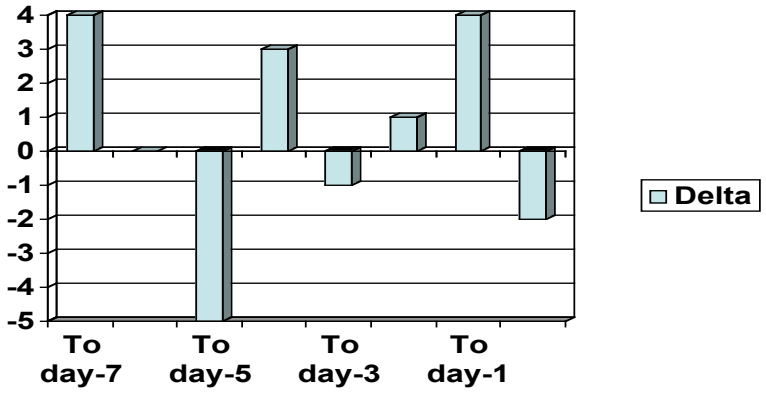
Syntax Directed Translation

Contents

- Rationale for Attribute Grammars
- Step by step example
  - Problem
  - Parsing
  - Calculation variants
  - Attribution

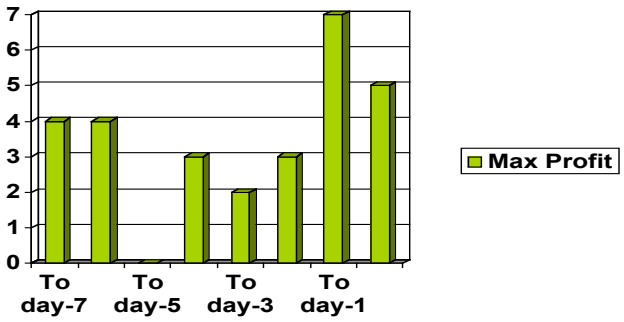
Problem

- Given: Dow Jones daily fluctuations (deltas)



Problem

- What is the maximum profit possible?



- 7 (if sold yesterday)

Example

- Input

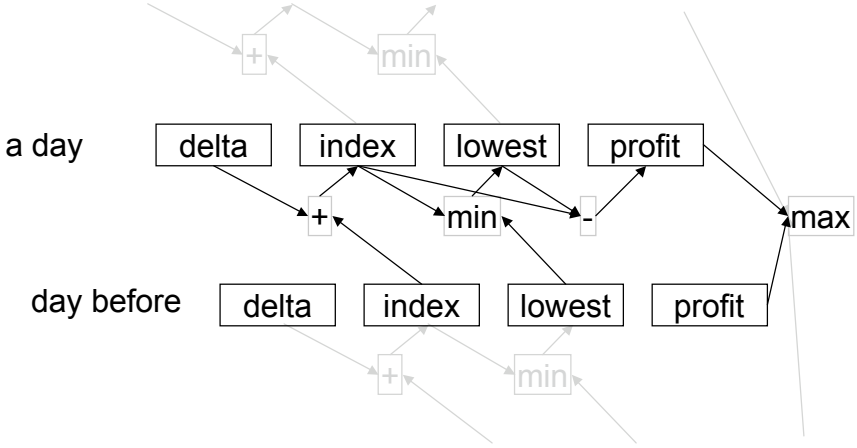
-2	4	1	-1	3	-5	0	4
today	yesterday	today-2	...			...	today-7

- Output
  - 7

Information needed

- Input: daily delta
- For each day
  - The index:
    today's delta + yesterdays index
  - The lowest index:
    today's index `min` yesterday's lowest
  - The profit:
    today's index - today's lowest
- Output: maximum of profit of all days

Information flow between aspects



List solution

- Delta's in list, intermediate results too

```
type DJSequence a = [a]

dj_bestinvest :: DJSequence Int -> Int
dj_bestinvest deltas
  = dj_maxprofit profits
    where profits = dj_profit indexes lowests
          lowests = dj_lowest indexes
          indexes = dj_index deltas

dj_ bestinvest [-2,4,1,-1,3,-5,0,4]
```

List solution

- E.g. dj_index, returns list with indexes:

```
dj_index :: DJSequence Int -> DJSequence Int
dj_index deltas
  = init (foldr (\today_delta past_indexes
                    -> today_delta + (head past_indexes) : past_indexes
                 )
                 [0]
                 deltas
         )
```

- Or (handier)

```
dj_index deltas
  = snd  (foldr (\today_delta (past_index,past_indexes)
                    -> let today_index = today_delta + past_index
                       in (today_index, today_index : past_indexes)
                 )
                 (0,[])
                 deltas
         )
```

yesterday's delta also passed around

List solution

- Other functions follow same pattern

```
dj_lowest indexes
  = snd  (foldr (\today_index (past_lowest,past_lowests)
                    -> let today_lowest = today_index `min` past_lowest
                       in (today_lowest, today_lowest : past_lowests)
                 )
                 (0,[])
                 indexes
         )
dj_profit indexes lowests
  =      (foldr (\(today_index,today_lowest) past_profits
                    -> let today_profit = today_index - today_lowest
                       in today_profit : past_profits
                 )
                 []
                 (zip indexes lowests)
         )
dj_maxprofit profits
  = foldr max 0 profits
```

Parsing

- Input is sequence of int's

```
string2int = foldl (\val dig -> (10*val + ord dig -ord '0')) 0

pPosInt = string2int <$> pList1 ('0' <..> '9')
pInt    = (negate <$ (pSym '-') `opt` id) <*> pPosInt
pSpaces = pList (pSym ' ')
pDJ     = pListSep pSpaces pInt

djparse inp = lst
              where (lst,_) = parse pDJ inp

dj' = djparse "-2 4 1 -1 3 -5 0 4"

dj_bestinvest dj'
```


Algebra solution

- Abstraction of folding on abstraction of a list

```
data DJSequence a
  = NextDay a (DJSequence a)
  | Start deriving Show

foldrDJSequence     (_      , start) Start
  = start
foldrDJSequence alg@(nextday, _    ) (NextDay i h)
  = nextday i (foldrDJSequence alg h)

dj :: DJSequence Int
dj = NextDay (-2)
      (NextDay 4
        (NextDay 1
          (NextDay (-1)
            (NextDay 3
              (NextDay (-5)
                (NextDay 0
                  (NextDay 4 Start
      ) ) ) ) ) ) )
```

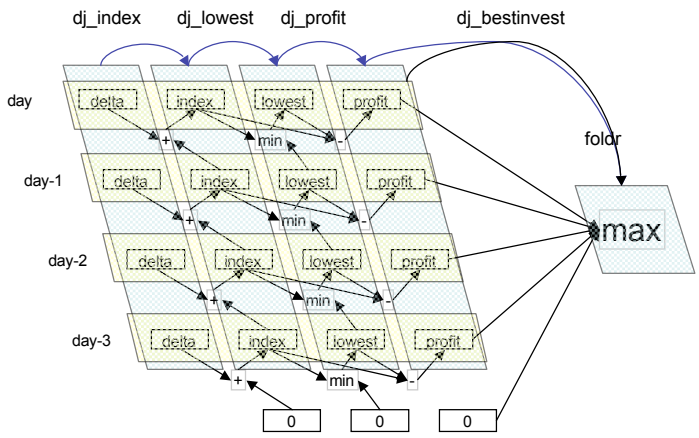
Algebra solution

- Structure differs, calculation is the same

```
dj_index deltas
  = snd  (foldrDJSequence
               ( (\today_delta (past_delta,past_indexes)
                    -> let today_index = today_delta + past_delta
                         in (today_index, NextDay today_index past_indexes)
                 )
               , (0,Start)
               )
               deltas
         )
```

- Structure
  - represents abstract syntax tree
  - as returned by parsing

Structure of calculation



Aspects

- For each aspect (index, low, profit) a separate function
- More efficient: all in one, no intermediate structures

```
dj_bestinvest' :: DJSequence Int -> Int
dj_bestinvest' deltas
  = let (maxprofit, _, _)
          = foldrDJSequence
              ( (\today_delta (past_maxprofit, past_low, past_index)
                   -> let today_index  = today_delta + past_index
                          today_low    = today_index `min` past_low
                          today_profit = today_index - today_low
                          maxprofit    = today_profit `max` past_maxprofit
                       in (maxprofit, today_low, today_index)
                )
              , (0,0,0)
              )
              deltas
        in maxprofit
```


Aspects

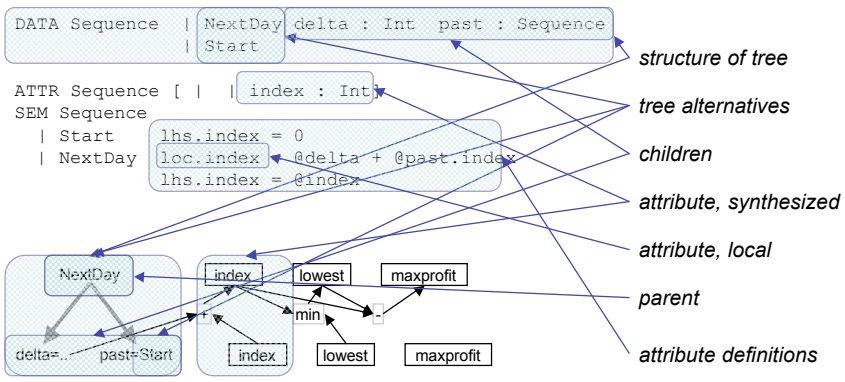
- Simultaneously calculation
  - More efficient
  - But all aspects are combined, breaks encapsulation
- Solution: Attribute Grammar
  - Describes aspects individually, but
  - AG compiler combines all aspect calculations into one program

Attribute Grammar

- Aspects: index, low, profit

```
DATA Sequence  | NextDay delta : Int  past : Sequence
               | Start

ATTR Sequence [ |  | index : Int]
SEM Sequence
  | Start    lhs.index      =  0
  | NextDay  loc.index      =  @delta + @past.index
             lhs.index      =  @index

ATTR Sequence [ |  | low : Int]
SEM Sequence
  | Start    lhs.low        =  0
  | NextDay  loc.low        =  @index `min` @past.low
             lhs.low        =  @low

ATTR Sequence [ |  | maxprofit : Int]
SEM Sequence
  | Start    lhs.maxprofit  =  0
  | NextDay  lhs.maxprofit  =  (@index - @low) `max` @past.maxprofit
```


Meaning of AG elements



AG compiler output

```
data Sequence = Sequence_NextDay (Int) (Sequence)
              | Sequence_Start
-- semantic domain
type T_Sequence = ( (Int),(Int),(Int))
-- cata
sem_Sequence :: (Sequence) ->
                (T_Sequence)
sem_Sequence ((Sequence_NextDay (_delta) (_past))) =
    (sem_Sequence_NextDay (_delta) ((sem_Sequence (_past))))
sem_Sequence ((Sequence_Start )) =
    (sem_Sequence_Start )

sem_Sequence_Start :: (T_Sequence)
sem_Sequence_Start  =
    let _lhsOindex :: (Int)
        _lhsOlow :: (Int)
        _lhsOmaxprofit :: (Int)
        (_lhsOindex@_) = 0
        (_lhsOlow@_) = 0
        (_lhsOmaxprofit@_) = 0
    in  ( _lhsOindex,_lhsOlow,_lhsOmaxprofit)
```


AG compiler output - II

```
sem_Sequence_NextDay :: (Int) ->
                        (T_Sequence) ->
                        (T_Sequence)
sem_Sequence_NextDay (delta_) (past_) =
    let _lhsOindex :: (Int)
        _lhsOlow :: (Int)
        _lhsOmaxprofit :: (Int)
        _pastIindex :: (Int)
        _pastIlow :: (Int)
        _pastImaxprofit :: (Int)
        ( _pastIindex,_pastIlow,_pastImaxprofit) =
            (past_ )
        (_lhsOindex@_) =
            _index
        (_index@_) =
            delta_ + _pastIindex
        (_lhsOlow@_) =
            _low
        (_low@_) =
            _index `min` _pastIlow
        (_lhsOmaxprofit@_) =
            (_index - _low) `max` _pastImaxprofit
    in  ( _lhsOindex,_lhsOlow,_lhsOmaxprofit)
```

Connecting AG generated functions to parser

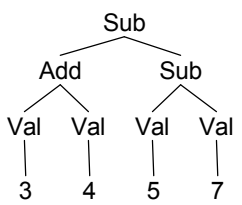
- By applying the generated functions

```
string2int = foldl (\val dig -> (10*val + ord dig -ord '0')) 0

pPosInt = string2int <$> pList1 ('0' <..> '9')
pInt    = (negate <$ (pSym '-') `opt` id) <*> pPosInt
pSpaces = pList (pSym ' ')
pDJ     = pFoldrSep (sem_Sequence_NextDay,sem_Sequence_Start) pSpaces pInt

dj_bestinvest' = maxprofit
                 where ((_,_,maxprofit),_) = parse pDJ "-2 4 1 -1 3 -5 0 4"
```


Example: expression

- Input: (3 +4)- (5 -7)
- Value output: 9
- String output: ((3+4)-(5-7))
- Intermediate structure:
- Grammar:

E: F | F '+' E | F '-' E
F: Int | '(' E ')'



Algebra solution

- Intermediate structure

```
data Expr = Add Expr Expr
          | Sub Expr Expr
          | Val Int

foldExpr alg@(add,sub,val) e
  = case e of
      (Add e1 e2) -> add (foldExpr alg e1) (foldExpr alg e2)
      (Sub e1 e2) -> sub (foldExpr alg e1) (foldExpr alg e2)
      (Val i    ) -> val i
```


Algebra solution

- Parser

E: F | F '+' E | F '-' E
F: Int | '(' E ')'

```
pSpaces   = pList (pSym ' ')
pSpaced p = p <* pSpaces
pInt      = pSpaced ((Val . string2int)
                     <$> pList1 ('0' <..> '9'))
pTok c    = pSpaced (pSym c)
pOper     = (   Add <$ pTok '+'
            <|> Sub <$ pTok '-'
            )
pExpr     = pSpaces *> pChainl pOper pFactor
pFactor   = (   pInt
            <|> pTok '(' *> pExpr <* pTok ')'
            )
```

Algebra solution

- Execution

```
showOp op e1 e2 = "(" ++ e1 ++ op ++ e2 ++ ")"

t filename
    = do input <- readFile filename
         let (result,errors) = parse pExpr input
         let value  = (foldExpr ((+),(-),id) result)
         let string = (foldExpr (showOp "+",showOp "-",show) result)
         putStr ("Input : " ++ input ++ "\n")
         putStr (if null errors then "" else "Errors:\n" ++ errors)
         putStr ("String: " ++ string ++ "\n")
         putStr ("Value : " ++ (show value) ++ "\n")
```

- Output

```
Input : (3 +4 )-  (5  -7)
String: ((3+4)-(5-7))
Value : 9
```


AG solution

- Structure + attribution

```
DATA Expr | Add e1,e2: Expr
          | Sub e1,e2: Expr
          | Val int: Int

SEM Expr [ | | value: Int ]
  | Add lhs.value  = @e1.value + @e2.value
  | Sub lhs.value  = @e1.value - @e2.value
  | Val lhs.value  = @int

SEM Expr [ | | string: String ]
  | Add lhs.string = "(" ++ @e1.string ++ "+" ++ @e2.string ++ ")"
  | Sub lhs.string = "(" ++ @e1.string ++ "-" ++ @e2.string ++ ")"
  | Val lhs.string = show @int
```

AG solution

- Parsing

```
pSpaces   = pList (pSym ' ')
pSpaced p = p <* pSpaces
pInt      = pSpaced ((sem_Expr_Val . string2int)
                     <$> pList1 ('0' <..> '9'))
pTok c    = pSpaced (pSym c)
pOper     = (   sem_Expr_Add <$ pTok '+'
            <|> sem_Expr_Sub <$ pTok '-'
            )
pExpr     = pSpaces *> pChainl pOper pFactor
pFactor   = (   pInt
            <|> pTok '(' *> pExpr <* pTok ')'
            )
```

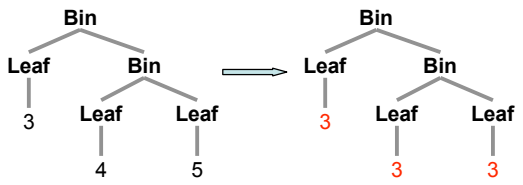

AG solution

- Execution

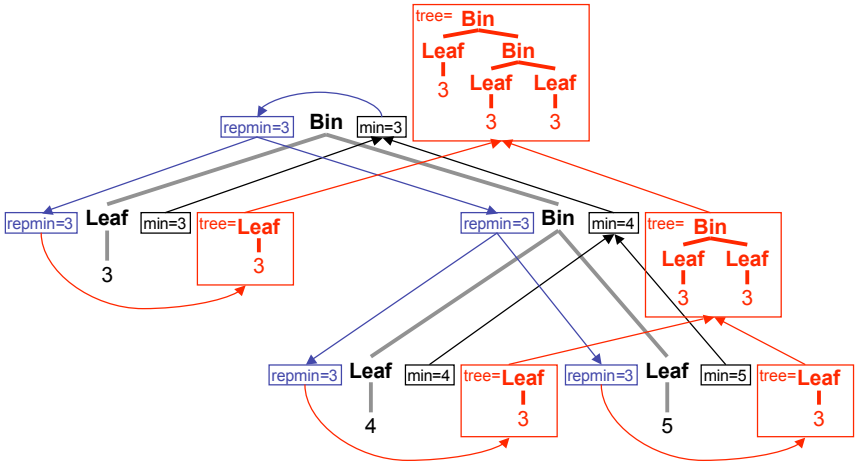
```
t filename
    = do input <- readFile filename
         let ((value,string),errors) = parse pExpr input
         putStr ("Input : " ++ input ++ "\n")
         putStr (if null errors then "" else "Errors:\n" ++ errors)
         putStr ("String: " ++ string ++ "\n")
         putStr ("Value : " ++ (show value) ++ "\n")
```

Example: Repmin

- *Rep*lace values in tree by *min*imal value of all values of the tree



Repmin calculation



Repmin code

```
DATA Root
   | Root  Tree

DATA Tree
   | Bin   lt,rt: Tree
   | Leaf  Int

SEM Tree [ | | min: Int ]
  | Bin   lhs.min = @lt.min `min` @rt.min
  | Leaf  lhs.min = @int

ATTR Root Tree [repmin: Int | |]

SEM Tree [| | tree: Tree ]
  | Bin   lhs.tree = Tree_Bin @lt.tree @rt.tree
  | Leaf  lhs.tree = Tree_Leaf @lhs.repmin

SEM Root
  | Root  tree.repmin = @tree.min
          lhs.tree    = @tree.tree

tr = Tree_Bin (Tree_Leaf 3) (Tree_Bin (Tree_Leaf 4) (Tree_Leaf 5))
tr' = sem_Root (Root_Root tr)
```