DELF · EXAMPLE 06, PHASE 5

Finding the start



One search, four ways Delf writes it, and one it learned on the way

```
function Start(const Input: TGrid<Char>): Optional<TPoint>;
```

THE JOB

Start searches the grid row by row for the guard, ‘^’

`Input.Rows`

The rows, top to bottom. Each row knows its Index.

`Row.Find('^')`

An Optional<Integer>: where ‘^’ is in the row, if anywhere.

`Optional<TPoint>`

Present(the point) at the first hit. Absent if no row has one.

Every version that follows does exactly this. They differ in two choices: how the loop holds Row, and how it asks Find’s answer.

BEFORE OPTIONAL · PHASE 4

Delphi’s Try shape: a Boolean and an out parameter

```
// Delphi's Try shape, on purpose: Start is only meaningful when this returns True
function TryStart(const Input: TArray<string>; out Start: TPoint): Boolean;
begin
  for var y := 0 to High(Input) do
    if Input[y].Contains('^') then begin Start := Point(Input[y].IndexOf('^'), y); Exit( True ) end;
  Exit( False );
end;

// the caller
var GuardPos : TPoint;
if TryStart(Input, GuardPos) then Seen[GuardPos] := True
                             else begin writeln('no start point found'); Exit; end;
```

Two answers, two channels

Whether there is a start comes back as the result; the start itself through Start.

Undefined on False

GuardPos is declared, and readable, whatever TryStart answered. Only the comment says not to.

What Optional changes

One answer, Present(point) or Absent, and no point to read when there isn’t one.

BEFORE OPTIONAL · A VARIANT, FOR THE SLIDES ONLY

Search each row once, and the sentinel comes back

```
// Delphi's Try shape, on purpose: Start is only meaningful when this returns True
function TryStart(const Input: TArray<string>; out Start: TPoint): Boolean;
begin
  for var y := 0 to High(Input) do
  begin
    const x := Input[y].IndexOf('^');
    if x ≥ 0 then begin Start := Point(x, y); Exit( True ) end;
  end;
  Exit( False );
end;
```

One search per row

IndexOf once, kept in x, where phase 4 asks Contains and then IndexOf.

The loop loses one line

A declaration and a test need begin … end.

A sentinel, -1

x >= 0 tests for IndexOf’s “not found”. Contains hid it; Find, next, answers an Optional instead.

Delf · a variant of 06-guard-patrol/4-grid’s TryStart · not in a file, not compiled

TWO CHOICES, FOUR VERSIONS

Every combination compiles and gives the same answer

	Search with `match … | Absent : {skip}`	Search with `if … is Present(x) then`
Loop with `for var Row`	**A** In the file today	**C** Narrowing, new on 24 Oct
Loop with `for const Row`	**B** Row can’t change	**D** Both

VERSION A · IN THE FILE TODAY

A match, with an Absent arm that does nothing

Loop: for var Row

Search: match, empty Absent arm

```
function Start(const Input: TGrid<Char>): Optional<TPoint>;
begin
  for var Row in Input.Rows do
    match Row.Find('^') of
      | Present(x) : Exit( Present(Point(x, Row.Index)) )
      | Absent     : {skip}
    end;
  Exit( Absent );
end;
```

SPEC §4.5a calls an empty arm on an Optional a smell. The {skip} comment is there to say the loop goes on; delfc drops it in the output.

VERSION B

The same match, and a loop variable that can’t change

Loop: for const Row

Search: match, empty Absent arm

```
function Start(const Input: TGrid<Char>): Optional<TPoint>;
begin
  for const Row in Input.Rows do
    match Row.Find('^') of
      | Present(x) : Exit( Present(Point(x, Row.Index)) )
      | Absent     : {skip}
    end;
  Exit( Absent );
end;
```

Delphi has no for const (E2029, experiment 25), so delfc writes for var Row {const}. The intent reaches the output only as a comment.

VERSION C · NARROWING

Ask only about the case you care about

Loop: for var Row

Search: if … is Present(x)

```
function Start(const Input: TGrid<Char>): Optional<TPoint>;
begin
  for var Row in Input.Rows do
    if Row.Find('^') is Present(x) then Exit( Present(Point(x, Row.Index)) );
  Exit( Absent );
end;
```

if S is P then A means exactly match S of | P : A | _ : end. There is no Absent arm to leave empty, and x is in scope only after then.

VERSION D · BOTH

Narrowing, and a loop variable that can’t change

```
function Start(const Input: TGrid<Char>): Optional<TPoint>;
begin
  for const Row in Input.Rows do
    if Row.Find('^') is Present(x) then Exit( Present(Point(x, Row.Index)) );
  Exit( Absent );
end;
```

The loop’s body goes from four lines to one. Nothing in it can change Row, x or what Find answered.

VERSION D, WITHOUT PRESENT · NOT DELF

The same, but Exit with the bare point

```
function Start(const Input: TGrid<Char>): Optional<TPoint>;
begin
  for const Row in Input.Rows do
    if Row.Find('^') is Present(x) then Exit( Point(x, Row.Index) );
  Exit( Absent );
end;
```

A TPoint, not an Optional

The signature promises an Optional<TPoint>. This Exit hands back a TPoint.

SPEC §8

No implicit conversions: nothing converts silently, so this isn't Delf as decided.

delfc lets it through

It copies the line unchanged, and dcc32 catches it: E2010 Incompatible types: 'Optional<TPoint>' and 'System.Types.TPoint'.

A DESIGN CHOICE

Write Present, or let the signature supply it?

```
function Start(…): Optional<TPoint>;
  … Exit( Present(Point(x, Row.Index)) );   // Delf: say it
  … Exit( Point(x, Row.Index) );          // the signature could imply it
```

The two exits match

Absent has to be written anyway. With Present, each Exit names its arm, and the line reads the same wherever it is moved.

Nesting is where wrapping guesses

Optional<Optional<T>> keeps both layers (user, 19 Oct). Returning an Optional<T> there: wrap it, or pass it as is? Written out, there is no guess.

Other languages split

Swift and Kotlin wrap a plain value silently; Rust, Haskell, OCaml and Scala make you write Some or Just.

The cost is one word

The filter: safety without red tape. Present is a word at the one place a value becomes optional; if it recurs, a lift (next) removes it.

ELSEWHERE · FROM MEMORY, TO BE CHECKED

How other languages say “maybe a point”

Language	Absence is	A value goes in	Asking
Delphi	nil, -1, False + out	as is	if TryStart(I, P) then
Go	a zero value + ok	as is	if p, ok := start(g); ok
Java	Optional<T>, a library class; null remains	Optional.of(p)	o.ifPresent(p -> …)
C#	T?, null; checked by warnings	silently	if (s is Point p)
TypeScript	T | undefined, a union	silently	if (s !== undefined)
Kotlin	T?, null, in the type	silently	if (s != null) s?.x
Swift	Optional<T>, an enum; T?	silently	if let p = start(g)
Rust	Option<T>, an enum; no null	Some(p)	if let Some(p) = start(g)
Haskell	Maybe a; no null	Just p	case start g of Just p -> …
Delf	Optional<T>, a sum	Present(p)	if Start(G) is Present(P)

Go’s comma-ok is Delphi’s Try shape. Delf’s narrowing is Rust’s and Swift’s if let. The split on the third column is slide 11’s question.

ABSENT IS KINDA LIKE NIL

Every Delphi class reference is already an optional, unannounced

```
// Delphi: the type doesn't say
function FindGuard: TGuard;
…
const G := FindGuard;
if Assigned(G) then G.Patrol;
// forget the test: an access violation
```

```
// Delf: the type says it
function FindGuard: Optional<TGuard>;
…
if FindGuard is Present(G) then G.Patrol;
// no G to use without the test
```

One nil, three meanings

Not made yet, freed (FreeAndNil), or absent by design. Only the last is what Absent means.

nil isn’t even the worst

A freed object that wasn’t nilled still passes Assigned. Experiment 16: FreeAndNil through a getter dangles.

Optional<TGuard> isn’t a TGuard

No Free, no Assigned, no = nil (SPEC §5). Present(nil) and Absent must stay different.

The open half: non-nil objects

If a TGuard could never be nil, Optional would carry all the absence. A type to invent (concepts map, ◯).

TEACHING POINTS · LIFTING

Present lifts a value; Map lifts a function

A value

`Present : (T → Optional<T>)`

Takes a point into the world where it might not be. Haskell calls it pure (or Just); every such world has one.

A function

`Map : (T → U) to (Optional<T> → Optional<U>)`

The function never sees Absent. Map passes it along.

```
// x → a point, lifted: no is, no Present, no Absent
Row.Find('^').Map(λx → Point(x, Row.Index))   // Optional<TPoint>
```

- Point(x, Row.Index) is a plain function of x. Lifted, it works on Find's answer directly.
- That leaves the search: the first row whose answer is Present. That's FindMap (slide 19).
- The laws: Map(Id) changes nothing; Map(f) then Map(g) is Map(g ∘ f). paskell.md has them.

```
function Start(const Input: TGrid<Char>): Optional<TPoint>;
begin
  for var Row {const} in Input.Rows do
    begin
      var __dpp_1 := Row.Find('^');
      case __dpp_1.Kind of
        TOptionalKind.Present :
          begin
            const x = __dpp_1.Value;
            Exit( Optional<TPoint>.Present(Point(x, Row.Index)) );
          end;
      else
        ;
      end
    end;
  Exit( Optional<TPoint>.Absent );
end;
```

The usual case, one line of Delf becoming twelve

Find’s answer is held once, in a temporary.

x becomes a constant at the top of the arm.

Present and Absent are written in full from the expected type.

C’s output is the same but for the loop header: for var Row.

delfc output, unedited · 06-guard-patrol/5-grid-optional

WHERE THE OUTPUT DIFFERS

Only in the else: who catches a tag that is neither arm

A and B

Both arms covered; a corrupt tag raises

```
  TOptionalKind.Absent  : ;
else
  __dpp_Unreachable('TOptionalKind', Ord(__dpp_1.Kind));
end
```

C and D

The _ arm takes the rest; a corrupt tag is skipped

```
else
  ;
end
```

The tail of each case, indentation trimmed · observation by Claude, not yet reviewed

A, B, C, D ON DCC32 AND DCC64

Four versions, one answer

```
start: (4, 6)
turns: 10
visited: 41
```

Each one compiled with no warnings on Win32 and Win64 (Delphi 13), and printed the same three lines on both.

VERSION D, WITHOUT PRESENT · NOT DELF

The same, but Exit with the bare point

```
function Start(const Input: TGrid<Char>): Optional<TPoint>;
begin
  for const Row in Input.Rows do
    if Row.Find('^') is Present(x) then Exit( Point(x, Row.Index) );
  Exit( Absent );
end;
```

A TPoint, not an Optional

The signature promises an Optional<TPoint>. This Exit hands back a TPoint.

SPEC §8

No implicit conversions: nothing converts silently, so this isn't Delf as decided.

delfc lets it through

It copies the line unchanged, and dcc32 catches it: E2010 Incompatible types: 'Optional<TPoint>' and 'System.Types.TPoint'.

BEYOND THE FOUR · PROPOSED BY CLAUDE, IN DELFC SINCE 25 OCT

Write the search once, and Start has no loop at all

```
function Start(const Input: TGrid<Char>): Optional<TPoint>;
begin
  Exit( Input.Rows.FindMap(λRow → Row.Find('^')
                                  .Map(λx → Point(x, Row.Index))
                          ) );
end;
```

```
function FindMap<T, U>(const Xs: TArray<T>; const F: (T → Optional<U>)): Optional<U>;
begin
  for const X in Xs do
    if F(X) is Present(Y) then Exit( Present(Y) );
  Exit( Absent );
end;
```

No loop, no match, no Absent in Start. The empty branch doesn’t vanish: it moves into FindMap, written once for every search.

FindMap, after Rust’s find_map (from memory, unchecked) · a sketch for review

BEYOND THE FOUR · THE SAME SHAPE, IN DELPHI'S WORDS

Slide 19 with anonymous methods for its lambdas

```
function Start(const Input: TGrid<Char>): Optional<TPoint>;
begin
  Exit( Input.Rows.FindMap(
                           function(Row: TGridRow<Char>): Optional<TPoint>
                           begin
                             Result := Row.Find('^')
                                         .Map(
                                              function(x: Integer): TPoint
                                              begin
                                                Result := Point(x, Row.Index)
                                              end
                                             )
                           end
                          )
      );
end;
```

The dot chain is slide 19's; each λ is spelled out with its types. Not Delphi as it stands: an array has no FindMap and Optional no method Map. The next slide is what delfc writes.

The user's layout · slide 19's colours · not compiled

BEYOND THE FOUR · THE SAME CALL, IN DELPHI

What delfc writes for slide 19’s Start

```
function Start(const Input: TGrid<Char>): Optional<TPoint>;
begin
  Exit( __dpp_Free.FindMap<TGridRow<Char>, TPoint>(Input.Rows,
          function(Row: TGridRow<Char>): Optional<TPoint>
          begin
            Result := TOptional.Map<Integer, TPoint>(Row.Find('^'),
              function(x: Integer): TPoint
              begin
                Result := Point(x, Row.Index);
              end);
          end) );
end;
```

Every type written

Two pairs of type arguments and both methods’ signatures. dcc32 infers none of them through an anonymous method (experiment 42).

Inside out

No dot chain: FindMap and Map are plain calls, the receiver their first argument. Read from the middle.

FindMap in a class

Delphi has no generic free functions, so delfc puts them in a carrier, __dpp_Free.

delfc’s output, laid out by hand · 🧪 dcc32 and dcc64: start (4, 6), turns 10, visited 41, no warnings

BEYOND THE FOUR · THE TYPES IN SLIDE 19'S CALL

Find, Map, and what they make together

```
// Find, on each row of the grid
Row.Find : (Char → Optional<Integer>)   // Row is a TGridRow<Char>
function TGridRow<T>.Find(const Value: T): Optional<Integer>;
```

```
// Map, the prelude's: O.Map(F) is Map(O, F)
Map : ((Optional<T>, (T → U)) → Optional<U>)
function Map<T, U>(const O: Optional<T>; const F: (T → U)): Optional<U>;
```

```
Row.Find('^')                  : Optional<Integer>    // T is Integer
λx → Point(x, Row.Index)       : (Integer → TPoint)   // U is TPoint
Map(Row.Find('^'), λx → …)     : Optional<TPoint>
```

The receiver gives T, the lambda's body gives U: what delfc writes on slide 21, TOptional.Map<Integer, TPoint>.

Demo.OptionalGrids' Find · the prelude's Map · Delf's arrow notation (SPEC §6)

UFCS · UNIFORM FUNCTION CALL SYNTAX

Helpers without the helper

Delf the dot means: the left is the first argument

```
function FindMap<T, U>(const Xs: TArray<T>;
  const F: (T → Optional<U>)): Optional<U>;

// one function, two spellings
FindMap(Input.Rows, λRow → …)
Input.Rows.FindMap(λRow → …)
```

Delphi a helper, and only one

```
type
  TMyHelper = record helper for string
    function IsPostcode: Boolean;
  end;

S.Trim.IsPostcode
// E2003 Undeclared identifier: 'Trim'
```

One helper per type

The nearest wins, and yours hides the RTL's. In 15,273 corpus files, one helper for string, in a demo.

A real method wins

A free function is used only where the type has no such method, so nothing that compiles today changes meaning.

The door found one

delfc's own TPoint helper hid the RTL's Offset, which SynEdit calls. Renamed Delta.

SPEC §6 · docs/ufcs.md · 🧪 E2003 on dcc32, 25 Oct · experiment 32

WHERE DELFC STOPS TODAY

Writing the types out only moves the error inward

Written in Start	delfc said
Input.Rows.FindMap(λRow -> …)	cannot infer U for FindMap
FindMap<TGridRow<Char>, TPoint>(…)	cannot infer U for Map
… .Map<TPoint>(λx -> …)	cannot tell this lambda’s result type

The expected type

Exit(…) expects an Optional<TPoint>, which fixes U. delfc doesn’t use it.

The RTL’s result types

delfc can’t see that Point(…) answers a TPoint.

Probably a bug

It accepted Map<TPoint>, one of Map’s two type arguments, and wrote it out so.

delfc on a scratch copy, 24 Oct · not compiled, since it doesn’t transpile

BEYOND START · SIDE STEP 5B

Inline, the lambda takes its type from Map

```
// 5: named, so annotated
const CharToCell := λ(Ch: Char) → if Ch = '#' then Wall else Open;
const Grid  := Input.Map(CharToCell);

// 5b: inline, and Grid and Seen read the same way
const Grid  := Input.Map(λch → if ch = '#' then Wall else Open);
var   Seen  := Input.Map(λch → False);
```

Named, unannotated: an error

const CharToCell := λch -> … gets “no expected function type for this lambda”. Nothing beside it says what ch is.

What delfc writes for 5b

```
Input.Map<TCell>(function(ch: Char): TCell begin
Result := …; end)
```

APPENDIX

if let, guard let, and ?

How Swift and Rust test, bind and unwrap, and where each one meets Delf

Swift and Rust from Claude’s memory, unchecked against their references

APPENDIX · IF LET

Test, bind and use, in one step

Swift

```
if let p = start(grid) {
  print(p)  // a Point, not a Point?
} else {
  print("no start")
}
```

Rust

```
if let Some(p) = start(&grid) {
  println!("{p:?}");
} else {
  println!("no start");
}
```

Delf

```
if Start(Input) is Present(P)
  then writeln(P.ToString)
  else writeln('no start');
```

Rust: a match with a _ arm

Any pattern can follow let. if let P = e { A } else { B } means match e { P => A, _ => B }: Delf's definition of narrowing, word for word (SPEC §5).

Which way round

Swift and Rust put the pattern first, let P = e. Delf and C# put the value first, e is P, which reads like Delphi's own O is TObject, so Delf reuses is.

APPENDIX · FOUR FORMS

Where each one meets Delf

Swift / Rust	What it does	Delf today
if let Some(p) = e	test, bind, use inside	if e is Present(P) then · rung 1, in delfc
while let Some(x) = s.pop()	loop while it matches	while … is … · rung 2, an error today
if let a = x, a > 0 if let Some(a) = x && a > 0	the binding in the rest of the condition	is Present(C) and (C = Alive) · rung 2, todo-naming.md
guard let p = e else { return } let Some(p) = e else { return };	bind for the rest of the scope; the else must leave	the arm-declared variable (SPEC §4.3), below

```
// phase 5's main block: the arm with the point declares GuardPos; every other arm must leave
match Start(Input) of
  | Present(Pos) : var GuardPos := Pos
  | Absent       : begin writeln('no start point found'); Exit; end
end;
Seen[GuardPos] := True;  // a TPoint from here on
```

Phase 5 reached guard let's rule on its own (user, 23 Oct). Theirs takes one line; Delf's takes a two-arm match.

APPENDIX · RUST'S ?

Unwrap, or leave: a plain value past the guard point

Rust

```
fn first_step(g: &Grid) → Option<Point> {
  let p = start(g)?;  // None: return None
  Some(p + UP)
}
```

Delf, if it had ? (🟡 SPEC §12)

```
function FirstStep(…): Optional<TPoint>;
begin
  const Pos := Start(Input)?;
  Exit( Present(Pos + Step(Up)) );
end;
```

Delf today: the same, with the arm-declared variable

```
match Start(Input) of
  | Present(P) : var Pos := P
  | Absent     : Exit( Absent )
end;
Exit( Present(Pos + Step(Up)) );
```

? is a guard let

whose else is fixed: return the absence. One character, on the line it guards.

Only in its own world

The function must return an Optional (or a Result). A main block uses the match.

Bind, as a statement

One step per line, no nesting: the end of 03's guard staircase (endnote 1).

APPENDIX · ? ON AN ARDUINO UNO

Where ? wouldn’t go, and the function that lets it

```
// one character: look it up, or give up on it
fn flash_char(led: &mut Led, ch: char) → Option<()> {
  let (_, code) = morse::MORSE_CODE.iter()
                  .find(|(m, _)| *m == ch)?;  // not there: None
  for blip in code.chars() {  // code: a plain &str now
    let delay = match blip { '.' ⇒ 200, '-' ⇒ 500, _ ⇒ 0 };
    led.set_high(); arduino_hal::delay_ms(delay);
    led.set_low();  arduino_hal::delay_ms(delay);
  }
  Some(())
}

fn main() → ! {
  let dp   = arduino_hal::Peripherals::take().unwrap();
  let pins = arduino_hal::pins!(dp);
  let mut led = pins.d13.into_output();
  for ch in "HELLO WORLD !".chars() {
    let _ = flash_char(&mut led, ch);  // skip what the table lacks
  }
  loop {}
}
```

The original: if let around it all

In main.rs the whole body sat inside if let Some(s) = dict_entry { … }, after a first try with is_some() and unwrap().

Why ? wouldn’t go

main returns !, never, so there is no None to return. And in the loop, ? leaves the function, not the character.

So: a function in Option’s world

flash_char returns Option<()>. One ? on the lookup, and code is a verified &str for the rest of the body.

At the top, unwrap halts

take() is an Option too. In main, unwrap panics, and panic_halt makes that a halt: Delf’s top level would Halt.

APPENDIX · A FUNCTION, OR ITS CALL

A bare name: the function, or a call?

```
// Delf
function Answer: Integer;

var N    := Answer;                    // a call: 42
var Get  : (() → Integer) := Answer;   // the function
Twice(Answer)                         // the function
Call(Answer)                          // T from Answer

var Make : (() → TStringList) := TStringList.Create;
var Made := λ → TStringList.Create;    // nothing expected
// both out as an anonymous method (dcc32
// refuses the bare constructor, E2010):
//  function: TStringList
//    begin Result := TStringList.Create; end
```

```
// Rust
fn answer() → i32 { 42 }

let n    = answer();          // a call: () always
let get  = answer;            // the function
twice(answer)                 // the function
let make = Vec::<i32>::new;   // a constructor is one too

// from the user's main.rs:
// for ch in out_string do ;  = Pascal
for ch in out_string.chars() {
  …
  led.set_high();  // () with nothing to pass
}
```

Pascal: brackets optional

A parameterless call needs none, so a bare name means two things. dcc32 decides by what is expected: a function type takes the function, for every routine but a constructor.

Rust: brackets always

A bare name is never a call: answer is the function, answer() the call. No ambiguity, and () on every call: .chars(), .iter(), .unwrap().

Delf: Pascal's, made whole

Where a function is expected, the bare name is the function, constructors too; a call there is Answer(). Anywhere else, a call. With nothing expected, say it in code, λ ->: a type hint could, and won't (next slide).

Appendix · Delf: function_values.delf and experiment 46, on dcc32 and dcc64 · Rust from general knowledge, not compiled · main.rs: D:\Rust\rs-arduino

APPENDIX · TYPE HINTS

Type hints: Delf's {:: T} beside Haskell's ::

```
// Delf: a comment, so it can't change the program
{:: ((TArray<T>, (T → Optional<U>)) → Optional<U>)}
function FindMap<T, U>(…): Optional<U>;

const N := O.Map(λx → Absent)
          {:: Optional<Optional<Integer>>};
// hint: U from it; assert: an error if not

const P := Row.Find('^').Map(λx → …) {:: ?};
// query, delfc: P is Optional<TPoint>

O.Map<_, TPoint>(…)  // _: the slot delfc infers

var Make := TStringList.Create {:: (() → TStringList)};
// would make a function, not a list: not used
```

```
-- Haskell: code, so it can
findMap :: [t] → (t → Maybe u) → Maybe u


n = read "5" :: Int     -- 5
d = read "5" :: Double  -- 5.0: the annotation
--  picks which read runs

f :: Int → Int
f x = _ + x         -- typed hole: GHC says _ :: Int

:t findMap        -- GHCi: asks the type


-- curried: a → b → c; Delf: ((A, B) → C)
```

Three roles, one form

{:: T} is a hint where inference stalls and an assert that stops the build; {:: ?} asks. Haskell's annotation guides and is checked; holes and :t ask.

Haskell's :: can change the program

read "5" :: Int and :: Double run different code: the type chooses. Fair in Haskell, where the annotation is code like any other.

Delf's never does, by choice

Deleting a comment must never change the program. A hint could turn TStringList.Create into a function; Delf asks for λ -> in code instead.

SLIDE 10, UPDATED 25 OCT · NOT DELF

The bare point, now stopped by delfc

```
function Start(const Input: TGrid<Char>): Optional<TPoint>;
begin
  for const Row in Input.Rows do
    if Row.Find('^') is Present(x) then Exit( Point(x, Row.Index) );
  Exit( Absent );
end;
```

delfc, at the Exit

```
error: Point(x, Row.Index) is TPoint and the result is Optional<TPoint>: Delf converts nothing implicitly (SPEC §8); write Present(Point(x, Row.Index))
```

Said in the Delf

Not E2010 from dcc32 in the output: the line you wrote, with the fix.

Rung 1

Exit only, a Delf sum on one side. Assignments, bindings and arguments next.

SLIDE 16, UPDATED 25 OCT

No difference left: all four catch a corrupt tag

A, B, C and D

_ is written as what it stands for, Absent; the else is the trapdoor

```
  TOptionalKind.Absent  : ;
else
  __dpp_Unreachable('TOptionalKind', Ord(__dpp_1.Kind));
end
```

The rule (user)

“_ should mean any other valid value, and we still have a trapdoor for any other random garbage.”

Closed types only

Sums, enums, Boolean, closed ranges. On Integer, Char or string, _ is still the else.

🧪 A tag set to 7

Narrowed, it raises: Delf: a match met TOptionalKind ordinal 7, which it does not cover.

The tail of each case, indentation trimmed · delfc at ecdfa65 · SPEC §9.3

25 OCT · A PROGRAMMING MOMENT

A question of principle fixed FindMap

1 · The question

Should the signature imply Present?

Convenience against principle (slides 10–11). Philosophy says nothing converts implicitly; the maths says an Optional<TPoint> is another world, and Present is the way in (slide 14).

→

2 · The answer, enforced

delfc checks every Exit

A bare point where an Optional is the result is now delfc's error, with the fix, not dcc32's E2010 (slide 31).

→

3 · The redesign

To check a value, know its type

delfc had to learn what values are, down to the RTL: Point(...) answers a TPoint.

→

4 · The bonus

FindMap goes through

Slide 24's "cannot infer U" was that same missing type. Nobody aimed at it.

```
Exit( Input.Rows.FindMap(λRow →
  Row.Find('^').Map(λx → Point(x, Row.Index))) );
```

```
start: (4, 6)
turns: 10
visited: 41
```

No type arguments written · 🧪 dcc32 and dcc64, a scratch copy of phase 5 · delfc at 45294c0

GENERIC TYPE INFERENCE · 25 OCT

Types from inside, and from outside

Inside what the arguments say

```
Row.Find('^')       // Optional<Integer>: T
  .Map(λx → Point(x, Row.Index))
  // the body answers a TPoint: U
```

The receiver gives T; the lambda's body gives U. Up from the leaves: slide 33's FindMap.

Outside where the value goes

```
function F(…): Optional<Optional<Integer>>;
…
  Exit( O.Map(λx → Absent) );
```

Absent of what? The body can't say. The signature can: U is Optional<Integer>.

Three outsides

An Exit's result, a typed variable, a parameter. Each now tells the call its type.

Outside first

Then the arguments. Only an outside already settled: not an outer call still working out its own.

dcc32 does neither here

From outside, E2531; through an anonymous method, E2010 (🧪 experiment 42). So delfc writes every type argument.

SPEC §6 · 🧪 test expected_types.delf, dcc32 and dcc64 · delfc at 4e6a51e

GENERIC TYPE INFERENCE · 25 OCT

When neither side says: write one, and _ the rest

```
// no type on N, a body that says nothing: write U
const N := O.Map<_, Optional<Integer>>(λx → Absent);

// delfc writes every slot: dcc32 takes all or none
const N = TOptional.Map<Integer, Optional<Integer>>(O, …);
```

O.Map<Optional<Integer>>(...), one of two

`error: Map has 2 type parameters, <T, U>, and 1 written; _ stands for one delfc infers: Map<_, Optional<Integer>>`

Slot by slot (user)

The declaration's list, with the dot or without. _ is "work this one out", as it is "any other" in a match.

Rare by design

Inside and outside come first. Usually nothing is written.

_ is a Delphi name

Two corpus projects declare a type _. None writes one as a type argument, so only Delf reads it so.

SPEC §6 · 🧪 test type_arg_holes.delf, dcc32 and dcc64 · delfc at 8409395 · Rust has _ in type arguments (from memory)

THE DOOR · 21–25 OCT

The door: rename the file, and it still builds

> *“not force a dev to HAVE TO scope all their enums before they even get a look in the door”*
>
> The user, 21 Oct. Nobody should have to fix a whole file before writing their first match.

```
// Guard.pas, renamed Guard.delf
{ delf: migrate }
…
X := Ord(North);
// a warning, not an error:
// TDir.North in Delf
```

Not features, but rules

Features bite only once used. What a renamed file meets at once is the rules: enum values qualified, a < b, c > (d) as type arguments.

The marker holds it open

{ delf: migrate } turns those rules into warnings. It comes out when the warnings are gone, and the file is plain Delf.

How big is it?

Experiment 32 runs every corpus file twice, as .pas and as .delf. Renamed, a file should get nothing new, and come out unchanged.

Next: with strict conversions the default, a renamed file without the marker hears of every implicit one. So the door is “rename, and add the marker”.

docs/migration.md · experiment 32, experiments/surveys/door.py · 15,220 files in the large corpus

THE DOOR SURVEY · 25 OCT

Twenty-six calls in a row stalled the whole survey

The code a 3 KB test unit in delphimvcframework

```
AEngine
  .AddController(TTestServerController)
  .AddController(TTestPrivateServerController)
  … 26 calls in all
  .AddMiddleware(TMVCCompressionMiddleware.Create);
```

What delfc did typed each receiver twice

```
// the type of X.F(…):
resolve the call  → the type of X
then the member   → the type of X

2 × 2 × … × 2 = 226 ≈ 67 million
```

What you saw

All 12 cores at 100% at the start; later about 20%. 15,265 of 15,273 files done, and nothing written for 18 minutes.

Old, and slow before

The morning's run got through it, slowly: part of its 65 minutes. With more done per step, it never finished.

Never → 0.1 seconds

Each type remembered for one question. The 300-file sample: identical results.

TestServerEngineConfigU.pas · delfc ff03acd · test fluent_chain.dpr: 40 links, which the morning's delfc didn't finish in 30 s

ANOTHER PASCAL · 25 OCT

Unleashed Pascal: the same ground, from the other end

Unleashed	Delf
match … of 'go': run; _: skip	match with _, strings too
var (a, b) := pair	tuples and destructuring
if / case / try as values	conditional expressions
array[10] of T	near the user's sized arrays
async / await, lock	TAsync, a library
defer, autofree	nothing yet
is not, $'{name}', step	not discussed

A fork of the compiler

Free Pascal, changed. Delf keeps dcc32 the compiler and writes plain Delphi.

Opt in, by a mode

{$mode unleashed}, and old code keeps compiling. Delf's door is close, but it flags some valid Delphi on purpose.

The question to ask it

Its match is first-match dispatch. Does it check every case is covered? That's what Delf cares most about.

github.com/unleashedpascal/compiler · features from its README, unchecked · docs/references.md

ANOTHER PASCAL · 25 OCT

Same syntax, different guarantees

Unleashed convenience on Pascal's rules

```
var p: TPoint := (3, 4);
// the fields in order: reorder them,
// and every (3, 4) swaps meaning

match d of
  North: Go;
  East:  Turn;
end;
// South, West: nothing happens
```

Delf checks, and the convenience with them

```
var P: TPoint := (3, 4);
// error: a Tuple where a TPoint is expected
// name the fields (X: 3; Y: 4)

match D of
  | North : Go
  | East  : Turn
end;
// error: doesn't cover South, West
```

The same words

match ... of, _, (a, b), destructuring, the swap: written almost alike.

Their match

Value dispatch, first match wins. No bindings, so no payloads. Coverage only for a match used as a value, by a written _.

Their tuples

Structural, and records take them by shape. Fields convert as they go: an Int32 into a Byte.

Unleashed: its match.md and tuples.md, unchecked · Delf: delfc's errors, at 19064eb · docs/references.md