Compiler Construction

Week 3: Parsing II

Sjaak Smetsers Mart Lubbers Jordy Aaldering

2024/2025 KW3

Radboud University

Recap

Advanced parser combinators

Beyond top-down parsing

Shift-reduce parsers

Conclusion

Recap

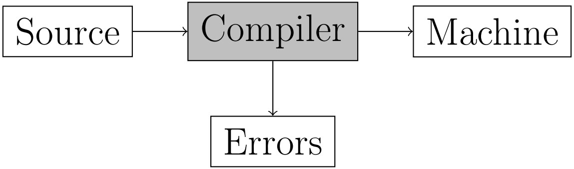
What was a compiler again?

Abstract view on a compiler

Compiler

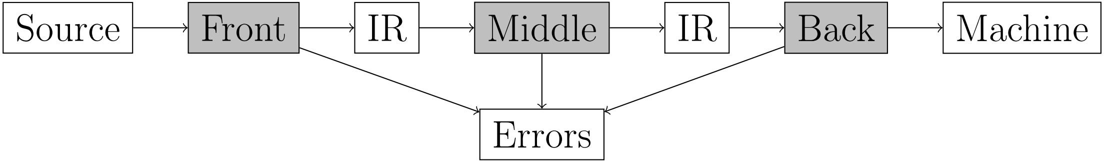
What was a compiler again?

Abstract view on a compiler

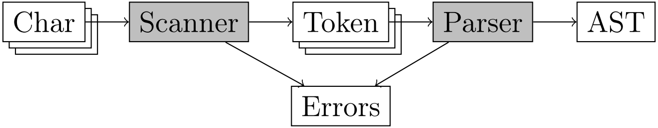


What was a compiler again?

Three pass compiler



Frontend



Advanced parser combinators

Chain rule

Remember our transformed grammar

Chain rule

Remember our transformed grammar

$$
\begin{array}{lcllll}
expr & ::= & fact & \{[+-] & fact & \}^* \\
fact & ::= & pow & \{[*/] & pow & \}^* \\
pow & ::= & basic & \hat{} & pow & \\
 & | & basic & & & \\
basic & ::= & '('expr\ ')' & & & \\
 & | & int & & & \\
 & | & id & & &
\end{array}
$$

Advanced combinators

Translate to parser combinators

```
data Expr
    = BinOp Char Expr Expr
    | Val   Int
    | Id    String
```

Advanced combinators

Translate to parser combinators

```
data Expr
    = BinOp Char Expr Expr
    | Val   Int
    | Id    String

pExpr  =  pChainl (op '+' <|> op '-') pFact
pFact  =  pChainl (op '*' <|> op '/') pPow
```

Advanced combinators

Translate to parser combinators

```
data Expr
    = BinOp Char Expr Expr
    | Val   Int
    | Id    String

pExpr  =  pChainl (op '+' <|> op '-') pFact
pFact  =  pChainl (op '*' <|> op '/') pPow
pPow   =  BinOp <$> pBasic <*> symbol '^' <*> pPow
      <|> pBasic
```

Advanced combinators

Translate to parser combinators

```
data Expr
    = BinOp Char Expr Expr
    | Val   Int
    | Id    String

pExpr  =  pChainl (op '+' <|> op '-') pFact
pFact  =  pChainl (op '*' <|> op '/') pPow
pPow   =  BinOp <$> pBasic <*> symbol '^' <*> pPow
      <|> pBasic
pBasic =  parens pExpr
      <|> Val . read <$> some pDigit
      <|> Id <$> some pAlpha
```

Chain left combinator

```
op :: Char → Parser Char (Expr → Expr → Expr)
op c = BinOp <$> symbol c
```

Chain left combinator

```
pChainl :: Parser s (a → a → a) → Parser s a → Parser s a
pChainl op p = foldl (&) <$> p <*> many (flip <$> op <*> p)
```

Chain left combinator

$$upper \quad ::= \; lower\ \{op\ \ lower\ \}^*$$

pChainl op p

Chain left combinator

$$upper ::= lower \; \{op \;\; lower\}^*$$

$$\text{pChainl op p} \;=\; \text{p} <^*> \text{many (op} <^*> \text{p)}$$

Chain left combinator

$$upper ::= lower \; \{op \;\; lower\}^*$$

$$\text{pChainl op p} \;=\; \text{p} <^*> \text{many (op} <^*> \text{p)}$$

[a]

Chain left combinator

$$upper \quad ::= \; lower \; \{op \;\; lower\}^*$$

$$\text{pChainl op p} \quad = \quad \text{p} <\!*\!> \text{many (flip} <\!\$\!> \text{op} <\!*\!> \text{p)}$$

Apply the operator to the list: `[a->a]`

Chain left combinator

$$upper \quad ::= \; lower \; \{op \;\; lower\}^*$$

$$\text{pChainl op p} \quad = \quad \text{foldl } (\&) <\$> \;\; \text{p} <^*> \text{many (flip} <\$> \text{op} <^*> \text{p)}$$

Glue the list together, `(&) = flip ($)`

Chaining chains

Chaining chains

```
pChainr :: Parser s (a → a → a) → Parser s a → Parser s a
pChainr op p = (&) <$> p <*> (flip <$> op <*> pChainr op p) <|> p
```

Chaining chains

```
pChainr :: Parser s (a → a → a) → Parser s a → Parser s a
pChainr op p = (&) <$> p <*> (flip <$> op <*> pChainr op p) <|> p

pExpr :: Parser Char Expr
pExpr = flip (foldr ($))
    [ pChainl (op '+' <|> op '-')
    , pChainl (op '*' <|> op '/')
    , pChainr (op '^')
    ] $    parens pExpr
       <|> Val . read <$> some pDigit
       <|> Id <$> some pAlpha
```

Parsec (simplified)

```
import Text.Parsec
import Text.Parsec.Expr
import Text.Parsec.String
```

Parsec (simplified)

```haskell
import Text.Parsec
import Text.Parsec.Expr
import Text.Parsec.String
```

```haskell
buildExpressionParser
  :: OperatorTable a
  → Parser a
  → Parser a

type OperatorTable a = [[Operator a]]

data Operator a
  = Infix (Parser (a → a → a))
      Assoc
  | Prefix  (Parser (a → a))
  | Postfix (Parser (a → a))

data Assoc
  = AssocNone
  | AssocLeft
  | AssocRight
```

Parsec (simplified)

```
import Text.Parsec
import Text.Parsec.Expr
import Text.Parsec.String

expr :: Parser Expr
expr = buildExpressionParser
    [
        ...
    ] basic
```

```
buildExpressionParser
  :: OperatorTable a
  → Parser a
  → Parser a

type OperatorTable a = [[Operator a]]

data Operator a
  = Infix (Parser (a → a → a))
      Assoc
  | Prefix  (Parser (a → a))
  | Postfix (Parser (a → a))

data Assoc
  = AssocNone
  | AssocLeft
  | AssocRight
```

Parsec (simplified)

```
import Text.Parsec
import Text.Parsec.Expr
import Text.Parsec.String

expr :: Parser Expr
expr = buildExpressionParser
    [
        ...
    ] basic

basic :: Parser Expr
basic = char '(' *> expr <* char ')'
    <|> Val . read <$> many1 digit
    <|> Id <$> many1 letter
```

```
buildExpressionParser
  :: OperatorTable a
  → Parser a
  → Parser a

type OperatorTable a = [[Operator a]]

data Operator a
  = Infix (Parser (a → a → a))
        Assoc
  | Prefix  (Parser (a → a))
  | Postfix (Parser (a → a))

data Assoc
  = AssocNone
  | AssocLeft
  | AssocRight
```

Parsec (simplified)

```
import Text.Parsec
import Text.Parsec.Expr
import Text.Parsec.String

expr :: Parser Expr
expr = buildExpressionParser
    [ [ Infix (op '+') AssocLeft
      , Infix (op '-') AssocLeft ]
    , [ Infix (op '*') AssocLeft
      , Infix (op '/') AssocLeft ]
    , [ Infix (op '^') AssocRight]
    ] basic
```

```
buildExpressionParser
  :: OperatorTable a
  → Parser a
  → Parser a

type OperatorTable a = [[Operator a]]

data Operator a
  = Infix (Parser (a → a → a))
      Assoc
  | Prefix  (Parser (a → a))
  | Postfix (Parser (a → a))

data Assoc
  = AssocNone
  | AssocLeft
  | AssocRight
```

Beyond top-down parsing

Error handling

$> 90\%$ of the programs are wrong

Error handling

$>$ 90% of the programs are wrong

What happens when the parse fails

Error handling

$> 90\%$ of the programs are wrong

What happens when the parse fails

`[]`

Error handling

- Change type to:

```
newtype Parser e s a = Parser {runParser :: [s] → ([(a, [s])], [e])}
```

Swierstra, S. Doaitse. "Combinator parsers: From toys to tools". Electronic Notes in Theoretical Computer Science 41.1 (2001): 38–59.

Error handling

- Change type to:

```
newtype Parser e s a = Parser {runParser :: [s] → ([(a, [s])], [e])}
```

- Error recovery

Swierstra, S. Doaitse. "Combinator parsers: From toys to tools". Electronic Notes in Theoretical Computer Science 41.1 (2001): 38–59.

Error handling

- Change type to:

```
newtype Parser e s a = Parser {runParser :: [s] → ([(a, [s])], [e])}
```

- Error recovery
- Skip tokens until we can parse again

Swierstra, S. Doaitse. "Combinator parsers: From toys to tools". Electronic Notes in Theoretical Computer Science 41.1 (2001): 38–59.

Error handling

- Change type to:

```
newtype Parser e s a = Parser {runParser :: [s] → ([(a, [s])], [e])}
```

- Error recovery
- Skip tokens until we can parse again
- Continuation based parsers

Swierstra, S. Doaitse. "Combinator parsers: From toys to tools". Electronic Notes in Theoretical Computer Science 41.1 (2001): 38–59.

Error handling

- Change type to:

```
newtype Parser e s a = Parser {runParser :: [s] → ([(a, [s])], [e])}
```

- Error recovery
- Skip tokens until we can parse again
- Continuation based parsers
  - Collect info on different branches

Swierstra, S. Doaitse. "Combinator parsers: From toys to tools". Electronic Notes in Theoretical Computer Science 41.1 (2001): 38–59.

Error handling

- Change type to:

```
newtype Parser e s a = Parser {runParser :: [s] → ([(a, [s])], [e])}
```

- Error recovery
- Skip tokens until we can parse again
- Continuation based parsers
  - Collect info on different branches
  - Postpone when there is no preference

Swierstra, S. Doaitse. "Combinator parsers: From toys to tools". Electronic Notes in Theoretical Computer Science 41.1 (2001): 38–59.

Error handling

- Change type to:

```
newtype Parser e s a = Parser {runParser :: [s] → ([(a, [s])], [e])}
```

- Error recovery
- Skip tokens until we can parse again
- Continuation based parsers
  - Collect info on different branches
  - Postpone when there is no preference
  - Can be extended with recovery

Swierstra, S. Doaitse. "Combinator parsers: From toys to tools". Electronic Notes in Theoretical Computer Science 41.1 (2001): 38–59.

Scannerless parsing

Pros

- One less phase

Cons

Visser, Eelco. Scannerless generalized-LR parsing. Universiteit van Amsterdam. Programming Research Group, 1997.

Scannerless parsing

Pros

- One less phase
- One grammar

Cons

Visser, Eelco. Scannerless generalized-LR parsing. Universiteit van Amsterdam. Programming Research Group, 1997.

Scannerless parsing

Pros

- One less phase
- One grammar
- Context free tokens possible

Cons

Visser, Eelco. Scannerless generalized-LR parsing. Universiteit van Amsterdam. Programming Research Group, 1997.

Scannerless parsing

Pros

- One less phase
- One grammar
- Context free tokens possible
- Layout conservation

Cons

Visser, Eelco. Scannerless generalized-LR parsing. Universiteit van Amsterdam. Programming Research Group, 1997.

Scannerless parsing

Pros

- One less phase
- One grammar
- Context free tokens possible
- Layout conservation
- Compositionality

Cons

Visser, Eelco. Scannerless generalized-LR parsing. Universiteit van Amsterdam. Programming Research Group, 1997.

Scannerless parsing

Pros

- One less phase
- One grammar
- Context free tokens possible
- Layout conservation
- Compositionality
- Library support (`Text.Parsec.Token`)*

Cons

Visser, Eelco. Scannerless generalized-LR parsing. Universiteit van Amsterdam. Programming Research Group, 1997.

Scannerless parsing

Pros

- One less phase
- One grammar
- Context free tokens possible
- Layout conservation
- Compositionality
- Library support (`Text.Parsec.Token`)*

Cons

- Complexity

Visser, Eelco. Scannerless generalized-LR parsing. Universiteit van Amsterdam. Programming Research Group, 1997.

Scannerless parsing

Pros

- One less phase
- One grammar
- Context free tokens possible
- Layout conservation
- Compositionality
- Library support (`Text.Parsec.Token`)*

Cons

- Complexity
- Efficiency

Visser, Eelco. Scannerless generalized-LR parsing. Universiteit van Amsterdam. Programming Research Group, 1997.

Scannerless parsing

Pros

- One less phase
- One grammar
- Context free tokens possible
- Layout conservation
- Compositionality
- Library support (`Text.Parsec.Token`)*

Cons

- Complexity
- Efficiency
- Separation of concerns

Visser, Eelco. Scannerless generalized-LR parsing. Universiteit van Amsterdam. Programming Research Group, 1997.

Shift-reduce parsers

Parsing in practice

Top-down parsers

- Start at root
- Backtracking may be required
- Slow (er)

Parsing in practice

Top-down parsers

- Start at root
- Backtracking may be required
- Slow (er)

Bottom-up parsers

- Fast
- Start at the leafs
- Lookahead
- Delay the decision
- Store fragments

Parser generators

- Almost always $LR(1)$

Parser generators

- Almost always $LR(1)$
- Suitable for almost any language

Parser generators

- Almost always $LR(1)$
- Suitable for almost any language
- Automatically generate tables

Parser generators

- Almost always $LR(1)$
- Suitable for almost any language
- Automatically generate tables
- Efficient parsers

Parser generators

- Almost always $LR(1)$
- Suitable for almost any language
- Automatically generate tables
- Efficient parsers
- Errors are recognized quickly

Parser generators

- Almost always $LR(1)$
- Suitable for almost any language
- Automatically generate tables
- Efficient parsers
- Errors are recognized quickly
- Generators:

Parser generators

- Almost always $LR(1)$
- Suitable for almost any language
- Automatically generate tables
- Efficient parsers
- Errors are recognized quickly
- Generators:
  - yacc: yet another compiler compiler (1970)

Parser generators

- Almost always $LR(1)$
- Suitable for almost any language
- Automatically generate tables
- Efficient parsers
- Errors are recognized quickly
- Generators:
  - yacc: yet another compiler compiler (1970)
  - bison: GNU version of yacc with extensions

Parser generators

- Almost always $LR(1)$
- Suitable for almost any language
- Automatically generate tables
- Efficient parsers
- Errors are recognized quickly
- Generators:
  - yacc: yet another compiler compiler (1970)
  - bison: GNU version of yacc with extensions
  - antlr, menhir, lark, ply, lrparsing, plyplus, pyleri, ...

Parser generators

- Almost always $LR(1)$
- Suitable for almost any language
- Automatically generate tables
- Efficient parsers
- Errors are recognized quickly
- Generators:
  - yacc: yet another compiler compiler (1970)
  - bison: GNU version of yacc with extensions
  - antlr, menhir, lark, ply, lrparsing, plyplus, pyleri, ...
  - etc.

LL(k) versus LR(k)

LL(k)

Left-to-right-parsing Leftmost-derivation

LL(k) versus LR(k)

LL(k)

Left-to-right-parsing Leftmost-derivation

LR(k)

Left-to-right-parsing Rightmost-derivation

LL(k) versus LR(k)

LL(k)

Left-to-right-parsing Leftmost-derivation

LR(k)

Left-to-right-parsing Rightmost-derivation

- Consume tokens
- Transform to rule when possible
- Stack tokens
- Suits most languages
- Right-associativity

What's powering this

Shift-reduce parser

What's powering this

Shift-reduce parser

```haskell
type Input   = [Item]
type Grammar = [Item]
type Rule    = [Item]

data Item
    = T  String         -- terminal (Operator)
    | Id String         -- terminal (Identifier)
    | End               -- End of input
    | NT String Rule    -- Non-terminal
```

What's powering this

Shift-reduce parser

```haskell
type Input   = [Item]
type Grammar = [Item]
type Rule    = [Item]

data Item
    = T  String         -- terminal (Operator)
    | Id String         -- terminal (Identifier)
    | End               -- End of input
    | NT String Rule    -- Non-terminal

exprInput = [Id "x", T "+", Id "y", T "+", Id "z", End]
```

What's powering this

Encoding a grammar

$$
\begin{aligned}
S \quad & ::= expr \\
expr \quad & ::= expr \quad + basic \\
& \quad\ \mid \ basic \\
basic \quad & ::= id
\end{aligned}
$$

What's powering this

Encoding a grammar

$$
\begin{aligned}
S \quad &::= expr \\
expr \quad &::= expr \ + basic \\
&\ \ \mid \ basic \\
basic \quad &::= id
\end{aligned}
$$

```
exprGrammar :: Grammar
exprGrammar =
    [ NT "S"      [NT "expr" [], End]
    , NT "expr"   [NT "expr" [], T "+", NT "basic" []]
    , NT "expr"   [NT "basic" []]
    , NT "basic"  [Id ""]
    ]
```

What's powering this

Shift-reduce

- State

What's powering this

Shift-reduce

- State
- Transition

What's powering this

Shift-reduce

- State
- Transition
- Stack

What's powering this

Shift-reduce

- State
- Transition
- Stack

What's powering this

Shift-reduce

- State
- Transition
- Stack

```
type LRStack = [(Item, State)]
type State   = Int
```

What's powering this

Shift-reduce

- State
- Transition
- Stack

```
type LRStack = [(Item, State)]
type State   = Int
```

Transitions

```
data Action
    = Shift State
    | GoTo State
    | Reduce String Int
    | Accept
    | Error
```

What's powering this

Example table

$$
\begin{aligned}
S &::= expr \\
expr &::= expr \quad + basic \\
&\quad \mid \quad basic \\
basic &::= id
\end{aligned}
$$

What's powering this

Example table

$$
\begin{aligned}
S &::= expr \\
expr &::= expr \quad + basic \\
&\quad \mid \quad basic \\
basic &::= id
\end{aligned}
$$

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

What's powering this

Example table

$$
\begin{aligned}
S &::= expr \\
expr &::= expr \quad + basic \\
&\quad\ \mid \quad basic \\
basic &::= id
\end{aligned}
$$

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

Start at 1

What's powering this

Example table

$$
\begin{aligned}
S &::= expr \\
expr &::= expr \; + basic \\
&\;\;\mid \;\; basic \\
basic &::= id
\end{aligned}
$$

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

Start at 1

Shift (pop t) goto 5

What's powering this

Example table

$$
\begin{aligned}
S &::= expr \\
expr &::= expr \quad + basic \\
&\quad \mid \quad basic \\
basic &::= id
\end{aligned}
$$

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

Start at 1

Shift (pop t) goto 5

Accept

What's powering this

Example table

$$
\begin{aligned}
S &::= expr \\
expr &::= expr \quad + basic \\
&\quad \mid \quad basic \\
basic &::= id
\end{aligned}
$$

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

Start at 1

Shift (pop t) goto 5

Accept

Reduce to expr pop 1

Example table

$$
\begin{aligned}
S &::= expr \\
expr &::= expr \ + basic \\
&\quad\mid \ basic \\
basic &::= id
\end{aligned}
$$

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

Start at 1

Shift (pop t) goto 5

Accept

Reduce to expr pop 1

Goto (pop nt) goto 3

What's powering this

Example table

$$
\begin{aligned}
S &::= \mathit{expr} \\
\mathit{expr} &::= \mathit{expr} \ + \mathit{basic} \\
&\ \ \mid \ \ \mathit{basic} \\
\mathit{basic} &::= \mathit{id}
\end{aligned}
$$

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

Start at 1

Shift (pop t) goto 5

Accept

Reduce to expr pop 1

Goto (pop nt) goto 3

Empty: Error

What's powering this

Encoding the table

```
type SRTable = State → Item → Action
```

What's powering this

Encoding the table

```
type SRTable = State → Item → Action

exprTable :: State → Item → Action
exprTable 1 (Id x)          = Shift 5
exprTable 1 (NT "expr" _)   = GoTo 2
exprTable 1 (NT "basic" _)  = GoTo 3
exprTable 2 End             = Accept
exprTable 3 (T "+")         = Shift 4
exprTable 3 End             = Reduce "expr" 1
exprTable 4 (Id x)          = Shift 5
exprTable 4 (NT "expr" _)   = GoTo 6
exprTable 4 (NT "basic" _)  = GoTo 3
exprTable 5 (T "+")         = Reduce "basic" 1
exprTable 5 End             = Reduce "basic" 1
exprTable 6 End             = Reduce "expr" 3
exprTable _ _               = Error
```

What's powering this

Execute one step

```
lrstep :: SRTable → LRStack → Input → (LRStack, Input, Action)
```

What's powering this

Execute one step

```
lrstep :: SRTable → LRStack → Input → (LRStack, Input, Action)
lrstep table stack@((_, state):_) input@(sym:rest) = case table state sym
    of
  action@(Shift n)
```

What's powering this

Execute one step

```
lrstep :: SRTable → LRStack → Input → (LRStack, Input, Action)
lrstep table stack@((_, state):_) input@(sym:rest) = case table state sym
    of
  action@(Shift n)      → (((sym, n):stack), rest, action)
```

What's powering this

Execute one step

```
lrstep :: SRTable → LRStack → Input → (LRStack, Input, Action)
lrstep table stack@((_, state):_) input@(sym:rest) = case table state sym
    of
  action@(Shift n)     → (((sym, n):stack), rest, action)
  action@(GoTo n)
```

What's powering this

Execute one step

```
lrstep :: SRTable → LRStack → Input → (LRStack, Input, Action)
lrstep table stack@((_, state):_) input@(sym:rest) = case table state sym
    of
  action@(Shift n)      → (((sym, n):stack), rest, action)
  action@(GoTo n)       → (((sym, n):stack), rest, action)
```

What's powering this

Execute one step

```
lrstep :: SRTable → LRStack → Input → (LRStack, Input, Action)
lrstep table stack@((_, state):_) input@(sym:rest) = case table state sym
    of
  action@(Shift n)      → (((sym, n):stack), rest, action)
  action@(GoTo n)       → (((sym, n):stack), rest, action)
  action@(Accept)
```

What's powering this

Execute one step

```
lrstep :: SRTable → LRStack → Input → (LRStack, Input, Action)
lrstep table stack@((_, state):_) input@(sym:rest) = case table state sym
    of
  action@(Shift n)      → (((sym, n):stack), rest, action)
  action@(GoTo n)       → (((sym, n):stack), rest, action)
  action@(Accept)       → (stack, input, action)
```

What's powering this

Execute one step

```
lrstep :: SRTable → LRStack → Input → (LRStack, Input, Action)
lrstep table stack@((_, state):_) input@(sym:rest) = case table state sym
    of
  action@(Shift n)      → (((sym, n):stack), rest, action)
  action@(GoTo n)       → (((sym, n):stack), rest, action)
  action@(Accept)       → (stack, input, action)
  action@(Error)
```

What's powering this

Execute one step

```
lrstep :: SRTable → LRStack → Input → (LRStack, Input, Action)
lrstep table stack@((_, state):_) input@(sym:rest) = case table state sym
    of
  action@(Shift n)      → (((sym, n):stack), rest, action)
  action@(GoTo n)       → (((sym, n):stack), rest, action)
  action@(Accept)       → (stack, input, action)
  action@(Error)        → (stack, input, action)
```

What's powering this

Execute one step

```
lrstep :: SRTable → LRStack → Input → (LRStack, Input, Action)
lrstep table stack@((_, state):_) input@(sym:rest) = case table state sym
    of
  action@(Shift n)      → (((sym, n):stack), rest, action)
  action@(GoTo n)       → (((sym, n):stack), rest, action)
  action@(Accept)       → (stack, input, action)
  action@(Error)        → (stack, input, action)
  action@(Reduce nt n)
```

What's powering this

Execute one step

```
lrstep :: SRTable → LRStack → Input → (LRStack, Input, Action)
lrstep table stack@((_, state):_) input@(sym:rest) = case table state sym
    of
  action@(Shift n)      → (((sym, n):stack), rest, action)
  action@(GoTo n)       → (((sym, n):stack), rest, action)
  action@(Accept)       → (stack, input, action)
  action@(Error)        → (stack, input, action)
  action@(Reduce nt n)  → (drop n stack, item:input, action)
    where item = NT nt (reverse (map fst (take n stack)))
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                  input
-----                                  -----
[(End,1)]                  [(Id "x"), End]
```

What's powering this

	id	+	End	expr	basic
1	**S5**			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                         input
-----                                         -----
[(End,1)]                          [(Id "x"), End]
  action: S 5
```

What's powering this

	id	+	End	expr	basic
1	**S5**			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                  input
-----                                                  -----
[(End,1)]                                      [(Id "x"), End]
  action: S 5
[(End,1),(Id "x", 5)]                                    [End]
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	**Rb1**		
6			Re3		

```
stack                                                   input
-----                                                   -----
[(End,1)]                                   [(Id "x"), End]
  action: S 5
[(End,1),(Id "x", 5)]                                 [End]
  action: R basic 1
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	**Rb1**		
6			Re3		

```
stack                                                    input
-----                                                    -----
[(End,1)]                                       [(Id "x"), End]
  action: S 5
[(End,1),(Id "x", 5)]                                     [End]
  action: R basic 1
[(End,1)]                            [NT "basic" [Id "x"],End]
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	**G3**
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                input
-----                                                -----
[(End,1)]                                   [(Id "x"), End]
  action: S 5
[(End,1),(Id "x", 5)]                                 [End]
  action: R basic 1
[(End,1)]                         [NT "basic" [Id "x"],End]
  action: G 3
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	**G3**
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                     input
-----                                                     -----
[(End,1)]                                        [(Id "x"), End]
  action: S 5
[(End,1),(Id "x", 5)]                                      [End]
  action: R basic 1
[(End,1)]                             [NT "basic" [Id "x"],End]
  action: G 3
[(End,1),(NT "basic" [Id "x"], 3)]                         [End]
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	**Re1**		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                    input
-----                                                    -----
[(End,1)]                                       [(Id "x"), End]
  action: S 5
[(End,1),(Id "x", 5)]                                     [End]
  action: R basic 1
[(End,1)]                            [NT "basic" [Id "x"],End]
  action: G 3
[(End,1),(NT "basic" [Id "x"], 3)]                        [End]
  action: R expr 1
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	**Re1**		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                  input
-----                                                  -----
[(End,1)]                                       [(Id "x"), End]
  action: S 5
[(End,1),(Id "x", 5)]                                     [End]
  action: R basic 1
[(End,1)]                            [NT "basic" [Id "x"],End]
  action: G 3
[(End,1),(NT "basic" [Id "x"], 3)]                        [End]
  action: R expr 1
[(End,1)]                [NT "expr" [NT "basic" [Id "x"]],End]
```

What's powering this

	id	+	End	expr	basic
1	S5			**G2**	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                  input
-----                                                  -----
[(End,1)]                                    [(Id "x"), End]
  action: S 5
[(End,1),(Id "x", 5)]                                  [End]
  action: R basic 1
[(End,1)]                         [NT "basic" [Id "x"],End]
  action: G 3
[(End,1),(NT "basic" [Id "x"], 3)]                     [End]
  action: R expr 1
[(End,1)]            [NT "expr" [NT "basic" [Id "x"]],End]
  action: G 2
```

What's powering this

	id	+	End	expr	basic
1	S5			**G2**	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                input
-----                                                -----
[(End,1)]                                   [(Id "x"), End]
  action: S 5
[(End,1),(Id "x", 5)]                                 [End]
  action: R basic 1
[(End,1)]                         [NT "basic" [Id "x"],End]
  action: G 3
[(End,1),(NT "basic" [Id "x"], 3)]                    [End]
  action: R expr 1
[(End,1)]            [NT "expr" [NT "basic" [Id "x"]],End]
  action: G 2
[(End,1),(NT "expr" [NT "basic" [Id "x"]], 2)]        [End]
```

What’s powering this

	id	+	End	expr	basic
1	S5			G2	G3
2			**Acc**		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                   input
-----                                                   -----
[(End,1)]                                        [(Id "x"), End]
  action: S 5
[(End,1),(Id "x", 5)]                                      [End]
  action: R basic 1
[(End,1)]                            [NT "basic" [Id "x"],End]
  action: G 3
[(End,1),(NT "basic" [Id "x"], 3)]                         [End]
  action: R expr 1
[(End,1)]            [NT "expr" [NT "basic" [Id "x"]],End]
  action: G 2
[(End,1),(NT "expr" [NT "basic" [Id "x"]], 2)]   [End]
  action: Acc
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                    input
-----                                    -----
[(End,1)]                  [Id "x", T "+", End]
```

What's powering this

	id	+	End	expr	basic
1	**S5**			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                         input
-----                                         -----
[(End,1)]                          [Id "x", T "+", End]
  action: S 5
```

What's powering this

	id	+	End	expr	basic
1	**S5**			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                      input
-----                                      -----
[(End,1)]                       [Id "x", T "+", End]
  action: S 5
[(End,1),(Id "x",5)]                    [T "+",End]
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		**Rb1**	Rb1		
6			Re3		

```
stack                                              input
-----                                              -----
[(End,1)]                              [Id "x", T "+", End]
  action: S 5
[(End,1),(Id "x",5)]                           [T "+",End]
  action: R basic 1
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		**Rb1**	Rb1		
6			Re3		

```
stack                                                     input
-----                                                     -----
[(End,1)]                                   [Id "x", T "+", End]
  action: S 5
[(End,1),(Id "x",5)]                                  [T "+",End]
  action: R basic 1
[(End,1)]                        [NT "basic" [Id "x"],T "+",End]
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	**G3**
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                    input
-----                                                    -----
[(End,1)]                                   [Id "x", T "+", End]
  action: S 5
[(End,1),(Id "x",5)]                                  [T "+",End]
  action: R basic 1
[(End,1)]                      [NT "basic" [Id "x"],T "+",End]
  action: G 3
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	**G3**
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                    input
-----                                                    -----
[(End,1)]                                 [Id "x", T "+", End]
  action: S 5
[(End,1),(Id "x",5)]                                [T "+",End]
  action: R basic 1
[(End,1)]                     [NT "basic" [Id "x"],T "+",End]
  action: G 3
[(End,1),(NT "basic" [Id "x"],3)]                   [T "+",End]
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		**S4**	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                        input
-----                                                        -----
[(End,1)]                                   [Id "x", T "+", End]
  action: S 5
[(End,1),(Id "x",5)]                                  [T "+",End]
  action: R basic 1
[(End,1)]                         [NT "basic" [Id "x"],T "+",End]
  action: G 3
[(End,1),(NT "basic" [Id "x"],3)]                     [T "+",End]
  action: S 4
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		**S4**	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                    input
-----                                                    -----
[(End,1)]                                 [Id "x", T "+", End]
  action: S 5
[(End,1),(Id "x",5)]                                [T "+",End]
  action: R basic 1
[(End,1)]                      [NT "basic" [Id "x"],T "+",End]
  action: G 3
[(End,1),(NT "basic" [Id "x"],3)]                   [T "+",End]
  action: S 4
[(End,1),(NT "basic" [Id "x"],3),(T "+",4)]               [End]
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5		X	G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                 input
-----                                                 -----
[(End,1)]                               [Id "x", T "+", End]
  action: S 5
[(End,1),(Id "x",5)]                              [T "+",End]
  action: R basic 1
[(End,1)]                   [NT "basic" [Id "x"],T "+",End]
  action: G 3
[(End,1),(NT "basic" [Id "x"],3)]                 [T "+",End]
  action: S 4
[(End,1),(NT "basic" [Id "x"],3),(T "+",4)]           [End]
  action: Error
```

What's powering this

	id	+	End	expr	basic
1	S5			G2	G3
2			Acc		
3		S4	Re1		
4	S5			G6	G3
5		Rb1	Rb1		
6			Re3		

```
stack                                                   input
-----                                                   -----
[(End,1)]                                [Id "x", T "+", End]
  action: S 5
[(End,1),(Id "x",5)]                                [T "+",End]
  action: R basic 1
[(End,1)]                      [NT "basic" [Id "x"],T "+",End]
  action: G 3
[(End,1),(NT "basic" [Id "x"],3)]                   [T "+",End]
  action: S 4
[(End,1),(NT "basic" [Id "x"],3),(T "+",4)]              [End]
  action: Error
```

What error to give?

How to generate the transition table

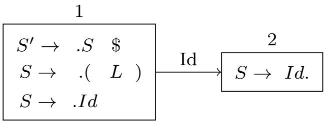
$$
\begin{array}{lll}
S' & ::= & S\ \$ \\
S & ::= & (\ L\ )\ |\ Id \\
L & ::= & S\ \ \ \ \ |\ L\ ,\ S
\end{array}
$$

1

$$
\boxed{\begin{array}{l}
S' \rightarrow\ .S\ \ \$ \\
S \rightarrow\ .(\ \ L\ \ ) \\
S \rightarrow\ .Id
\end{array}}
$$

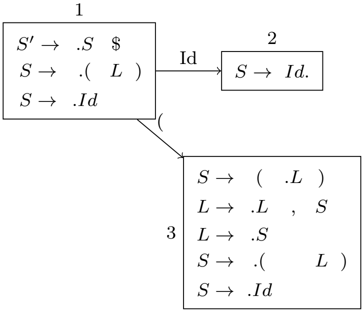
How to generate the transition table

$$
\begin{array}{lll}
S' & ::= & S\ \$ \\
S & ::= & (\ L\ )\ |\ Id \\
L & ::= & S\ \ \ \ \ |\ L\ ,\ S
\end{array}
$$



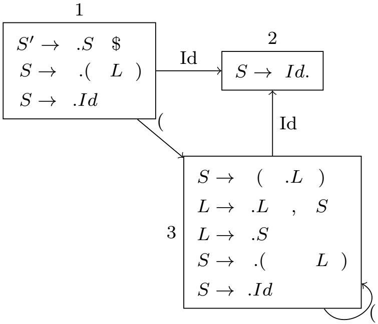
How to generate the transition table

$$
\begin{array}{lll}
S' & ::= & S\ \$ \\
S & ::= & (\ L\ )\ |\ Id \\
L & ::= & S\ \ \ \ |\ L\ ,\ S
\end{array}
$$



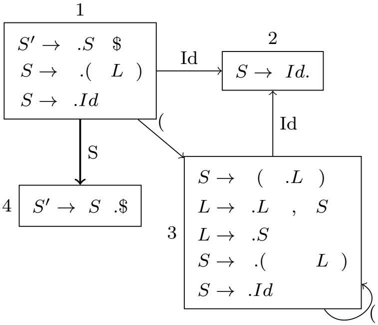
How to generate the transition table

$$
\begin{array}{lll}
S' & ::= S\ \$ & \\
S & ::= (\ L\ ) & \mid Id \\
L & ::= S & \mid L\ ,\ S
\end{array}
$$



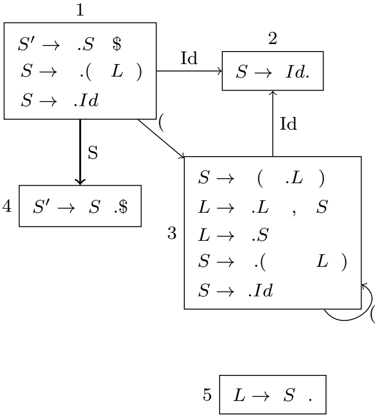
How to generate the transition table

$$
\begin{aligned}
S' &::= S\ \$ \\
S &::= (\ L\ )\ |\ Id \\
L &::= S\ \ |\ L\ ,\ S
\end{aligned}
$$



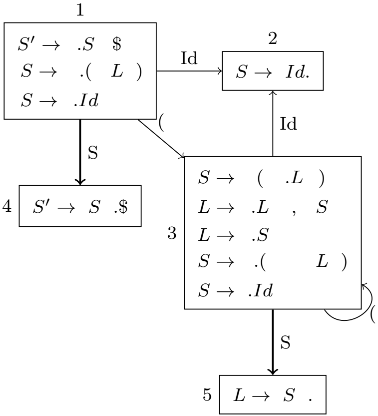
How to generate the transition table

$$\begin{aligned}
S' &::= S\ \$ \\
S &::= (\ L\ )\ |\ Id \\
L &::= S\ \ |\ L\ ,\ S
\end{aligned}$$



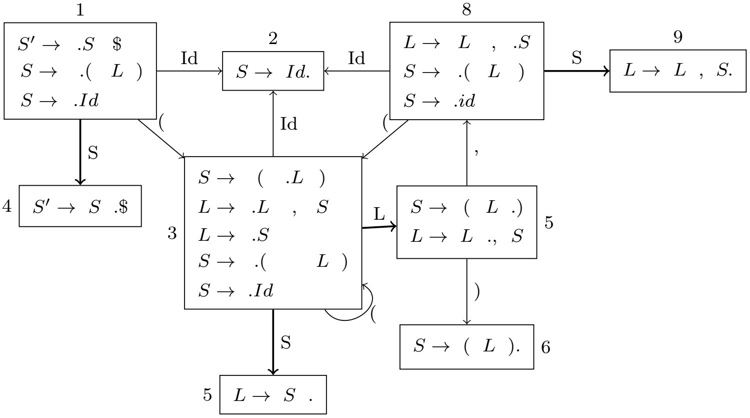
How to generate the transition table

$$
\begin{array}{lll}
S' & ::= & S\ \$ \\
S & ::= & (\ L\ )\ |\ Id \\
L & ::= & S \quad |\ L\ ,\ S
\end{array}
$$



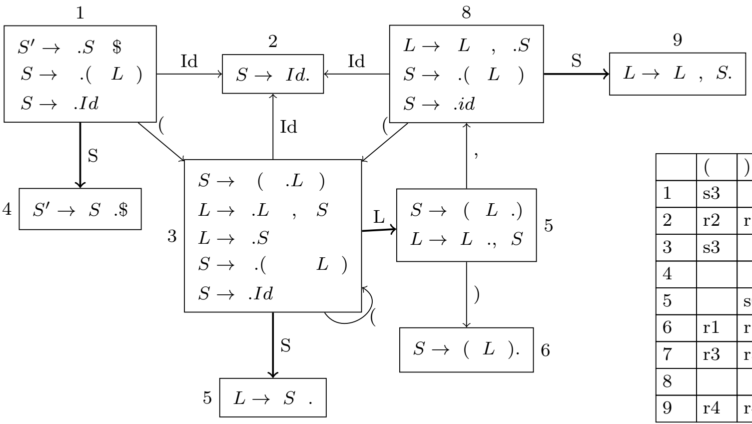
How to generate the transition table

$$
\begin{array}{lll}
S' & ::= & S\ \$ \\
S & ::= & (\ L\ )\ |\ Id \\
L & ::= & S \qquad |\ L\ ,\ S
\end{array}
$$



How to generate the transition table

$$
\begin{array}{lll}
S' & ::= S\ \$ & \\
S & ::= (\ \ L\ ) & \mid Id \\
L & ::= S & \mid L\ \ ,\ S
\end{array}
$$



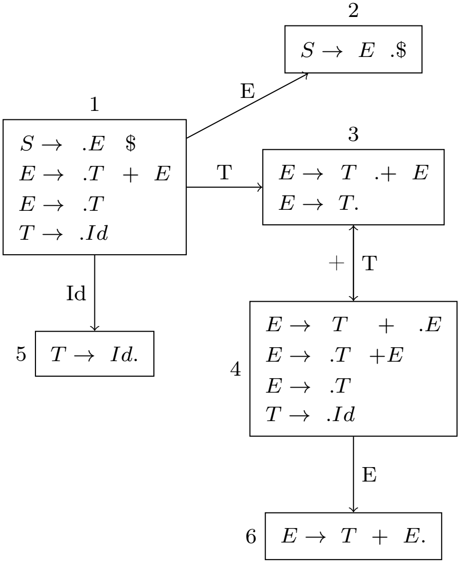
	(	)	Id	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					ac		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8						g9	
9	r4	r4	r4	r4	r4		

How to generate the transition table (2)

$$
\begin{array}{lcl}
S' & ::= & S \;\; \$ \\
S & ::= & (\quad L \;) \\
 & | & Id \\
L & ::= & S \\
 & | & L \;\; , \;\; S
\end{array}
$$

	(	)	Id	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					ac		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8						g9	
9	r4	r4	r4	r4	r4		

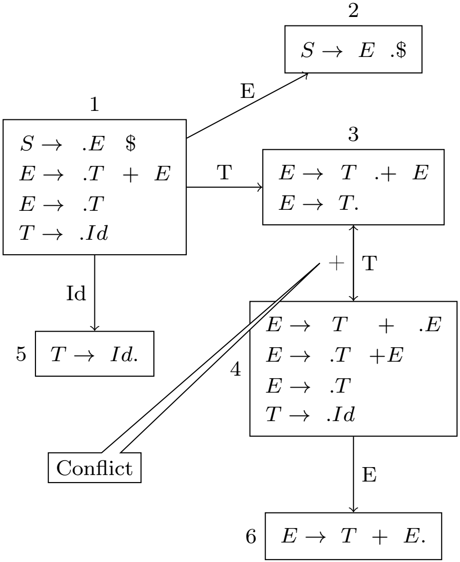
One token lookahead is not always sufficient



$$
\begin{array}{llll}
S & ::= & E & \$ \\
E & ::= & T & + \ E \\
 & | & T & \\
T & ::= & Id &
\end{array}
$$

	Id	+	$	E	T
1	s5			g2	g3
2			acc		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r4	r3		
6	r1	r1	r1		

One token lookahead is not always sufficient



$$
\begin{array}{llllll}
S & ::= & E & \$ & \\
E & ::= & T & + & E \\
 & | & T & & \\
T & ::= & Id & &
\end{array}
$$

	Id	+	$	E	T
1	s5			g2	g3
2			acc		
3	r2	**s4,r2**	r2		
4	s5			g6	g3
5	r3	r4	r3		
6	r1	r1	r1		

Notes on the data types

- AST $\neq$ Parse tree
- Tokens include location?
- AST includes location? (easy in OO, complexer in FP)
- Annotate with types?
- $\ldots$

Conclusion

Conclusion

- Shift-reduce parser are powerful and efficient

Conclusion

- Shift-reduce parser are powerful and efficient
- Producing tables is detention work

Conclusion

- Shift-reduce parser are powerful and efficient
- Producing tables is detention work
- Scannerless parsing has up and downsides

Conclusion

- Shift-reduce parser are powerful and efficient
- Producing tables is detention work
- Scannerless parsing has up and downsides
- Making a parser is easy

Conclusion

- Shift-reduce parser are powerful and efficient
- Producing tables is detention work
- Scannerless parsing has up and downsides
- Making a parser is easy
- Making a good parser is difficult

Conclusion

- Shift-reduce parser are powerful and efficient
- Producing tables is detention work
- Scannerless parsing has up and downsides
- Making a parser is easy
- Making a good parser is difficult
- Error handling is difficult

Conclusion

- Shift-reduce parser are powerful and efficient
- Producing tables is detention work
- Scannerless parsing has up and downsides
- Making a parser is easy
- Making a good parser is difficult
- Error handling is difficult
- Parser combinators are often interchangable

Conclusion

- Shift-reduce parser are powerful and efficient
- Producing tables is detention work
- Scannerless parsing has up and downsides
- Making a parser is easy
- Making a good parser is difficult
- Error handling is difficult
- Parser combinators are often interchangable
- Next week: Presentations, aftewards: typing with Sjaak

Conclusion

- Shift-reduce parser are powerful and efficient
- Producing tables is detention work
- Scannerless parsing has up and downsides
- Making a parser is easy
- Making a good parser is difficult
- Error handling is difficult
- Parser combinators are often interchangable
- Next week: Presentations, aftewards: typing with Sjaak
- Presentation (8 groups, 10 minutes per presentation including questions), report