スキー合宿勉強会1日目

Haskell,Dynet,TreeLSTM

2018/03/07

吉川将司

今回やること

- Haskellで深層学習
  - Dynetライブラリ
- TreeLSTMを用いて含意関係認識(RTE)
  - ２文の間に含意関係が成り立つか予測
  - SNLIデータセット(Bowman+, 2016)

Dynet(Neubig+, 2017)

- 深層学習ライブラリ(C++, Python)
  - CMUのNLPに強い研究室が開発
    - 系列、木など複雑な構造と相性◯
- Haskellラッパー作ったよ(吉川)
  - 構造データとさらに相性◯？
  - ちょっとメンテナンスが大変…

余談: tensorflow-haskellは dependently-typed

```haskell
{-# LANGUAGE DataKinds, ScopedTypeVariables #-}

import             Data.Maybe (fromJust)
import             Data.Vector.Sized (Vector, fromList)
import             TensorFlow.DepTyped

test :: IO (Vector 8 Float)
test = runSession $ do
  (x :: Placeholder "x" '[4,3] Float) <- placeholder
  let elems1 = fromJust $ fromList [1,2,3,4,1,2]
      elems2 = fromJust $ fromList [5,6,7,8]
      (w :: Tensor '[3,2] '[] Build Float) = constant elems1
      (b :: Tensor '[4,1] '[] Build Float) = constant elems2
      y = (x `matMul` w) `add` b

  let (inputX :: TensorData "x" [4,3] Float) =
        encodeTensorData . fromJust $ fromList [1,2,3,4,1,0,7,9,5,3,5,4]

  runWithFeeds (feed x inputX :~~ NilFeedList) y
```

- 真似したい!
- 疑問: 我々NLPerは実行時にshapeを決めたい(実験等)
  - 存在量化？複雑な演算はできるか？(concatして行列かける、など)

SNLIデータセット

- 含意関係認識タスクのデータセット
  - ２文に対して含意関係があるか
  - 木構造のアノテーション付き
  - 学習データ55万ペア、評価1万ペア
    - (Haskell)Stringだとデータ読み込みが終わらない…

People jump over a mountain crevasse on a rope.

→ People are jumping outside.

LSTM

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i \boldsymbol{x}_t + \boldsymbol{U}_i \boldsymbol{h}_{t-1} + \boldsymbol{b}_i)$$

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_f \boldsymbol{x}_t + \boldsymbol{U}_f \boldsymbol{h}_{t-1} + \boldsymbol{b}_f)$$

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o \boldsymbol{x}_t + \boldsymbol{U}_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o)$$

$$\boldsymbol{u}_t = \tanh(\boldsymbol{W}_u \boldsymbol{x}_t + \boldsymbol{U}_u \boldsymbol{h}_{t-1} + \boldsymbol{b}_u)$$

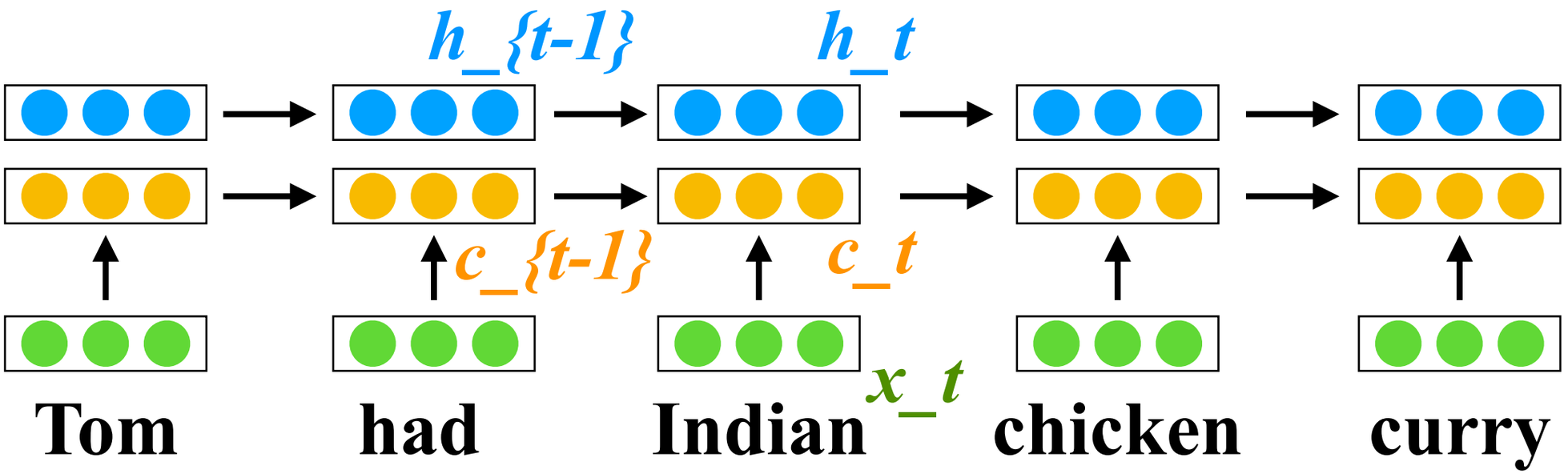
$$\boldsymbol{c}_t = \boldsymbol{c}_{t-1} \otimes \boldsymbol{f}_t + \boldsymbol{u}_t \otimes \boldsymbol{i}_t$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \otimes \tanh(\boldsymbol{c}_t)$$

入力(input)、忘却(forget)、
出力(output)ゲート

u: 新しい情報

セル:これまでの情報を忘れ、
新しい情報を入れる



TreeLSTM(Tai+, 2015)

$$\boldsymbol{i} = \sigma(\boldsymbol{W}_i\boldsymbol{x}_t + \boldsymbol{U}_i\boldsymbol{h}_L + \boldsymbol{V}_i\boldsymbol{h}_R + \boldsymbol{b}_i)$$

$$\boldsymbol{f}_L = \sigma(\boldsymbol{W}_{fL}\boldsymbol{x}_t + \boldsymbol{U}_{fL}\boldsymbol{h}_L + \boldsymbol{V}_{fL}\boldsymbol{h}_R + \boldsymbol{b}_{fL})$$

$$\boldsymbol{f}_R = \sigma(\boldsymbol{W}_{fR}\boldsymbol{x}_t + \boldsymbol{U}_{fR}\boldsymbol{h}_L + \boldsymbol{V}_{fR}\boldsymbol{h}_R + \boldsymbol{b}_{fR})$$

$$\boldsymbol{o} = \sigma(\boldsymbol{W}_o\boldsymbol{x}_t + \boldsymbol{U}_o\boldsymbol{h}_L + \boldsymbol{V}_o\boldsymbol{h}_R + \boldsymbol{b}_o)$$

$$\boldsymbol{u} = \tanh(\boldsymbol{W}_u\boldsymbol{x}_t + \boldsymbol{U}_u\boldsymbol{h}_L + \boldsymbol{V}_u\boldsymbol{h}_R + \boldsymbol{b}_u)$$

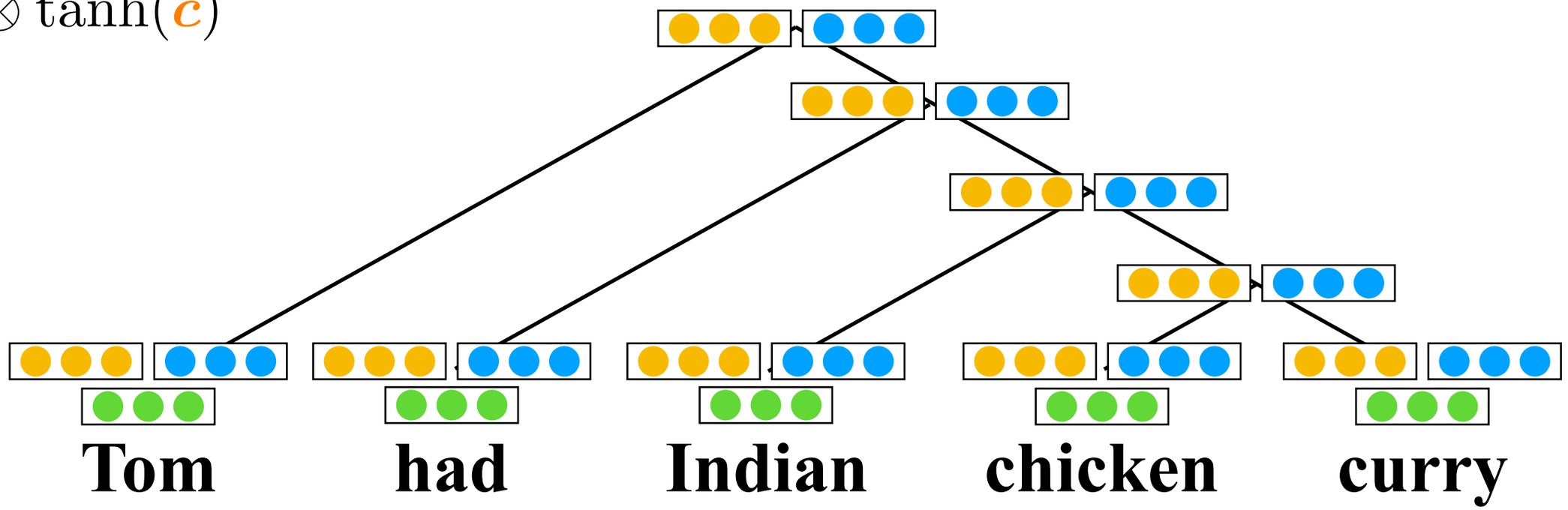
$$\boldsymbol{c} = \boldsymbol{c}_L \otimes \boldsymbol{f}_L + \boldsymbol{c}_R \otimes \boldsymbol{f}_R + \boldsymbol{u} \otimes \boldsymbol{i}$$

$$\boldsymbol{h} = \boldsymbol{o} \otimes \tanh(\boldsymbol{c})$$

同様にゲート

u: 新しい情報

セル:同様に忘却と新しい入力



TreeLSTM(Tai+, 2015)

$$\boldsymbol{i} = \sigma(\boldsymbol{W}_i \boldsymbol{x}_t + \boldsymbol{U}_i \boldsymbol{h}_L + \boldsymbol{V}_i \boldsymbol{h}_R + \boldsymbol{b}_i)$$

$$\boldsymbol{f}_L = \sigma(\boldsymbol{W}_{fL} \boldsymbol{x}_t + \boldsymbol{U}_{fL} \boldsymbol{h}_L + \boldsymbol{V}_{fL} \boldsymbol{h}_R + \boldsymbol{b}_{fL})$$

$$\boldsymbol{f}_R = \sigma(\boldsymbol{W}_{fR} \boldsymbol{x}_t + \boldsymbol{U}_{fR} \boldsymbol{h}_L + \boldsymbol{V}_{fR} \boldsymbol{h}_R + \boldsymbol{b}_{fR})$$

$$\boldsymbol{o} = \sigma(\boldsymbol{W}_o \boldsymbol{x}_t + \boldsymbol{U}_o \boldsymbol{h}_L + \boldsymbol{V}_o \boldsymbol{h}_R + \boldsymbol{b}_o)$$

$$\boldsymbol{u} = \tanh(\boldsymbol{W}_u \boldsymbol{x}_t + \boldsymbol{U}_u \boldsymbol{h}_L + \boldsymbol{V}_u \boldsymbol{h}_R + \boldsymbol{b}_u)$$

$$\boldsymbol{c} = \boldsymbol{c}_L \otimes \boldsymbol{f}_L + \boldsymbol{c}_R \otimes \boldsymbol{f}_R + \boldsymbol{u} \otimes \boldsymbol{i}$$

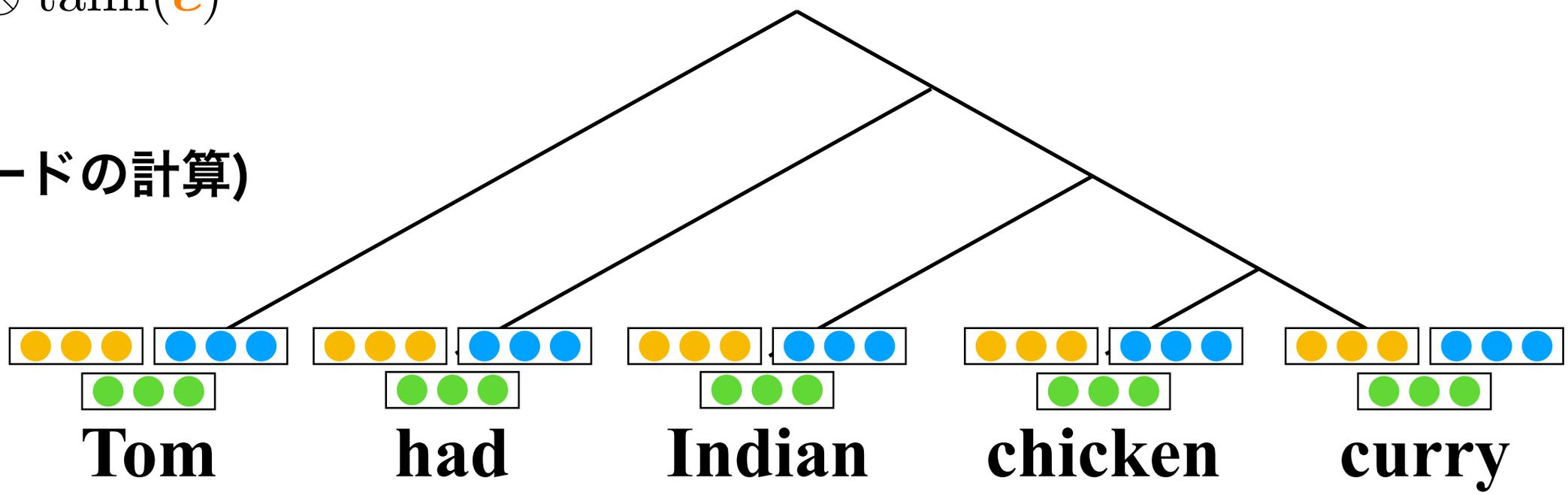
$$\boldsymbol{h} = \boldsymbol{o} \otimes \tanh(\boldsymbol{c})$$

同様にゲート

u: 新しい情報

セル:同様に忘却と新しい入力

(終端ノードの計算)



TreeLSTM(Tai+, 2015)

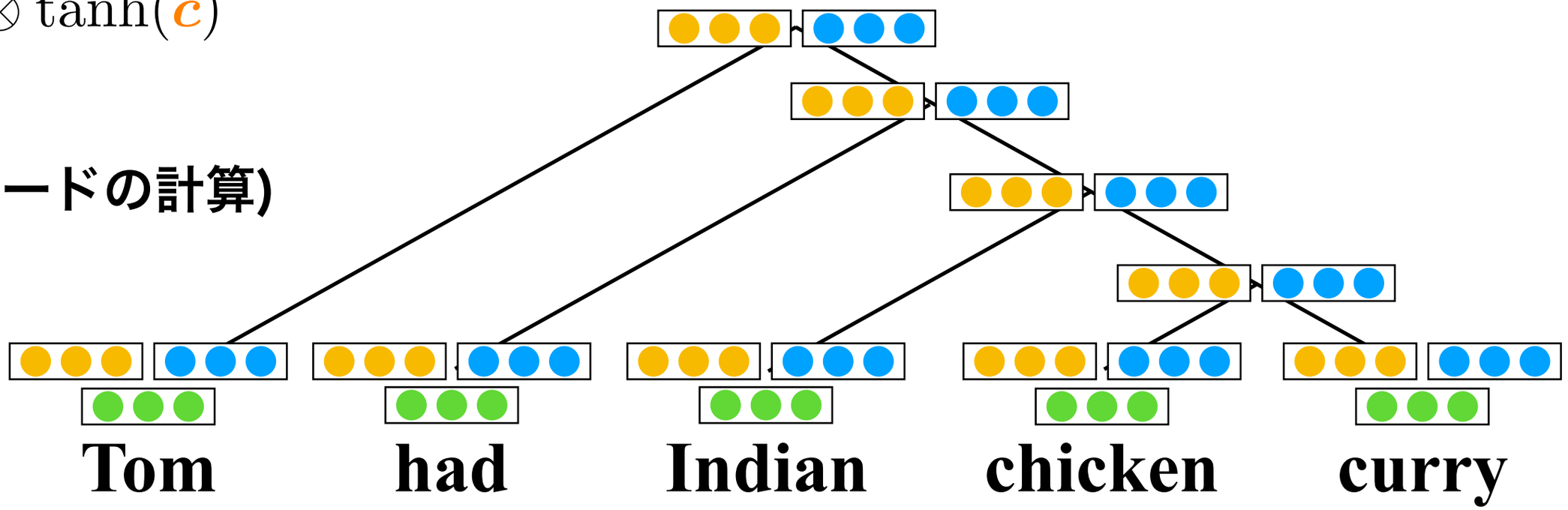
$$\boldsymbol{i} = \sigma(\boldsymbol{W}_i\boldsymbol{x}_t + \boldsymbol{U}_i\boldsymbol{h}_L + \boldsymbol{V}_i\boldsymbol{h}_R + \boldsymbol{b}_i)$$
$$\boldsymbol{f}_L = \sigma(\boldsymbol{W}_{fL}\boldsymbol{x}_t + \boldsymbol{U}_{fL}\boldsymbol{h}_L + \boldsymbol{V}_{fL}\boldsymbol{h}_R + \boldsymbol{b}_{fL})$$
$$\boldsymbol{f}_R = \sigma(\boldsymbol{W}_{fR}\boldsymbol{x}_t + \boldsymbol{U}_{fR}\boldsymbol{h}_L + \boldsymbol{V}_{fR}\boldsymbol{h}_R + \boldsymbol{b}_{fR})$$
$$\boldsymbol{o} = \sigma(\boldsymbol{W}_o\boldsymbol{x}_t + \boldsymbol{U}_o\boldsymbol{h}_L + \boldsymbol{V}_o\boldsymbol{h}_R + \boldsymbol{b}_o)$$
$$\boldsymbol{u} = \tanh(\boldsymbol{W}_u\boldsymbol{x}_t + \boldsymbol{U}_u\boldsymbol{h}_L + \boldsymbol{V}_u\boldsymbol{h}_R + \boldsymbol{b}_u)$$
$$\boldsymbol{c} = \boldsymbol{c}_L \otimes \boldsymbol{f}_L + \boldsymbol{c}_R \otimes \boldsymbol{f}_R + \boldsymbol{u} \otimes \boldsymbol{i}$$
$$\boldsymbol{h} = \boldsymbol{o} \otimes \tanh(\boldsymbol{c})$$

同様にゲート

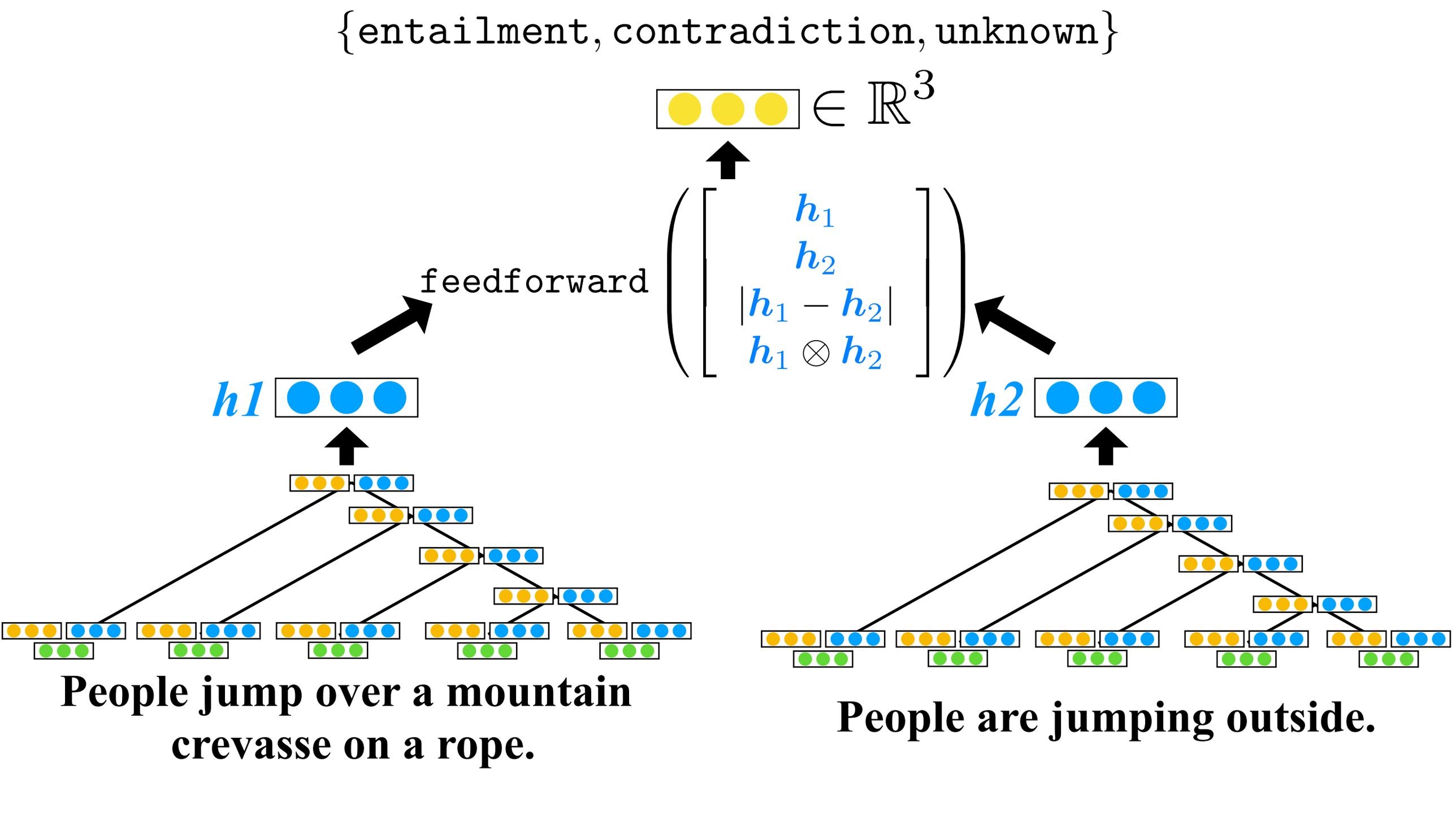
u: 新しい情報

セル:同様に忘却と新しい入力

(非終端ノードの計算)



TreeLSTMによるRTE



実装を見てみる

簡単な拡張

- パラメータのチューニング (勾配降下アルゴリズム等も)
- 事前学習した埋め込みベクトル
- TreeLSTMの前に文をLSTMで走査する
  - よりリッチな文脈情報を考慮
- SNLIのよりリッチな木を使う
  - 今回は単語だけの二分木

Advancedな拡張

- バッチ処理による高速化
    - できるらしい
- Gumbel softmax(Choi+, 2017), 強化学習(Yogatama+, 2016), CKY法(Maillard+, 2017)
    - 木構造も学習

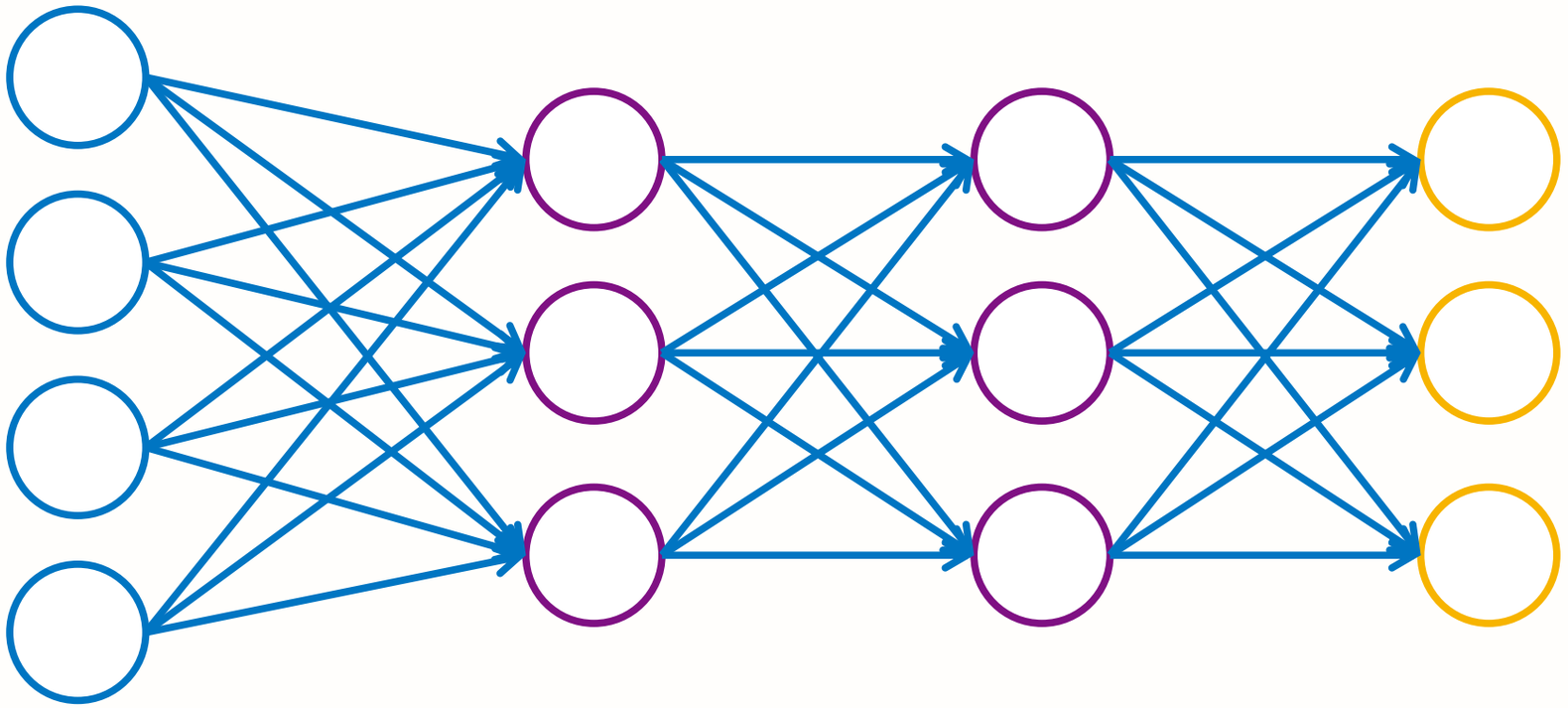
計算グラフについて

資料

PFN奥田さんのスライドより

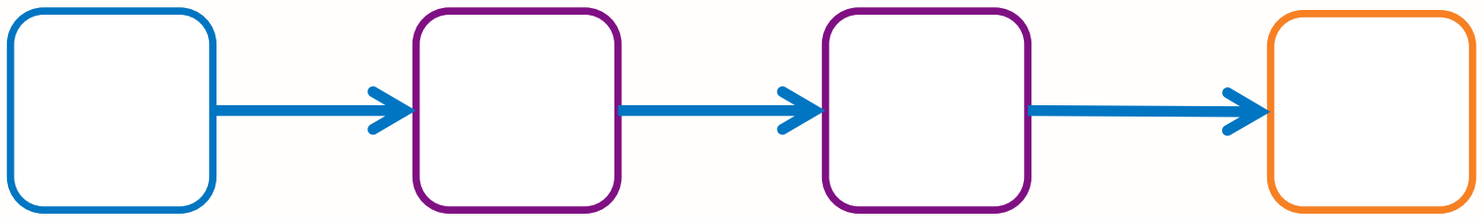
https://www.slideshare.net/ryokuta/chainer-59180103

ニューラルネット



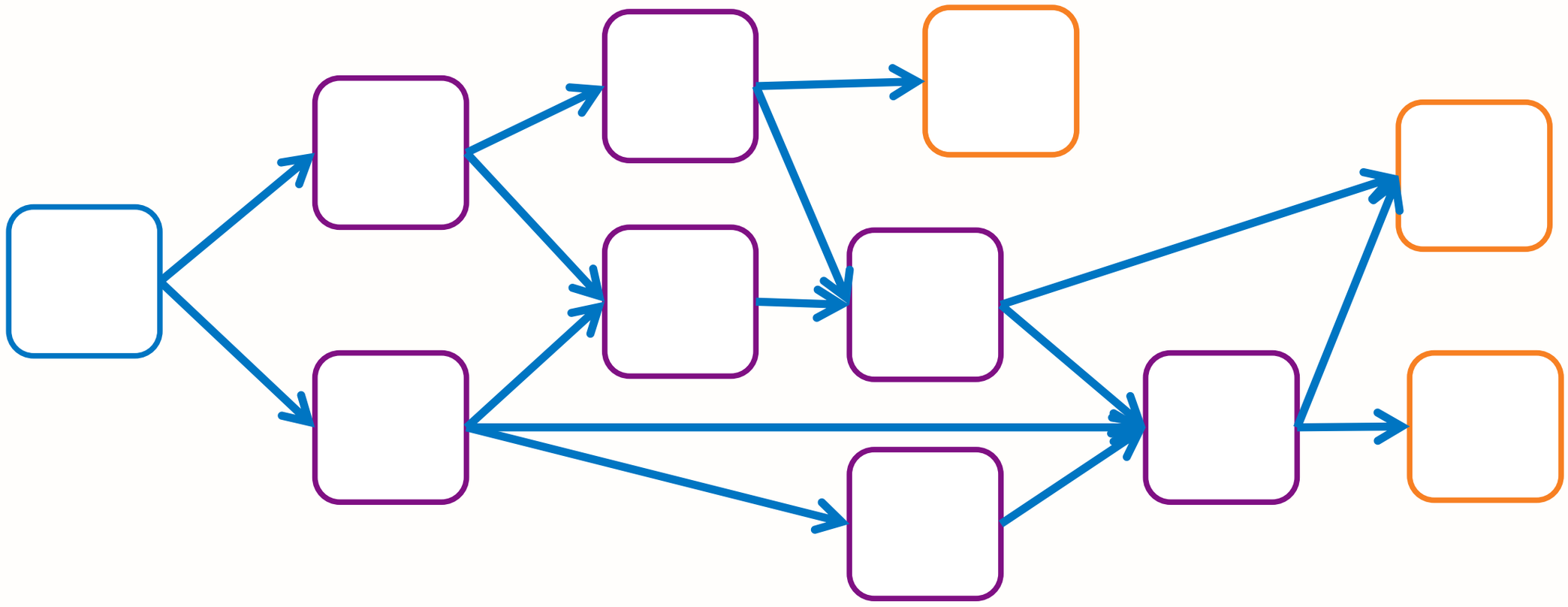
- 値が伝播していく有向グラフ
- エッジで重みをかけて、ノードに入るところで足し込み、ノードの中で非線形変換する
- 全体としては**巨大で複雑な関数**を表す

ニューラルネット＝合成関数



- ベクトルに対して線形・非線形な関数をたくさん適用する**合成関数**と捉えるとよい
- 各ノードはベクトルを保持する変数（テンソル）

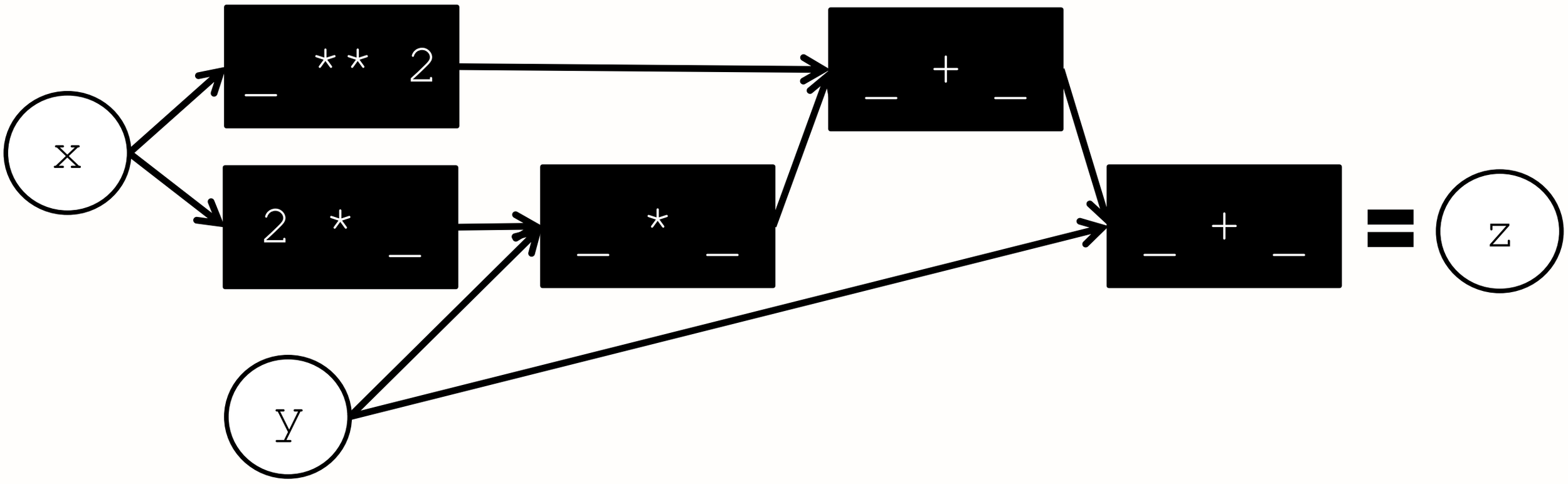
一般のニューラルネットは DAG = 計算グラフ



一般にはグラフが分岐したり合流したりする

- **分岐**：同じ変数を複数の場所でつかう
- **合流**：二つ以上の変数を受け取る関数を適用する

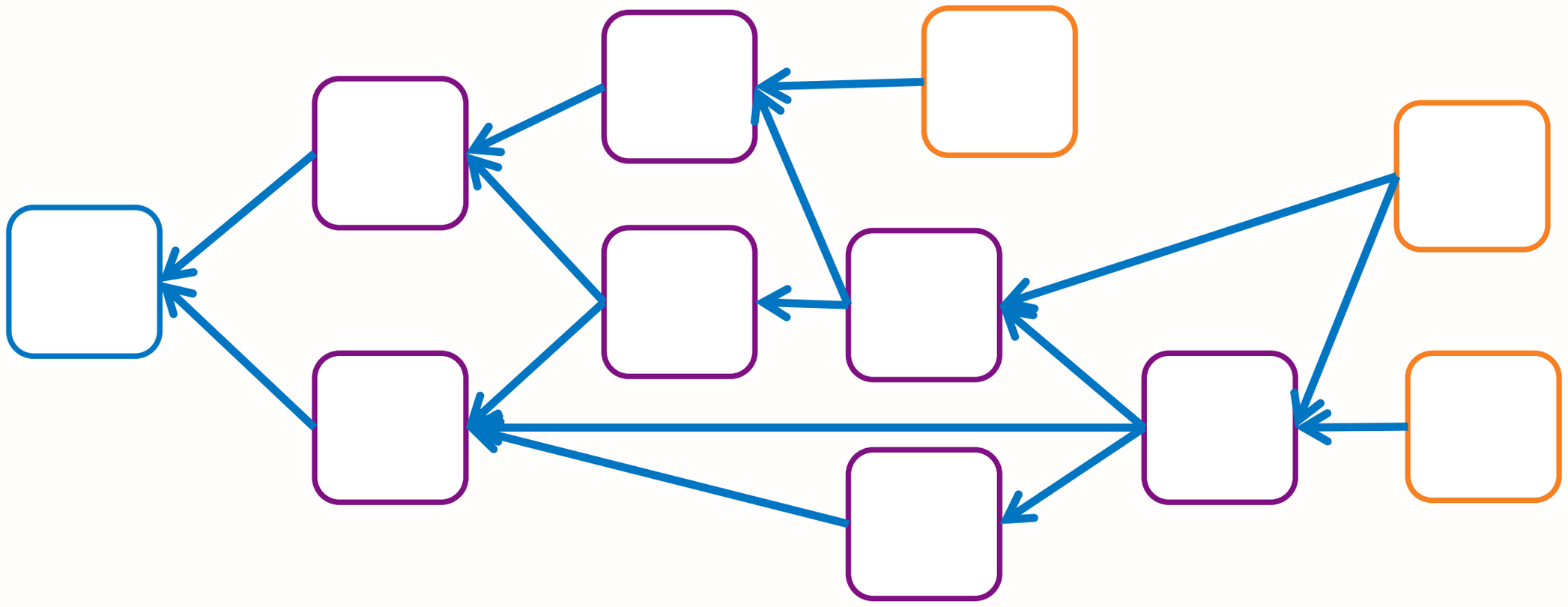
計算グラフの例



$$z = x ** 2 + 2 * x * y + y$$

Preferred Networks

誤差逆伝播は、計算グラフを逆向きにたどる



計算グラフと順伝播時の各変数の値があれば計算可能

ニューラルネットの学習方法

1. 目的関数の設計
   - 計算グラフを自分で設計する
2. 勾配の計算
   - 誤差逆伝播で機械的に計算できる
3. 最小化のための反復計算
   - 勾配を使って反復更新する

1 さえ設計すれば残りは
ほぼ自動化されている

おまけ

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```

((A (man selling)) ((to

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```

((A (man selling)) ((to

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```

(A (man selling)) ((to (

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```

A	(man selling)) ((to (a

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```

A man

selling)) ((to (a customer

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```

A man selling

)) ((to (a customer)) .

((A (man selling)) ((donuts (to (a customer))) .))

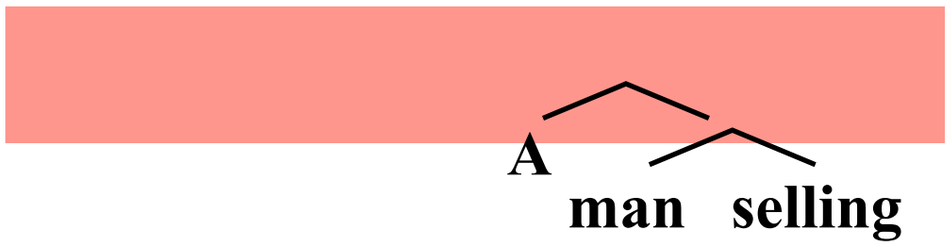
shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
     where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
           parse stack             "(" = stack
           parse stack              t  = Leaf t : stack
```

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```

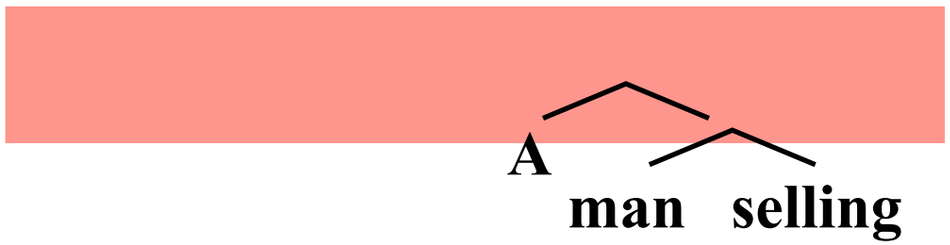


((to (a customer)) .))

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```

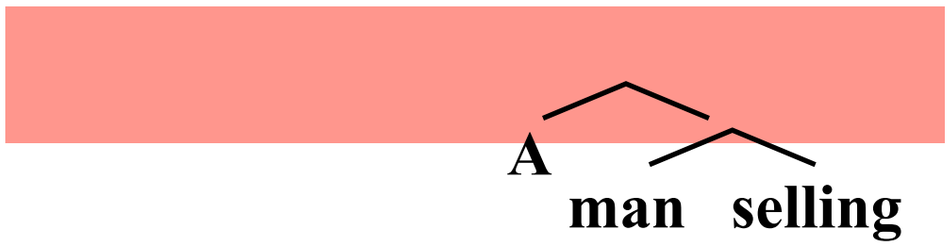


(to (a customer)) .))

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```



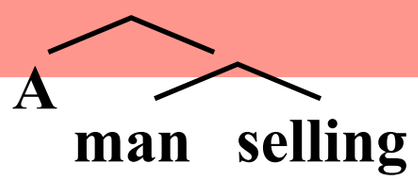
to (a customer)) .))

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```

to



(a customer)) .))

((A (man selling)) ((donuts (to (a customer))) .))

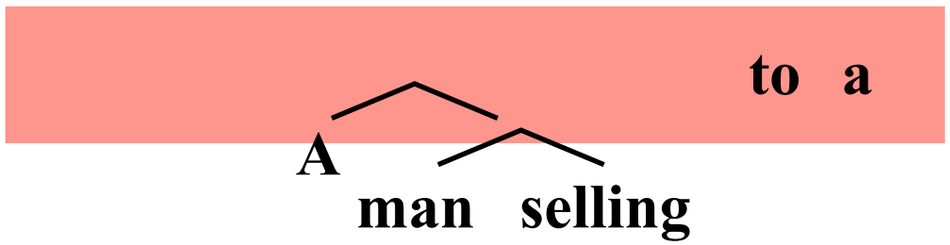
shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```



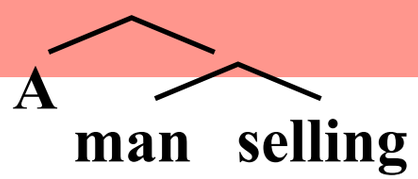
customer)) .))

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
     where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
           parse stack             "(" = stack
           parse stack              t  = Leaf t : stack
```

to a customer



)) .))

((A (man selling)) ((donuts (to (a customer))) .))

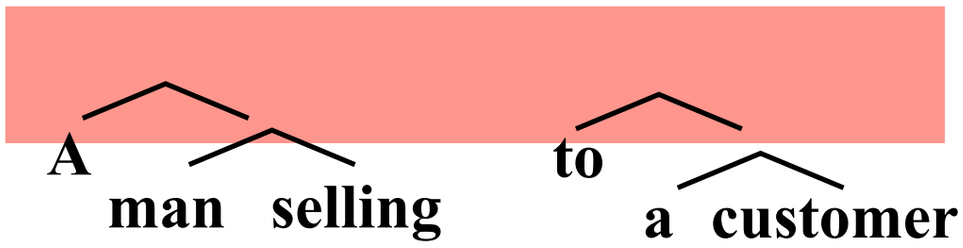
shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
     where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
           parse stack             "(" = stack
           parse stack              t  = Leaf t : stack
```

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```

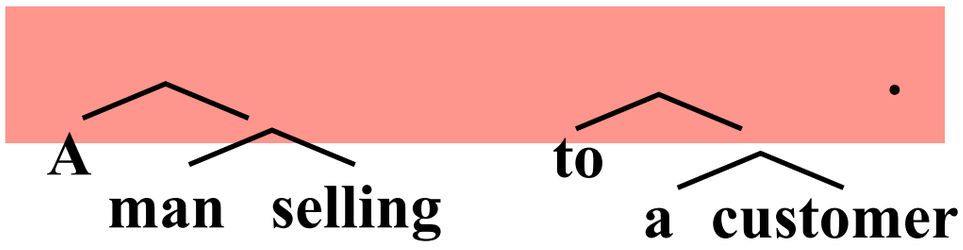


.))

((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

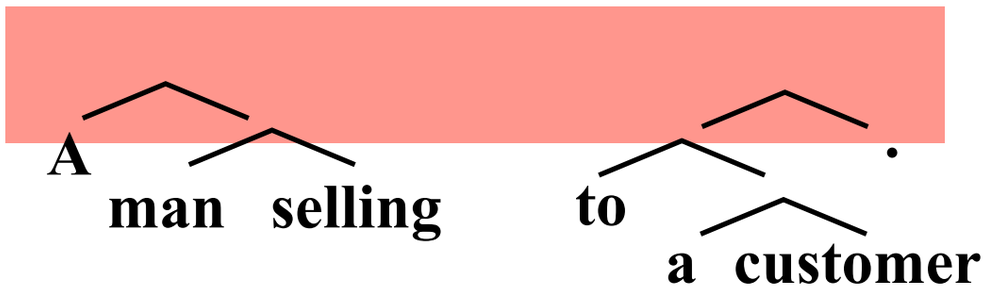
```
parser :: [Token] -> Tree
parser = head . foldl parse []
     where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
           parse stack             "(" = stack
           parse stack              t  = Leaf t : stack
```



((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

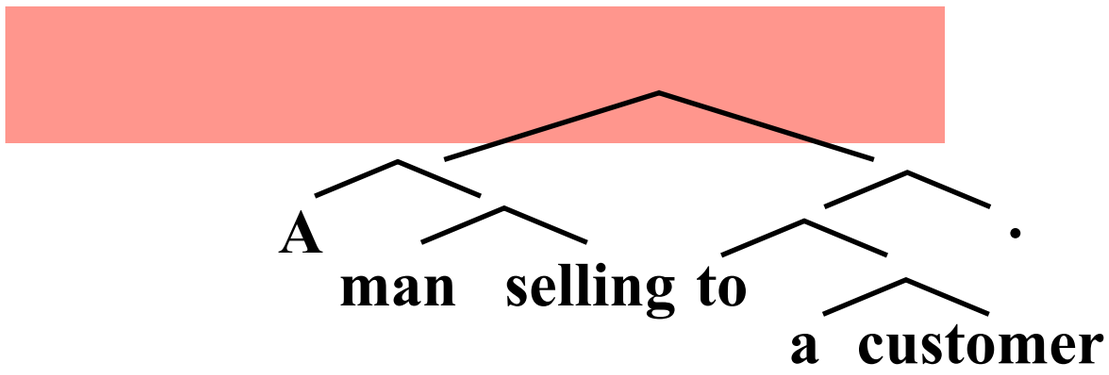
```
parser :: [Token] -> Tree
parser = head . foldl parse []
     where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
           parse stack             "(" = stack
           parse stack              t  = Leaf t : stack
```



((A (man selling)) ((donuts (to (a customer))) .))

shift-reduce法

```
parser :: [Token] -> Tree
parser = head . foldl parse []
    where parse (c2 : c1 : stack) ")" = Node c1 c2 : stack
          parse stack             "(" = stack
          parse stack              t  = Leaf t : stack
```



((A (man selling)) ((donuts (to (a customer))) .))