IT UNIVERSITY OF COPENHAGEN

Functional Programming

Patrick Bahr

Parser Combinators

Based on slides by Jesper Bengtson

Where we are

- Core principles of functional programming:
  - (Recursive) Algebraic data types
  - Higher-order and recursive functions
  - Pattern matching
  - Polymorphic types and functions

Weeks 1-3

Where we are

- Core principles of functional programming:
  - (Recursive) Algebraic data types
  - Higher-order and recursive functions
  - Pattern matching
  - Polymorphic types and functions

Weeks 1-3

- F# specific features:
  - Module system and interface files
  - Imperative features (loops, references)

Week 4

Where we are

- Computational behaviour of functional programs:
  - Memory model of F#
  - Tail recursion & optimisations
  - Lazy evaluation & sequences

Weeks 5 & 6

Where we are

- Computational behaviour of functional programs:
  - Memory model of F#
  - Tail recursion & optimisations
  - Lazy evaluation & sequences

Weeks 5 & 6

- Programming patterns for functional programs:
  - Monads & railway-oriented programming
  - Computation expressions
  - Combinator libraries & parsers

Weeks 7 & 8

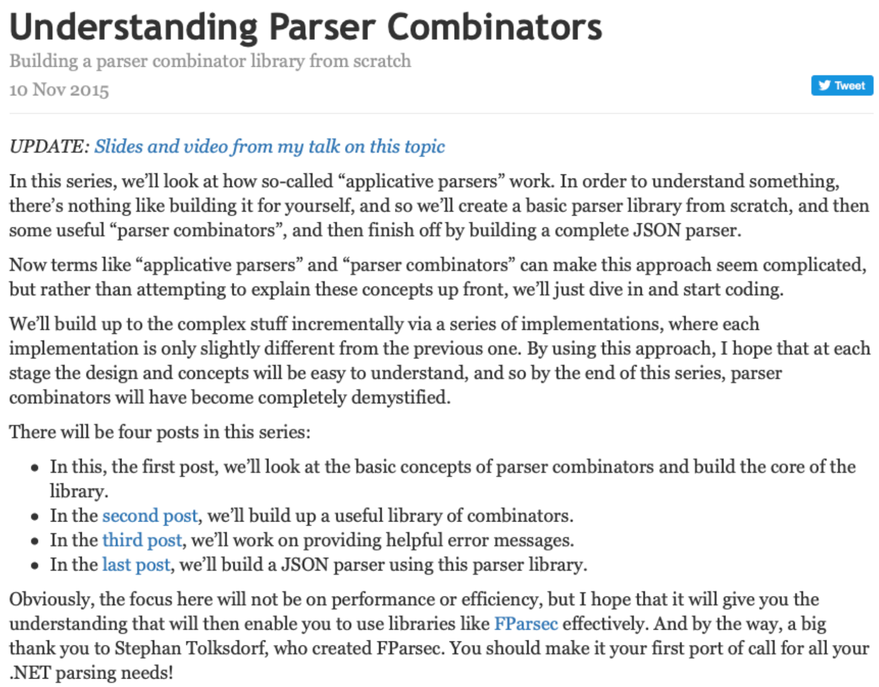
This week

- Context-free grammars
- Parser for arithmetic expression language
- Parser combinators

This week

- Context-free grammars
- Parser for arithmetic expression language
- Parser combinators

Based on



Understanding Parser Combinators

Building a parser combinator library from scratch

10 Nov 2015

Tweet

UPDATE: Slides and video from my talk on this topic

In this series, we'll look at how so-called "applicative parsers" work. In order to understand something, there's nothing like building it for yourself, and so we'll create a basic parser library from scratch, and then some useful "parser combinators", and then finish off by building a complete JSON parser.

Now terms like "applicative parsers" and "parser combinators" can make this approach seem complicated, but rather than attempting to explain these concepts up front, we'll just dive in and start coding.

We'll build up to the complex stuff incrementally via a series of implementations, where each implementation is only slightly different from the previous one. By using this approach, I hope that at each stage the design and concepts will be easy to understand, and so by the end of this series, parser combinators will have become completely demystified.

There will be four posts in this series:

- In this, the first post, we'll look at the basic concepts of parser combinators and build the core of the library.
- In the second post, we'll build up a useful library of combinators.
- In the third post, we'll work on providing helpful error messages.
- In the last post, we'll build a JSON parser using this parser library.

Obviously, the focus here will not be on performance or efficiency, but I hope that it will give you the understanding that will then enable you to use libraries like FParsec effectively. And by the way, a big thank you to Stephan Tolksdorf, who created FParsec. You should make it your first port of call for all your .NET parsing needs!

https://fsharpforfunandprofit.com/posts/understanding-parser-combinators/

By Scott Wlaschin

Part I

Context-Free Grammars

Context-free grammars

Context-free grammars are used to describe languages such as programming languages or data formats

Ingredients of CFGs:

- Terminal symbols
- Non-terminal symbols
- Production rules

Each production rule rewrites a non-terminal symbol to strings of non-terminal and terminal symbols

Context-free grammars

Context-free grammars are used to describe languages such as programming languages or data formats

Ingredients of CFGs:

- Terminal symbols
- Non-terminal symbols
- Production rules

Example

$A \rightarrow A + A$
$A \rightarrow x$

Each production rule rewrites a non-terminal symbol to strings of non-terminal and terminal symbols

Context-free grammars

Context-free grammars are used to describe languages such as programming languages or data formats

Ingredients of CFGs:

- Terminal symbols
- Non-terminal symbols
- Production rules

Example

A → A + A
A → x

Each production rule rewrites a non-terminal symbol to strings of non-terminal and terminal symbols

Context-free grammars

Context-free grammars are used to describe languages such as programming languages or data formats

Ingredients of CFGs:

- Terminal symbols
- Non-terminal symbols
- Production rules

Example

$A \rightarrow A + A$

$A \rightarrow x$

Each production rule rewrites a non-terminal symbol to strings of non-terminal and terminal symbols

Context-free grammars

Context-free grammars are used to describe languages such as programming languages or data formats

Ingredients of CFGs:

- Terminal symbols
- Non-terminal symbols
- Production rules

Example

A → A + A
A → x

Short hand notation (aka EBNF)

```
A ::= A + A | x
```

Each production rule rewrites a non-terminal symbol to strings of non-terminal and terminal symbols

Grammar of arithmetic expressions

Grammar for arithmetic expressions with variables

```
N ::= <integers>
V ::= <variable names>

A ::= N | V | A + A | A * A | ( A ) | - A
```

Grammar of arithmetic expressions

Grammar for arithmetic expressions with variables

```
N ::= <integers>
V ::= <variable names>
```

We won't write explicit rules for N and V

```
A ::= N | V | A + A | A * A | ( A ) | - A
```

Grammar of arithmetic expressions

Grammar for arithmetic expressions with variables

```
N ::= <integers>
V ::= <variable names>
```

We won't write explicit rules for N and V

```
A ::= N | V | A + A | A * A | ( A ) | - A
```

In the notation you know from *Discrete Mathematics:*

```
A → N        A → A + A        A → ( A )
A → V        A → A * A        A → - A
```

Grammar of arithmetic expressions

Grammar for arithmetic expressions with variables

```
N ::= <integers>
V ::= <variable names>

A ::= N | V | A + A | A * A | ( A ) | - A
```

Terminals

Grammar of arithmetic expressions

Grammar for arithmetic expressions with variables

```
N ::= <integers>
V ::= <variable names>

A ::= N | V | A + A | A * A | ( A ) | - A
```

Non-terminals

Grammar of arithmetic expressions

Grammar for arithmetic expressions with variables

```
N ::= <integers>
V ::= <variable names>

A ::= N | V | A + A | A * A | ( A ) | - A
```

Examples: Which of these are generated by the grammar?

```
x * (1 + y)        5 + -y        x + (5 * 2)
x (1 + y)          5 + *y        x + ()
```

Grammar of arithmetic expressions

Grammar for arithmetic expressions with variables

```
N ::= <integers>
V ::= <variable names>

A ::= N | V | A + A | A * A | ( A ) | - A
```

Examples:

Which of these are generated by the grammar?

x * (1 + y) ✓

x (1 + y) ✗

5 + -y ✓

5 + *y ✗

x + (5 * 2) ✓

x + () ✗

Abstract syntax trees

We can represent an expression described by this grammar by a data type called an **abstract syntax tree** (AST) or **expression tree**.

```
type aExp =
 | N of int
 | V of string
 | Add of aExp * aExp
 | Mul of aExp * aExp
 | Neg of aExp
```

For example:

```
Add (N 3, Mul (N 5, V "x"))
```

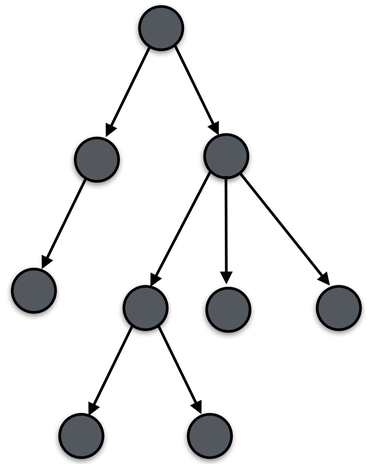
represents the expression 3 + (5 * x)

Parsers

In general, parsers translate unstructured input (text, binary, bit streams …) into a data structure

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec elementum sem id lectus efficitur convallis. Cras diam diam, elementum in libero sed, egestas dictum augue. Ut mauris lacus, interdum et dui sit amet, vestibulum mattis sapien. Nulla quis nibh urna. Maecenas sit amet nisi et est vehicula sagittis. Mauris a magna efficitur, varius lectus eget, imperdiet massa. Vivamus sodales nisl eget risus eleifend, at sollicitudin odio aliquet. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Lorem ipsum dolor sit amet, consectetur adipiscing elit.

parse

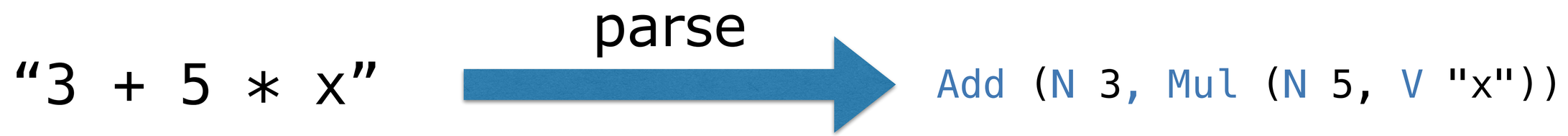


unstructured input

abstract syntax tree

Parsers

In general, parsers translate unstructured input (text, binary, bit streams …) into a data structure



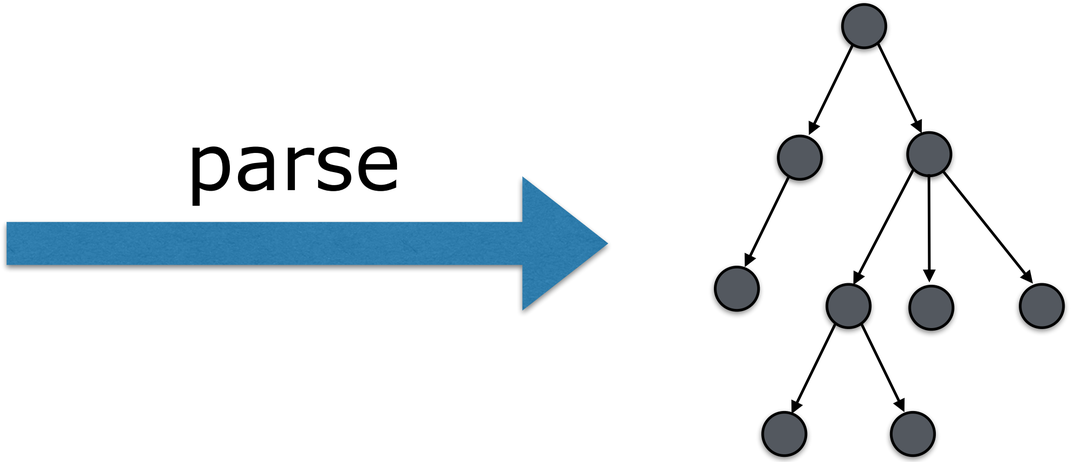
unstructured input

abstract syntax tree

Parsers

In general, parsers translate unstructured input (text, binary, bit streams …) into a data structure

```
01100110 11001110 01011110 10111001
10010101 01110111 11100110 11010101
11100001 01100001 00101110 11110100
00010100 01111111 00110000 00011001
00011000 01000110 10010010 01101010
10000011 00110110 01010011 01010101
11010010 10110100 11100000 00000010
11100010 11010000 11001100 01101001
10001111 01111111 01110111 00101000
00100011 01011110 10111001 00100010
11100000 11110110 00011010 10010100
00101010 00110111 11110001 00100001
```



unstructured input

abstract syntax tree

The job of a parser

Given a grammar

```
N ::= <integers>
V ::= <variables>

A ::= N      | V
    | A + A  | A * A
    |  -A    | ( A )
```

and a data type

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

turn “3 + 5 * x” into `Add (N 3, Mul (N 5, V "x"))`

The job of a parser

Given a grammar and a data type

```
N ::= <integers>
V ::= <variables>

A ::= N     | V
    | A + A | A * A
    |  -A   | ( A )
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

turn “3 + 5 * x” into `Add (N 3, Mul (N 5, V "x"))`

Precedence

Given a grammar

```
N ::= <integers>
V ::= <variables>

A ::= N     | V
    | A + A | A * A
    |  -A   | ( A )
```

How do we ensure that

```
3 + 5 * x
```

parses into

```
Add (N 3, Mul (N 5, V "x"))
```

but not

```
Mul (Add (N 3, N 5), V "x")
```

?

Precedence

Given a grammar

```
N ::= <integers>
V ::= <variables>

A ::= N      | V
    | A + A  | A * A
    |  -A    | ( A )
```

How do we ensure that

```
3 +(5 * x)
```

this

parses into

```
Add (N 3, Mul (N 5, V "x"))
```

but not

```
Mul (Add (N 3, N 5), V "x")
```

?

Precedence

Given a grammar

```
N ::= <integers>
V ::= <variables>

A ::= N      | V
    | A + A  | A * A
    |  -A    | ( A )
```

How do we ensure that

not this `(3 + 5)* x`

parses into

```
Add (N 3, Mul (N 5, V "x"))
```

but not

```
Mul (Add (N 3, N 5), V "x")
```

?

Precedence

Given a grammar

```
N ::= <integers>
V ::= <variables>

A ::= N      | V
    | A + A  | A * A
    |  -A    | ( A )
```

How do we ensure that

not this `(3 + 5)* x`

parses into

```
Add (N 3, Mul (N 5, V "x"))
```

but not

```
Mul (Add (N 3, N 5), V "x")
```

Problem: A parser would read the input left-to-right and apply the production rules in order. That would yield the wrong result!

Grammar of arithmetic expressions

Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```

Grammar of arithmetic expressions

Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```

This is better.
We ensure that precedence order is maintained.

Grammar of arithmetic expressions

Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```

This is better.
We ensure that precedence order is maintained.

Let's try to parse this:

```
3 + 5 * x
```

Grammar of arithmetic expressions

Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```

```
3 + 5 * x
```

Grammar of arithmetic expressions

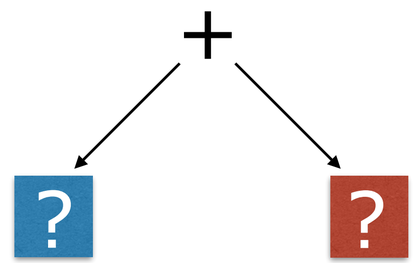
Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```



3 + 5 * x

Grammar of arithmetic expressions

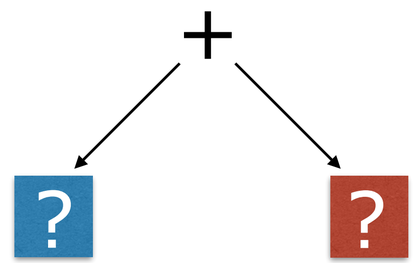
Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```



3 + 5 * x

Grammar of arithmetic expressions

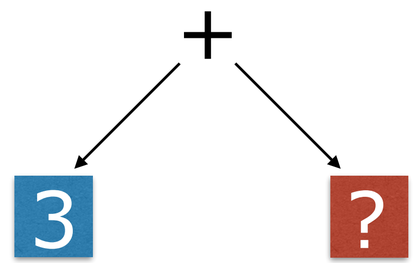
Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```



3 + 5 * x

Grammar of arithmetic expressions

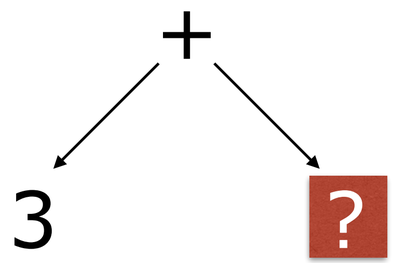
Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```



3 + 5 * x

Grammar of arithmetic expressions

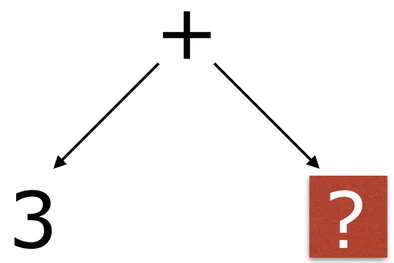
Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```




```
3 + 5 * x
```

Grammar of arithmetic expressions

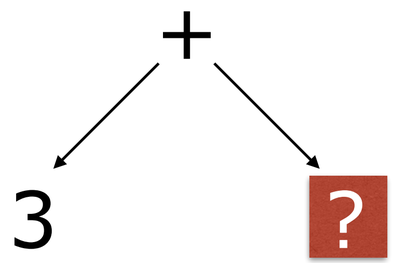
Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```




```
3 + 5 * x
```

Grammar of arithmetic expressions

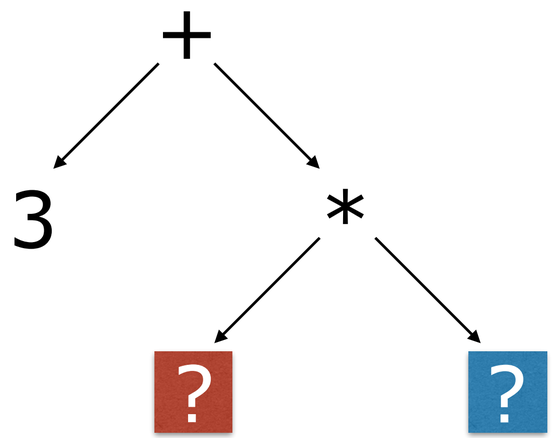
Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```



3 + 5 * x

Grammar of arithmetic expressions

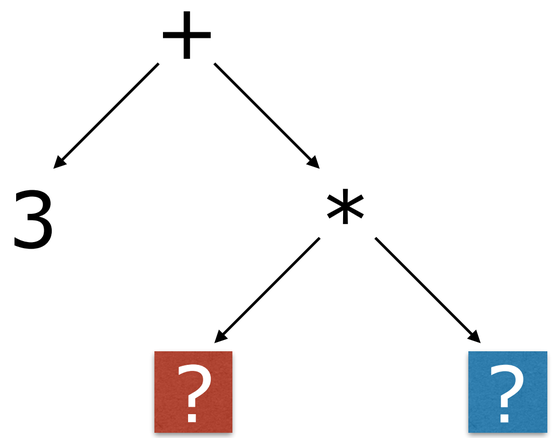
Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```



3 + 5 * x

Grammar of arithmetic expressions

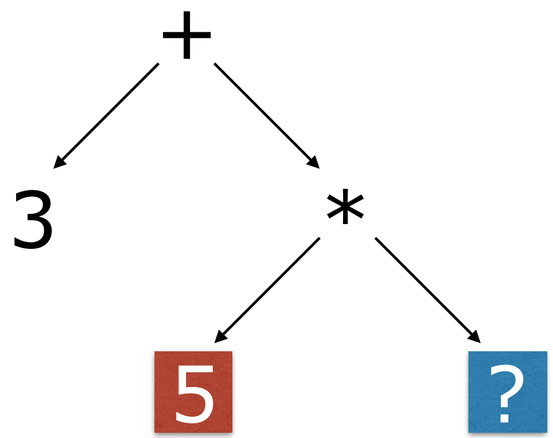
Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```



3 + 5 * x

Grammar of arithmetic expressions

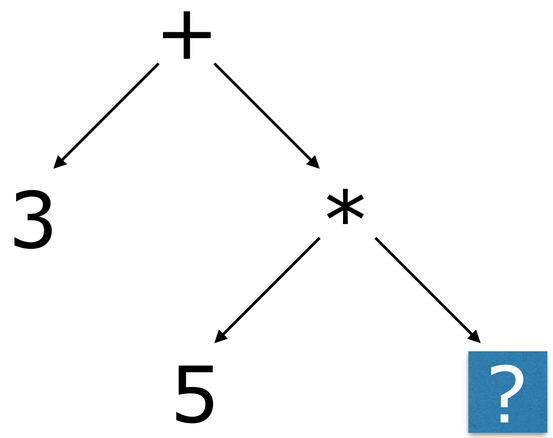
Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```




```
3 + 5 * x
```

Grammar of arithmetic expressions

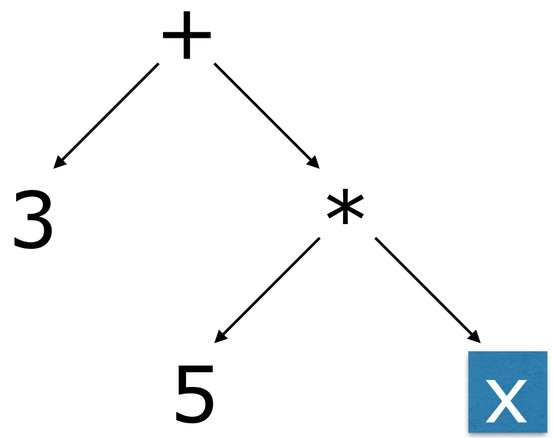
Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```




```
3 + 5 * x
```

Grammar of arithmetic expressions

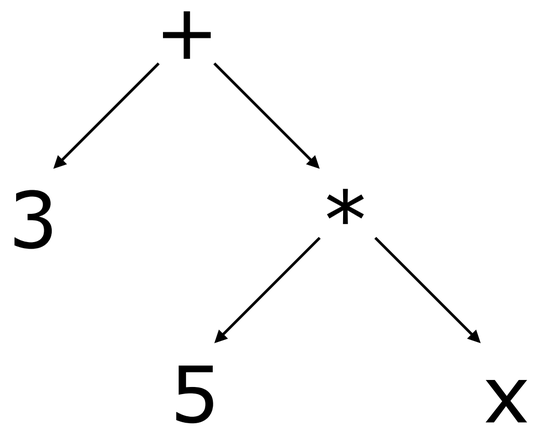
Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```




```
3 + 5 * x
```

Left-recursive grammars

Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```

- Note: To pick the right rules to use, we had to look ahead in the input.

Left-recursive grammars

Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```

- Note: To pick the right rules to use, we had to look ahead in the input.
- We want to create grammars so that the right rule can be picked by only reading the next character of the input.

Left-recursive grammars

Revised grammar:

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N | V | - A | ( T )
```

A **left-recursive** parser is recursive on the leftmost symbol

By just looking at the next symbol we cannot know which rule to pick

Remove left recursions

It is relatively straightforward to remove left recursions

Left-recursive grammar

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N   | V
    | - A | ( T )
```

Non-left-recursive grammar

```
N ::= <integers>
V ::= <variables>

T ::= P + T | P

P ::= A * P | A

A ::= N   | V
    | - A | ( T )
```

Remove left recursions

It is relatively straightforward to remove left recursions

Left-recursive grammar

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N   | V
    | - A | ( T )
```

Non-left-recursive grammar

```
N ::= <integers>
V ::= <variables>

T ::= P + T | P

P ::= A * P | A

A ::= N   | V
    | - A | ( T )
```

Remove left recursions

It is relatively straightforward to remove left recursions

Left-recursive grammar

```
N ::= <integers>
V
T
P
A
    | - A | ( T )
```

Non-left-recursive grammar

```
N ::= <integers>
V ::= <variables>

T ::= P + T | P

P ::= A * P | A

A ::= N   | V
    | - A | ( T )
```

We only need the next input symbol to choose a path through the grammar

Remove left recursions

It is relatively straightforward to remove left recursions

Left-recursive grammar

```
N ::= <integers>
V
T
P
A
    | - A | ( T )
```

We only need the next input symbol to choose a path through the grammar

Non-left-recursive grammar

```
N ::= <integers>
    | - A | ( T )
```

Moreover: left-recursive grammars have the problem that parsing might get stuck in an infinite loop!

Remove left recursions

It is relatively straightforward to remove left recursions

Left-recursive grammar

```
N ::= <integers>
V ::= <variables>

T ::= T + T | P

P ::= P * P | A

A ::= N   | V
    | - A | ( T )
```

Non-left-recursive grammar

```
N ::= <integers>
```

Moreover: left-recursive grammars have the problem that parsing might get stuck in an infinite loop!

```
    | - A | ( T )
```

Summary

- We use **context-free grammars** to describe languages and data formats
- **Abstract syntax trees** represent data described by a context-free grammar
- A **parser** turns unstructured data (i.e. text, binary) into abstract syntax trees
- We have to revise grammars in order to account for **operator precedence**
- We want to avoid **left-recursion**

Part II

Implementing a Parser

Implementing a parser

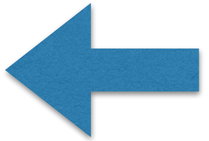
There are different ways to implement a parser

- knuckle down and implement it **from scratch**
- use a **parser generator**, which takes a grammar as input and generates the source code for a parser
- use a library for implementing a parser (called **parser combinator** library)

Implementing a parser

There are different ways to implement a parser

- knuckle down and implement it **from scratch**
- use a **parser generator**, which takes a grammar as input and generates the source code for a parser
- use a library for implementing a parser (called **parser combinator** library)



This is where FP shines

Parser combinators

- Small parsers for simple terms (e.g. integers)
- Combinators that compose them to larger parsers
- Parsers are first-class: can be passed as arguments
- FParsec: industry-grade parser combinator library

Parser combinators

- Small parsers for simple terms (e.g. integers)
- Combinators that compose them to larger parsers
- Parsers are first-class: can be passed as arguments
- FParsec: industry-grade parser combinator library

- FParsec is efficient but rather complicated.
- Instead, we look at a simpler library (courtesy of Jesper) that is largely compatible with FParsec

Plan for the rest of the lecture

Goal: Build a complete parser for the arithmetic expression language.

- **First:** Use parser combinators (as black boxes) to build the parser.
- **Afterwards:** Look into how these parser combinators are implemented.

Parser combinators

Parsers have the following type

```
Parser<'a>
```

Parsers turn (unstructured) text into (structured) data of type 'a

Parser combinators

Parsers have the following type

```
Parser<'a>
```

Output type of the parser

Parser combinators

Parsers have the following type

```
Parser<'a>
```

We can run a parser on a string:

```
run : Parser<'a> -> string -> 'a
```

(slightly simplified type)

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

- Goal: Write parsers for T, P and A
- All parsers have type Parser<aExp>

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

- Goal: Write parsers for T, P and A
- All parsers have type Parser<aExp>

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

- First step: Write parsers for the right-hand sides of the productions rules.

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

- Goal: Write parsers for T, P and A
- All parsers have type Parser<aExp>

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

- First step: Write parsers for the right-hand sides of the productions rules.
- Afterwards: Put it all together.

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
```

This is the parser for aExp of integers

It uses the following integer parser from the parser combinator library:

```
pint : Parser<int>
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
```

Mapping function for parsers

```
(|>>) : Parser<'a> -> ('a -> 'b)
        -> Parser<'b>
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
```

Mapping function for parsers

```
(|>>) : Parser<'a> -> ('a -> 'b)
        -> Parser<'b>
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

```
pint : Parser<int>

N : int -> aExp

pN : Parser<aExp>
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
let pV   = pid  |>> V
```

Parser for aExp of variables

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

```
pid : Parser<string>

V : string -> aExp

pV : Parser<aExp>
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
let pV   = pid  |>> V
```

Parser for aExp of variables

```
pid : Parser<string>

let pid = many1 letterChar |>> charListToStr
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
let pV   = pid  |>> V
```

Parser for aExp of variables

```
pid : Parser<string>

let pid = many1 letterChar |>> charListToStr
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

```
charListToStr : char list -> string

letterChar : Parser<char>

many1 : Parser<'a> -> Parser <'a list>
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
let pV   = pid  |>> V
```

Parser for aExp of variables

```
pid : Parser<string>

let pid = many1 letterChar |>> charListToStr
```

parses a letter

many1 p repeats p one or more times

```
type aExp =
| N of int
|
|
|
|
```

```
charListToStr : char list -> string

letterChar : Parser<char>

many1 : Parser<'a> -> Parser <'a list>
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = (pchar '-' >>. pA) |>> Neg
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

```
pchar : char -> Parser<char>

(>>.) : Parser<'a> ->
        Parser<'b> ->
        Parser<'b>

pA : Parser<aExp>
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = (pchar '-' >>. pA) |>> Neg
```

```
pchar : char -> Parser<char>

(>>.) : Parser<'a> ->
        Parser<'b> ->
        Parser<'b>

pA : Parser<aExp>
```

Assume for now that the parser for A is already defined.

```
t
|
|
|
|
|                    Exp
| Neg of aExp
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = (pchar '-' >>. spaces >>. pA) |>> Neg
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

```
pchar : char -> Parser<char>

(>>.) : Parser<'a> ->
        Parser<'b> ->
        Parser<'b>

pA : Parser<aExp>

spaces : Parser<char list>
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = (pchar '-' >>. spaces >>. pA) |>> Neg

let pPar = between (pchar '(') pT (pchar ')')
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

```
pT : Parser<aExp>

between : Parser <'a> ->
          Parser <'b> ->
          Parser <'c> ->
          Parser <'b>
between l p r = l >>. p .>> r
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = (pchar '-' >>. spaces >>. pA) |>> Neg

let pPar = between (pchar '(' >>. spaces) pT
                   (spaces >>. pchar ')')
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

```
pT : Parser<aExp>

between : Parser <'a> ->
          Parser <'b> ->
          Parser <'c> ->
          Parser <'b>
between l p r = l >>. p .>> r
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = (pchar '-' >>. spaces >>. pA) |>> Neg

let pPar = between (pchar '(' >>. spaces) pT
                   (spaces >>. pchar ')')

let pAdd = (pP .>> pchar '+' .>>. pT) |>> Add
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

```
(.>>.) : Parser<'a> ->
         Parser<'b> ->
         Parser<'a * 'b>
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = (pchar '-' >>. spaces >>. pA) |>> Neg

let pPar = between (pchar '(' >>. spaces) pT
                   (spaces >>. pchar ')')

let pAdd = (pP .>> (spaces >>. pchar '+' >>. spaces)
               .>>. pT) |>> Add
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

```
(.>>.) : Parser<'a> ->
         Parser<'b> ->
         Parser<'a * 'b>
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

```
let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = (pchar '-' >>. spaces >>. pA) |>> Neg

let pPar = between (pchar '(' >>. spaces) pT
                   (spaces >>. pchar ')')

let pAdd = (pP .>> (spaces >>. pchar '+' >>. spaces)
                .>>. pT) |>> Add

let pMul = (pA .>> (spaces >>. pchar '*' >>. spaces)
                .>>. pP) |>> Mul
```

Parser for arithmetic expressions

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
type aExp =
| N of int
| V of string
| Add of aExp * aExp
| Mul of aExp * aExp
| Neg of aExp
```

```
let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = (pchar '-' >>. spaces >>. pA) |>> Neg

let pPar = between (pchar '(' >>. spaces) pT
                   (spaces >>. pchar ')')

let pAdd = (pP .>> (spaces >>. pchar '+' >>. spaces)
               .>>. pT) |>> Add

let pMul = (pA .>> (spaces >>. pchar '*' >>. spaces)
               .>>. pP) |>> Mul
```

We are almost done.
What do we do about pA, pT, and pP?

Putting it all together

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let rec pT = (pP .>> (spaces .>>. pchar '+' .>>. spaces) .>>. pT |>> Add)
             <|> pP
and pP = ...

and pA = ...
```

```
(<|>) : Parser <'a> -> Parser <'a> -> Parser <'a>
```

Putting it all together

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

- F# will not accept this definition!
- Only functions may be defined recursively!
  (otherwise the right-hand is evaluated immediately)
- We have to use some trickery instead.

```
let rec pT = (pP .>> (spaces .>>. pchar '+' .>>. spaces) .>>. pT |>> Add)
             <|> pP
and pP = ...

and pA = ...
```

```
(<|>) : Parser <'a> -> Parser <'a> -> Parser <'a>
```

Recursive production rules

```
let pT, tref = createParserForwardedToRef<aExp>()

  …

do tref := (pP .>> (spaces >>. pchar '+' >>. spaces)
                   .>>. pT |>> Add) <|> pP
```

- We create a dummy value for pT and a reference to it:

```
createParserForwardedToRef : unit -> Parser<'a> * Parser<'a> ref
```

- pT can be used in declarations immediately
- but the reference **must** be set before the parser is used

The complete parser definition

```
let pT, tref = createParserForwardedToRef<aExp>()
let pP, pref = createParserForwardedToRef<aExp>()
let pA, aref = createParserForwardedToRef<aExp>()
```

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

The complete parser definition

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pT, tref = createParserForwardedToRef<aExp>()
let pP, pref = createParserForwardedToRef<aExp>()
let pA, aref = createParserForwardedToRef<aExp>()

let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = (pchar '-' >>. spaces >>. pA) |>> Neg

let pPar = between (pchar '(' >>. spaces) pT (spaces >>. pchar ')')
let pAdd = (pP .>> (spaces >>. pchar '+' >>. spaces).>>. pT) |>> Add
let pMul = (pA .>> (spaces >>. pchar '*' >>. spaces) >>. pP) |>> Mul
```

The complete parser definition

```
T ::= P + T | P

P ::= A * P | A

A ::= N | V
    | -A | ( T )
```

```
let pT, tref = createParserForwardedToRef<aExp>()
let pP, pref = createParserForwardedToRef<aExp>()
let pA, aref = createParserForwardedToRef<aExp>()

let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = (pchar '-' >>. spaces >>. pA) |>> Neg

let pPar = between (pchar '(' >>. spaces) pT (spaces >>. pchar ')')
let pAdd = (pP .>> (spaces >>. pchar '+' >>. spaces).>>. pT) |>> Add
let pMul = (pA .>> (spaces >>. pchar '*' >>. spaces) >>. pP) |>> Mul

do tref := pAdd <|> pP
do pref := pMul <|> pA
do aref := pNeg <|> pPar <|> pN <|> pV
```

Two improvements

```
let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = (pchar '-' >>. spaces >>. pA) |>> Neg

let pPar = between (pchar '(' >>. spaces) pT (spaces >>. pchar ')')
let pAdd = (pP .>> (spaces >>. pchar '+' >>. spaces).>>. pT) |>> Add
let pMul = (pA .>> (spaces >>. pchar '*' >>. spaces) >>. pP) |>> Mul
```

Two improvements

Use combinators for parsing binary/unary operators.
Implementing these combinators is part of the assignment

```
let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = (pchar '-' >>. spaces >>. pA) |>> Neg

let pPar = between (pchar '(' >>. spaces) pT (spaces >>. pchar ')')
let pAdd = (pP .>> (spaces >>. pchar '+' >>. spaces).>>. pT) |>> Add
let pMul = (pA .>> (spaces >>. pchar '*' >>. spaces) >>. pP) |>> Mul
```

Two improvements

Use combinators for parsing binary/unary operators.
Implementing these combinators is part of the assignment

```
let pN   = pint |>> N
let pV   = pid  |>> V

let pNeg = unop (pchar '-') pA |>> Neg

let pPar = between (pchar '(' >>. spaces) pT (spaces >>. pchar ')')
let pAdd = binop (pchar '+') pP pT |>> Add
let pMul = binop (pchar '*') pA pP |>> Mul
```

Two improvements

Use combinators for parsing binary/unary operators.
Implementing these combinators is part of the assignment

```
let pN   = pint |>> N <?> "integer"
let pV   = pid  |>> V <?> "variable"

let pNeg = unop (pchar '-') pA |>> Neg <?> "negated expression"

let pPar = between (pchar '(' >>. spaces) pT (spaces >>. pchar ')')
let pAdd = binop (pchar '+') pP pT |>> Add <?> "addition"
let pMul = binop (pchar '*') pA pP |>> Mul <?> "multiplication"
```

Attach labels to each parser to improve error messages:

```
<?> : Parser<'a> -> string -> Parser<'a>
```

Summary

- There are different approaches to writing parsers
- We used a **parser combinator** library
- A parser is built by writing **small parsers** for parts of the language
- Parsers are **combined** to form larger parsers **using parser combinators**
- In F# we have to be careful about **recursive parsers**

Part III

Parser Library

Implementing the Parser Library

```
type Parser<'a> = string -> Result<'a * string, ParseError>
```

```
type ParseError = string
```

```
type Result<'a, 'b> =
| Success of 'a
| Failure of 'b
```

Implementing the Parser Library


```
type ParseError = string
```

```
type Result<'a, 'b> =
| Success of 'a
| Failure of 'b
```

Implementing the Parser Library

input string

success

failure

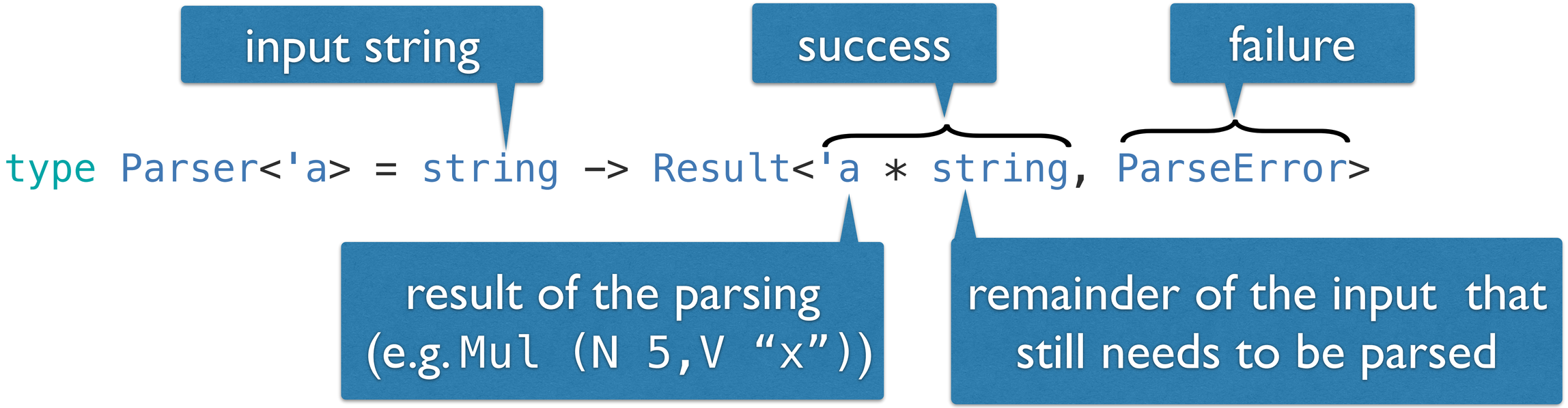
```
type Parser<'a> = string -> Result<'a * string, ParseError>
```

result of the parsing
(e.g. `Mul (N 5,V "x")`)

```
type ParseError = string
```

```
type Result<'a, 'b> =
| Success of 'a
| Failure of 'b
```

Implementing the Parser Library




```
type ParseError = string
```

```
type Result<'a, 'b> =
| Success of 'a
| Failure of 'b
```

Implementing the Parser Library

input string

success

failure

```
type Parser<'a> = string -> Result<'a * string, ParseError>
```

result of the parsing (e.g. `Mul (N 5,V "x")`)

remainder of the input that still needs to be parsed

```
type ParseError = string
```

```
type Result<'a, 'b> =
| Success of 'a
| Failure of 'b
```

NB: this is a slight simplification that does not support <?>

Implementing pchar

```
type Parser<'a> = string -> Result<'a * string, ParseError>



let pchar (ch : char) : Parser<char> = fun str ->
  if str.Length = 0 then
    Failure "No more input"
  else
    let first = str.[0]
    if first = ch then
      let remaining = str.[1..]
      Success (first,remaining)
    else
      Failure (sprintf "Expecting '%c'. Got '%c'" ch first)
```

Implementing .>>.

```
type Parser<'a> = string -> Result<'a * string, ParseError>

    (.>>.) : Parser<'a> -> Parser<'b> -> Parser<'a * 'b>


let (.>>.) p1 p2 = fun inp ->
  match p1 inp with
  | Failure err -> Failure err
  | Success (val1,rem1) ->
    match p2 rem1 with
    | Failure err -> Failure err
    | Success (val2,rem2) -> Success ((val1,val2),rem2)
```

Implementing |>>

```
type Parser<'a> = string -> Result<'a * string, ParseError>


(|>>) : Parser<'a> -> ('a -> 'b) -> Parser<'b>


let (|>>) p f = fun inp ->
  match p inp with
   | Success (value,rem) -> Success (f value, rem)
   | Failure err -> Failure err
```

Implementing >>. and .>>

```
type Parser<'a> = string -> Result<'a * string, ParseError>

(.>>.) : Parser<'a> -> Parser<'b> -> Parser<'a * 'b>

(.>>)  : Parser<'a> -> Parser<'b> -> Parser<'a>

(>>.)  : Parser<'a> -> Parser<'b> -> Parser<'b>

let (.>>) p1 p2 = (p1 .>>. p2) |>> fst

let (>>.) p1 p2 = (p1 .>>. p2) |>> snd
```

Implementing >>. and .>>

```
type Parser<'a> = string -> Result<'a * string, ParseError>
```

```
(<|>) : Parser<'a> -> Parser<'a> -> Parser<'a>
```

```
let (<|>) p1 p2 = fun inp ->
  match p1 inp with
   | Success result -> Success result
   | Failure err -> p2 inp
```

Implementing many

```
type Parser<'a> = string -> Result<'a * string, ParseError>

many : Parser<'a> -> Parser <'a list>

repeat : Parser<'a> -> string -> 'a list * string
```

helper function

```
let rec repeat p inp =
  match p inp with
  | Failure err -> ([], inp)
  | Success (value, rem1) ->
    let (values, rem2) = repeat p rem1
    (value::values, rem2)

let many p = fun inp -> Success (repeat p inp)
```

Implementing many1

```
type Parser<'a> = string -> Result<'a * string, ParseError>

many : Parser<'a> -> Parser <'a list>

many1 : Parser<'a> -> Parser <'a list>
```

similar to `many`, but with at least 1 repetition

```
let many1 p = (p .>>. many p) |>> (fun (v,vs) -> v :: vs)
```

Generalise pchar

```
pchar : char -> Parser<char>
```

Generalise pchar

```
pchar : char -> Parser<char>

let satisfy (p : char -> bool) : Parser<char> = fun str ->
  if str.Length = 0 then
    Failure "No more input"
  else
    let first = str.[0]
    if p first then
      let remaining = str.[1..]
      Success (first,remaining)
    else
      Failure (sprintf "Unexpected '%c'" first)
```

Generalise pchar

```
pchar : char -> Parser<char>

let satisfy (p : char -> bool) : Parser<char> = fun str ->
  if str.Length = 0 then
    Failure "No more input"
  else
    let first = str.[0]
    if p first then
      let remaining = str.[1..]
      Success (first,remaining)
    else
      Failure (sprintf "Unexpected '%c'" first)

let pchar c = satisfy (fun x -> x = c)
```

Refining the Parser type

```
type Parser<'a> = string -> Result<'a * string, ParseError>
```

Replace string type with this type

```
type TextInputState = {
    lines : string[]
    position : Position
}
```

- The parser will only change the position (more efficient)
- We can provide better error messages
  - We can give the position where the parser failed!

Implementing <?>

Our type definition do not support <?> yet.

```
type Parser<'a> = TextInputState ->
                  Result<'a * TextInputState, ParseError>

type ParseError = string
```

```
type Result<'a, 'b> =
    | Success of 'a
    | Failure of 'b
```

Implementing <?>

We extend the ParseError type with a `label` field

```
type Parser<'a> = TextInputState ->
                  Result<'a * TextInputState, ParseError>

type ParseError = {
    label : string
    input : TextInputState
    error : string
}

type Result<'a, 'b> =
    | Success of 'a
    | Failure of 'b
```

Implementing <?>

We extend the ParseError type with a `label` field

```
type Parser<'a> = TextInputState ->
                  Result<'a * TextInputState, ParseError>

type ParseError = {
    label : string
    input : TextInputState
    error : string
}

type Result<'a, 'b> =
    | Success of 'a
    | Failure of 'b
```

what did the parser expect (e.g. “integer”)

where did the parser fail

Implementing <?>

```
type Parser<'a> = TextInputState ->
                  Result<'a * TextInputState, ParseError>

type ParseError = {
    label : string
    input : TextInputState
    error : string }


(<?>) : Parser<'a> -> string -> Parser<'a>


let (<?>) p l = fun input ->
    match p input with
    | Success s  -> Success s
    | Failure pe -> Failure { pe with label = l}
```

Printing results

```
let printResult = function
      | Success (value,input) -> printfn "Success: %A" value
      | Failure err -> printfn "Error parsing %s at %A\n%s"
                               err.label err.input err.error
```

One last refinement

```
type Parser<'a> = {
  pfun  : TextInputState ->
          Result<'a * TextInputState, ParseError>
  label : string }
```

Label is attached to parser itself as well.

One last refinement

```
type Parser<'a> = {
  pfun  : TextInputState ->
          Result<'a * TextInputState, ParseError>
  label : string }
```

Label is attached to parser itself as well.

This allows us to update the label in the combinators, e.g.

```
let (.>>.) p1 p2 = {
    pfun = fun inp -> ... // defined as before
    label = "unknown" }
      <?> sprintf "(%s .>>. %s)" (p1.label) (p2.label)
```

Summary

- **A parser is a function** turning input into a **partial result + remaining input**
- **simple parsers** can be constructed by simply inspecting the input
- **parser combinators** take parsers as input and construct a new parser by distributing the input to the old parsers
- **error messages** can be adjusted by attaching **labels** to parsers