Compiler Construction

Week 2: Parsing I

Sjaak Smetsers Mart Lubbers Jordy Aaldering

2024/2025 KW3

Radboud University

Recap

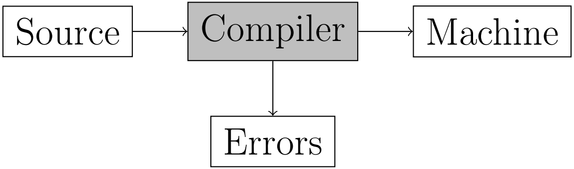
What was a compiler again?

Abstract view on a compiler

Compiler

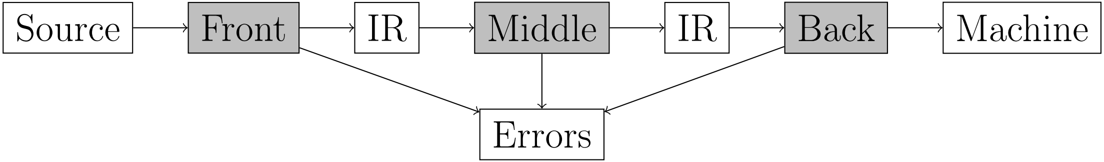
What was a compiler again?

Abstract view on a compiler

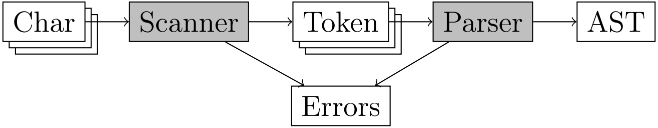


What was a compiler again?

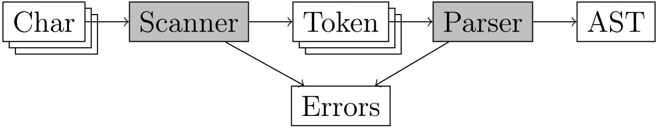
Three pass compiler



Frontend



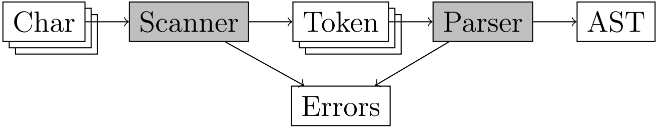
Frontend



Duty of the frontend

- Recognise the (context-free) syntax
- Produce IR
- Report errors

Frontend



Duty of the frontend

- Recognise the (context-free) syntax
- Produce IR
- Report errors

Duty of the scanner

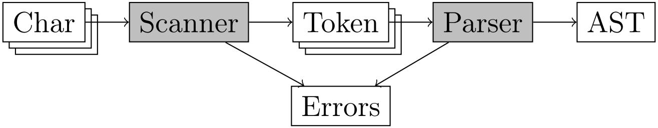
- Translate source code:

```
x = val + 42;
```

- to regular tokens

```
<id,x> <sym,=> ...
```

Frontend



Duty of the frontend

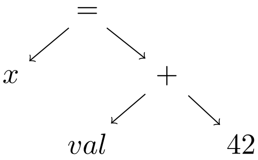
- Recognise the (context-free) syntax
- Produce IR
- Report errors

Duty of the scanner

- Translate source code:
  `x = val + 42;`
- to regular tokens
  `<id,x> <sym,=>` ...

Duty of the parser

- Translate tokens:
  `<id,x> <sym,=>` ...
- To an abstract syntax tree:



Scanning or parsing?

Where to draw the line

Scanning

Parsing

Scanning or parsing?

Where to draw the line

Scanning

- Words

Parsing

- Sentences (structure)

Scanning or parsing?

Where to draw the line

Scanning

- Words
- Regular grammar

Parsing

- Sentences (structure)
- Context free grammar

Scanning or parsing?

Where to draw the line

Scanning

- Words
- Regular grammar
- No recursion in the rules

Parsing

- Sentences (structure)
- Context free grammar
- Recursion allowed

Scanning or parsing?

Where to draw the line

Scanning

- Words
- Regular grammar
- No recursion in the rules
- Efficient

Parsing

- Sentences (structure)
- Context free grammar
- Recursion allowed
- Inefficient

Scanning or parsing?

Problems

- The lexer solves the easy problems

Scanning or parsing?

Problems

- The lexer solves the easy problems
- The parser solves the hard problems

Scanning or parsing?

Problems

- The lexer solves the easy problems
- The parser solves the hard problems
- Ambiguity

Scanning or parsing?
Problems

- The lexer solves the easy problems
- The parser solves the hard problems
- Ambiguity
  - Precendence

Scanning or parsing?

Problems

- The lexer solves the easy problems
- The parser solves the hard problems
- Ambiguity
  - Precendence
  - Associativity

Scanning or parsing?

Problems

- The lexer solves the easy problems
- The parser solves the hard problems
- Ambiguity
  - Precendence
  - Associativity
  - Dangling else

Context-free grammars

Context-free grammar (CFG)

and Backus-Naur form (BNF)

Context-free grammar (CFG)

and Backus-Naur form (BNF)

A CFG G is a 4-tuple

$$(V_t, V_n, S, P)$$

Context-free grammar (CFG)

and Backus-Naur form (BNF)

A CFG G is a 4-tuple

$$(V_t, V_n, S, P)$$

where

$$\begin{aligned} V_t &\quad \text{terminals} \\ V_n &\quad \text{non-terminals} \\ S &\quad \text{start } (S \in V_n) \\ P &\quad \text{productions} \end{aligned}$$

Context-free grammar (CFG)

and Backus-Naur form (BNF)

A CFG G is a 4-tuple

$$(V_t, V_n, S, P)$$

where

V_t	terminals
V_n	non-terminals
S	start ($S \in V_n$)
P	productions

Notation for grammars

$$
\begin{array}{lclccc}
goal & ::= & expr & & & \\
expr & ::= & expr & op & expr & \\
 & \mid & (& expr &) & \\
 & \mid & int & & & \\
 & \mid & id & & & \\
op & ::= & + & & & \\
 & \mid & * & & &
\end{array}
$$

terminal ($\in V_t$)

Context-free grammar (CFG)

and Backus-Naur form (BNF)

A CFG G is a 4-tuple

$$(V_t, V_n, S, P)$$

where

V_t	terminals
V_n	non-terminals
S	start ($S \in V_n$)
P	productions

Notation for grammars

$$\begin{array}{rcl} \mathit{goal} & ::= & \mathit{expr} \\ \mathit{expr} & ::= & \mathit{expr} \quad \mathit{op} \quad \mathit{expr} \\ & \mid & (\quad \mathit{expr} \quad) \\ & \mid & \mathit{int} \\ & \mid & \mathit{id} \\ \mathit{op} & ::= & + \\ & \mid & * \end{array}$$

terminal ($\in V_t$)

non-terminal ($\in V_n$)

Context-free grammar (CFG)

and Backus-Naur form (BNF)

A CFG G is a 4-tuple

$$(V_t, V_n, S, P)$$

where

V_t	terminals
V_n	non-terminals
S	start ($S \in V_n$)
P	productions

Notation for grammars

$$\begin{array}{lclll}
goal & ::= & expr & & \\
expr & ::= & expr & op & expr \\
 & \mid & (& expr &) \\
 & \mid & int & & \\
 & \mid & id & & \\
op & ::= & + & & \\
 & \mid & * & &
\end{array}$$

start (S)

terminal ($\in V_t$)

non-terminal ($\in V_n$)

Context-free grammar (CFG)

and Backus-Naur form (BNF)

A CFG G is a 4-tuple

$$(V_t, V_n, S, P)$$

where

V_t	terminals
V_n	non-terminals
S	start ($S \in V_n$)
P	productions

Notation for grammars

$$
\begin{array}{rcllll}
\boxed{goal & ::= & expr} & & & \text{production } (\in P) \\
expr & ::= & expr & op & expr & \\
\text{start } (S) & | & (& expr &) & \\
 & | & int & & & \text{terminal } (\in V_t) \\
 & | & id & & & \\
op & ::= & + & & & \\
 & | & * & & & \\
\text{non-terminal } (\in V_n) & & & & &
\end{array}
$$

Derivations

Example

```
x + 42 * 2
<id,x> <op,+> <int,42> <op,*> <int,2>
```

Rewriting

goal

Derivations

Example

```
x + 42 * 2
<id,x> <op,+> <int,42> <op,*> <int,2>
```

Rewriting

$$goal \Rightarrow \mathbf{expr}$$

Derivations

Example

`x + 42 * 2`

`<id,x> <op,+> <int,42> <op,*> <int,2>`

Rewriting

$$goal \Rightarrow expr$$

Derivations

Example

```
x + 42 * 2
<id,x> <op,+> <int,42> <op,*> <int,2>
```

Rewriting

$$\begin{aligned} goal &\Rightarrow expr \\ &\Rightarrow \mathbf{expr\ op\ expr} \end{aligned}$$

Derivations

Example

```
x + 42 * 2
<id,x> <op,+> <int,42> <op,*> <int,2>
```

Rewriting

$$
\begin{aligned}
goal &\Rightarrow expr \\
&\Rightarrow expr\ op\ expr
\end{aligned}
$$

Derivations

Example

```
x + 42 * 2
<id,x> <op,+> <int,42> <op,*> <int,2>
```

Rewriting

$$
\begin{aligned}
goal &\Rightarrow expr \\
&\Rightarrow expr\ op\ expr \\
&\Rightarrow id, x\ \mathbf{op\ expr}
\end{aligned}
$$

Derivations

Example

```
x + 42 * 2
<id,x> <op,+> <int,42> <op,*> <int,2>
```

Rewriting

$$
\begin{aligned}
goal &\Rightarrow expr \\
&\Rightarrow expr\ op\ expr \\
&\Rightarrow id, x\ op\ expr
\end{aligned}
$$

Derivations

Example

```
x + 42 * 2
<id,x> <op,+> <int,42> <op,*> <int,2>
```

Rewriting

$$\begin{aligned} goal &\Rightarrow expr \\ &\Rightarrow expr\ op\ expr \\ &\Rightarrow id, x\ op\ expr \\ &\Rightarrow id, x\ op, +\ \mathbf{expr} \end{aligned}$$

Derivations

Example

`x + 42 * 2`

`<id,x> <op,+> <int,42> <op,*> <int,2>`

Rewriting

$$
\begin{aligned}
goal &\Rightarrow expr \\
&\Rightarrow expr\ op\ expr \\
&\Rightarrow id, x\ op\ expr \\
&\Rightarrow id, x\ op, +\ expr
\end{aligned}
$$

Derivations

Example

`x + 42 * 2`

`<id,x> <op,+> <int,42> <op,*> <int,2>`

Rewriting

$$
\begin{aligned}
goal &\Rightarrow expr \\
&\Rightarrow expr\ op\ expr \\
&\Rightarrow id, x\ op\ expr \\
&\Rightarrow id, x\ op, +\ expr \\
&\Rightarrow id, x\ op, +\ \mathbf{expr\ op\ expr}
\end{aligned}
$$

Derivations

Example

`x + 42 * 2`

`<id,x> <op,+> <int,42> <op,*> <int,2>`

Rewriting

$$
\begin{aligned}
goal &\Rightarrow expr \\
&\Rightarrow expr\ op\ expr \\
&\Rightarrow id, x\ op\ expr \\
&\Rightarrow id, x\ op, +\ expr \\
&\Rightarrow id, x\ op, +\ expr\ op\ expr
\end{aligned}
$$

Derivations

Example

`x + 42 * 2`

`<id,x> <op,+> <int,42> <op,*> <int,2>`

Rewriting

$$
\begin{aligned}
goal &\Rightarrow expr \\
&\Rightarrow expr\ op\ expr \\
&\Rightarrow id, x\ op\ expr \\
&\Rightarrow id, x\ op, +\ expr \\
&\Rightarrow id, x\ op, +\ expr\ op\ expr \\
&\Rightarrow id, x\ op, +\ int, 42\ \mathbf{op\ expr}
\end{aligned}
$$

Derivations

Example

```
x + 42 * 2
<id,x> <op,+> <int,42> <op,*> <int,2>
```

Rewriting

$$
\begin{aligned}
goal &\Rightarrow expr \\
&\Rightarrow expr\ op\ expr \\
&\Rightarrow id, x\ op\ expr \\
&\Rightarrow id, x\ op, +\ expr \\
&\Rightarrow id, x\ op, +\ expr\ op\ expr \\
&\Rightarrow id, x\ op, +\ int, 42\ op\ expr
\end{aligned}
$$

Derivations

Example

`x + 42 * 2`

`<id,x> <op,+> <int,42> <op,*> <int,2>`

Rewriting

$$
\begin{aligned}
goal &\Rightarrow expr \\
&\Rightarrow expr\ op\ expr \\
&\Rightarrow id, x\ op\ expr \\
&\Rightarrow id, x\ op, +\ expr \\
&\Rightarrow id, x\ op, +\ expr\ op\ expr \\
&\Rightarrow id, x\ op, +\ int, 42\ op\ expr \\
&\Rightarrow id, x\ op, +\ int, 42\ op, *\ \mathbf{expr}
\end{aligned}
$$

Derivations

Example

`x + 42 * 2`

`<id,x> <op,+> <int,42> <op,*> <int,2>`

Rewriting

$$
\begin{aligned}
goal &\Rightarrow expr \\
&\Rightarrow expr\ op\ expr \\
&\Rightarrow id, x\ op\ expr \\
&\Rightarrow id, x\ op, +\ expr \\
&\Rightarrow id, x\ op, +\ expr\ op\ expr \\
&\Rightarrow id, x\ op, +\ int, 42\ op\ expr \\
&\Rightarrow id, x\ op, +\ int, 42\ op, *\ expr
\end{aligned}
$$

Derivations

Example

```
x + 42 * 2
<id,x> <op,+> <int,42> <op,*> <int,2>
```

Rewriting

$$
\begin{aligned}
goal &\Rightarrow expr \\
&\Rightarrow expr\ op\ expr \\
&\Rightarrow id, x\ op\ expr \\
&\Rightarrow id, x\ op, +\ expr \\
&\Rightarrow id, x\ op, +\ expr\ op\ expr \\
&\Rightarrow id, x\ op, +\ int, 42\ op\ expr \\
&\Rightarrow id, x\ op, +\ int, 42\ op, *\ expr \\
&\Rightarrow id, x\ op, +\ int, 42\ op, *\ int, 2
\end{aligned}
$$

Rewriting grammars

- Confluent?

Rewriting grammars

- Confluent?
- Referentially transparent?

Rewriting grammars

- Confluent?
- Referentially transparent?
- One grammar specifies all valid sentences

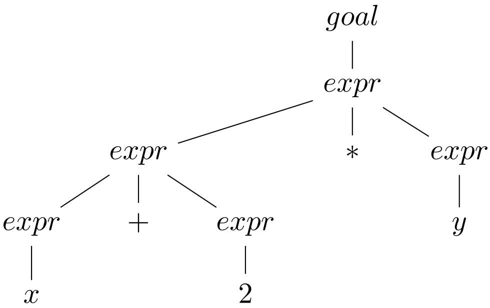
Using CFG's is problematic

Using CFG's is problematic, but not necessarily

Solving problems

Operator precedence

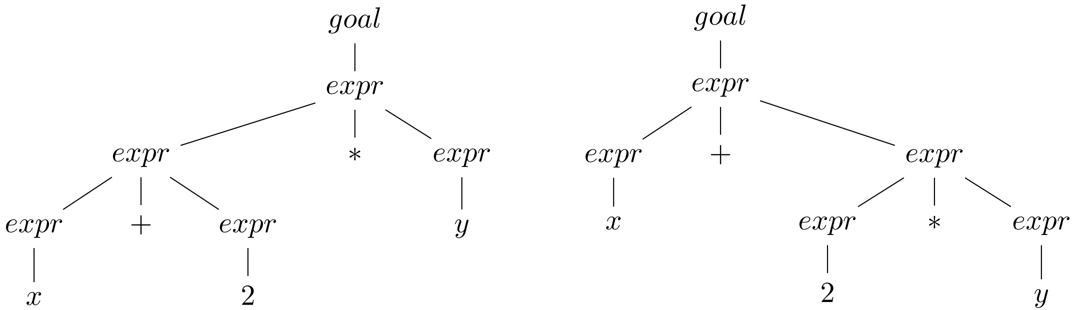
```
x + 2 * y
<id,x> <op,+> <int,2> <op,*> <id,y>
```



Solving problems

Operator precedence

```
x + 2 * y
<id,x> <op,+> <int,2> <op,*> <id,y>
```



Solving problems

Operator precedence: Solutions

- Preprocess and insert brackets[2]

[2] https://en.wikipedia.org/wiki/Operator-precedence_parser#Alternative_methods

Solving problems

Operator precedence: Solutions

- Preprocess and insert brackets[2]

[2] https://en.wikipedia.org/wiki/Operator-precedence_parser#Alternative_methods

Solving problems

Operator precedence: Solutions

- Preprocess and insert brackets[2]
  "The resulting formula is properly parenthesized, believe it or not."
  — *Donald E. Knuth*
- Postprocess and fix precedence

[2] `https://en.wikipedia.org/wiki/Operator-precedence_parser#Alternative_methods`

Solving problems

Operator precedence: Solutions

- Preprocess and insert brackets[2]
  "The resulting formula is properly parenthesized, believe it or not."
  — *Donald E. Knuth*
- Postprocess and fix precedence
- Shunting yard

[2] `https://en.wikipedia.org/wiki/Operator-precedence_parser#Alternative_methods`

Solving problems

Operator precedence: Solutions

- Preprocess and insert brackets[2]
  "The resulting formula is properly parenthesized, believe it or not."
  — *Donald E. Knuth*
- Postprocess and fix precedence
- Shunting yard
- Precedence climbing

[2] `https://en.wikipedia.org/wiki/Operator-precedence_parser#Alternative_methods`

Solving problems

Operator precedence: Solutions

- Preprocess and insert brackets[2]
  "The resulting formula is properly parenthesized, believe it or not."
  — *Donald E. Knuth*
- Postprocess and fix precedence
- Shunting yard
- Precedence climbing
- or. . .

[2] https://en.wikipedia.org/wiki/Operator-precedence_parser#Alternative_methods

Solving problems

Operator precedence: Solutions

- Preprocess and insert brackets[2]
  "The resulting formula is properly parenthesized, believe it or not."
  — *Donald E. Knuth*
- Postprocess and fix precedence
- Shunting yard
- Precedence climbing
- or…

[2] `https://en.wikipedia.org/wiki/Operator-precedence_parser#Alternative_methods`

Solving problems
Operator precedence: Solutions

- Preprocess and insert brackets[2]
  "The resulting formula is properly parenthesized, believe it or not."
  — *Donald E. Knuth*
- Postprocess and fix precedence
- Shunting yard
- Precedence climbing
- or. . .

Encode in the grammar

[2] https://en.wikipedia.org/wiki/Operator-precedence_parser#Alternative_methods

Solving problems

Operator precedence: Removing precedence ambiguity

Intuition

Top down, so parse from weakest to strongest precedence

In practice

$$
\begin{aligned}
goal &::= \\
sum &::= \\
fact &::= \\
basic &::=
\end{aligned}
$$

Solving problems

Operator precedence: Removing precedence ambiguity

Intuition

Top down, so parse from weakest to strongest precedence

In practice

$$\begin{aligned} goal &::= sum \\ sum &::= \\ fact &::= \\ basic &::= \end{aligned}$$

Solving problems

Operator precedence: Removing precedence ambiguity

Intuition

Top down, so parse from weakest to strongest precedence

In practice

$$
\begin{aligned}
goal &::= sum \\
sum &::= fact \;\; [+-] \; sum \\
fact &::= \\
basic &::=
\end{aligned}
$$

Solving problems

Operator precedence: Removing precedence ambiguity

Intuition

Top down, so parse from weakest to strongest precedence

In practice

$$
\begin{aligned}
goal &::= sum \\
sum &::= fact\ [+-]\ sum \\
&\ \ \mid\ \ fact \\
fact &::= \\
basic &::=
\end{aligned}
$$

Solving problems

Operator precedence: Removing precedence ambiguity

Intuition

Top down, so parse from weakest to strongest precedence

In practice

$$
\begin{array}{rcllll}
goal & ::= & sum & & \\
sum & ::= & fact & [+-] & sum \\
 & | & fact & & \\
fact & ::= & basic & * & fact \\
basic & ::= & & &
\end{array}
$$

Solving problems

Operator precedence: Removing precedence ambiguity

Intuition

Top down, so parse from weakest to strongest precedence

In practice

$$
\begin{array}{rcl}
goal & ::= & sum \\
sum & ::= & fact \quad [+-] \; sum \\
 & | & fact \\
fact & ::= & basic \quad * \quad fact \\
 & | & basic \\
basic & ::= &
\end{array}
$$

Solving problems

Operator precedence: Removing precedence ambiguity

Intuition

Top down, so parse from weakest to strongest precedence

In practice

$$
\begin{aligned}
goal &::= sum \\
sum &::= fact \quad [+-] \; sum \\
&\quad \mid \; fact \\
fact &::= basic \; * \quad fact \\
&\quad \mid \; basic \\
basic &::= int \\
&\quad \mid \; id
\end{aligned}
$$

Solving problems

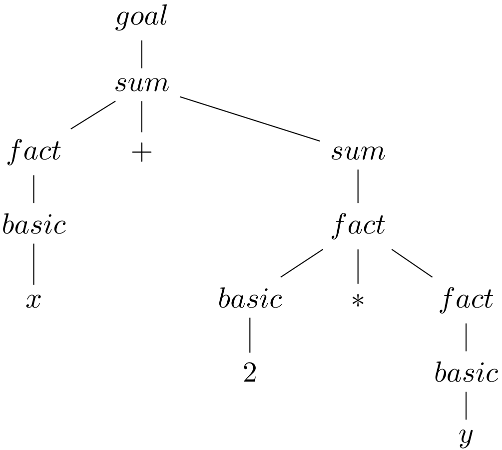
Operator precedence: Removing precedence ambiguity

Intuition

Top down, so parse from weakest to strongest precedence

In practice

$$
\begin{array}{rcllll}
goal & ::= & sum & & \\
sum & ::= & fact & [+-] & sum \\
 & | & fact & & \\
fact & ::= & basic & * & fact \\
 & | & basic & & \\
basic & ::= & int & & \\
 & | & id & &
\end{array}
$$



Solving problems

Operator associativity

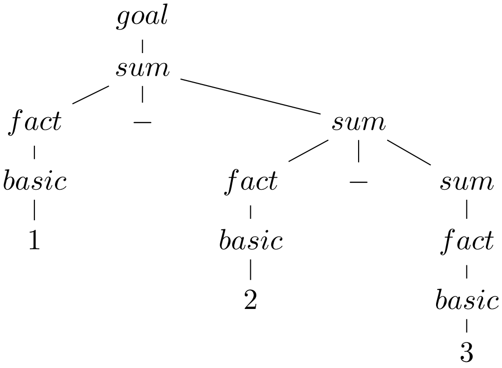
```
1-2-3
<int,1> <op,-> <int,2> <op,-> <int,3>
```

Solving problems

Operator associativity

`1-2-3`

`<int,1> <op,-> <int,2> <op,-> <int,3>`

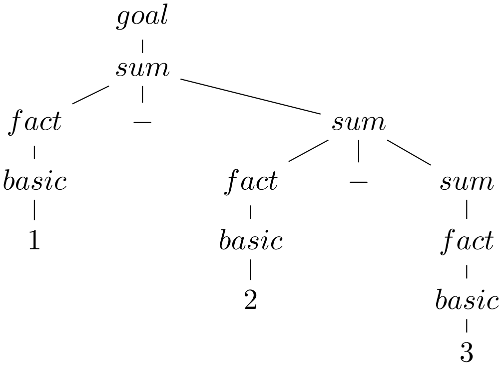


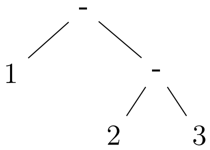
Solving problems

Operator associativity

`1-2-3`

`<int,1> <op,-> <int,2> <op,-> <int,3>`



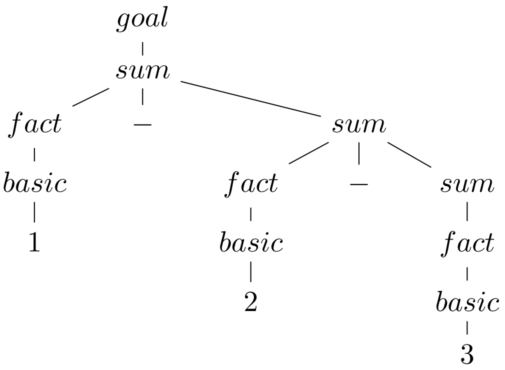


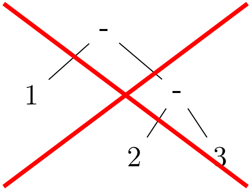
Solving problems

Operator associativity

`1-2-3`

`<int,1> <op,-> <int,2> <op,-> <int,3>`

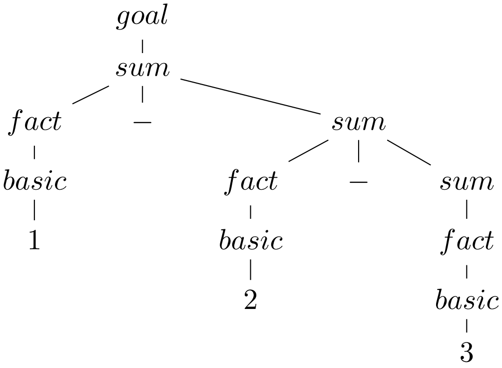


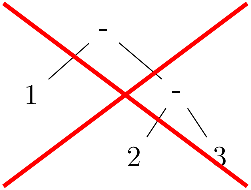


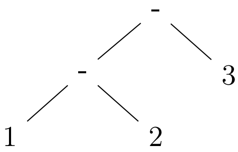
Solving problems

Operator associativity

```
1-2-3
<int,1> <op,-> <int,2> <op,-> <int,3>
```







Solving problems
Operator associativity

- Postprocess and fix precedence
- Shunting yard
- Precedence climbing
- or…

Encode in the grammar

Solving problems

Operator associativity: Solutions

Change order right?

Change order right?

$$
\begin{aligned}
goal &::= sum \\
sum &::= \color{red}{sum} \quad [+-]\ fact \\
&\quad\ \mid\ fact \\
fact &::= fact \quad * \quad basic \\
&\quad\ \mid\ basic \\
basic &::= int \\
&\quad\ \mid\ id
\end{aligned}
$$

Solving problems

Operator associativity: Solutions

Change order right?

$$
\begin{aligned}
goal &::= sum \\
sum &::= sum \quad [+-]\ fact \\
&\quad\ \mid\ fact \\
fact &::= fact \quad * \quad basic \\
&\quad\ \mid\ basic \\
basic &::= int \\
&\quad\ \mid\ id
\end{aligned}
$$

Tail recursion

Parse left associative operators greedy

Solving problems

Operator associativity: Solutions

Change order right?

$$
\begin{aligned}
goal &::= sum \\
sum &::= sum \quad [+-]\ fact \\
&\quad\mid\ fact \\
fact &::= fact \quad * \quad basic \\
&\quad\mid\ basic \\
basic &::= int \\
&\quad\mid\ id
\end{aligned}
$$

Tail recursion

Parse left associative operators greedy

$$
\begin{aligned}
goal &::= sum \\
sum &::= sum \quad [+-]\ fact \\
&\quad\mid\ fact \\
fact &::= fact \quad * \quad basic \\
&\quad\mid\ basic \\
basic &::= int \\
&\quad\mid\ id
\end{aligned}
$$

Solving problems

Operator associativity: Solutions

Change order right?

$$
\begin{aligned}
goal &::= sum \\
sum &::= sum \quad [+-]\ fact \\
&\quad |\ fact \\
fact &::= fact \quad * \quad basic \\
&\quad |\ basic \\
basic &::= int \\
&\quad |\ id
\end{aligned}
$$

Tail recursion

Parse left associative operators greedy

$$
\begin{aligned}
goal &::= sum \\
sum &::= fact \quad \{-\ fact\ \}^* \\
fact &::= fact \quad * \quad basic \\
&\quad |\ basic \\
basic &::= int \\
&\quad |\ id
\end{aligned}
$$

Change order right?

$$
\begin{array}{lcl}
goal & ::= & sum \\
sum & ::= & sum \quad [+-]\ fact \\
 & | & fact \\
fact & ::= & fact \quad * \quad basic \\
 & | & basic \\
basic & ::= & int \\
 & | & id
\end{array}
$$

Tail recursion

Parse left associative operators greedy

$$
\begin{array}{lcl}
goal & ::= & sum \\
sum & ::= & fact \quad \{-\ fact\ \}^* \\
fact & ::= & basic \quad \{*\ basic\ \}^* \\
basic & ::= & int \\
 & | & id
\end{array}
$$

Solving problems

Dangling else

$$\begin{aligned} stmt \ ::= & \ if \ \ expr \ \ then \ \ expr \\ | & \ \ if \ \ expr \ \ then \ \ expr \ \ else \ \ else \\ | & \ \ \ldots \end{aligned}$$

Solving problems

Dangling else

$$\begin{aligned} stmt \; ::= \; & if \;\; expr \;\; then \;\; expr \\ \mid \; & if \;\; expr \;\; then \;\; expr \;\; else \;\; else \\ \mid \; & \ldots \end{aligned}$$

```
if E1 then if E2 then S1 else S2
```

Solving problems

Dangling else

$$
\begin{aligned}
stmt \ ::= \ & if \ \ expr \ \ then \ \ expr \\
\mid \ & if \ \ expr \ \ then \ \ expr \ \ else \ \ else \\
\mid \ & \ldots
\end{aligned}
$$

```
if E1 then if E2 then S1 else S2
```

```
if E1 then (if E2 then S1) else S2
```

Solving problems

Dangling else

$$\begin{aligned} stmt \ ::= & \ if \ \ expr \ \ then \ \ expr \\ | & \ if \ \ expr \ \ then \ \ expr \ \ else \ \ else \\ | & \ \dots \end{aligned}$$

```
if E1 then if E2 then S1 else S2

if E1 then (if E2 then S1) else S2

if E1 then (if E2 then S1 else S2)
```

Solving problems

Dangling else

$$
\begin{aligned}
stmt \ ::= \ & if \ \ expr \ \ then \ \ expr \\
| \ \ & if \ \ expr \ \ then \ \ expr \ \ else \ \ else \\
| \ \ & \ldots
\end{aligned}
$$

```
if E1 then if E2 then S1 else S2

if E1 then (if E2 then S1) else S2

if E1 then (if E2 then S1 else S2)
```

Solution: endif, brackets, innermost else, refactor grammar

Intermezzo: Lexer hack for ANSI C

```
void fun()
{
  T (x);
}
```

Intermezzo: Lexer hack for ANSI C

```
void fun()
{
  T (x);
}
```

Function call or cast?

Intermezzo: Lexer hack for ANSI C

```
void fun()
{
  T (x);
}
```

```
void fun()
{
  T * x;
}
```

Function call or cast?

Intermezzo: Lexer hack for ANSI C

```
void fun()
{
  T (x);
}
```

Function call or cast?

```
void fun()
{
  T * x;
}
```

Multiplication or pointer declaration?

Left recursion

- Top down parsers cannot handle left recursion

Definition

Grammar G is left recursive if

$$\exists A \in V_n \text{ such that } A \Rightarrow^{+} A\alpha \text{ for some } \alpha$$

Left recursion

- Top down parsers cannot handle left recursion
- Infinite recursion

Definition

Grammar G is left recursive if

$$\exists A \in V_n \text{ such that } A \Rightarrow^{+} A\alpha \text{ for some } \alpha$$

Remove left recursion

Grammar

$$\begin{aligned} foo \ ::= & \ foo \ \ a \\ | & \ \ b \end{aligned}$$

Remove left recursion

Grammar

$$
\begin{aligned}
foo \ &::= foo \ \ a \\
&\ \ | \ \ b
\end{aligned}
$$

Language: ba^*

Remove left recursion

Grammar

$$
\begin{aligned}
foo &::= foo \;\; a \\
&\mid \;\; b
\end{aligned}
$$

Language: ba^*

Rewritten grammar

$$
\begin{aligned}
foo &::= b \;\; bar \\
bar &::= a \;\; bar \\
&\mid \;\; \epsilon
\end{aligned}
$$

Tricky

Removing immediate left recursion

Removing immediate left recursion

$$A ::= A\alpha_1 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \cdots \mid \beta_n$$

Removing immediate left recursion

$$A ::= A\alpha_1 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \cdots \mid \beta_n$$

$$\begin{aligned} A &::= \beta_1 A' \mid \cdots \mid \beta_n A' \\ A' &::= \alpha_1 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

Example I

Initial grammar

$$
\begin{aligned}
expr \;\; ::= &\; expr\ [+ - */\hat{}\,]expr \\
\mid &\; '(' \quad expr \quad ')' \\
\mid &\; int \\
\mid &\; id
\end{aligned}
$$

Example II

Fix precedence and associativity

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

Example III

Apply direct left recursion elimination

$$
\begin{aligned}
expr \;&::=\; fact\ expr' \\
expr' \;&::=\; [+-] fact\ expr' \\
&\quad\mid\ \epsilon \\
fact \;&::=\; pow\ fact' \\
fact' \;&::=\; [*/] pow\ fact' \\
&\quad\mid\ \epsilon \\
pow \;&::=\; basic\ \hat{}\ pow \\
&\quad\mid\ basic \\
basic \;&::=\; '('expr\ ')' \\
&\quad\mid\ int \\
&\quad\mid\ id
\end{aligned}
$$

Example IV

Or alternatively

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

Parsing strategies

Parsing in practice

Parsing in practice

- Given a grammar and a sentence

Parsing in practice

- Given a grammar and a sentence
- Find a derivation

Parsing in practice

- Given a grammar and a sentence
- Find a derivation

Parsing in practice

- Given a grammar and a sentence
- Find a derivation, or an error
- Two main strategies

Parsing in practice

- Given a grammar and a sentence
- Find a derivation, or an error
- Two main strategies

Parsing in practice

- Given a grammar and a sentence
- Find a derivation, or an error
- Two main strategies

1. Top-down

Parsing in practice

- Given a grammar and a sentence
- Find a derivation, or an error
- Two main strategies

1. Top-down
2. Bottom-up

Parsing in practice

Top-down parsers

- Start at root (S)
- Pick a production and try to match input
- Backtracking may be required
- Some grammars are predictive

Parsing in practice

Top-down parsers

- Start at root (S)
- Pick a production and try to match input
- Backtracking may be required
- Some grammars are predictive

Bottom-up parsers

- Start at the leafs
- Start in a valid state for the first token
- Consume, change state, encode possibilities
- Store state and sentential forms on a stack
- Topic of next week

Top-down parsing

Top-down parsers

Top-down parsers

- Start at root (S)

Top-down parsers

Top-down parsers

- Start at root (S)
- Pick an alternative and try to match input

Top-down parsers

Top-down parsers

- Start at root (S)
- Pick an alternative and try to match input
- When the terminal doesn't match: backtrack (lookahead)

Top-down parsers

Top-down parsers

- Start at root (S)
- Pick an alternative and try to match input
- When the terminal doesn't match: backtrack (lookahead)
- Find the next node

How to make a top-down parser

- Recursive descent parser

How to make a top-down parser

- Recursive descent parser
  - Function

How to make a top-down parser

- Recursive descent parser
  - Function
  - Recursive

How to make a top-down parser

- Recursive descent parser
  - Function
  - Recursive
  - Descent

How to make a top-down parser

- Recursive descent parser
  - Function
  - Recursive
  - Descent
- Combinators

How to make a top-down parser

- Recursive descent parser
  - Function
  - Recursive
  - Descent
- Combinators
- Libraries, bake your own

Let's jump right in!

Grammar

$$
\begin{aligned}
stmt \ ::= &\ id \ = expr \ ; \\
\mid &\ \{ \quad stmt\ ^* \ \}
\end{aligned}
$$

Let’s jump right in!

Grammar

$$
\begin{aligned}
stmt \;::=\;& id \;=\; expr \;\; ; \\
\mid\;& \{ \;\; stmt^{\,*} \;\; \}
\end{aligned}
$$

Haskell

```
data Stmt = Assignment IdToken Expr | Stmts [Stmt]

pStmt =  Assignment <$> satisfy isIdToken <* symbol '=' *> pExpr <*
    symbol ';'
     <|> Stmts      <$> symbol '{' *> many pStmt <* symbol '}'
```

What's powering this

P. Wadler, *How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages* (1985)

What's powering this

P. Wadler, *How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages* (1985)

failure	$[]$
deterministic	$[rs]$
nondeterministic	$[rs_1, \ldots, rs_n]$

What's powering this

P. Wadler, *How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages* (1985)

failure	$[]$
deterministic	$[rs]$
nondeterministic	$[rs_1, \ldots, rs_n]$

```
type Parser s a = [s] → [(a, [s])]
```

What's powering this

P. Wadler, *How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages* (1985)

failure	$[]$
deterministic	$[rs]$
nondeterministic	$[rs_1, \ldots, rs_n]$

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

Satisfy (`g ::= t`)

Recognize a symbol given a predicate

Satisfy (`g ::= t`)

Recognize a symbol given a predicate

```
satisfy :: (s → Bool) → Parser s s
```

Satisfy (`g ::= t`)

Recognize a symbol given a predicate

```
satisfy :: (s → Bool) → Parser s s
satisfy f = Parser $ \input → case input of
    (s:rest) | f s → [(s, rest)]
    _              → []
```

Satisfy (`g ::= t`)

Examples

```
runParser (satisfy isDigit) "123"
```

Satisfy (`g ::= t`)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
```

Satisfy (`g ::= t`)

Examples

```
runParser (satisfy isDigit) "123"
runParser (satisfy isDigit) "abc"
```

— $[('1', "23")]$

Satisfy (`g ::= t`)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
```

Satisfy (`g ::= t`)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
runParser (satisfy isDigit) "1"
```

Satisfy (`g ::= t`)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
runParser (satisfy isDigit) "1"   -- [('1',"")]
```

Satisfy (`g ::= t`)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
runParser (satisfy isDigit) "1"   -- [('1',"")]
runParser (satisfy isDigit) ""
```

Satisfy (g ::= t)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
runParser (satisfy isDigit) "1"   -- [('1',"")]
runParser (satisfy isDigit) ""    -- []
```

Satisfy (`g ::= t`)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
runParser (satisfy isDigit) "1"   -- [('1',"")]
runParser (satisfy isDigit) ""    -- []
```

Abbreviations

Satisfy (`g ::= t`)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
runParser (satisfy isDigit) "1"   -- [('1',"")]
runParser (satisfy isDigit) ""    -- []
```

Abbreviations

```
symbol s = satisfy (==s)
```

Satisfy (`g ::= t`)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
runParser (satisfy isDigit) "1"   -- [('1',"")]
runParser (satisfy isDigit) ""    -- []
```

Abbreviations

```
symbol s = satisfy (==s)
top = satisfy (\_ → True)
```

Satisfy (g ::= t)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
runParser (satisfy isDigit) "1"   -- [('1',"")]
runParser (satisfy isDigit) ""    -- []
```

Abbreviations

```
symbol s = satisfy (==s)
top = satisfy (\_ -> True)

runParser (symbol '.') "abc"
```

Satisfy (g ::= t)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
runParser (satisfy isDigit) "1"   -- [('1',"")]
runParser (satisfy isDigit) ""    -- []
```

Abbreviations

```
symbol s = satisfy (==s)
top = satisfy (\_ → True)

runParser (symbol '.') "abc"                -- []
```

Satisfy (g ::= t)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
runParser (satisfy isDigit) "1"   -- [('1',"")]
runParser (satisfy isDigit) ""    -- []
```

Abbreviations

```
symbol s = satisfy (==s)
top = satisfy (\_ → True)

runParser (symbol '.') "abc"                    -- []
runParser (symbol '.') ".bc"
```

Satisfy (`g ::= t`)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
runParser (satisfy isDigit) "1"   -- [('1',"")]
runParser (satisfy isDigit) ""    -- []
```

Abbreviations

```
symbol s = satisfy (==s)
top = satisfy (\_ → True)

runParser (symbol '.') "abc"                   -- []
runParser (symbol '.') ".bc"                   -- [('.',"bc")]
```

Satisfy (`g ::= t`)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
runParser (satisfy isDigit) "1"   -- [('1',"")]
runParser (satisfy isDigit) ""    -- []
```

Abbreviations

```
symbol s = satisfy (==s)
top = satisfy (\_ → True)

runParser (symbol '.') "abc"                    -- []
runParser (symbol '.') ".bc"                    -- [('.',"bc")]
runParser (symbol "hello") ["hello", "world"]
```

Satisfy (g ::= t)

Examples

```
runParser (satisfy isDigit) "123" -- [('1',"23")]
runParser (satisfy isDigit) "abc" -- []
runParser (satisfy isDigit) "1"   -- [('1',"")]
runParser (satisfy isDigit) ""    -- []
```

Abbreviations

```
symbol s = satisfy (==s)
top = satisfy (\_ → True)

runParser (symbol '.') "abc"                  -- []
runParser (symbol '.') ".bc"                  -- [('.',"bc")]
runParser (symbol "hello") ["hello", "world"] -- [("hello",["world"])]
```

Choice (`g ::= p1 | p2`)

Choose one or the other parser

Choice (`g ::= p1 | p2`)

Choose one or the other parser

```
infixl 3 <|>
(<|>) :: Parser s a → Parser s a → Parser s a
```

Choice (`g ::= p1 | p2`)

Choose one or the other parser

```
infixl 3 <|>
(<|>) :: Parser s a → Parser s a → Parser s a
x <|> y = Parser $ \t→runParser x t ++ runParser y t
```

Choice (`g :== p1 | p2`)

Examples

```
pLayout :: Parser Char Char
pLayout = symbol ' ' <|> symbol '\n' <|> symbol '\t'
```

Choice (g ::= p1 | p2)

Examples

```
pLayout :: Parser Char Char
pLayout = symbol ' ' <|> symbol '\n' <|> symbol '\t'

runParser pLayout " 123"
```

Choice (`g :== p1 | p2`)

Examples

```
pLayout :: Parser Char Char
pLayout = symbol ' ' <|> symbol '\n' <|> symbol '\t'

runParser pLayout " 123" -- [(' ',"123")]
```

```
pDigit = satisfy isDigit
pAlpha = satisfy isAlpha
pAlnum = pDigit <|> pAlpha
```

Choice (`g :== p1 | p2`)

Examples

```
pLayout :: Parser Char Char
pLayout = symbol ' ' <|> symbol '\n' <|> symbol '\t'
```

`runParser pLayout " 123"` — *[(' ',"123")]*

```
pDigit = satisfy isDigit
pAlpha = satisfy isAlpha
pAlnum = pDigit <|> pAlpha
```

`runParser (pDigit <|> pAlpha <|> pAlnum) "123"`

Choice (g :== p1 | p2)

Examples

```
pLayout :: Parser Char Char
pLayout = symbol ' ' <|> symbol '\n' <|> symbol '\t'
```

runParser pLayout " 123" — *[(' ',"123")]*

```
pDigit = satisfy isDigit
pAlpha = satisfy isAlpha
pAlnum = pDigit <|> pAlpha
```

runParser (pDigit <|> pAlpha <|> pAlnum) "123" — *[('1',"23"),('1',"23")]*

Sequence (`g ::= p q`)

Recognize multiple (non)terminals

Sequence (g ::= p q)

Recognize multiple (non)terminals

```
infixl 4 <*>
(<*>) :: Parser s (a → b) → Parser s a → Parser b
```

Sequence (`g ::= p q`)

Recognize multiple (non)terminals

```
infixl 4 <*>
(<*>) :: Parser s (a → b) → Parser s a → Parser b
l <*> r = Parser $ \t →
    [ (fa a, ts')
    | (fa, ts ) ← runParser l t
    , ( a, ts') ← runParser r ts
    ]
```

Sequence (`g ::= p q`)

Recognize multiple (non)terminals

```
infixl 4 <*>
(<*>) :: Parser s (a → b) → Parser s a → Parser b
l <*> r = Parser $ \t→
    [ (fa a, ts')
    | (fa, ts ) ← runParser l t
    , ( a, ts') ← runParser r ts
    ]
```

Return constant values

```
pure :: a → Parser s a
pure a = Parser $ \t→[(a, t)]
```

Sequence: bind

Right-hand determined by left-hand result

Sequence: bind

Right-hand determined by left-hand result

```
infixl 1 >>=
(>>=) :: Parser s a → (a → Parser s b) → Parser s b
```

Sequence: bind

Right-hand determined by left-hand result

```
infixl 1 >>=
(>>=) :: Parser s a → (a → Parser s b) → Parser s b
ma >>= a2mb = Parser $ \t→concat
    [ runParser (a2mb a) ts
    | (a, ts) ← runParser ma t
    ]
```

Sequence (`g :== p1 p2`)

Examples

```
pAlphaAndDigit = pure (,) <*> pAlpha <*> pDigit

runParser pAlphaAndDigit "a1b"
```

Sequence (`g :== p1 p2`)

Examples

```
pAlphaAndDigit = pure (,) <*> pAlpha <*> pDigit

runParser pAlphaAndDigit "a1b" -- [((a,1),"b")]
```

Sequence (`g :== p1 p2`)

Examples

```
pAlphaAndDigit = pure (,) <*> pAlpha <*> pDigit

runParser pAlphaAndDigit "a1b" -- [((a,1),"b")]

pAlphaAndDigit = pAlpha >>= \x → pDigit >>= \y → pure (x, y)
```

Expressive power `<*>` vs. `>>=`

Which has more expressive power?

Expressive power <*> vs. »=

Which has more expressive power?

```
pTwice :: Parser Char Char
pTwice = pAlpha >>= symbol
```

Expressive power <*> vs. »=

Which has more expressive power?

```
pTwice :: Parser Char Char
pTwice = pAlpha >>= symbol
```

Context sensitive!

Expressive power <*> vs. »=

Which has more expressive power?

```
pTwice :: Parser Char Char
pTwice = pAlpha >>= symbol
```

Context sensitive!

```
runParser pTwiceA "aabc"
```

Expressive power `<*>` vs. `»=`

Which has more expressive power?

```
pTwice :: Parser Char Char
pTwice = pAlpha >>= symbol
```

Context sensitive!

```
runParser pTwiceA "aabc" -- [('a',"bc")]
```

Expressive power `<*>` vs. `>>=`

Which has more expressive power?

```
pTwice :: Parser Char Char
pTwice = pAlpha >>= symbol
```

Context sensitive!

```
runParser pTwiceA "aabc" -- [('a',"bc")]
runParser pTwiceA "abc"
```

Expressive power <*> vs. »=

Which has more expressive power?

```
pTwice :: Parser Char Char
pTwice = pAlpha >>= symbol
```

Context sensitive!

```
runParser pTwiceA "aabc" -- [('a',"bc")]
runParser pTwiceA "abc"  -- []
```

Functor

Transform the result of a parser

Functor

Transform the result of a parser

```
infixr 7 <$>
(<$>) :: (a → b) → Parser s a → Parser s b
```

Functor

Transform the result of a parser

```
infixr 7 <$>
(<$>) :: (a → b) → Parser s a → Parser s b
f <$> p = Parser $ \t→[(f a, ts) | (a, ts)←runParser p t]
```

Functor

Transform the result of a parser

```
infixr 7 <$>
(<$>) :: (a → b) → Parser s a → Parser s b
f <$> p = Parser $ \t → [(f a, ts) | (a, ts) ← runParser p t]

runParser (digitToInt <$> pDigit) "1ab"
```

Functor

Transform the result of a parser

```
infixr 7 <$>
(<$>) :: (a → b) → Parser s a → Parser s b
f <$> p = Parser $ \t→[(f a, ts) | (a, ts)←runParser p t]

runParser (digitToInt <$> pDigit) "1ab" — [(1,"ab")]
```

Functor

Transform the result of a parser

```
infixr 7 <$>
(<$>) :: (a → b) → Parser s a → Parser s b
f <$> p = Parser $ \t → [(f a, ts) | (a, ts) ← runParser p t]

runParser (digitToInt <$> pDigit) "1ab" -- [(1,"ab")]

pAlphaAndDigit =
```

Functor

Transform the result of a parser

```
infixr 7 <$>
(<$>) :: (a → b) → Parser s a → Parser s b
f <$> p = Parser $ \t→[(f a, ts) | (a, ts)←runParser p t]

runParser (digitToInt <$> pDigit) "1ab" -- [(1,"ab")]

pAlphaAndDigit = (,) <$> pAlpha <*> pDigit
```

Forgetful sequence (`g ::= '(' t ')'`)

Parse but do not save

Forgetful sequence (`g ::= '(' t ')'`)

Parse but do not save

```
infixl 6 <*, *>
(<*) :: Parser s a → Parser s b → Parser s a

(*>) :: Parser s a → Parser s b → Parser s b
```

Forgetful sequence (`g ::= '(' t ')'`)

Parse but do not save

```
infixl 6 <*, *>
(<*) :: Parser s a → Parser s b → Parser s a
x <* y = (\x y→x) <$> x <*> y

(*>) :: Parser s a → Parser s b → Parser s b
x *> y = (\x y→y) <$> x <*> y
```

Forgetful sequence (`g ::= '(' t ')'`)

Parse but do not save

```
infixl 6 <*, *>
(<*) :: Parser s a → Parser s b → Parser s a
x <* y = (\x y→x) <$> x <*> y

(*>) :: Parser s a → Parser s b → Parser s b
x *> y = (\x y→y) <$> x <*> y

parens p = symbol '(' *> p <* symbol ')'
```

Forgetful sequence (g ::= '(' t ')')

Parse but do not save

```
infixl 6 <*, *>
(<*) :: Parser s a → Parser s b → Parser s a
x <* y = (\x y→x) <$> x <*> y

(*>) :: Parser s a → Parser s b → Parser s b
x *> y = (\x y→y) <$> x <*> y

parens p = symbol '(' *> p <* symbol ')'

runParser (parens pDigit) "(1)ab"
```

Forgetful sequence (g ::= '(' t ')')

Parse but do not save

```
infixl 6 <*, *>
(<*) :: Parser s a → Parser s b → Parser s a
x <* y = (\x y → x) <$> x <*> y

(*>) :: Parser s a → Parser s b → Parser s b
x *> y = (\x y → y) <$> x <*> y

parens p = symbol '(' *> p <* symbol ')'

runParser (parens pDigit) "(1)ab" -- [('1',"ab")]
```

Counting parenthesis depth

Counting parenthesis depth

```
parens  =  (+1) <$> (symbol '(' *> parens <* symbol ')')
       <|> pure 0

runParser parens "((()))"
```

Counting parenthesis depth

```
parens  =  (+1) <$> (symbol '(' *> parens <* symbol ')')
       <|> pure 0

runParser parens "((()))" -- [(3,""),(0,"((()))")]
```

Collecting results

Save results in a list

Collecting results

Save results in a list

```
infixr 6 <:>
(<:>) :: Parser s r → Parser s [r] → Parser s [r]
```

Collecting results

Save results in a list

```
infixr 6 <:>
(<:>) :: Parser s r → Parser s [r] → Parser s [r]
x <:> y = (:) <$> x <*> y
```

Collecting results

Save results in a list

```
infixr 6 <:>
(<:>) :: Parser s r → Parser s [r] → Parser s [r]
x <:> y = (:) <$> x <*> y
```

Abbreviations, Kleene star

```
many :: Parser s r → Parser s [r]
many p = p <:> many p <|> pure []

some :: Parser s r → Parser s [r]
some p = p <:> many p
```

Collecting results

Handle with care

Collecting results

Handle with care

```
runParser ((read :: String → Int) <$> many pDigit) "123."
```

Collecting results

Handle with care

```
runParser ((read :: String → Int) <$> many pDigit) "123."
  -- [(123,"."),
```

Collecting results

Handle with care

```
runParser ((read :: String → Int) <$> many pDigit) "123."
  -- [(123,"."), (12,"3."),
```

Collecting results

Handle with care

```
runParser ((read :: String → Int) <$> many pDigit) "123."
  — [(123,"."), (12,"3."), (1,"23."),
```

Collecting results

Handle with care

```
runParser ((read :: String → Int) <$> many pDigit) "123."
  -- [(123,"."), (12,"3."), (1,"23."), (*** Exception: Prelude.read:
     no parse
```

The better choice

Only parse the second if the first fails

The better choice

Only parse the second if the first fails

```
infixr 4 <<|>
(<<|>) :: (Parser s r) → (Parser s r) → Parser s r
```

The better choice

Only parse the second if the first fails

```
infixr 4 <<|>
(<<|>) :: (Parser s r) → (Parser s r) → Parser s r
x <<|> y = Parser $ \input→case runParser x input of
    []  → runParser y input
    res → res
```

The better choice

Only parse the second if the first fails

```
infixr 4 <<|>
(<<|>) :: (Parser s r) → (Parser s r) → Parser s r
x <<|> y = Parser $ \input→case runParser x input of
    []   → runParser y input
    res  → res

many p = p <:> many p <<|> pure []

runParser ((read :: String → Int) <$> many pDigit) "123."
```

The better choice

Only parse the second if the first fails

```
infixr 4 <<|>
(<<|>) :: (Parser s r) → (Parser s r) → Parser s r
x <<|> y = Parser $ \input→case runParser x input of
    []  → runParser y input
    res → res

many p = p <:> many p <<|> pure []

runParser ((read :: String → Int) <$> many pDigit) "123."
  -- [(123,".")]
```

Lookahead

- Arbitrary lookahead

Lookahead

- Arbitrary lookahead
- Combinators manage

Lookahead

- Arbitrary lookahead
- Combinators manage
- Reduced lookahead

Lookahead

- Arbitrary lookahead
- Combinators manage
- Reduced lookahead
- $LL(1)$: Left to right scan, left-most derivation, 1-token look ahead

Lookahead

- Arbitrary lookahead
- Combinators manage
- Reduced lookahead
- $LL(1)$: Left to right scan, left-most derivation, 1-token look ahead
- $LR(1)$: Left to right scan, right-most derivation, 1-token look ahead

Predictive parsing

Intuition

Which production to expand

Predictive parsing

Intuition

Which production to expand

Definition

For RHS $\alpha \in G$, define $FIRST(\alpha)$.

$$w \in V_t, w \in FIRST(\alpha) \text{ iff } \alpha \Rightarrow^{+} w\gamma.$$

Predictive parsing

Intuition

Which production to expand

Definition

For RHS $\alpha \in G$, define $FIRST(\alpha)$.

$$w \in V_t, w \in FIRST(\alpha) \text{ iff } \alpha \Rightarrow^{+} w\gamma.$$

Property

For any two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$

$$FIRST(\alpha) \bigcap FIRST(\beta) = \emptyset$$

Predictive parsing

Intuition

Which production to expand

Definition

For RHS $\alpha \in G$, define $FIRST(\alpha)$.

$$w \in V_t, w \in FIRST(\alpha) \text{ iff } \alpha \Rightarrow^{+} w\gamma.$$

Property

For any two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$

$$FIRST(\alpha) \bigcap FIRST(\beta) = \emptyset$$

Lookahead of 1!

Left factoring

Can we transform grammars to have this property?

Left factoring

Can we transform grammars to have this property?

Sometimes

Left factoring

Can we transform grammars to have this property?

Sometimes

- For each non-terminal A

Left factoring

Can we transform grammars to have this property?

Sometimes

- For each non-terminal A
- Find the longest prefix α common to two or more alternatives

Left factoring

Can we transform grammars to have this property?

Sometimes

- For each non-terminal A
- Find the longest prefix α common to two or more alternatives
- Replace all of the A productions

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n$$

with

$$\begin{aligned} A &\to \alpha A' \\ A' &\to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

Left factoring

Can we transform grammars to have this property?

Sometimes

- For each non-terminal A
- Find the longest prefix α common to two or more alternatives
- Replace all of the A productions

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n$$

with

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

- Repeat

Left factoring in general

Question

By left factoring and eliminating left-recursion, can we transform an arbitrary context-free grammar to a form where it can be predictively parsed with a single token look ahead?

Left factoring in general

Question

By left factoring and eliminating left-recursion, can we transform an arbitrary context-free grammar to a form where it can be predictively parsed with a single token look ahead?

Answer

Left factoring in general

Question

By left factoring and eliminating left-recursion, can we transform an arbitrary context-free grammar to a form where it can be predictively parsed with a single token look ahead?

Answer

Undecidably hard

Left factoring in general

Question

By left factoring and eliminating left-recursion, can we transform an arbitrary context-free grammar to a form where it can be predictively parsed with a single token look ahead?

Answer

Undecidably hard

Example

$$\{a^n 0 b^n | n \geq 1\} \bigcup \{a^n 1 b^{2n} | n \geq 1\}$$

Why left factoring?

- Efficiency

Why left factoring?

- Efficiency
  Parser combinators can handle arbitrary backtracking

Why left factoring?

- Efficiency
  Parser combinators can handle arbitrary backtracking
- Requirement

Why left factoring?

- Efficiency
  Parser combinators can handle arbitrary backtracking
- Requirement
  Recursive descent parsers require a store of arbitrary size

Conclusion

Conclusion

- Not all grammars are easy to parse

Conclusion

- Not all grammars are easy to parse
- Problem? Change the grammar

Conclusion

- Not all grammars are easy to parse
- Problem? Change the grammar
  - Left recursion

Conclusion

- Not all grammars are easy to parse
- Problem? Change the grammar
  - Left recursion
  - Precedence

Conclusion

- Not all grammars are easy to parse
- Problem? Change the grammar
  - Left recursion
  - Precedence
  - Associativity

Conclusion

- Not all grammars are easy to parse
- Problem? Change the grammar
  - Left recursion
  - Precedence
  - Associativity
  - Left factoring

Conclusion

- Not all grammars are easy to parse
- Problem? Change the grammar
  - Left recursion
  - Precedence
  - Associativity
  - Left factoring
- Parser combinators

Conclusion

- Not all grammars are easy to parse
- Problem? Change the grammar
  - Left recursion
  - Precedence
  - Associativity
  - Left factoring
- Parser combinators
  - Easy translations

Conclusion

- Not all grammars are easy to parse
- Problem? Change the grammar
  - Left recursion
  - Precedence
  - Associativity
  - Left factoring
- Parser combinators
  - Easy translations
  - Hides input passing

Next week

Homework

- Start implementing your parser

Next week

Homework

- Start implementing your parser
- Assignment is on Brightspace

Next week

Homework

- Start implementing your parser
- Assignment is on Brightspace
- You may use parser combinator libraries or parser generators

Next week

Homework

- Start implementing your parser
- Assignment is on Brightspace
- You may use parser combinator libraries or parser generators

Next week

Homework

- Start implementing your parser
- Assignment is on Brightspace
- You may use parser combinator libraries or parser generators

Next week

- Chain rule

Next week

Homework

- Start implementing your parser
- Assignment is on Brightspace
- You may use parser combinator libraries or parser generators

Next week

- Chain rule
- Bottom-up parsers

Next week

Homework

- Start implementing your parser
- Assignment is on Brightspace
- You may use parser combinator libraries or parser generators

Next week

- Chain rule
- Bottom-up parsers
- Parser generators

Next week

Homework

- Start implementing your parser
- Assignment is on Brightspace
- You may use parser combinator libraries or parser generators

Next week

- Chain rule
- Bottom-up parsers
- Parser generators
- Scannerless parsers