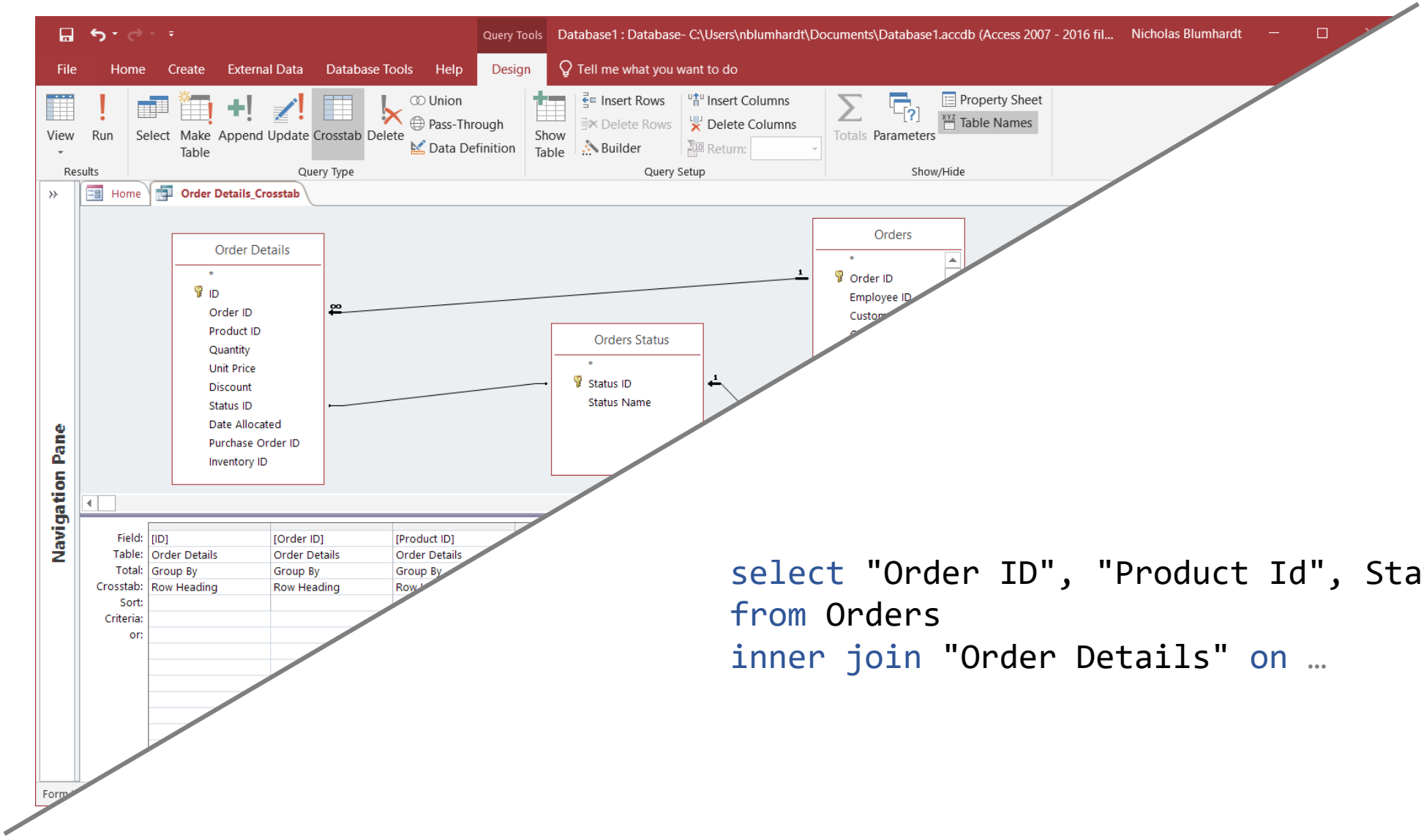
Parsing in C# from First Principles

Nicholas Blumhardt

@nblumhardt

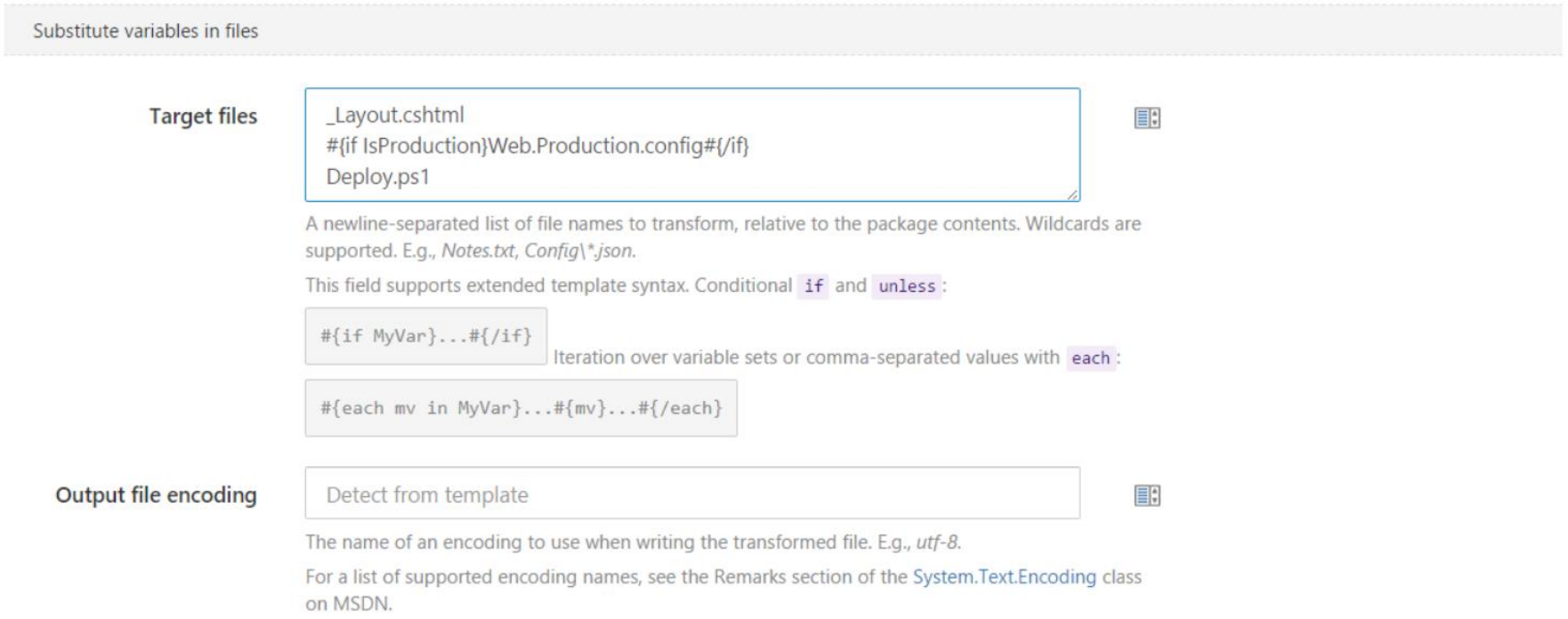
Why languages?

Advantages of expressing a problem in a domain-specific language




```
select "Order ID", "Product Id", Status
from Orders
inner join "Order Details" on …
```

Query Designer vs SQL / Different goals for different use cases


Substitute variables in files
Target files
_Layout.cshtml
#{if IsProduction}Web.Production.config#{/if}
Deploy.ps1
A newline-separated list of file names to transform, relative to the package contents. Wildcards are supported. E.g., Notes.txt, Config*.json.
This field supports extended template syntax. Conditional if and unless:
#{if MyVar}...#{/if}
Iteration over variable sets or comma-separated values with each:
#{each mv in MyVar}...#{mv}...#{/each}
Output file encoding
Detect from template
The name of an encoding to use when writing the transformed file. E.g., utf-8.
For a list of supported encoding names, see the Remarks section of the System.Text.Encoding class on MSDN.

Flexible, structured events — log file convenience.

Why Serilog?

Like many other libraries for .NET, Serilog provides diagnostic logging to files, the console, and **elsewhere**. It is easy to set up, has a clean API, and is portable between recent .NET platforms.

Unlike other logging libraries, Serilog is built with powerful *structured event data* in mind.

Text formatting with a twist

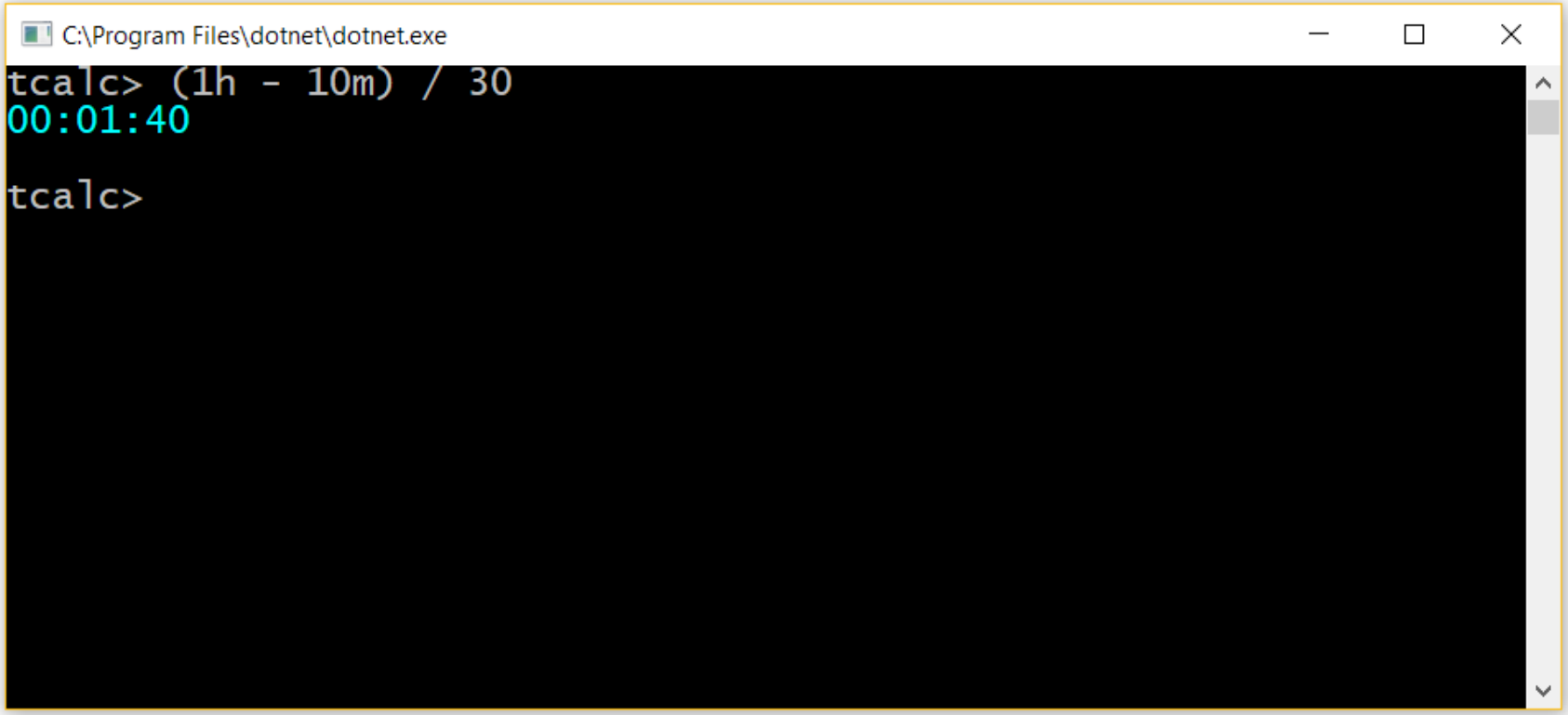
Serilog *message templates* are a simple DSL extending .NET format strings. Parameters can be named, and their values are serialized as properties on the event for incredible searching and sorting flexibility:

```
var position = new { Latitude = 25, Longitude = 134 };
var elapsedMs = 34;

log.Information("Processed {@Position} in {Elapsed:000} ms.", position, elapsedMs);
```

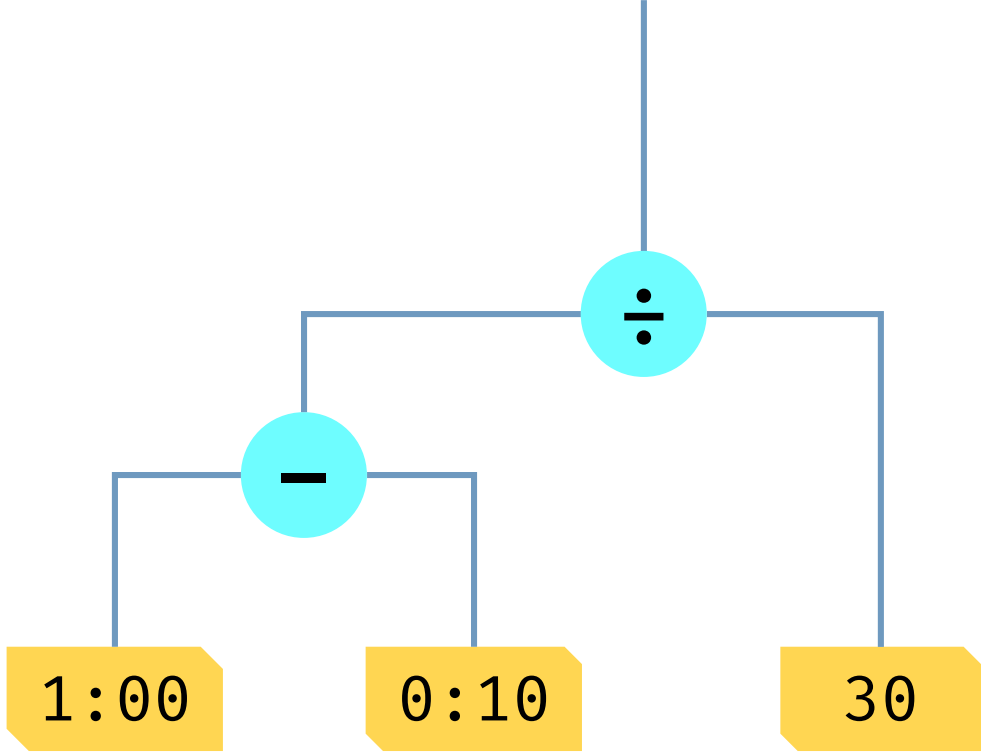
This example records two properties, `Position` and `Elapsed` along with the log event. The properties captured in the example, in JSON format, would appear like:

```
{"Position": {"Latitude": 25, "Longitude": 134}, "Elapsed": 34}
```

Serilog / Structured events from format strings



tcalc.exe / Evaluate expressions with numeric and duration values



(1h − 10m) / 30

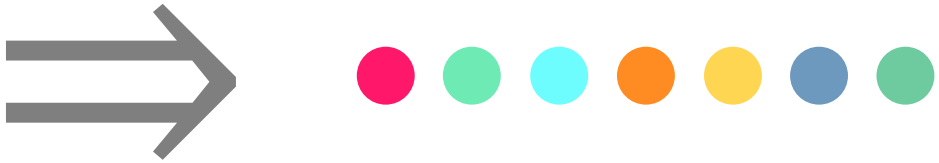
Evaluating Expressions

- Leaf nodes such as durations or numbers are easy to process
- Binary operations are evaluated by recursively evaluating their left and right sides, before dispatching

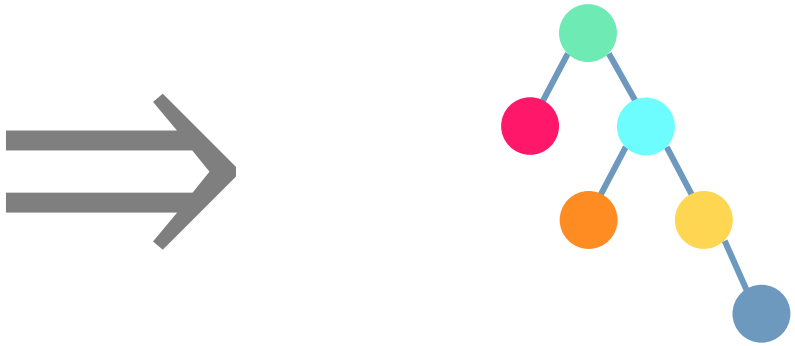
```
public static Result Evaluate(Expression expression)
{
    if (expression == null) throw new ArgumentNullException(nameof(expression));

    switch (expression)
    {
        case DurationValue duration:
            return new DurationResult(duration.Value);
        case NumericValue numeric:
            return new NumericResult(numeric.Value);
        case BinaryExpression binary:
            return DispatchOperator(
                Evaluate(binary.Left), Evaluate(binary.Right), binary.Operator);
        default:
            throw new ArgumentException($"Unsupported expression {expression}.");
    }
}

static Result DispatchOperator(Result left, Result right, Operator @operator)
{
    if (left == null) throw new ArgumentNullException(nameof(left));
    if (right == null) throw new ArgumentNullException(nameof(right));

    switch (@operator)
    {
        case Operator.Add:
            return DispatchAdd(left, right);
        case Operator.Subtract:
            return DispatchSubtract(left, right);
        case Operator.Multiply:
```

Character Sequences / If we could only pile these up in just the right way…

Regular Expressions

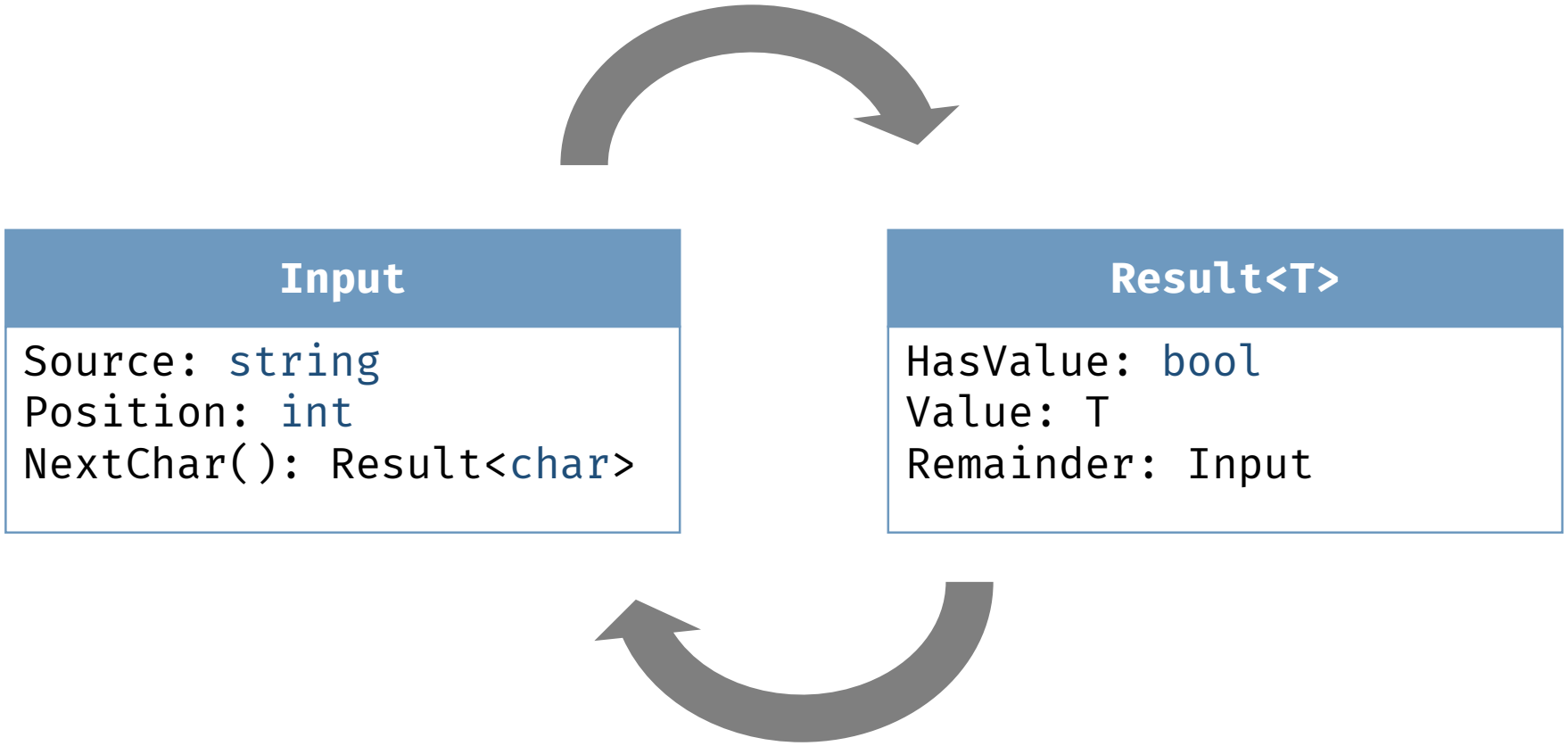


Parsers



A Parser is a Function

```
// Natural("123") -> 123

static int Natural(string input)
{
    // Magic goes here
}
```



Input and Result / The only two ingredients you need to create tasty parsers at home


```
1

1 * 10 + 2

(1 * 10 + 2) * 10 + 3
```

Parsing Natural Numbers

- If nothing can be parsed, no result is produced and the whole input remains un-parsed
- Keep moving forwards as characters are used
- Return the value, and whatever input remains un-parsed

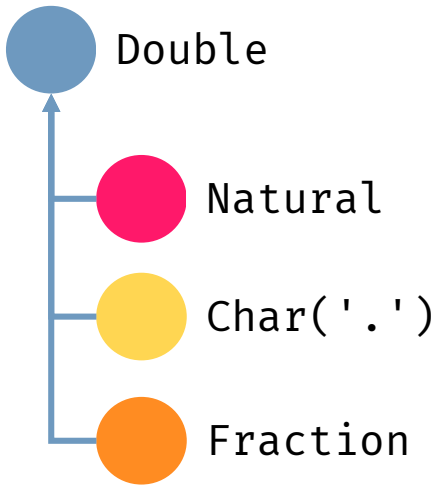
```
// Natural("123") → 123

static Result<int> Natural(Input input)
{
    var next = input.NextChar();
    if (!next.HasValue || !char.IsDigit(next.Value))
        return Result.Empty<int>(input);

    Input remainder;
    var val = 0;
    do
    {
        val = 10 * val +
            CharUnicodeInfo.GetDigitValue(next.Value);
        remainder = next.Remainder;
        next = remainder.NextChar();
    } while (next.HasValue && char.IsDigit(next.Value));

    return Result.Value(val, remainder);
}
```

Parsing a Single Character

65,536 parsers for the price of one!

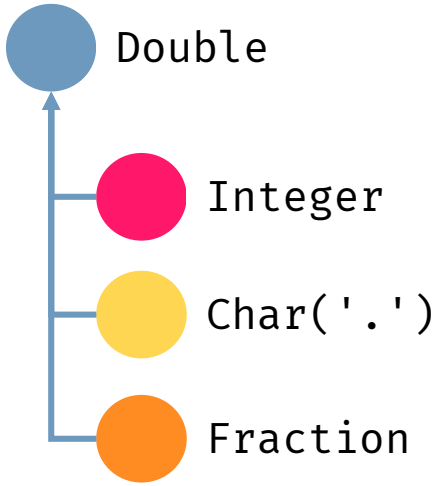
```
// Char("a", 'a') → 'a'
// Char("a", 'b') → Empty

static Result<char> Char(Input input, char c)
{
    var next = input.NextChar();
    if (!next.HasValue || next.Value != c)
        return Result.Empty<char>(input);

    return next;
}
```

Combining Parsers




```
// Double("123.456") → 123.456

static Result<double> Double(Input input)
{
    var whole = Natural(input);
    if (!whole.HasValue)
        return Result.Empty<double>(whole.Remainder);

    var dot = Char(whole.Remainder, '.');
    if (!dot.HasValue)
        return Result.Empty<double>(dot.Remainder);

    var fraction = Fraction(dot.Remainder);
    if (!fraction.HasValue)
        return Result.Empty<double>(fraction.Remainder);

    return Result.Value(whole.Value + fraction.Value,
                        fraction.Remainder);
}
```

Combining Parsers




```
// Double("123.456") → 123.456

static Result<double> Double(Input input)
{
    var whole = Integer(input);
    if (!whole.HasValue)
        return Result.Empty<double>(whole.Remainder);

    var dot = Char(whole.Remainder, '.');
    if (!dot.HasValue)
        return Result.Empty<double>(dot.Remainder);

    var fraction = Fraction(dot.Remainder);
    if (!fraction.HasValue)
        return Result.Empty<double>(fraction.Remainder);

    return Result.Value(whole.Value + fraction.Value,
                        fraction.Remainder);
}
```

Reset! / © Paul Goyette – CC BY-NC-SA 2.0
https://www.flickr.com/photos/pgoyette/110076905/

Parsers as Functions

`Parser<T>` encodes the common signature of parser functions

`Many()` converts a parser for an item, into a parser for a list of items

```
delegate Result<T> Parser<T>(Input input);

// Smiley(":-(") → '☹'
// Smiley.Many()(":-(:-)") → ['☹', '☺']

static Parser<List<T>> Many<T>(this Parser<T> item)
{
    return input =>
    {
        var many = new List<T>();
        var next = item(input);
        while (next.HasValue)
        {
            next.Add(itemResult.Value);
            next = item(next.Remainder);
        }
        return Result.Value(many, next.Remainder);
    };
}
```

Revised Char()

Instead of parsing the input directly, `Char()` now returns a parser that recognises the character

```
static Parser<char> Char(char c)
{
    return input =>
    {
        var next = input.NextChar();
        if (!next.HasValue || next.Value != c)
            return Result.Empty<char>(input);

        return next;
    };
}
```

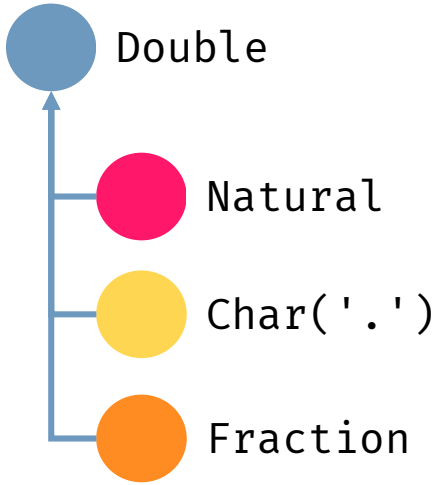
Then( ) and Return( )

Then( ) automates the sequencing of parsers

Return( ) always succeeds with a constant value

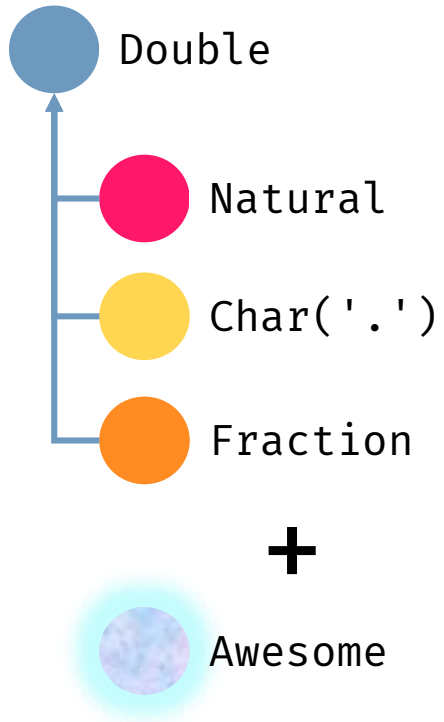
```
static Parser<U> Then<T, U>(
    this Parser<T> first,
    Func<T, Parser<U>> makeSecond)
{
    return input =>
    {
        var rf = first(input);
        if (!rf.HasValue)
            return Result.Empty<U>(rf.Remainder);

        return makeSecond(rf.Value)(rf.Remainder);
    };
}

static Parser<T> Return<T>(T value)
{
    return input => Result.Value(value, input);
}
```

Combining Parsers, Take Two!




```
// Double("123.456") -> 123.456

static readonly Parser<double> Double =
    Integer.Then(whole =>
        Char('.').Then(dot =>
            Fraction.Then(fraction =>
                Return(whole + fraction))));
```

Combining Parsers, LINQ Style…




```
// Double("123.456") -> 123.456

static readonly Parser<double> Double =
    from whole in Integer
    from _ in Char('.')
    from fraction in Fraction
    select whole + fraction;
```

```
Then(p)
Optional()
Value(v)
Or(p)
AtLeastOnce()
Many()
AtLeastOnceDelimitedBy(p)
```

```
ManyDelimitedBy(p)
Commented(p)
Token()
Log()
Positioned()
Repeat(n)
Where(r)
```

```
Except(p)
Named(n)
Cast()
Between(p, q)
AtEnd()
Lookahead()
…
```

Superpower / https://github.com/datalust/superpower

```
public static TextParser<TimeSpan> Magnitude { get; } =
    Character.EqualTo('d').Value(TimeSpan.FromDays(1))
        .Or(Character.EqualTo('h').Value(TimeSpan.FromHours(1)))
        .Or(Span.EqualTo("ms").Try().Value(TimeSpan.FromMilliseconds(1)))
        .Or(Character.EqualTo('m').Value(TimeSpan.FromMinutes(1)))
        .Or(Character.EqualTo('s').Value(TimeSpan.FromSeconds(1)));
```

(1h – 10m) / 30

```
public static TextParser<Expression> Duration { get; } =
    Numerics.DecimalDouble
        .Token()
        .Then(d => Magnitude.Select(m => m * d))
        .Select(ts => (Expression)new DurationValue(ts));
```

(1h – 10m) / 30

Duration / Composing `DecimalDouble` and `Magnitude`

Testing Parsers

Parsers are just code, so, their tests are just code, too

- `Duration` is the parser we're testing

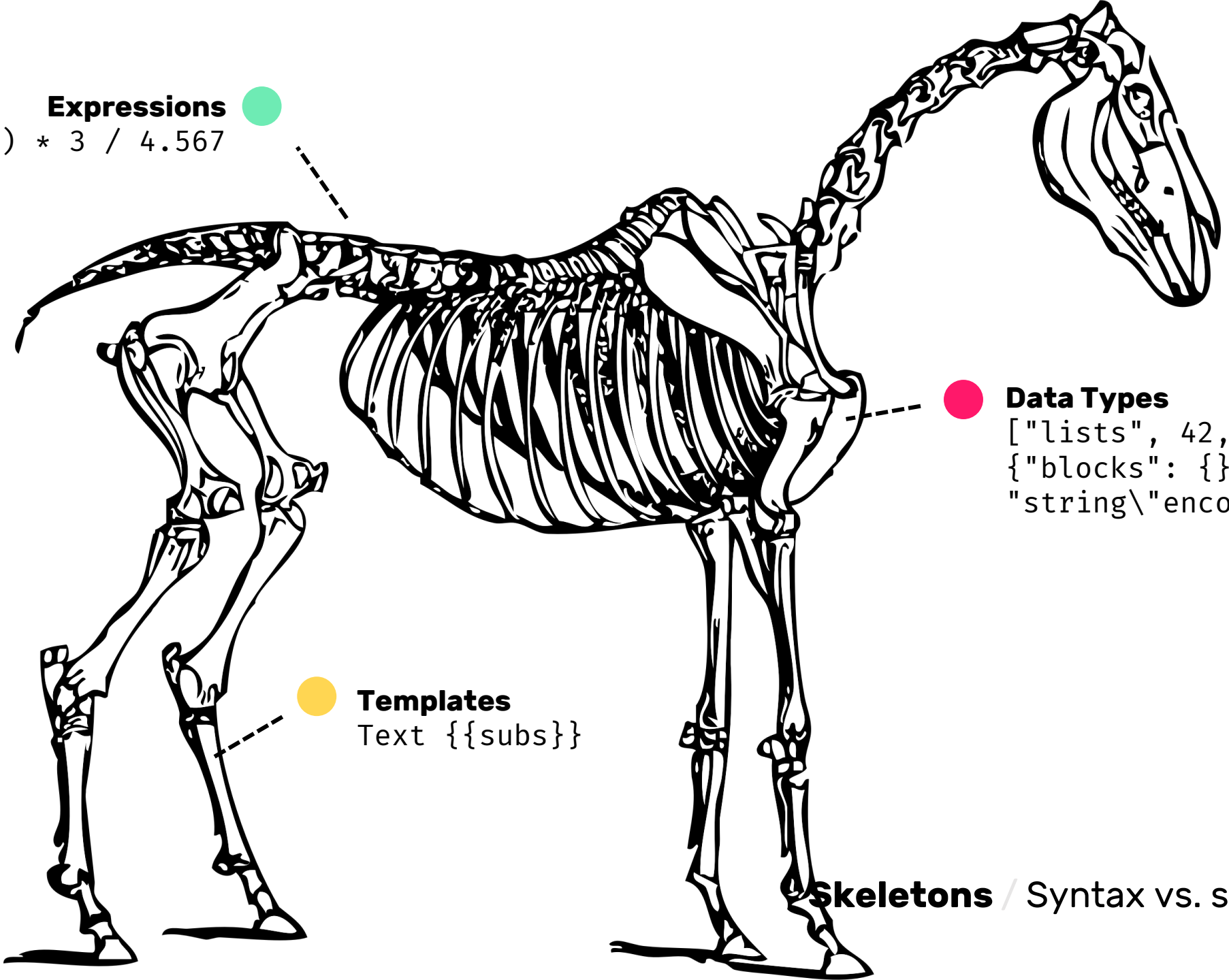
```
[Fact]
public void DurationsAreParsed()
{
    var ok = TestParser.TryParseAll(
        ExpressionParser.Duration,
        "150h",
        out var expr,
        out var err),

    Assert.True(ok, err);

    var duration =
        Assert.IsType<DurationValue>(expr);

    Assert.Equal(
        TimeSpan.FromHours(150),
        duration.Value);
}
```

```
public static TextParser<Operator> Op(char symbol, Operator op) =>
    Character.EqualTo(symbol)
        .Token()
        .Value(op);

public static TextParser<Operator> Add { get; } = Op('+', Operator.Add);
public static TextParser<Operator> Subtract { get; } = Op('-', Operator.Subtract)
public static TextParser<Operator> Multiply { get; } = Op('*', Operator.Multiply)
public static TextParser<Operator> Divide { get; } = Op('/', Operator.Divide);
```

(1h − 10m) / 30

```
public static TextParser<Expression> Literal { get; } =
    Duration.Try().Or(Number);

static TextParser<Expression> Factor { get; } =
    (from lparen in Character.EqualTo('(').Token()
     from expr in Parse.Ref(() => Expression)
     from rparen in Character.EqualTo(')').Token()
     select expr)
    .Or(Literal);

static TextParser<Expression> Term { get; } =
    Parse.Chain(Multiply.Or(Divide), Factor, BinaryExpression.Create);

static TextParser<Expression> Expression { get; } =
    Parse.Chain(Add.Or(Subtract), Term, BinaryExpression.Create);
```

Expression / The recursive machinery for parsing with operator precedence



Expressions
`(1 + 2) * 3 / 4.567`

Data Types
`["lists", 42, null]`
`{"blocks": {}}`
`"string\"encoding"`

Templates
`Text {{subs}}`

Skeletons / Syntax vs. semantics

```
static TextParser<Expression> Source { get; } = Expression.AtEnd();

public static bool TryParse(
       string input, out Expression expr, out string error)
{
    var result = Source(new TextSpan(input));
    if (result.HasValue)
    {
        expr = result.Value;
        error = null;
        return true;
    }
    expr = null;
    error = result.ToString();
    return false;
}
```

Source / Wiring up the `Expression` parser to handle complete expressions

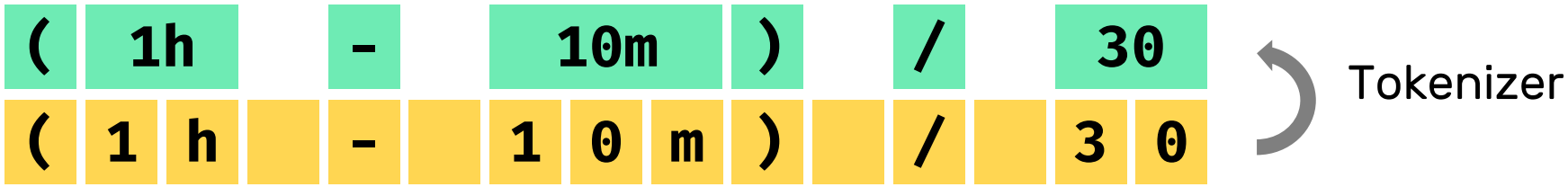
Demo

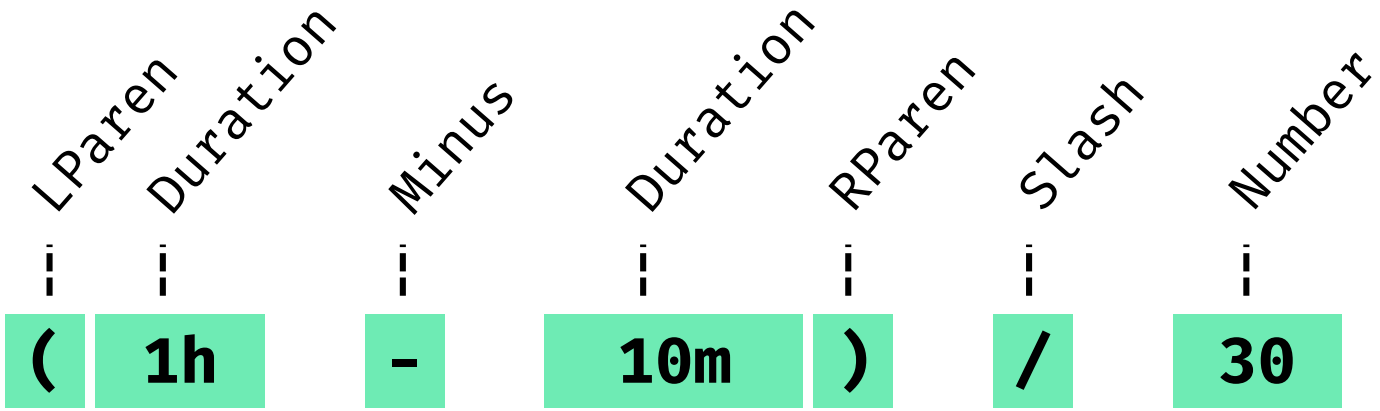
Let’s take a closer look at
how `tcalc.exe` behaves

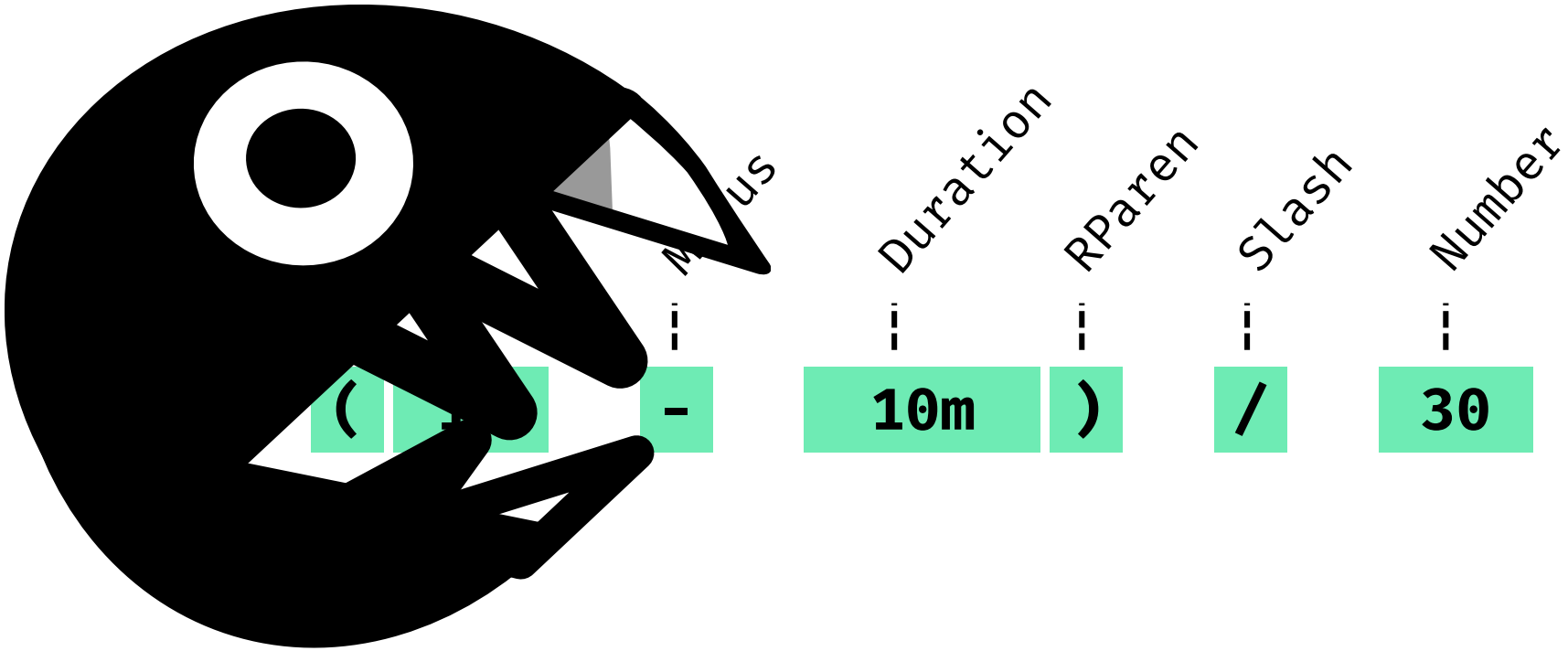
Token-driven Parsing

Working with higher-level lexical elements


(1 h - 10 m) / 30


(1h - 10m) / 30
(1 h - 1 0 m) / 3 0
Tokenizer


LParen
Duration
Minus
Duration
RParen
Slash
Number
(
1h
-
10m
)
/
30


Minus
Duration
RParen
Slash
Number
(
-
10m
)
/
30

`tcalc` Token Kinds

Every token identified in the source is tagged with a kind

- The `enum` member name is used in expectations and error messages
- Examples power more succinct expectations: `expected `+``

```
public enum ExpressionToken
{
    Number,
    Duration,

    [Token(Example = "+")]
    Plus,

    [Token(Example = "-")]
    Minus,

    [Token(Example = "*")]
    Asterisk,

    [Token(Example = "/")]
    Slash,

    [Token(Example = "(")]
```

```
var tokenizer = new TokenizerBuilder<ExpressionToken>()
    .Match(Character.EqualTo('+'), ExpressionToken.Plus)
    .Match(Character.EqualTo('-'), ExpressionToken.Minus)
    .Match(Character.EqualTo('*'), ExpressionToken.Asterisk)
    .Match(Character.EqualTo('/'), ExpressionToken.Slash)
    .Match(Duration, ExpressionToken.Duration, requireDelimiters: true)
    .Match(Numerics.Decimal, ExpressionToken.Number, requireDelimiters: true)
    .Match(Character.EqualTo('('), ExpressionToken.LParen)
    .Match(Character.EqualTo(')'), ExpressionToken.RParen)
    .Ignore(Span.WhiteSpace)
    .Build();
```

45d6

Tokenization in Superpower / `TokenizerBuilder` tries recognizers from top-to-bottom

```
public static TextParser<Expression> Number { get; } =
    Numerics.DecimalDouble
        .Token()
        .Select(num => (Expression)new NumericValue(num));
```

```
public static TokenListParser<ExpressionToken, Expression> Number { get; } =
    Token.EqualTo(ExpressionToken.Number)
        .Apply(Numerics.DecimalDouble)
        .Select(num => (Expression)new NumericValue(num));
```

```
static TextParser<Expression> Factor { get; } =
    (from lparen in Character.EqualTo('(').Token()
     from expr in Parse.Ref(() => Expression)
     from rparen in Character.EqualTo(')').Token()
     select expr)
    .Or(Literal);
```

```
static TokenListParser<ExpressionToken, Expression> Factor { get; } =
    (from lparen in Token.EqualTo(ExpressionToken.LParen)
     from expr in Parse.Ref(() => Expression)
     from rparen in Token.EqualTo(ExpressionToken.RParen)
     select expr)
    .Or(Literal);
```

- Demo

Adding support for
comments to `tcalc.exe`

Text parsers

\+ Easiest to get started with: only one level of abstraction

Developer-quality error reporting

\- More emergent complexity – especially around character-level ambiguities and backtracking

Token parsers

\+ Simpler and cleaner at each level of abstraction

End-user-quality error reporting

\- More machinery to understand (and teach!)

Languages rock

Parsers are programs

You can write programs

You can write parsers!

Even pretty good ones, if you're prepared to tinker for a while

So don't be afraid to give it a shot :-)

- Thank you!

https://github.com/nblumhardt/tcalc

 @nblumhardt

http://nblumhardt.com

Superpower
https://github.com/datalust/superpower

Sprache
https://github.com/sprache/Sprache

Pidgin
https://github.com/benjamin-hodgson/Pidgin

10m

Questions / AMA on Level 4 in the next session