Lecture 16: Final Summary

Talen en Compilers 2023-2024, period 2

Lawrence Chonavel

Department of Information and Computing Sciences, Utrecht University

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

What we built

```
char* s = "hello";
while(putchar(*s++));
```

```
SSeq
 (SDecAsg "s"
  (EStrLit "hello there"))
 (SWhile
  (EFunCall "putchar"
   [EDeref (EPostInc (EVar "s"))])
  (SBlock []))
```

Validator

Code generator

Parser

Printer

```
char* s = "hello";
while(putchar(*s++)) {
};
```

Evaluator

```
Result {
  ret_val = 0;
  stdout = "hello\0";
}
```

```
.LC0:
.string "hello"
main:
pushq %rbx
movl $.LC0, %ebx
.L2:
movq stdout(%rip), %rsi
movsbl (%rbx), %edi
addq $1, %rbx
call putc
testl %eax, %eax
jne .L2
popq %rbx
ret
```

🌪️ Lecture Summary

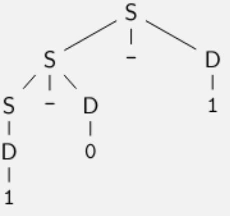
Concrete and abstract syntax

The grammar and the datatype describe the language.

concrete: **abstract** syntax:

concrete	abstract
S → S−D | D	**data** S = Minus S D | SingleDigit D
D → 0 | 1	**data** D = Zero | One

The string 1−0−1 corresponds to the parse tree



Haskell

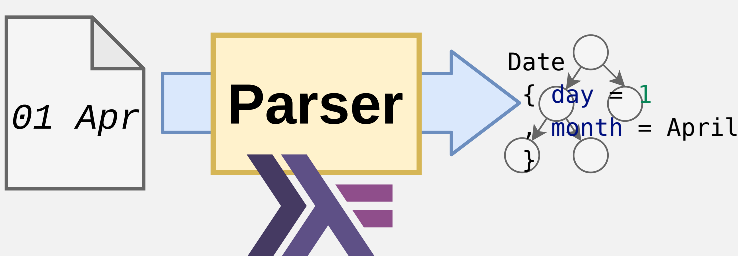
```
Minus (Minus (SingleDigit One)
             Zero)
      One
```

Lecture 2: BNF, Theory

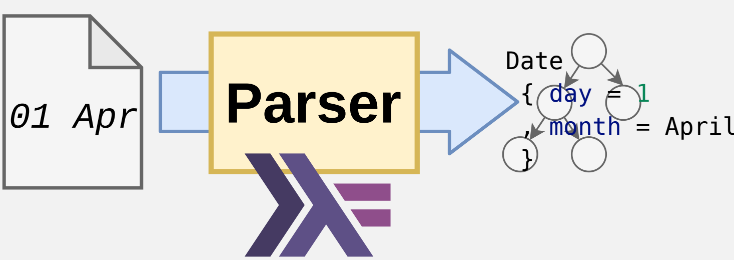
Summary

Grammar	A way to describe a language inductively.
Production	A rewrite rule in a grammar.
Context-free	The class of grammars/languages we consider.
Nonterminal	Auxiliary symbols in a grammar.
Terminal	Alphabet symbols in a grammar.
Derivation	Successively rewriting from a grammar until we reach a sentence.
Parse tree	Tree representation of a derivation.
Ambiguity	Multiple parse trees for the same sentence.
Abstract syntax	(Haskell) Datatype corresponding to a grammar.
Semantic function	Function defined on the abstract syntax.

Lecture 3: Parser Combinators (👶)

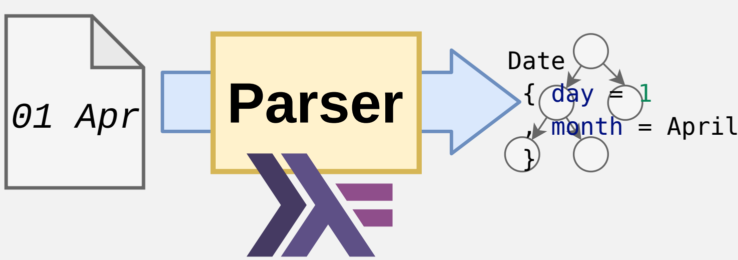


Lecture 3: Parser Combinators (👶)




```
type Parser a = String -> [(a,String)]
```

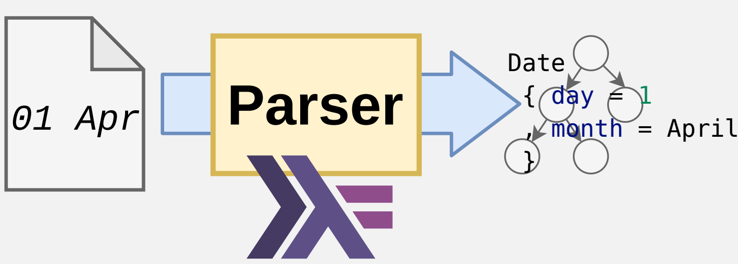
Lecture 3: Parser Combinators (👶)




```
type Parser a = String -> [(a,String)]
```

```
<$> ::          (a -> b) -> Parser a -> Parser b
<*> :: Parser (a -> b) -> Parser a -> Parser b
```

Lecture 3: Parser Combinators (👶)




```
type Parser a = String -> [(a,String)]
```

```
<$> ::          (a -> b) -> Parser a -> Parser b
<*> :: Parser (a -> b) -> Parser a -> Parser b
```

```
parseDate6 :: Parser Date
parseDate6 = Date <$> parseDay <*> parseMonth
```

Lecture 4: Parser Combinators (👦)

```
<$  ::            a -> Parser b -> Parser a
<*  ::     Parser a -> Parser b -> Parser a
<|> ::     Parser a -> Parser a -> Parser a
```


Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Lecture 4: Parser Combinators (👦)

```
<$  ::          a -> Parser b -> Parser a
<*  ::   Parser a -> Parser b -> Parser a
<|> ::   Parser a -> Parser a -> Parser a
```

```
type Parser' tok a = [tok] -> [(a,[tok])]
```

Lecture 5: Parser Combinators (👴)

```
chainl :: Parser a → Parser (a→a→a) → Parser a
chainr :: Parser a → Parser (a→a→a) → Parser a
```

Lecture 5: Parser Combinators (👴)

```
chainl :: Parser a → Parser (a→a→a) → Parser a
chainr :: Parser a → Parser (a→a→a) → Parser a
```

```
gen :: [(Char, a→a→a)] → Parser a → Parser a
```


Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Lecture 5: Parser Combinators (👴)

```
chainl :: Parser a → Parser (a→a→a) → Parser a
chainr :: Parser a → Parser (a→a→a) → Parser a
```

```
gen :: [(Char, a→a→a)] → Parser a → Parser a
```

```
e1,e3 :: Parser Exp
e1 = foldr gen e3
  [ [('+', Plus), ('-', Minus)]
  , [('*', Times)]
  ]
e3 =  parenthesised e1
  <|> Nat <$> natural
```

Lecture 6: RegExp

```
<|> :: R → R → R
<+> :: R → R → R
many   :: R → R
many1 :: R → R
option   :: R → R
symbol   :: Char → R
satisfy :: (Char → Bool) → R
```

```
r1|r2
r1r2
r*
r+
r?
c
\d \s \S [a-z] ...
```

```
{}|{-?\d+(\.\d+)?}
|{(-?\d+(\.\d+)?,)+-?\d+(\.\d+)?}
```

Ubiquity

code eclipse sublime idea xcode notepad++ emacs vim vi ed grep awk sed find perl python javascript lua …

[Faculty of Science
Information and Computing
Sciences]

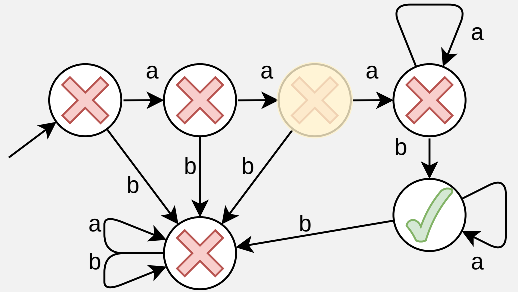
Lecture 7: RegExp implementation via FSMs

```
step :: Event -> Memory -> Memory
step [☼] _ = State {color=W,on=True}
step [☽] _ = State {color=Y,on=True}
step [⏻] s = s {on = not (on s)}
```

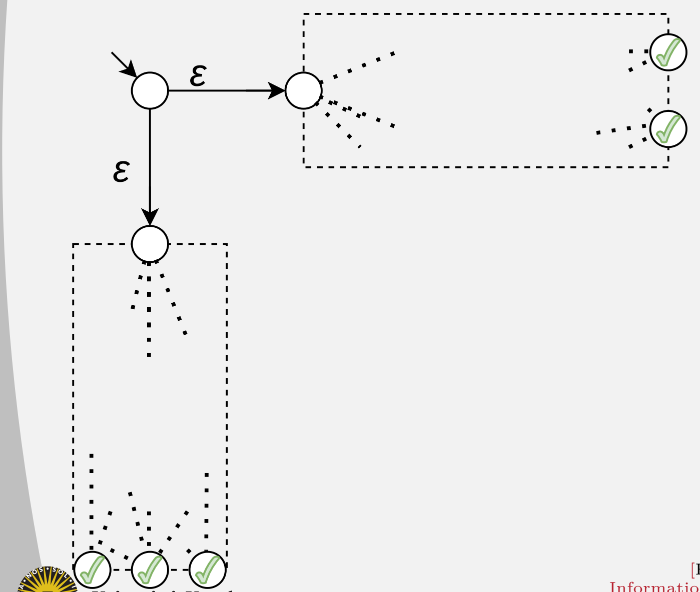
```
color :: YellowOrWhite
on :: Bool
```

```
genOut :: Memory -> Output
genOut (State {color=_,on=False}) = [lamp off]
genOut (State {color=W,on=True})  = [white lamp]
genOut (State {color=Y,on=True})  = [yellow lamp]
```

Lecture 7: RegExp implementation via FSMs



Lecture 7: RegExp implementation via FSMs



Lecture 7: RegExp implementation via FSMs

```
n2d :: NFAε sy st → DFA sy (Set st)
n2d (NFAε step εsteps genOut s0) = Moore
  { s0 = reachable εsteps (s0 nfa)
  , step = \sy → Set.unions . Set.map
      (reachable εsteps . step nfa sy)
  , genOut = any genOut }
```

Exercise 1

Write the type of the algebra for the following datatype:

```
data Expr v = Var v
            | App (Expr v) (Expr v)
            | Lam v (Expr v)
```

This represents λ-expressions in which variables are represented by values of type v (the **λ-calculus**).

```
type ExprAlgebra v r = (v → r, r → r → r, v → r → r)
foldExpr :: ExprAlgebra v r → Expr v → r
foldExpr (var, app, lam) = f
  where f (Var v)     = var v
        f (App x y) = app (f x) (f y)
        f (Lam v e) = lam v (f e)
```

Lecture 9: `fold`, for evaluation

Evaluation

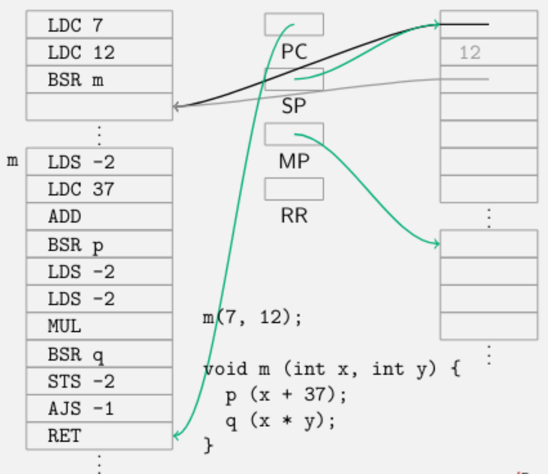
Directly:

$$\begin{aligned} &\text{eval} :: \text{E} \rightarrow \text{Int} \\ &\text{eval}\ (\text{Add}\ e_1\ e_2) = \text{eval}\ e_1 + \text{eval}\ e_2 \\ &\text{eval}\ (\text{Neg}\ e) \qquad = \text{negate}\ (\text{eval}\ e) \\ &\text{eval}\ (\text{Num}\ n) \qquad = n \end{aligned}$$

Using foldE:

$$\begin{aligned} &\text{eval} :: \text{E} \rightarrow \text{Int} \\ &\text{eval} = \text{foldE}\ ((+), \text{negate}, \text{id}) \end{aligned}$$

Lecture 10: code generation, SSM



Lecture 12: checks

```
har* s = "hello";
while
  putchar(*s+
);
```

```
.LC0:
.string "hello"
main:
pushq
movl
.L2:
movq stdout(%rip), %rsi
movsbl (%rbx), %edi
addq $1,
call putc
testl %eax, %eax
jne .L2
popq %rbx
ret
```

Comp

Mozilla Crash Reporter

We're Sorry

Firefox had a problem and crashed. We'll try to restore your tabs and windows when it restarts.

To help us diagnose and fix the problem, you can send us a crash report.

- [ ] Tell Mozilla about this crash so they can fix it

Details…

Add a comment (comments are publicly visible)

- [x] Include the address of the page I was on

Quit Firefox | Restart Firefox

#$!@ #$!@

Lecture 12: checks

```
char* s = "hello";
while(putchar(*s++));
```

HLint
ormolu
...

Parser

```
SSeq
 (SDecAsg "s"
  (EStrLit "hello there"))
 (SWhile
  (EFunCall "putchar"
   [EDeref (EPostInc
(EVar "s"))])
  (SBlock []))
```

scope-checker
type-checker
termination-checker
borrow-checker
...

Code generator

```
.LC0:
.string "hello"
main:
pushq %rbx
movl $.LC0, %ebx
.L2:
movq stdout(%rip), %rsi
movsbl (%rbx), %edi
addq $1, %rbx
call putc
testl %eax, %eax
jne .L2
popq %rbx
ret
```

SAST
bugprove
...
size-limit
tests

Lecture 12: checks

```
char* s = "hello";
while(putchar(*s++));
```

```
SSeq
 (SDecAsg "s"
  (EStrLit "hello there"))
 (SWhile
  (EFunCall "putchar"
   [EDeref (EPostInc
(EVar "s"))])
  (SBlock []))
```

```
.LC0:
.string "hello"
main:
pushq %rbx
movl $.LC0, %ebx
.L2:
movq stdout(%rip), %rsi
movsbl (%rbx), %edi
addq $1, %rbx
call putc
testl %eax, %eax
jne .L2
popq %rbx
ret
```

Parser

Code generator

scope-checker
type-checker

borrow-checker
...

Lecture 13: poking the limits of RegExp/BNF

C
o o
D B
p l
x y
start S A E

Lecture 13: poking the limits of RegExp/BNF

S

A

A

A

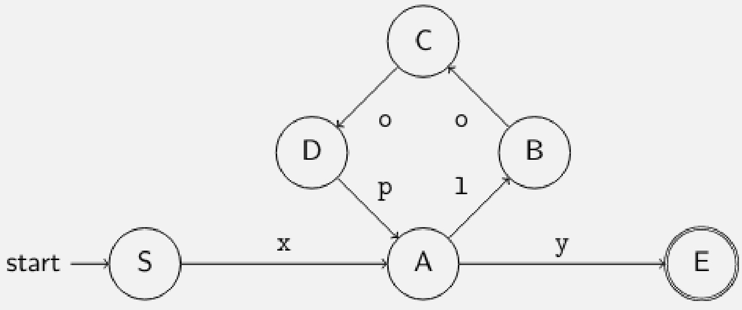
A

u v x y

v x

v w x

Lecture 13: poking the limits of RegExp/BNF



Pumping Lemma for regular languages

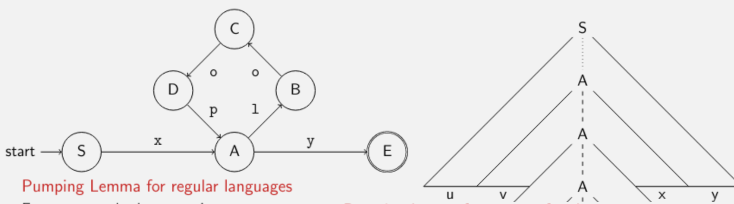
For every regular language L,
there exists an $n \in \mathbb{N}$

such that for every word xyz in L with $|y| \geqslant n$,

we can split y into three parts, $y = uvw$, with $|v| > 0$,

cience
outing
ences]

Lecture 13: poking the limits of RegExp/BNF



Pumping Lemma for regular languages

For every regular language L,
there exists an $n \in \mathbb{N}$

such that for every word xyz in L with $|y| \geqslant n$,

we can split y into three parts, $y = uvw$, with $|\mathrm{v}$

such that for every $i \in \mathbb{N}$, we have $xuv^iwz \in L$.

Pumping lemma for context-free languages

For every context-free language L,

- there exists a number $n \in \mathbb{N}$ such that
- for every word $z \in L$ with $|z| \geqslant n$,
- we can split z into five parts, $z = uvwxy$, with $|vx| > 0$ and $|vwx| \leqslant n$, such that
- for every $i \in \mathbb{N}$, we have $uv^iwx^iy \in L$.

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Lecture 14: nanopass

98
-XGeneralisedNewtypeDeriving
-XTypeFamilies

98

98
-XRebindableSyntax
-XTemplateHaskell
-XGADTs
-XUndecidableInstances

where is "putStrLn" defined?

LSP

System.IO (282,00) -(283,38)

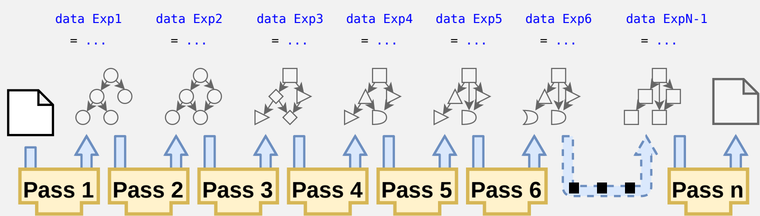
GHC

x86_64

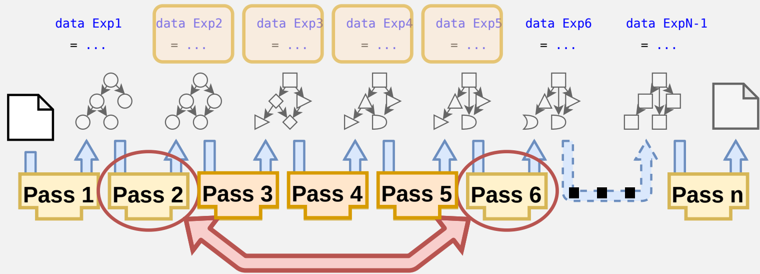
x86

armv7hf

Lecture 14: nanopass



Lecture 14: nanopass



Lecture 15: optimization

```
for (int i = 0; i < n; i++)
{doStuff (i);
}
```

```
for (int i = 0; i < n - 4; i += 4)
{doStuff (i);
 doStuff (i + 1);
 doStuff (i + 2);
 doStuff (i + 3);
}
```

Lecture 15: optimization

```
for (int i = 0; i < n; i++)
{doStuff (i);
}
```

```
for (int i = 0; i < n - 4; i += 4)
{doStuff (i);
 doStuff (i + 1);
 doStuff (i + 2);
 doStuff (i + 3);
}
```

Risky!

- When is it safe?
- When does it improve the code?
- When does it degrade the code?

Lecture 15: optimization

```
for (int i = 0; i < n; i++)
{doStuff (i);
}
```

```
for (int i = 0; i < n - 4; i += 4)
{doStuff (i);
 doStuff (i + 1);
 doStuff (i + 2);
 doStuff (i + 3);
}
```

🎲 Risky!

- When is it safe? 💣
- When does it improve the code? ✨
- When does it degrade the code? 💩
- Usually need analysis 🔎

Revision Guide

Parsing

Rest of Compiler

Revision Guide

Parsing

Parser
Combinators

Rest of
Compiler

Revision Guide

Parsing

Parser
Combinators

Parser
Generators

Rest of
Compiler

Revision Guide

Parsing

Parser
Combinators

Rest of
Compiler

Parser
Generators

BNF

Revision Guide

Parsing

Parser Combinators

Rest of Compiler

Parser Generators

BNF

Normal forms

Revision Guide

Parsing

Parser Combinators

Parser Generators

BNF

Normal forms

Rest of Compiler

RegExp

Revision Guide

Parsing
Parser
Combinators
Rest of
Compiler
Parser
Generators
BNF
Normal forms
NFA
RegExp

Revision Guide

Parsing

Parser
Combinators

Rest of
Compiler

Parser
Generators

DFA

BNF

NFA

Normal forms

RegExp

Revision Guide

Parsing

Parser Combinators

Parser Generators

BNF

Normal forms

Rest of Compiler

DFA

nfa2dfa

NFA

RegExp

Revision Guide

Rest of Compiler

Parsing

Parser Combinators

Parser Generators

DFA

BNF

nfa2dfa

NFA

Normal forms

Theory

RegExp

Revision Guide

Parsing

Parser Combinators

Rest of Compiler

Parser Generators

DFA

BNF

nfa2dfa

NFA

Normal forms

Folds

Theory

RegExp

Revision Guide

Parsing
Parser
Combinators
Rest of
Compiler
Parser
Generators
BNF
Normal forms
DFA
nfa2dfa
NFA
Evaluation
Folds
Theory
RegExp

Revision Guide

Parsing
Parser Combinators
Parser Generators
BNF
Normal forms
Theory
Rest of Compiler
Checks
Evaluation
Folds
DFA
NFA
nfa2dfa
RegExp

Revision Guide

Parsing

Parser Combinators

Rest of Compiler

Checks

Optimisation

Evaluation

Parser Generators

BNF

Normal forms

DFA

nfa2dfa

NFA

Folds

Theory

RegExp

Revision Guide

Parsing
Parser Combinators
Parser Generators
BNF
Normal forms
Theory
Rest of Compiler
Checks
Optimisation
Nanopass
Evaluation
Folds
DFA
NFA
nfa2dfa
RegExp

🎤 Requests, Q&A

😔 The End?

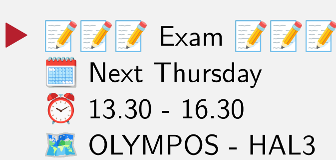
😀 Not the end!

- `mcpd` Concepts of programming language design
  - ➕ more on checking (semantics, meta-theory, rule notation)
  - ➕ more on evaluation (interpreter design)
  - ➕ more on nanopass (more lowering)
  - ➕ language design
- `dsl` Domain Specific Languages
  - ➕ more about parsing (how to avoid it)
  - ➕ language design
- `afp` Advanced Functional Programmming
  - ➕ more about haskell programming
- `b3stv` Software Testing & Verification
  - ➕ more about checking

All of the above: UU Software Technology MSc track

https://ics.uu.nl/docs/vakken/mcpd etc.

Now the end.



- Exam
  Next Thursday
  13.30 - 16.30
  OLYMPOS - HAL3
- Feedback wanted

only 5 questions