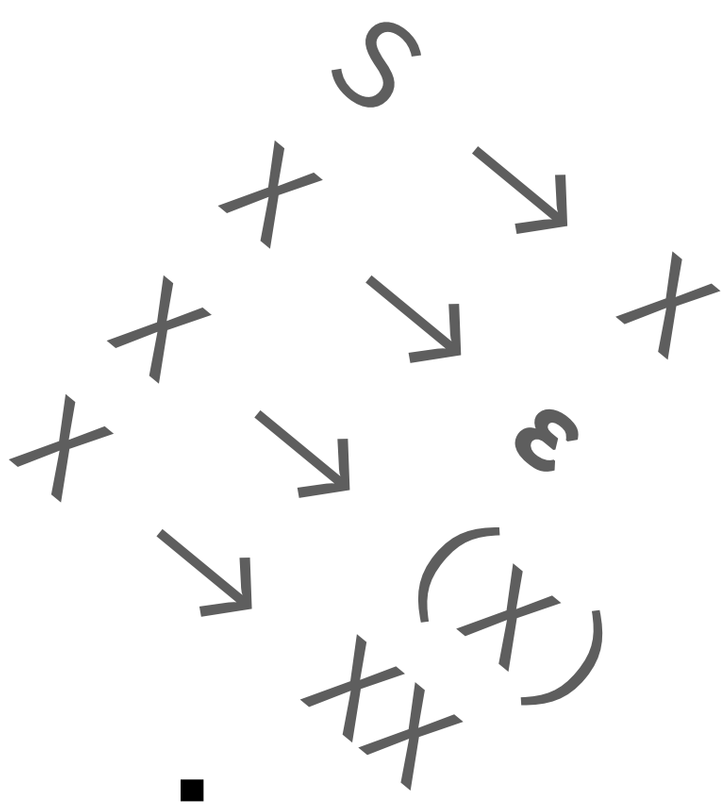

Tabling and Parsing

$$
\begin{aligned}
S &\rightarrow X \\
X &\rightarrow \varepsilon \\
X &\rightarrow (X)X
\end{aligned}
$$

What is Tabling Again?

```
p :: (Int,Int)
p = (1,2) ? flip swap

swap :: (a,b) → (b,a)
swap (x,y) = (y,x)
```

```
p :: [(Int,Int)]
p = [(1,2)] ++ swap p
swap :: [(a,b)] → [(b,a)]
swap xs = [(y,x) | (x,y) ← xs]
```

What is Tabling Again?

```
p :: [(Int,Int)]
p = [(1,2)] ++ swap p
swap :: [(a,b)] → [(b,a)]
swap xs = [(y,x) | (x,y) ← xs]
```

```
> p
[(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),
(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),
(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),
(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),(1,2),(2,1),
(1,2),(2,1),…
```

Tabling to the Rescue

```
type Open s = s → s
```

open recursion

```
p :: MonadPlus m
  ⇒ Open (() → m (Int,Int))
p p () = return (1,2) <|> swap p ()

swap :: Functor f ⇒ f (a,b) → f (b,a)
swap = fmap (\(x,y) → (y,x))
```

> fix p ()
[(1,2),(2,1),…

> runTbl p ()
Set.fromList [(1,2),(2,1)]

Parsing Time!

```
type Symbol = Char

isTerminal :: Symbol → Bool
isTerminal = not . isUpper
```

```
type Grammar = [Production]
data Production = Symbol :⟶ [Symbol]
  deriving Show
```

Parsing Time!

```
par :: Grammar
par = [ 'S' :→ "X",
        'X' :→ "",
        'X' :→ "(X)",
        'X' :→ "XX"
      ]
```

"", "()", "(())", "()()",...

Earley Parsing

Configurations:

$$X \rightarrow \alpha \,.\, \beta : k$$

non-terminal symbol

parsed prefix (reversed)

expected suffix

position of S in the input string

```
data Conf =
  Symbol :-•-> ([Symbol],[Symbol],Int)
```

Earley Parsing

Initialisation

$$S \rightarrow .\gamma : 0 \in C_0 \textbf{ for all } S \rightarrow \gamma \in G$$

production for the start symbol

```
guard (k == 0) >> return (s0 grammar) where
  s0 (s :→ g:_) = s :-•-> ([],g,0)
```

Earley Parsing

Prediction

$$C_k = \{Y \to \ .\gamma : k \mid X \to \alpha\ .\ Y\beta : l \in C_k, Y \to \gamma \in G\}$$

new configuration

next non-terminal Y

production for Y

```
do x :-•-> (a,y:b,j) ← states k
   guard (not $ isTerminal y)
   z :⟶ gamma ← msum (map return grammar)
   guard (z == y)
   return (y :-•-> ([],gamma,k))
```

Earley Parsing

Scanning

$$C_{k+1} = \{X \to \alpha w_k \,.\, \beta : l \mid X \to \alpha \,.\, w_k \beta : l \in C_k\}$$

new configuration

next terminal in the input $w = w_0 w_1 \cdots w_n$

```
do x :-•-> (a,wk:b,l) ← states (k - 1)
   guard (wk == w !! (k - 1))
   return (x :-•-> (wk:a,b,l))
```

Earley Parsing

Completion

$$C_k = \{X \to \alpha Y \,.\, \beta : j \mid Y \to \gamma \,.\, : l \in C_k, X \to \alpha \,.\, Y\beta : j \in C_l\}$$

unblocked configuration for X

completed configuration for Y

configuration for X blocked on Y

```
do y :-•-> (g,[],l) ← states k
   x :-•-> (a,z:b,j) ← states l
   guard (z == y)
   return (x :-•-> (z:a,b,j))]
```

Earley Parsing

Recognition

$$w \in L(G) \iff S \to \gamma\,. : 0 \in C_{|w|}$$

configurations after reading |w| terminals

```
earley :: Grammar → String → Bool
earley g@(s :→ w:_) str =
  let sFinal = s :-•-> (reverse w,[],0)
      n      = length str
  in sFinal`S.member` runTbl (states g str) n
```

```
states grammar str states k
  | k < 0 = Fail
  | otherwise = msum [
      guard (k == 0) >> return (s0 grammar) where
        s0 (s :→ g) = s :-•-> ([],g,0),
      do x :-•-> (a,y:b,j) ← states k
         guard (not $ isTerminal y)
         z :⟶ gamma ← msum (map return grammar)
         guard (z == y)
         return (y :-•-> ([],gamma,k)),
      do x :-•-> (a,wk:b,l) ← states (k - 1)
         guard (wk == w !! (k - 1))
         return (x :-•-> (wk:a,b,l)),
      do y :-•-> (g,[],l) ← states k
         x :-•-> (a,z:b,j) ← states l
         guard (z == y)
         return (x :-•-> (z:a,b,j))]
earley :: Grammar → String → Bool
earley g@(s :→ w:_) str =
  let sFinal = s :-•-> (reverse w,[],0)
      n      = length str
  in sFinal`S.member` runTbl (states g str) n
```

```
data ParseTree = PT Symbol [Symbol] deriving (Eq,Ord)
states grammar str states k
  | k < 0 = mzero
  | otherwise = msum [
     guard (k == 0) >> return (s0 grammar,[]) where
       s0 (s :→ g) = s :-•-> ([],g,0),
     do x :-•-> ((a,y:b,j),ptx) ← states k
        guard (not $ isTerminal y)
        z :⟶ gamma ← msum (map return grammar)
        guard (z == y)
        return (y :-•-> ([],gamma,k),[]),
     do (x :-•-> (a,wk:b,l),ptx) ← states (k - 1)
        guard (wk == w !! (k - 1))
        return (x :-•-> (wk:a,b,l),PT wk []:ptx),
     do (y :-•-> (g,[],l),pty) ← states k
        (x :-•-> (a,z:b,j),ptx) ← states l
        guard (z == y)
        return (x :-•-> (y:a,b,j),PT y (reverse pty):ptx)]
earleyParseTrees :: Grammar → String → [ParseTree]
earleyParseTrees g@(s :→ w:_) str =
  let sFinal (conf,pts) =
        [PT s reverse pts | conf == s :-•-> (reverse w,[],0)]
      n = length str
  in concatMap sFinal . S.toList . runTbl (states g str) $ n
```

```
earleyParseTrees :: Grammar → String → [ParseTree]
earleyParseTrees g@(s :→ w:_) str =
  let sFinal (conf,pts) =
        [PT s reverse pts | conf == s :-•-> (reverse w,[],0)]
      n = length str
  in concatMap sFinal . S.toList . runTbl (states g str) $ n
```

```
earley :: Grammar → String → Bool
earley grammar = not . null . earleyParseTrees grammar
```

Identical?

What's Next

- ✦Parser Combinators - Applicative
- ✦Returning values - Hasochism
- ✦Other algorithms - A zoo (chart parsers, LL(k), DFA minimisation)