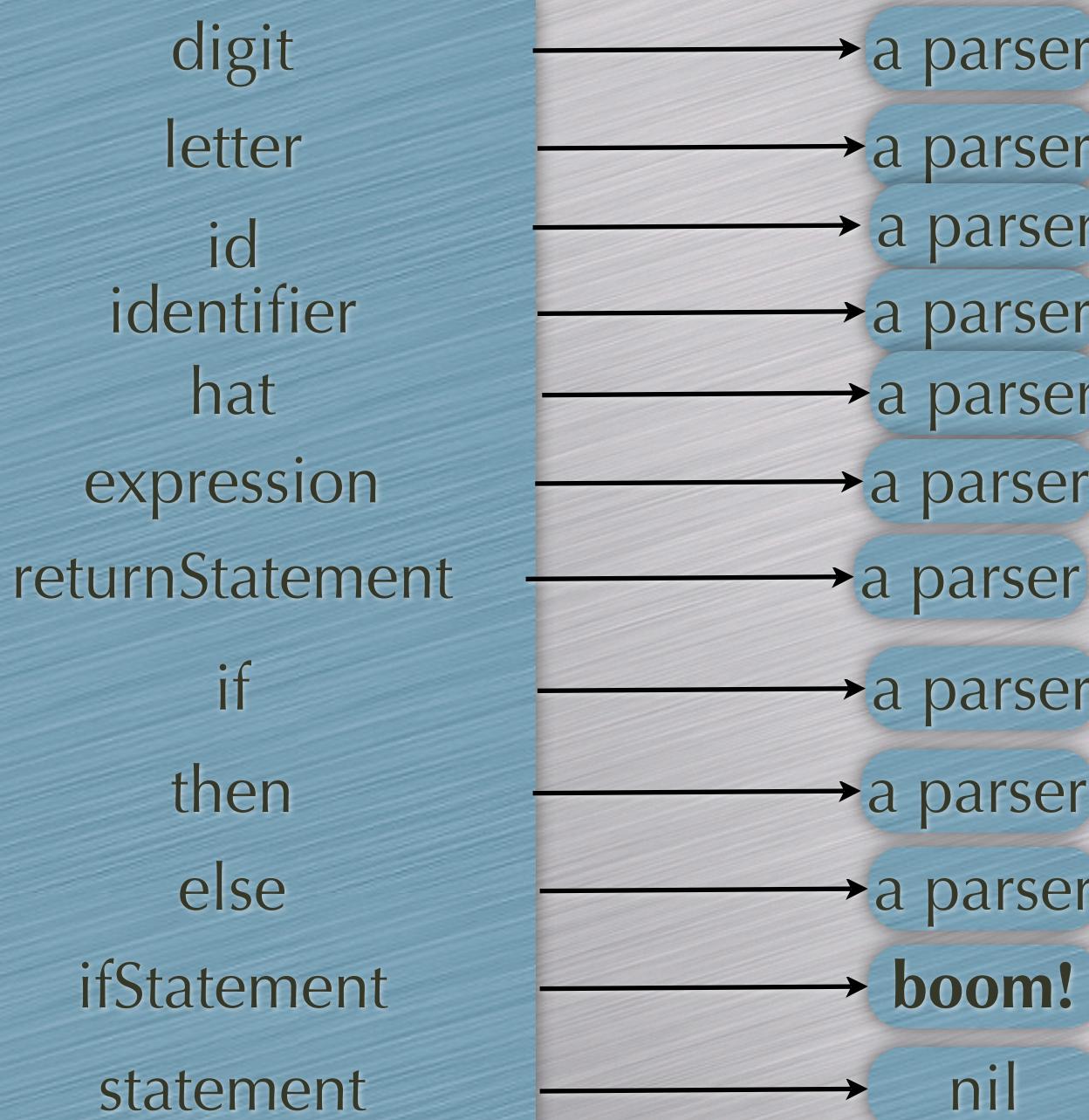
Mutual Recursion



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].

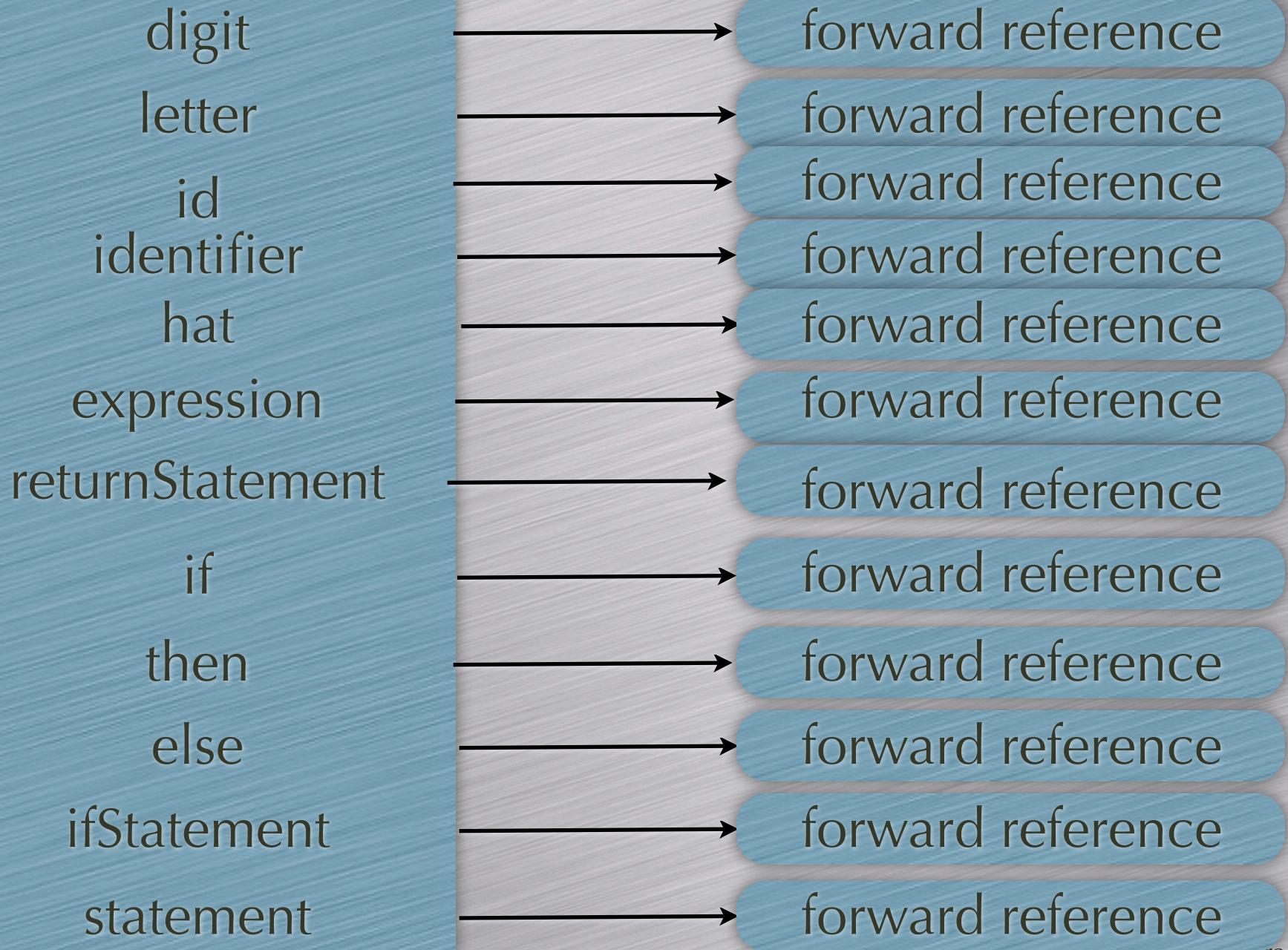
statement = *ifStatement* | [returnStatement].

Being Reflective rather than Lazy

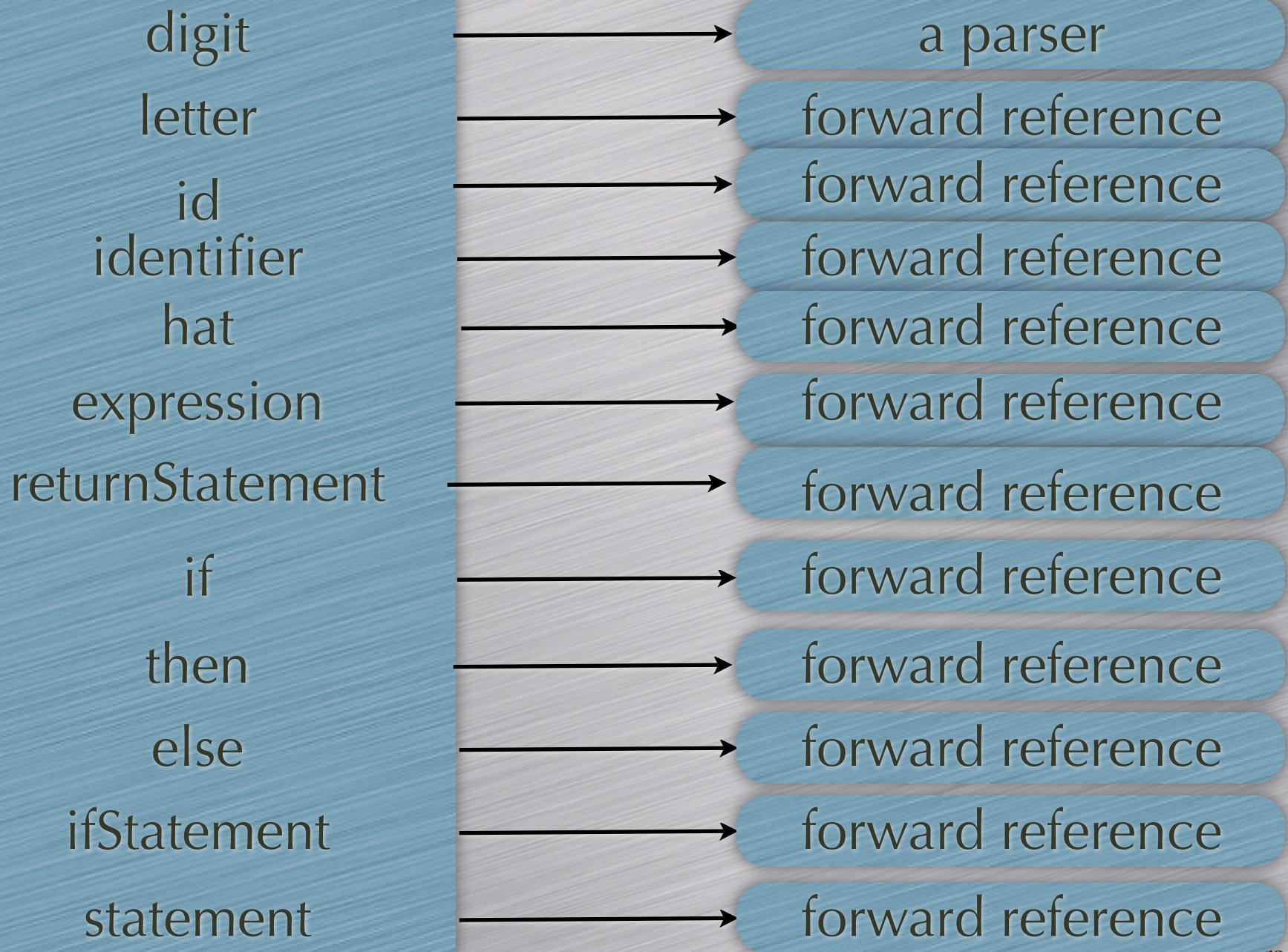
Use Reflection instead of Laziness.

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

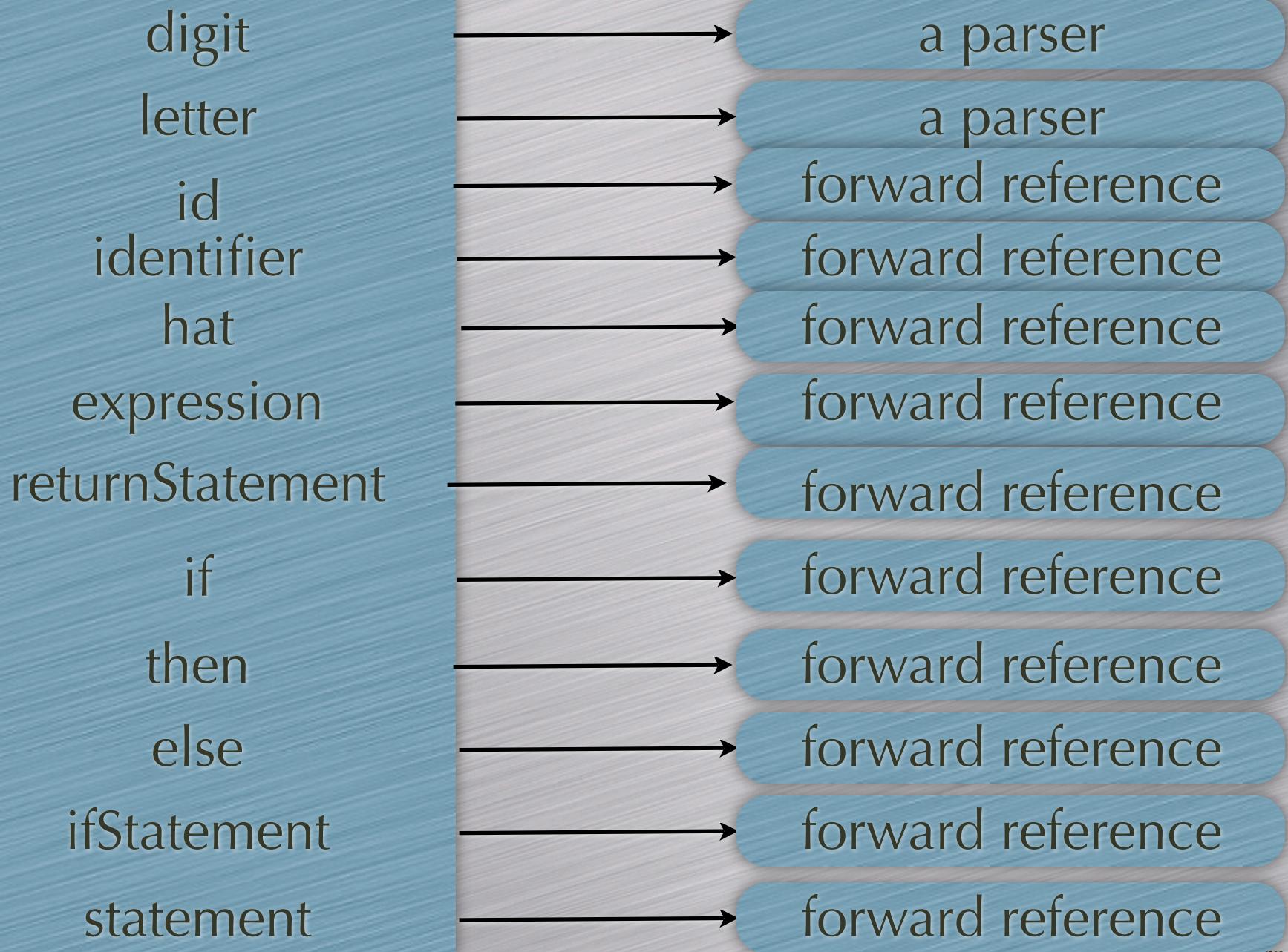
Mutual Recursion



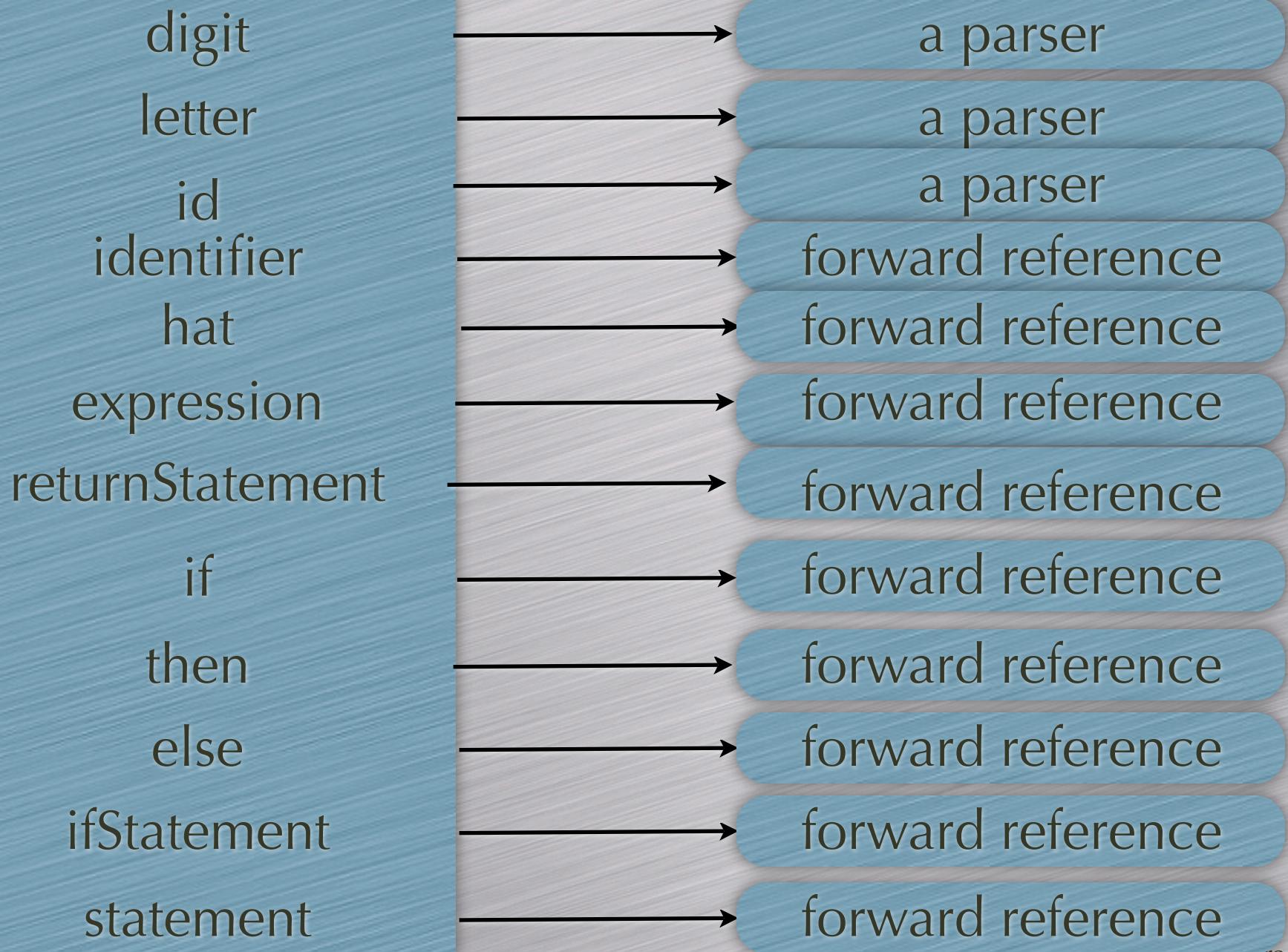
Mutual Recursion



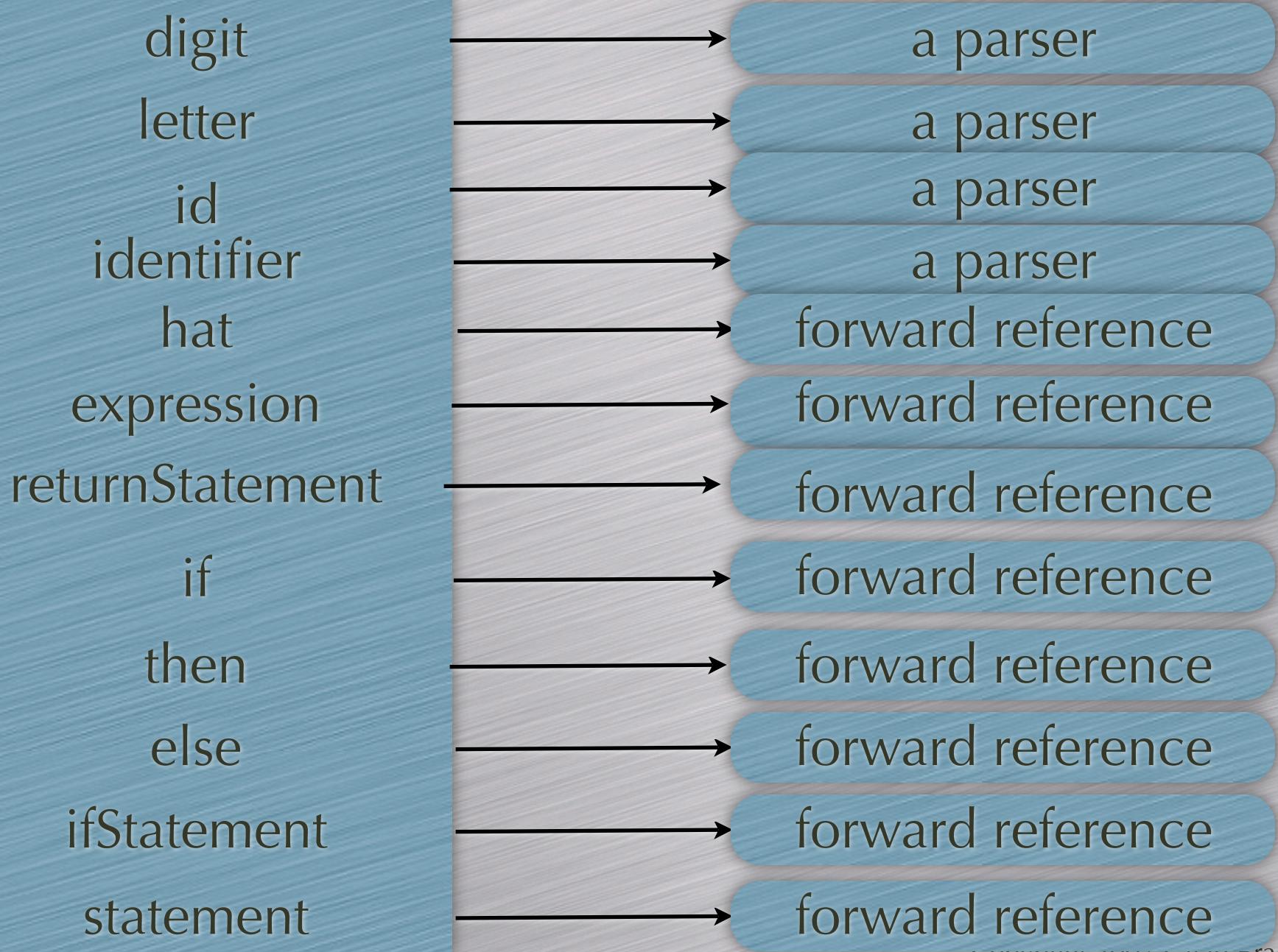
Mutual Recursion



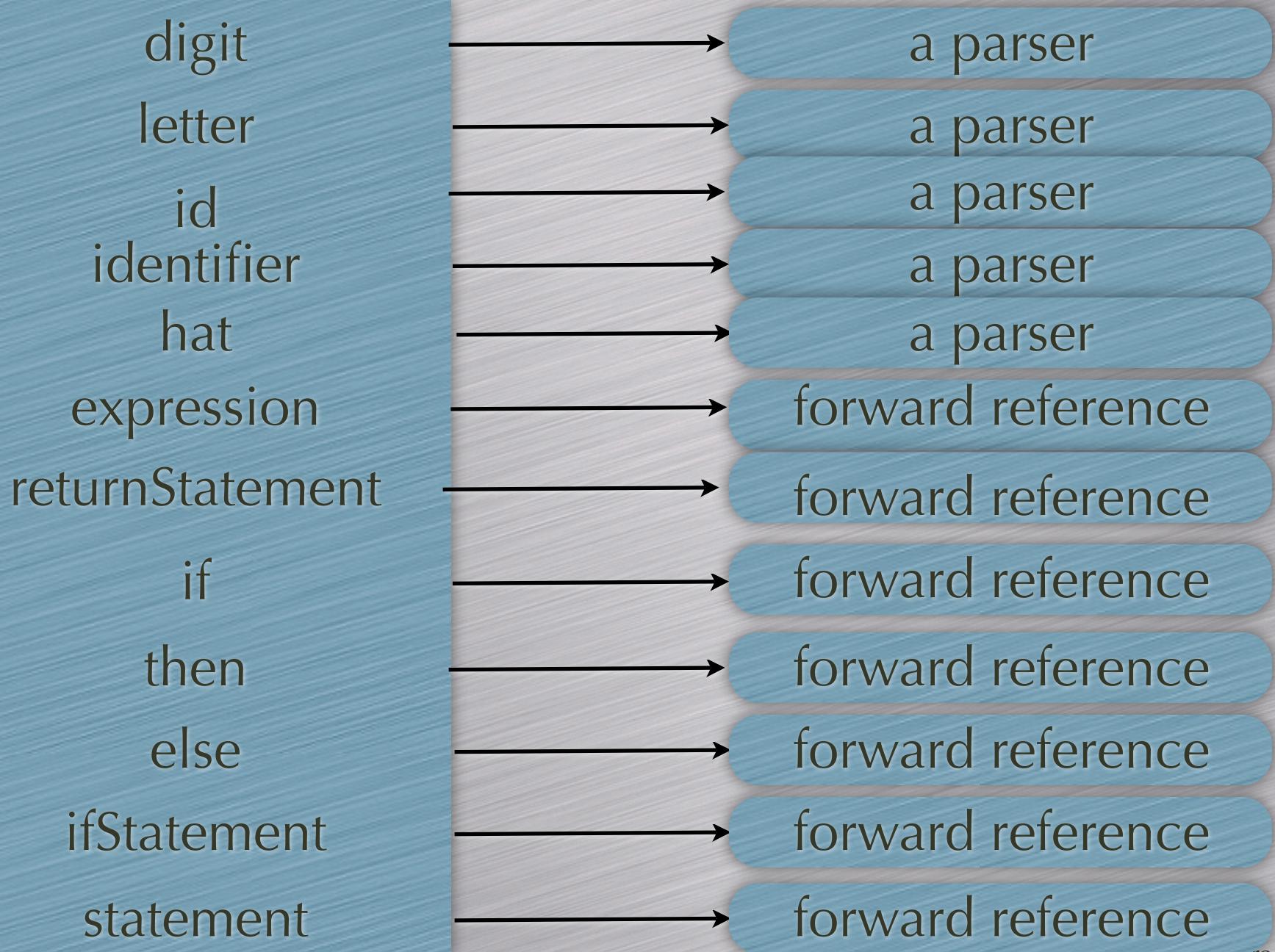
Mutual Recursion



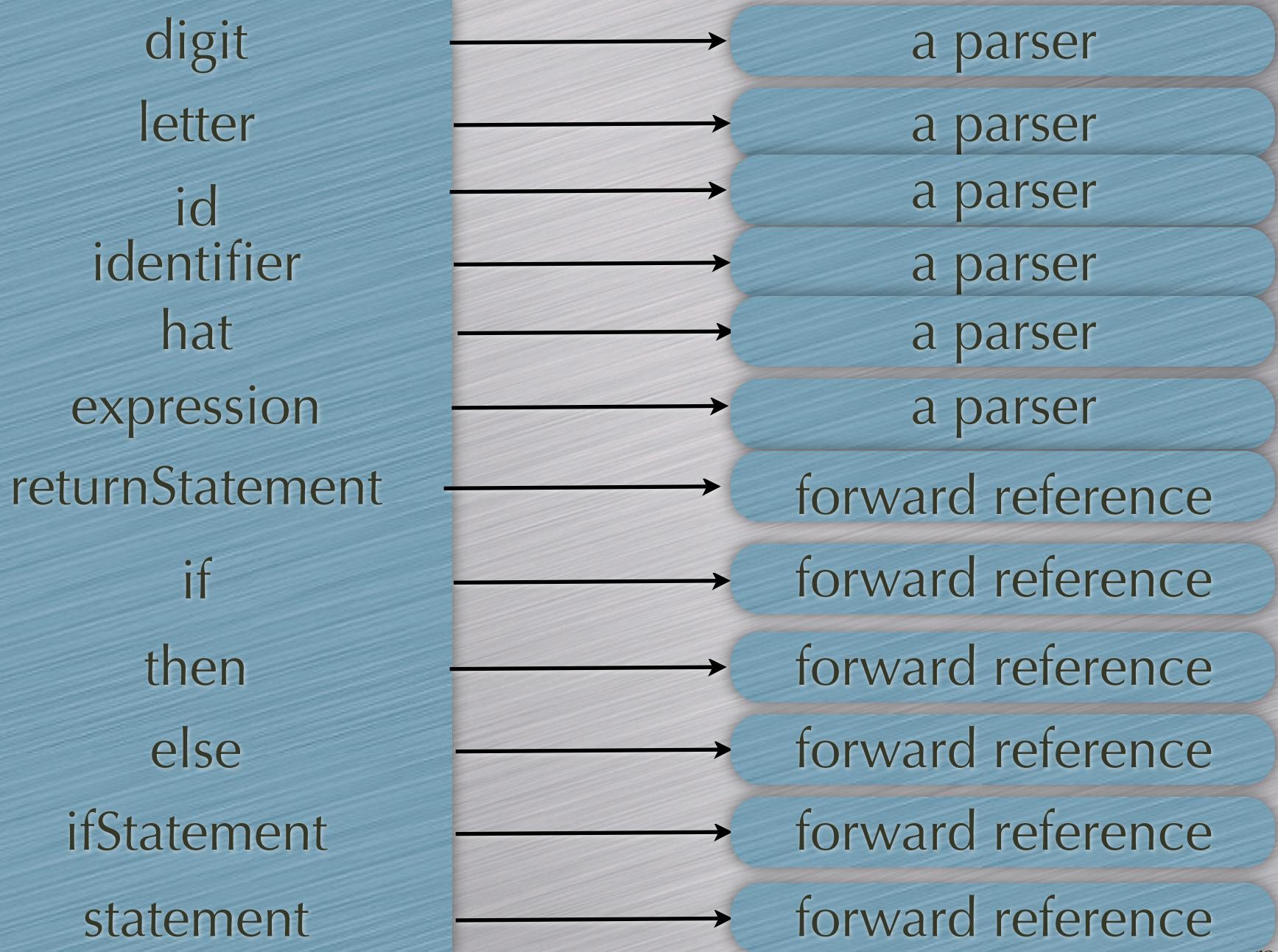
Mutual Recursion



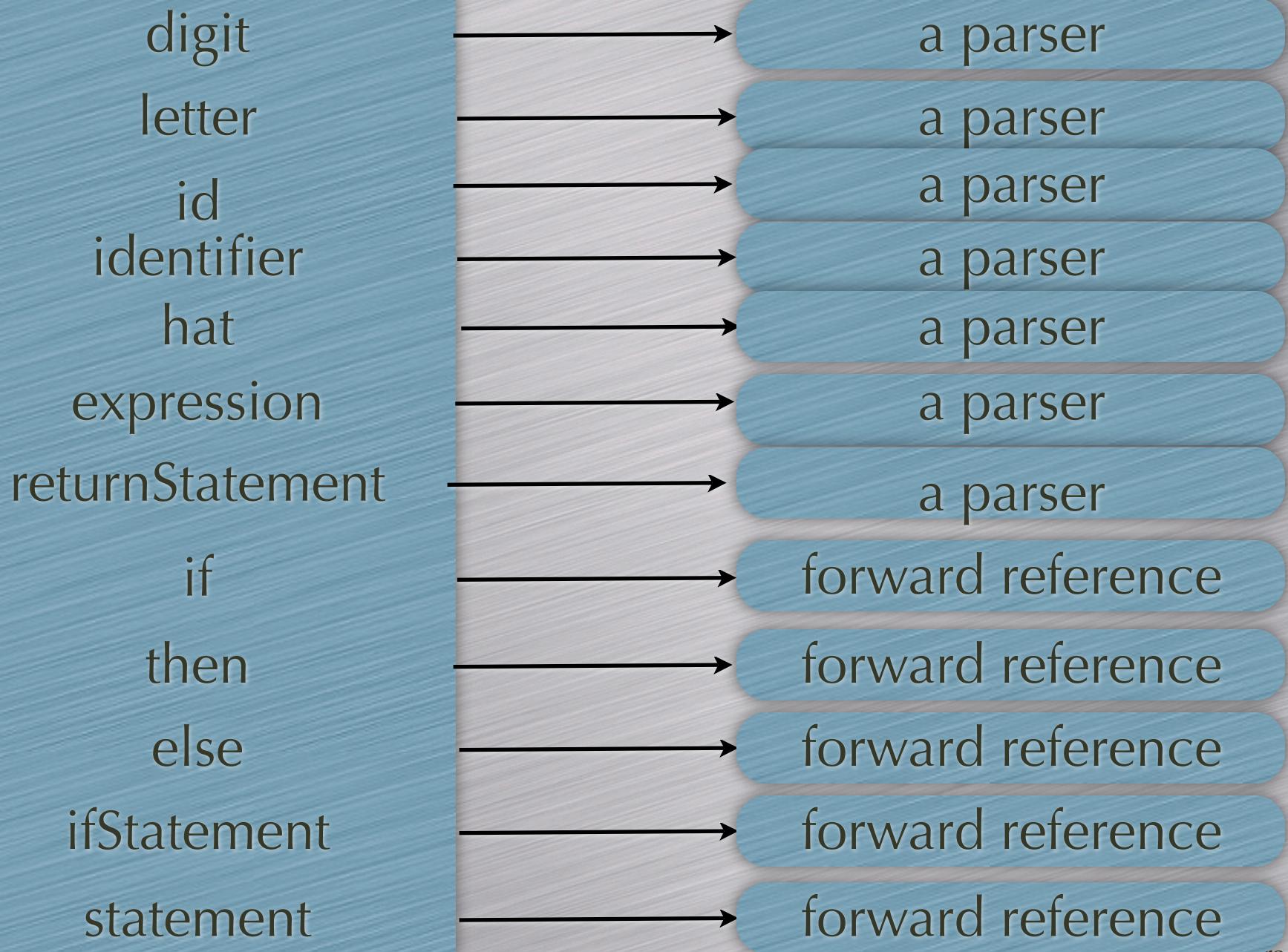
Mutual Recursion



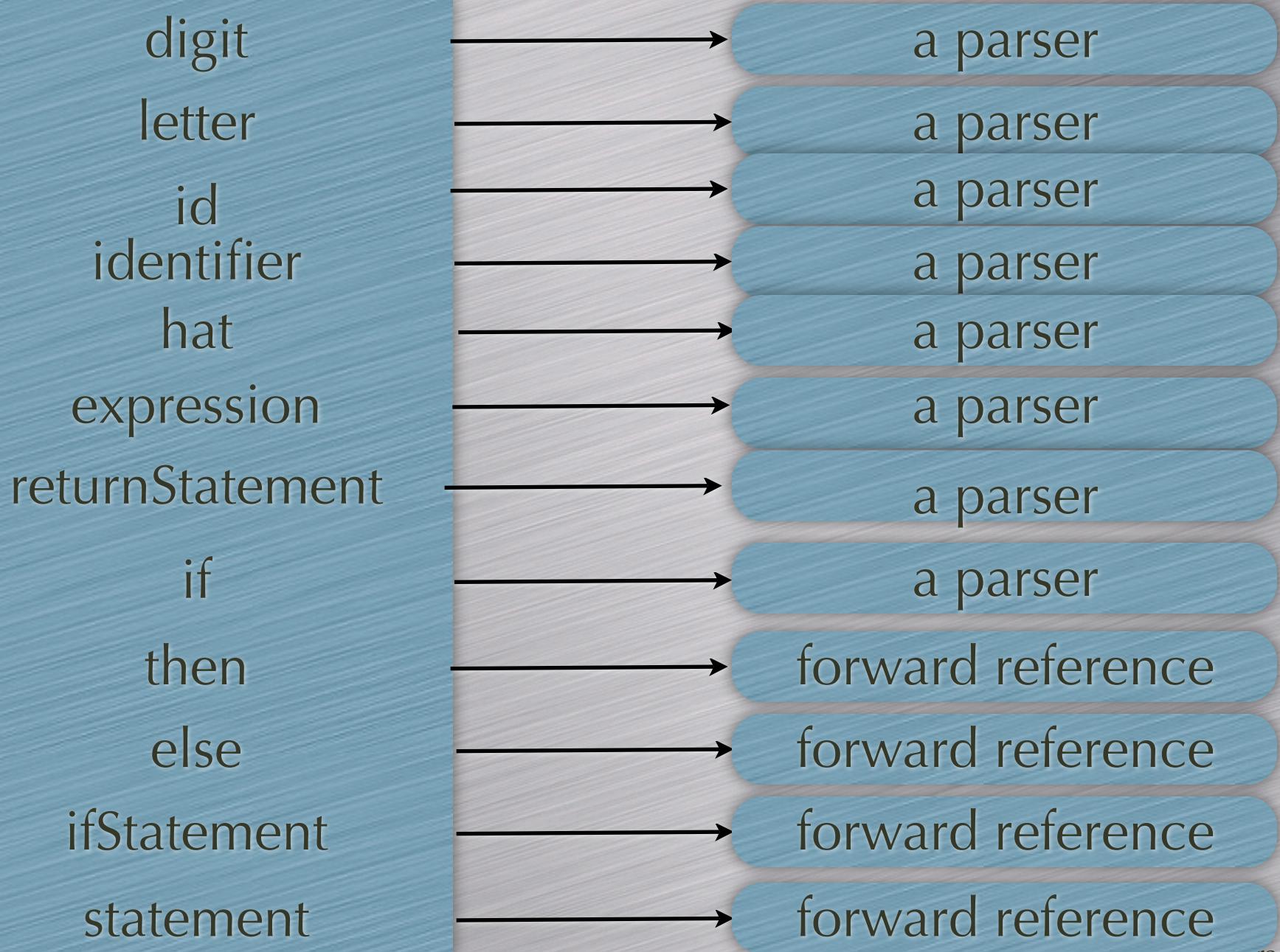
Mutual Recursion



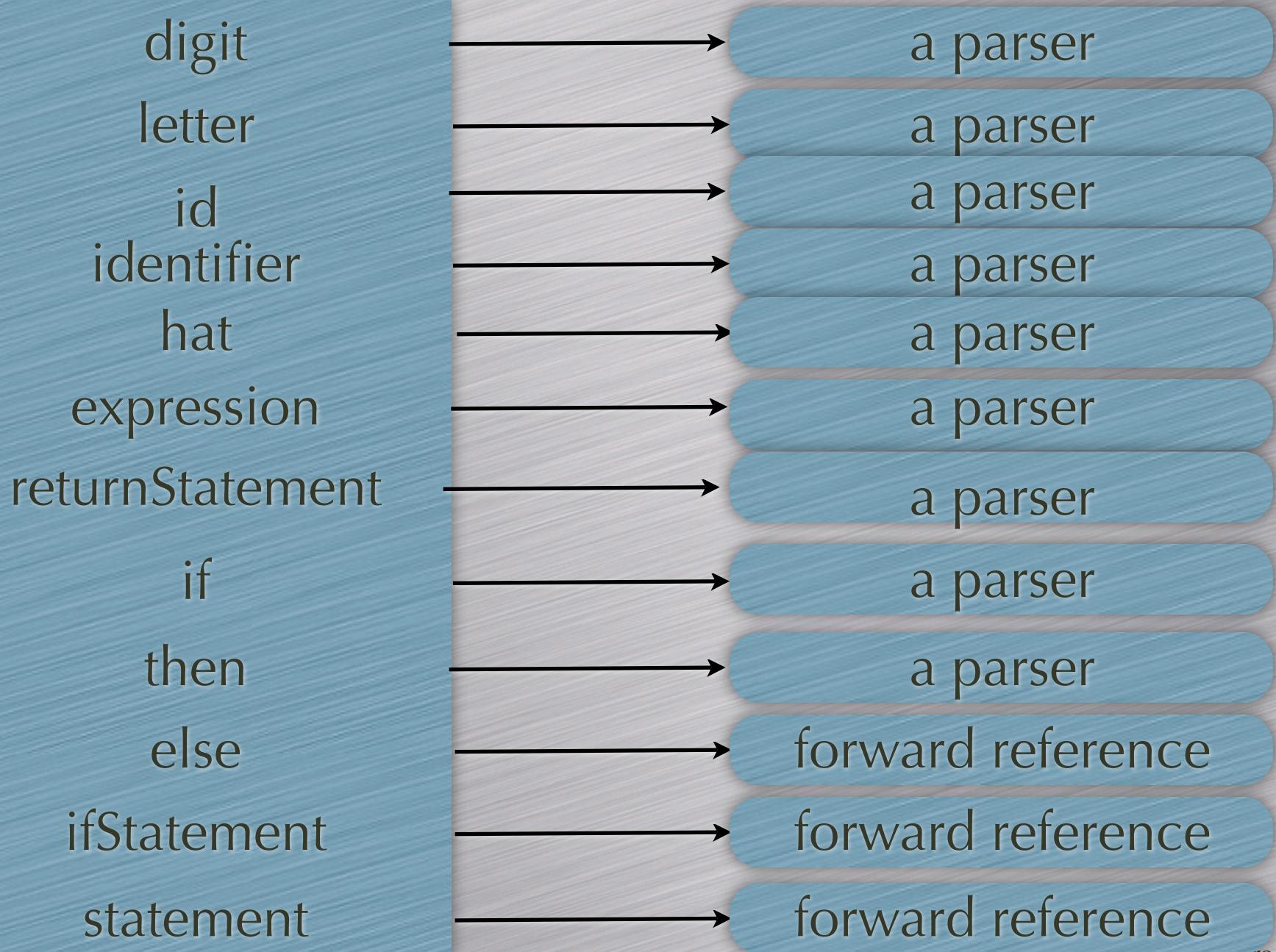
Mutual Recursion



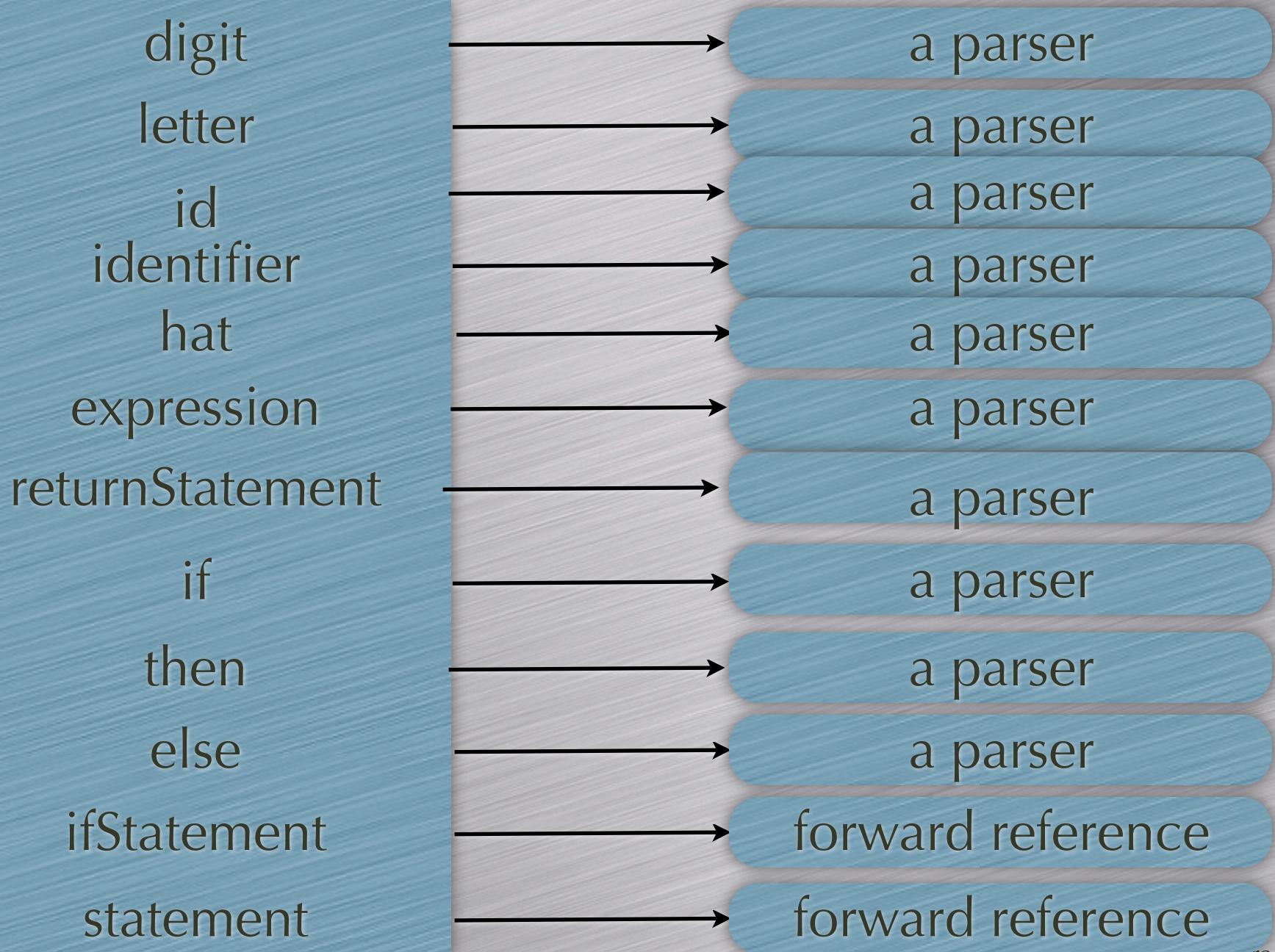
Mutual Recursion



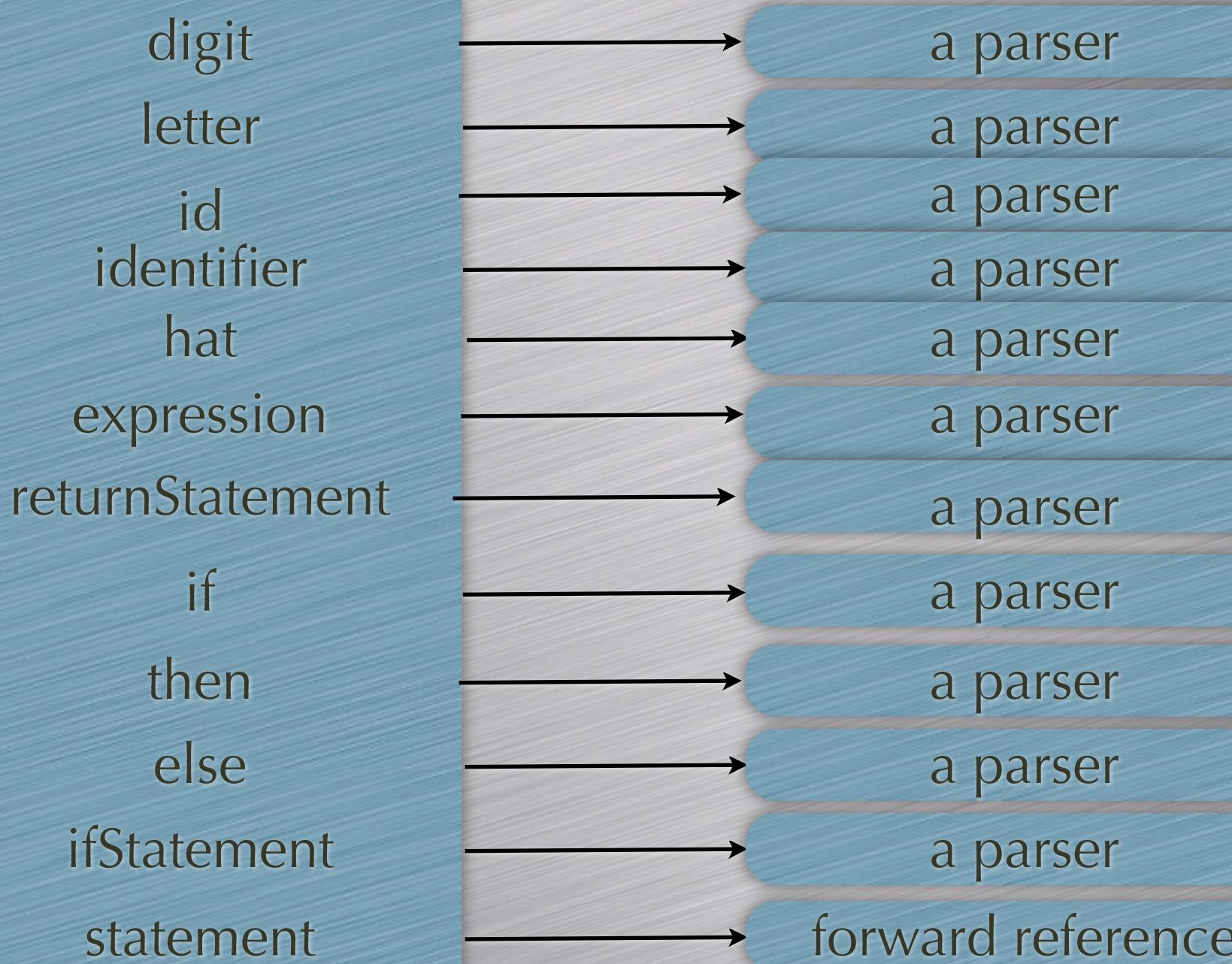
Mutual Recursion



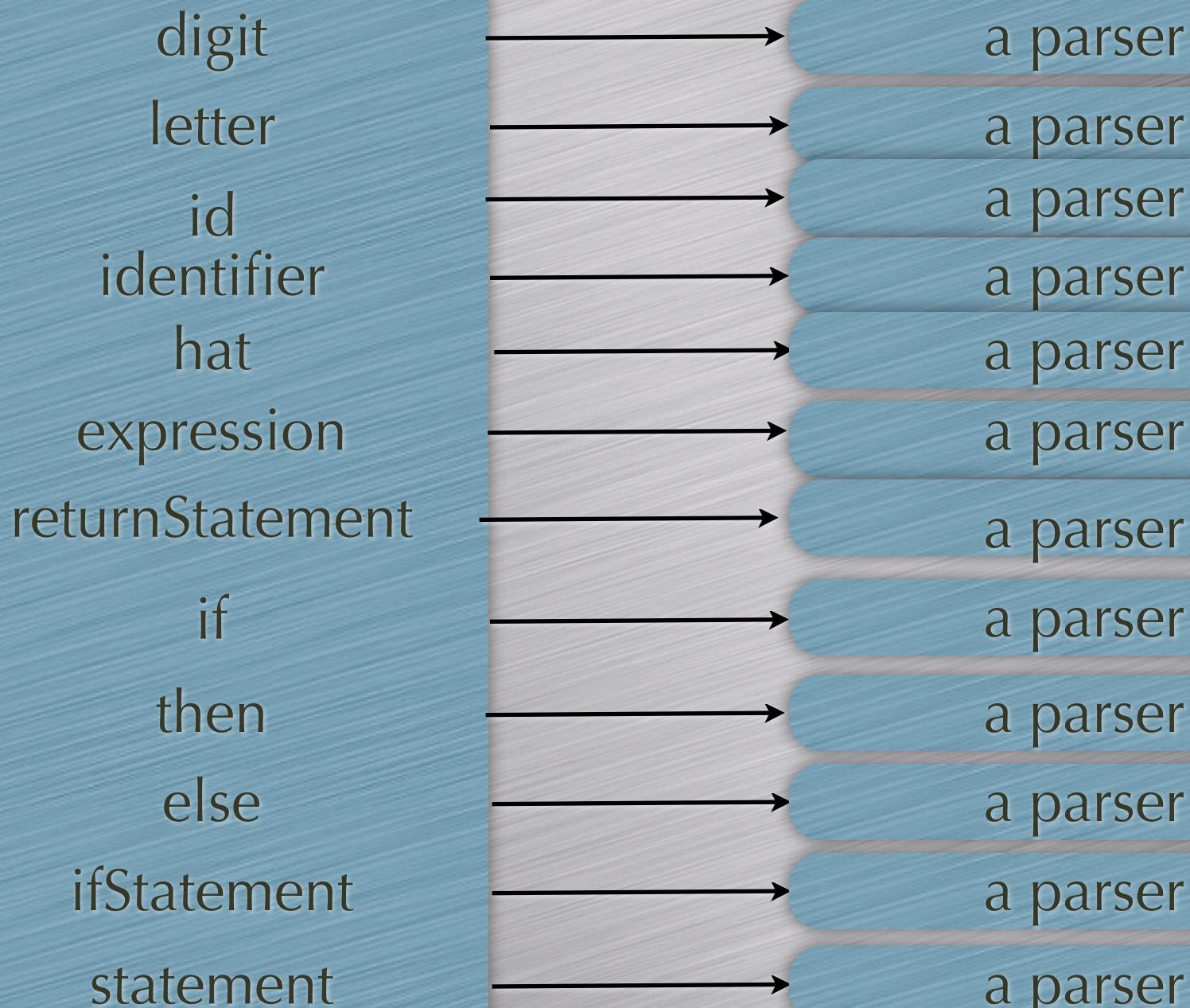
Mutual Recursion



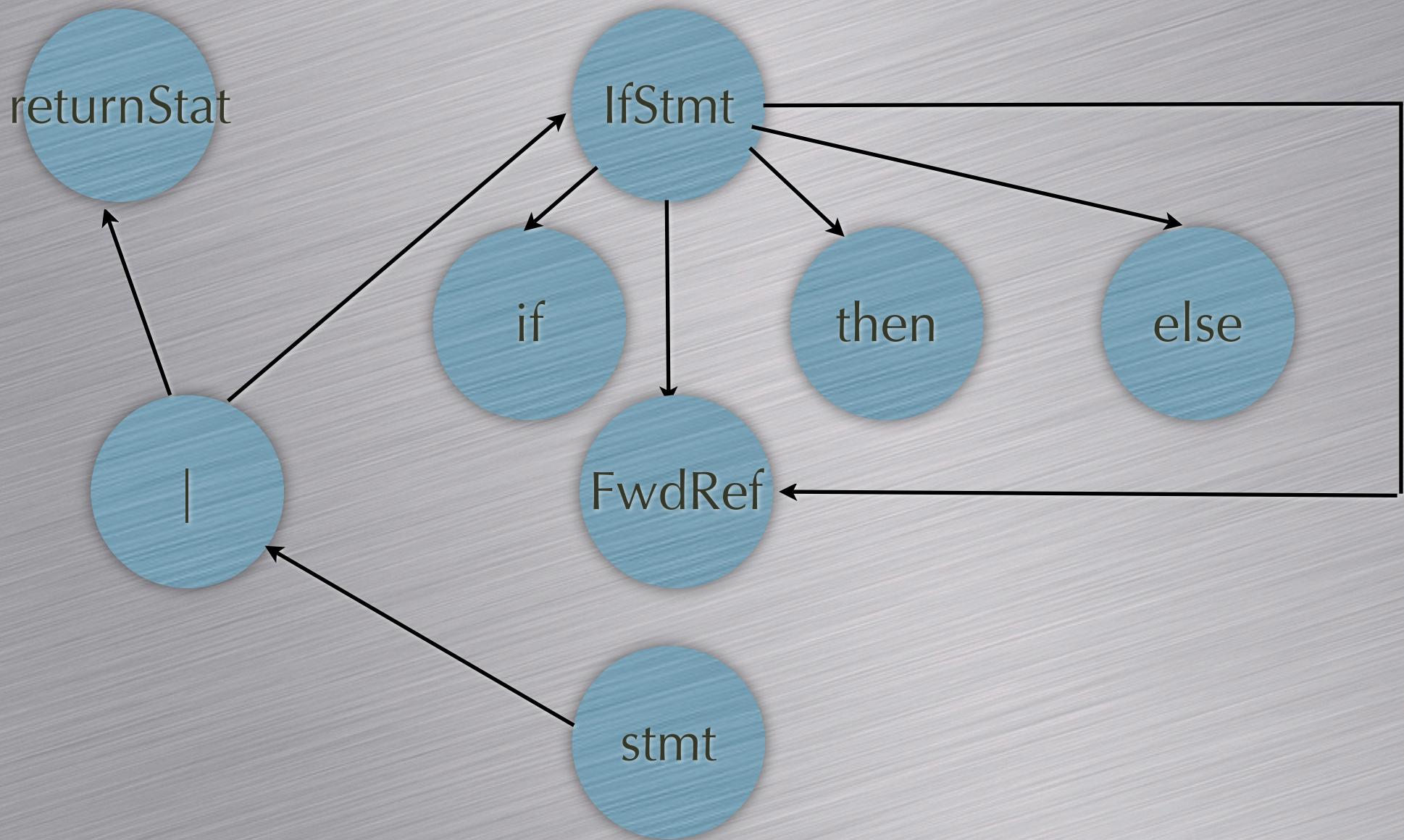
Mutual Recursion



Mutual Recursion



Mutual Recursion



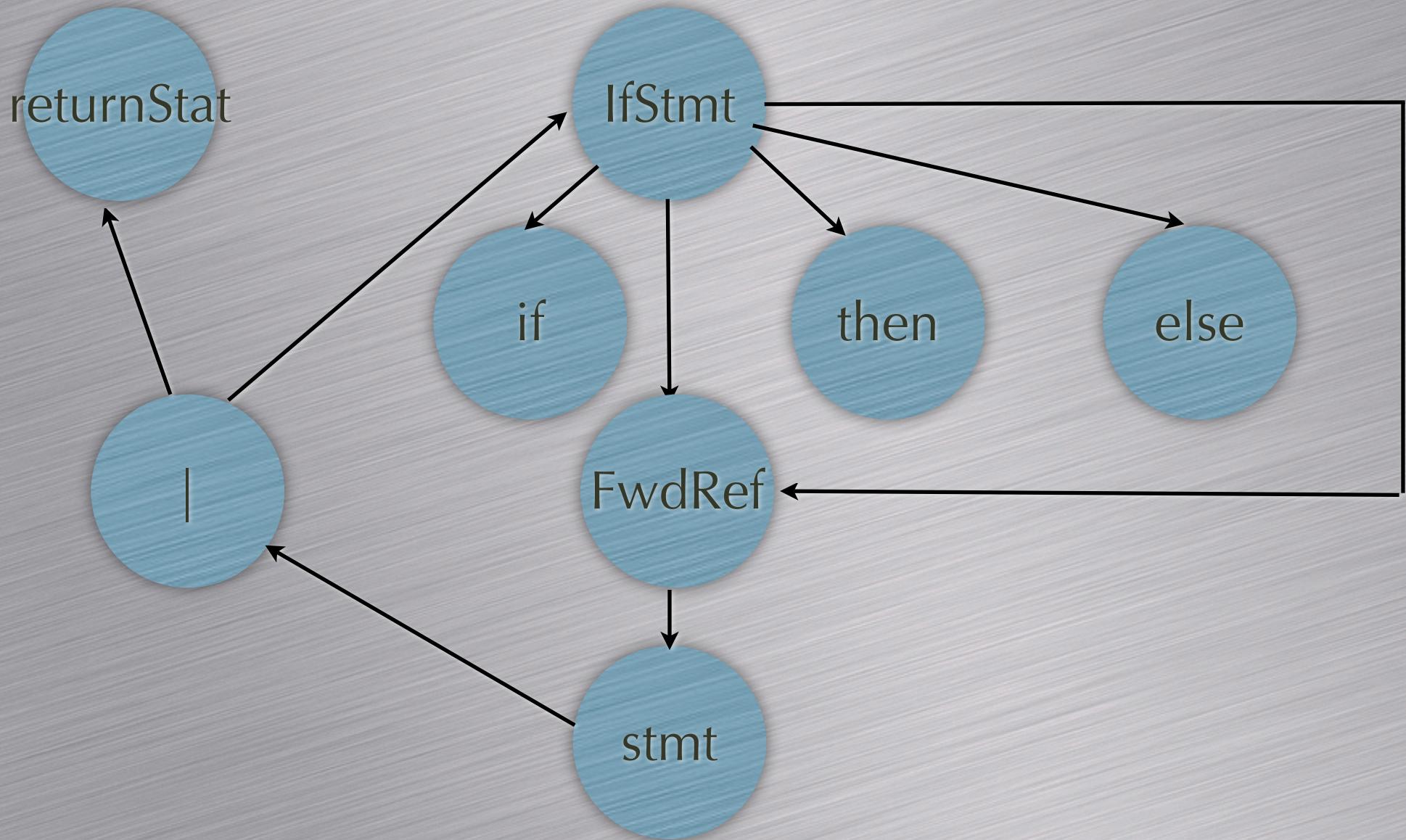
Being Reflective rather than Lazy

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

The screenshot shows a software interface titled "CombinatorialParsing". The main window contains a code editor with the following text:

```
Array = platform Collections Array    delete
BlockContext = platform BlockContext   delete
Error = platform Error    delete
OrderedCollection = platform Collections OrderedCollection   delete
SortedCollection = platform Collections SortedCollection   delete
show definition...
```

Below the code editor, there is a section titled "Classes" with the following list of parser classes:

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

At the bottom of the interface, there is a section titled "Methods" with the following buttons:

add method | expand | collapse

Nested Classes

Classes

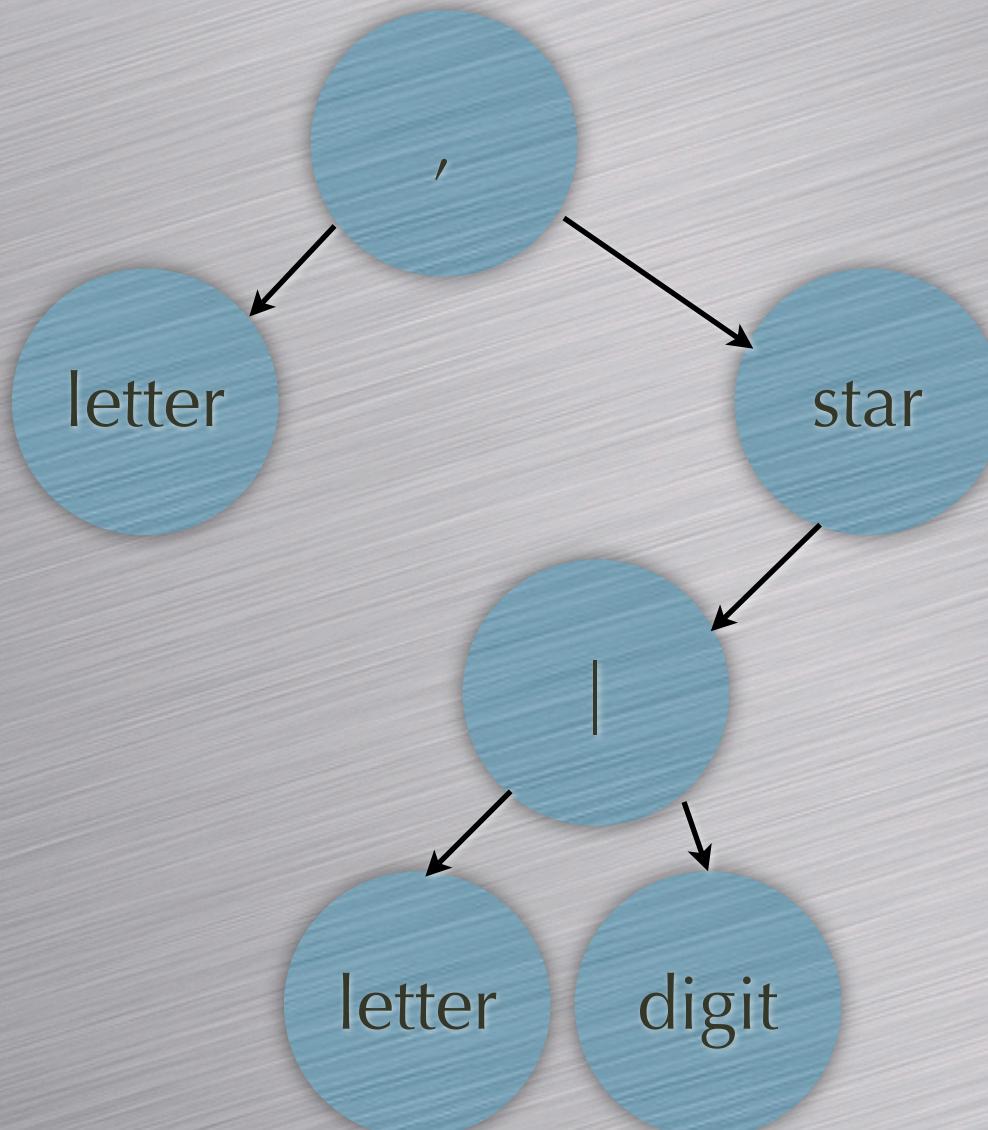
[add class](#) | [expand](#) | [collapse](#)

- ▶ [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

- ▶ SequentialParser
- ▶ StarParser

How it Works



$id = \text{letter}, (\text{letter} \mid \text{digit})^*$.

Modules

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



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

```
)
```



```
....
```

```
)
```

Modules up Close

CombinatorialParsing

GrammarExamples

classic

edit

add slot

▼ CombinatorialParsing usingLib: platform =

(no comment)

Array = platform Collections Array delete

BlockContext = platform BlockContext delete

Error = platform Error delete

OrderedCollection = platform Collections OrderedCollection delete

SortedCollection = platform Collections SortedCollection delete

show definition...

Classes add class | expand | collapse

Copyright 2005-2007 Gilad Bracha

But, this version uses Closures

ifStatement = *if*, [*expression*], [*then*], [*statement*],
[*else*], [*statement*].

statement = *ifStatement* | [*returnStatement*].

Closureless Version

BlocklessCombinatorialParsing

BlocklessCombinatorialParsing usingLib: platform = CombinatorialParsing usingLib: platform

A submodule that does not strictly require blocks to be used as inputs to the sequencing and alternating combinators

Dictionary = platform Collections Dictionary

List = platform Collections OrderedCollection

SuperAlternatingParser = super AlternatingParser

SuperCombinatorialParser = super CombinatorialParser

SuperSequentialParser = super SequentialParser

[show definition...](#)

Classes

- ▶ AlternatingParser
- ▶ CombinatorialParser
- ▶ ExecutableGrammar
- ▶ ForwardReferenceParser
- ▶ SequentialParser

Methods

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

The screenshot shows a software interface titled "AlternatingParser" with a toolbar at the top. The main area displays a snippet of code:

```
A parser that parses either P or Q.  
pO    delete  
pfun   delete  
qO    delete  
qfun   delete  
show definition...
```

Below this, there are sections for "Classes" and "Methods". The "Methods" section is expanded, showing:

- combineErrors:and:at:with: [refst](#) [delete](#)
- either:or: [refst](#) [delete](#)

The "either:or:" method is expanded, showing its implementation:

```
[either: pf1 or: pf2 = (  
  
    self assert:[(pf1 isKindOfClass: BlockContext) and: [pf2  
isKindOfClass: BlockContext]].  
    pf1:: pf1.  
    pf2:: pf2  
)  
  
p    refst delete  
parse:inContext:ifError: refst delete
```

Closure Version

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

Closureless Alternation

The screenshot shows a software interface for an 'AlternatingParser'. The title bar says 'AlternatingParser' with a red close button. Below the title are navigation icons: back, forward, home, and refresh. The current tab is 'GrammarExamples'. A search icon is also present.

The main content area displays the code for the `either:or:` method. The code is as follows:

```
either: pf1 or: pf2 = (
    pfun:: pf1.
    qfun:: pf2
)
```

Below the code, there are buttons for 'classic' (selected), 'edit', 'add slot', and 'show definition...'. There are also buttons for 'add class | expand | collapse' under 'Classes' and 'add method | expand | collapse' under 'Methods'.

Hierarchy Inheritance

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

Modules Using Modules

The screenshot shows a software interface for managing module definitions. At the top, there's a toolbar with icons for back, forward, home, refresh, and search. The title bar says "GrammarExamples". On the right side, there are buttons for "classic" (selected), "edit", and "add slot". A vertical scroll bar is on the right edge.

The main area displays a module definition:

```
GrammarExamples usingLib: platform parsingWith: parserLib =  
(no comment)
```

Below this, several platform components are listed with delete links:

- AST = platform AST [delete](#)
- Grammar = parserLib ExecutableGrammar [delete](#)
- ReturnStatAST = platform ReturnStatAST [delete](#)
- SuperParser = ExampleParser1 mixin lbo ExampleGrammar2 [delete](#)
- VariableAST = platform VariableAST [delete](#)

A "show definition..." link is located below the component list.

Below the component list, there's a section titled "Classes" with a "add class" button and "expand/collapse" links. The classes listed are:

- ExampleGrammar1
- ExampleGrammar2
- ExampleParser1
- ExampleParser2
- IfStatAST

At the bottom, there's a "Methods" section with a "add method" button and "expand/collapse" links.

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)
 - Omerta (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