Andrew Blinn

Kevin Li

Cyrus Omar

FPLab @ UMich

Type-directed Prompt Construction for LLM-powered Programming Assistants

October 2023 Experience Report

Hazel Demo: Todo MVU

hazel Examples Void

```
#Render main TODO view#
#Should be a brown box with rounded corners and nice padding #
#Labelled Hazel TODOs. Use comics sans#
let view: (Action -> Event, Model) -> Node =
  fun go, (buffer, todos) ->
    Div(
      [
        Style([
          BackgroundColor("#946"),
          BorderRadius("0.3em"),
          Padding("1em")]),
        Create("class", "app")
      ],
      [
        Div([
          Style([
            FontFamily("'Comic Sans MS', cursive, sans-serif"),
            Color("#fff"),
            Margin("1em"),
            Display("flex"),
            JustifyContent("center")]),
          Create("class", "header")
        ], [Text("Hazel TODOs")]),
        buffer(go),
        add_button(go),
        todos_deck(go,todos)
      ]
    ) in
```

EXP Function literal : (Action -> Event, Model) -> Node equals expected type

Backup in case of technical difficulties

```
#A box titled "Hazel TODOs".#
#The box has nice padding and vivid colors#
#It uses comics sans. Obviously.#
let view: Model -> Node =
  fun (buffer, todos) ->
    Div(
      [
        Style([
          Display("flex"),
          FlexDirection("column"),
          Gap("1em"),
          Padding("1em"),
          BorderRadius("1em"),
          BackgroundColor("#4CAF50"),
          Color("white"),
          FontFamily("Comic Sans MS, cursive, sans-serif")
        ])
      ],
      [
        Div([], [Text("Hazel TODOs")]),
        buffer,
        add_button,
        todos_deck(todos)
      ]
    ) in

Render("todo_app", Model.init, view, update)
```

```
#A box titled "Hazel TODOs".#
#The box has nice padding and vivid colors#
#It uses comics sans. Obviously.#
let view: Model -> Node =
  fun (buffer', todos) ->
    Div(
      [
        Style([
          Display("flex"),
          FlexDirection("column"),
          Gap("1em"),
          Padding("1em"),
          BorderRadius("1em"),
          BackgroundColor("#4CAF50"),
          Color("white"),
          FontFamily("Comic Sans MS, cursive, sans-serif")
        ])
      ],
      [
        Div([], [Text("Hazel TODOs")]),
        buffer,
        add_button,
        todos_deck(todos)
      ]
    ) in

Render("todo_app", Model.init, view, update)
```

Intro

- Large language models have changed the landscape for code completion

- But how do they get fed? Poor or missing information can lead to hallucination

- We have a wealth of semantic information available that can be used to inform and constrain generation

LLMs vs Types

Types & LLMs: Opportunities for Intervention

- **Before prompting**
  Types informing prompt creation
- **After prompting**
  Types for validation + correction

Types & LLMs: Opportunities for Intervention

- **Before prompting**
  Types informing prompt creation
- **After prompting**
  Types for validation + correction
- **During prompting (Future work)**
  Types constraining productions by-token
- **Between prompting (Future work)**
  Types informing multi-step strategies

Prompt Imagineering

Prompt Construction: Challenges

- Limited context window length;
  can't just throw everything in

- Per-token costs

- Even with larger windows, models aren't
  equally attentive to the whole length

Prompt Construction: Challenges

- Limited context window length;
  can't just throw everything in
- Per-token costs
- Even with larger windows, models aren't
  equally attentive to the whole length
- **Need to distill pertinent data**

```
type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in
```

Prompt Construction: Approaches

- Basic approach:
  - Truncate current file up to caret to context window length
- Current approaches: Heuristics for related code:
  - Last active tabs (Github Co-pilot)
  - Imports via dependency analysis (RepoCoder: Zhang et al. 2023)
  - RAG: Vector databases to retrieve ‘similar’ code

Running Example: Update

- We're going to try to get our assistant to complete the entire core update logic for our Todo app
- We use this example instead of the view function shown you as it's easier to evaluate via unit tests
- We'll use to versions: An easier, shorter one with helpers provided in sketch, and a harder version without helpers

Simpler Version

```
let add: Model -> [Todo] =
  fun (description, todos) ->
    if description $== ""
    then todos
    else (description, false) :: todos in
let remove: (Int, [Todo]) -> [Todo]=
  fun (index, todos) ->
    List.filteri(fun i, _ -> i!= index, todos) in
let toggle: (Int, [Todo]) -> [Todo]=
  fun (index, todos) ->
    List.mapi(
      fun i, (description, done) ->
        (description, if i == index then !done else done),
        todos) in

let update: (Model, Action) -> Model =
  fun ((input: String, todos: [Todo]), action) ->
    case action
      | AddTodo => (input, add(input, todos))
      | RemoveTodo(i) => (input, remove(i, todos))
      | ToggleTodo(i) => (input, toggle(i, todos))
      | UpdateBuffer(s) => (s, todos) end in
```

Complex Version

```
let update: (Model, Action) -> Model =
  fun ((input: String, todos: [Todo]), action) ->
    let add: Model -> [Todo] =
      fun (description, todos) ->
        if description $== ""
        then todos
        else (description, false) :: todos in
    let remove: (Int, [Todo]) -> [Todo]=
      fun (index, todos) ->
        List.filteri(fun i, _ -> i!= index, todos) in
    let toggle: (Int, [Todo]) -> [Todo]=
      fun (index, todos) ->
        List.mapi(
          fun i, (description, done) ->
            (description, if i == index then !done else done),
            todos) in
    case action
      | AddTodo => (input, add(input, todos))
      | RemoveTodo(i) => (input, remove(i, todos))
      | ToggleTodo(i) => (input, toggle(i, todos))
      | UpdateBuffer(s) => (s, todos) end in
```

Simpler Version

```
let add: Model -> [Todo] =
  fun (description, todos) ->
    if description $== ""
    then todos
    else (description, false) :: todos in
let remove: (Int, [Todo]) -> [Todo]=
  fun (index, todos) ->
    List.filteri(fun i, _ -> i!= index, todos) in
let toggle: (Int, [Todo]) -> [Todo]=
  fun (index, todos) ->
    List.mapi(
      fun i, (description, done) ->
        (description, if i == index then !done else done),
        todos) in

let update: (Model, Action) -> Model =
  fun ((input: String, todos: [Todo]), action) ->
    case action
      | AddTodo => (input, add(input, todos))
      | RemoveTodo(i) => (input, remove(i, todos))
      | ToggleTodo(i) => (input, toggle(i, todos))
      | UpdateBuffer(s) => (s, todos) end in
```

Complex Version

```
let update: (Model, Action) -> Model =
  fun ((input: String, todos: [Todo]), action) ->
    let add: Model -> [Todo] =
      fun (description, todos) ->
        if description $== ""
        then todos
        else (description, false) :: todos in
    let remove: (Int, [Todo]) -> [Todo]=
      fun (index, todos) ->
        List.filteri(fun i, _ -> i!= index, todos) in
    let toggle: (Int, [Todo]) -> [Todo]=
      fun (index, todos) ->
        List.mapi(
          fun i, (description, done) ->
            (description, if i == index then !done else done),
            todos) in
    case action
      | AddTodo => (input, add(input, todos))
      | RemoveTodo(i) => (input, remove(i, todos))
      | ToggleTodo(i) => (input, toggle(i, todos))
      | UpdateBuffer(s) => (s, todos) end in
```

Without prompt engineering

- The model doesn't know hazel. Hence: Frankencode

```
let update: (Model, Action) -> Model =
  fun ((input: String, todos: [Todo]), action) ->
    match action with
    | Add ->
      let todo = { id = newId (); text = input; completed = false } in
      ("", todo :: todos)
      | Remove id ->
        ("", List.filter (fun todo -> todo.id <> id) todos)
        | UpdateInput input -> (input, todos)
          | Toggle id ->
            let todos = List.map (fun todo ->
              if todo.id = id then { todo with completed = not todo.completed } else todo) todos in
            (input, todos)
            | ClearCompleted ->
              let todos = List.filter (fun todo -> not todo.completed) todos in
              (input, todos) in
```

Without type information

- May do okay if everything is in a small single file
- But otherwise, no great way to guess names

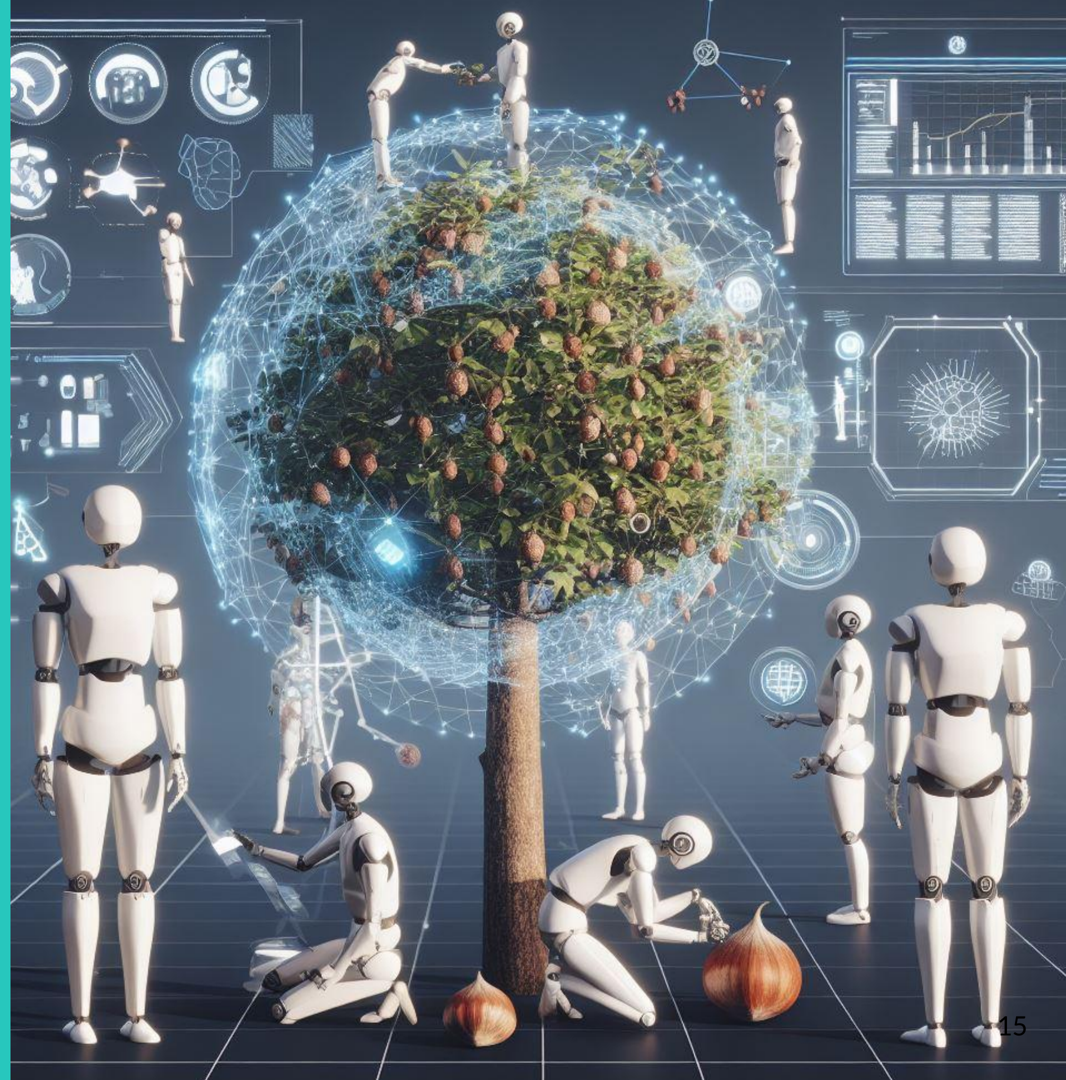
Ground truth

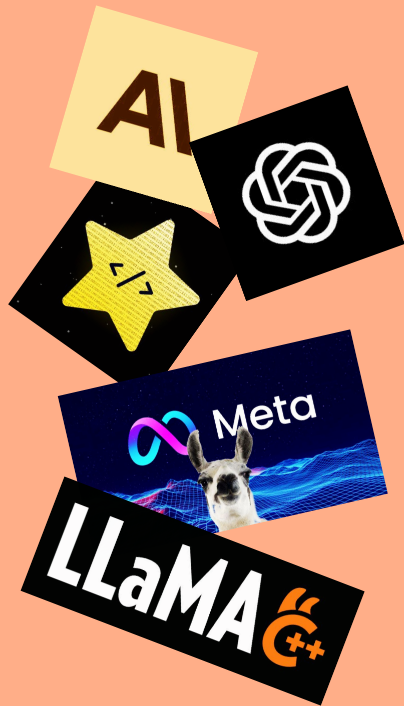
```
let update: (Model, Acti
  fun ((buffer: String, t
    case action
      | AddTodo => (inpu
      | RemoveTodo(i) =>
      | ToggleTodo(i) =>
      | UpdateBuffer(s)
```

Hallucinations

```
let update: (Model, Action) -> Model =
  fun ((buffer: String, todos: [Todo]), action) ->
    case todo_action
      | AddTodo => (buffer, text::todos)
      | DeleteTodo => (buffer, List.remove_at(index, todos))
      | UpdateBuffer => (new_buffer, todos)
      | NoAction => (buffer, todos)
```

Hazel LLM Assistant





LLMs for code completion: The model landscape

- We're PL people, not ML people.
  We're mostly using models as black boxes

Commercial APIs

- Smart (out of the box)
- Convenient

Local models

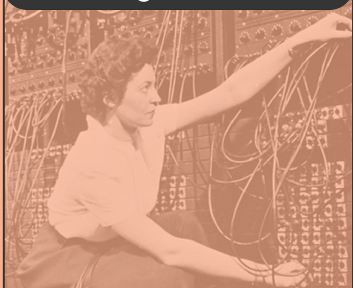
- Flexible
- Reproducible

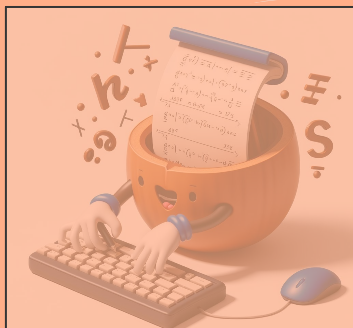
LLMs for code completion: The model landscape

- For our initial experiments, we picked OpenAI GPT4 via Microsoft Azure

- Quickest way out-of-the box to start generating somewhat syntactically & semantically reasonable code

Hazel LLM Assistant: Conversational Architecture





System Message: Generic Instructions

User Message: Program Sketch

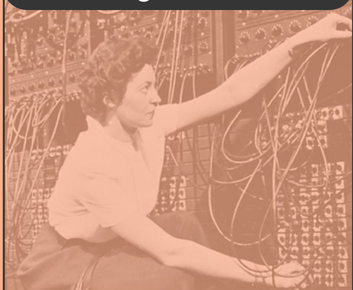
Language Server: Type Information

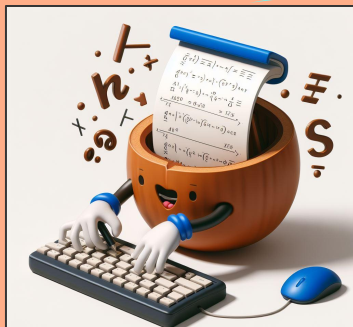
Assistant Message: Suggested completion

Language Server: Type errors, if any

Assistant Message: Corrected completion, if necessary

Hazel LLM Assistant: Conversational Architecture





System Message: Generic Instructions

User Message: Program Sketch

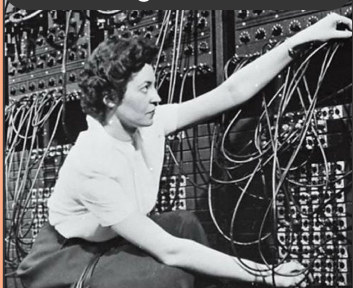
Language Server: Type Information

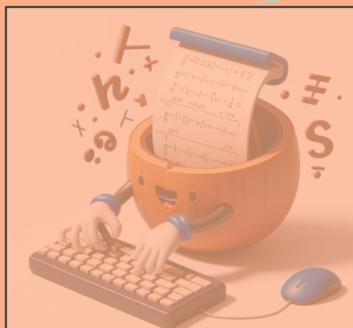
Assistant Message:
Suggested completion

Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary

Hazel LLM Assistant: Conversational Architecture





System Message: Generic Instructions

User Message: Program Sketch

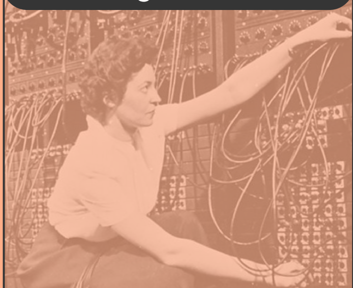
Language Server: Type Information

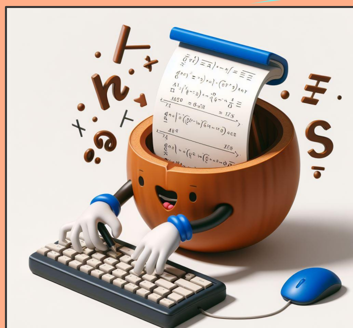
Assistant Message:
Suggested completion

Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary

Hazel LLM Assistant: Conversational Architecture





System Message: Generic Instructions

User Message: Program Sketch

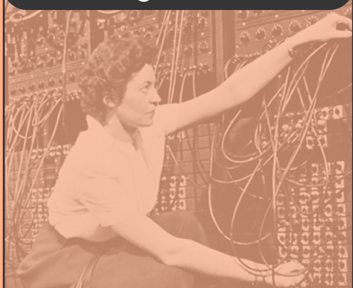
Language Server: Type Information

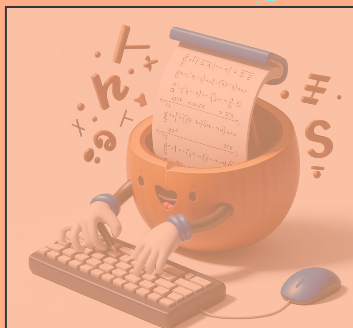
Assistant Message:
Suggested completion

Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary

Hazel LLM Assistant: Conversational Architecture





System Message: Generic Instructions

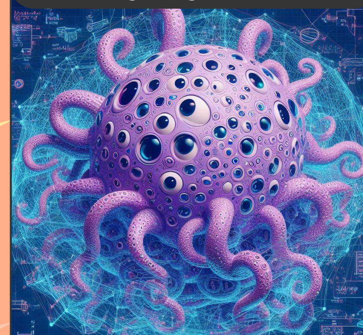
User Message: Program Sketch

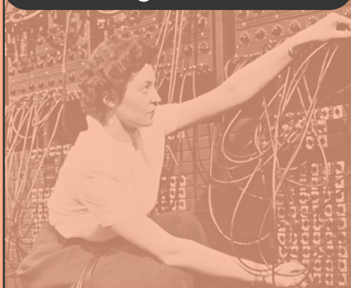
Language Server: Type Information

Assistant Message:
Suggested completion

Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary





Hazel LLM Assistant: Conversational Architecture

System Message: Generic Instructions

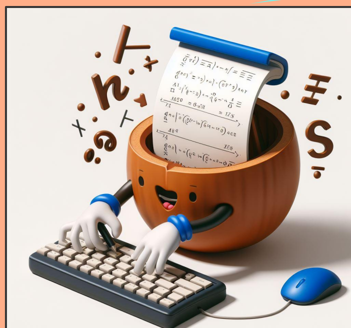
User Message: Program Sketch

Language Server: Type Information

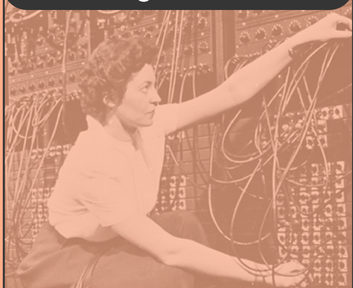
Assistant Message:
Suggested completion

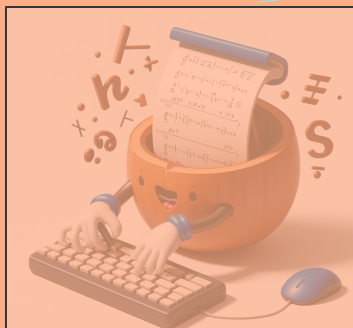
Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary



Hazel LLM Assistant: Conversational Architecture





System Message: Generic Instructions

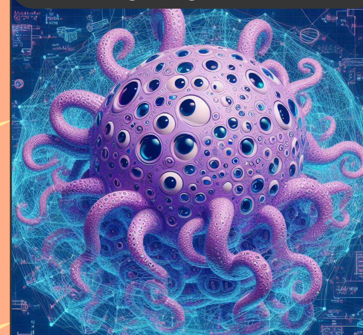
User Message: Program Sketch

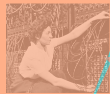
Language Server: Type Information

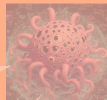
Assistant Message:
Suggested completion

Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary







System Message (Always the same)

- Task Description

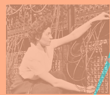
```
let main_prompt = [
  "CODE COMPLETION INSTRUCTIONS:",
  "- Reply with a functional, idiomatic replacement for the program hole marked '??' in the provided program sketch",
  //" - Your replacement suggestion doesn't have to be complete; it's okay to leave holes (marked '?') in your completi
  "- Reply only with a single replacement term for the unqiue distinguished hole marked '??'",
```

- Syntax Specification

```
let hazel_syntax_notes = [
  "HAZEL SYNTAX NOTES:",
  "- Hazel uses C-style function application syntax, with parenthesis around comma-separated arguments",
  "- Function application is ALWAYS written using parentheses and commas: use 'function(arg1, arg2)'. DO NOT just use s
  "- Function parameters are ALWAYS commas separated: 'fun arg1, arg2 -> <exp>'. DO NOT use spaces to separate function
```

- Input/Output Examples (Few-shot)

```
let samples = [
  ("let merge_sort: [Int]->[Int] =\n??\nin\nmerge_sort([4,1,3,7,2])",
   Type.expected(Some(Ana(Arrow(Int, Int))), ~ctx=[]),
   "fun list ->\nlet split: [Int]->([Int],[Int]) = fun left, right -> ?\nin\nlet merge: ([Int],[Int])->[Int]= ?\nin\nle
  ("type MenuItem =\n+ Breakfast(Int, Int)\n+ Lunch(Float)\nin\nlet per_lunch_unit = 0.95 in\nlet price: MenuItem-> Flo
   Type.expected(Some(Ana(Var("MenuItem"))), ~ctx=[]),
   "fun m ->\ncase m\n| Breakfast(x, y) => ??\n| Lunch(f) => f *. per_lunch_unit\nend",
```



INSTRUCTIONS

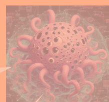
SKETCH

TYPES

COMPLETION

ERRORS

CORRECTION

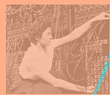


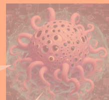
System Message (Always the same)

- **Task Description**

```
list(string)
let main_prompt = [
  "CODE COMPLETION INSTRUCTIONS:",
  "- Reply with a functional, idiomatic replacement for the program hole marked '??' in the provided program sketch",
  "- Reply only with a single replacement term for the unqiue distinguished hole marked '??'",
  "- Reply only with code",
  "- DO NOT suggest more replacements for other holes in the sketch (marked '?'), or implicit holes",
  "- DO NOT include the program sketch in your reply",
  "- DO NOT include a period at the end of your response and DO NOT use markdown",
```

- Syntax Specification
- Input/Output Examples (Few-shot)



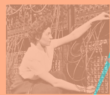


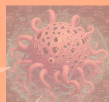
System Message (Always the same)

- Task Description
- **Syntax Specification**

```
list(string)
let hazel_syntax_notes = [
  "HAZEL SYNTAX NOTES:",
  "- Hazel uses C-style function application syntax, with parenthesis around comma-separated arguments",
  "- Function application is ALWAYS written using parentheses and commas: use 'function(arg1, arg2)'. DO
  "- Function parameters are ALWAYS commas separated: 'fun arg1, arg2 -> <exp>'. DO NOT use spaces to se
  "- There is no dot accessor notation for tuples; DO NOT use tuple.field. use pattern matching for des
  "- The following ARE NOT Hazel keywords. DO NOT use these keywords: switch, with, of, rec. ALWAYS omi
  "- Pattern matching is ALWAYS written a 'case ... end' expression. Cases MUST END in an 'end' keyword.
  "- The ONLY way to define a named function is by using a function expression nested in a let expressio
  "- No 'rec' keyword is necessary for 'let' to define a recursive function. DO NOT use the 'rec' keywor
```

- Input/Output Examples (Few-shot)



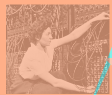


System Message (Always the same)

Language Server

- Task Description
- Syntax Specification
- **Input/Output Examples (Few-shot)**

```
  "fun x:Int -> ??",
),
(
  "let get: Option_int => Int = " ++
  "  case Some(5)" ++
  "  | Some(x) => ??" ++
  "  | None => 0 end",
  Type.expected(Some(Ana(Int)), ~ctx=[]),
  "x",
),
(
```



INSTRUCTIONS

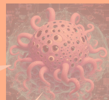
SKETCH

TYPES

COMPLETION

ERRORS

CORRECTION



System Message: Preliminary Results

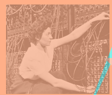
Sketch

```
let update: (Model, Action) -> Model =
  fun (input, todos), action ->
    ?? in
```

Reference Solution

```
let update: (Model, Action) -> Model =
  fun ((input: String, todos: [Todo]), action) ->
    let add: Model -> [Todo] =
      fun (description, todos) ->
        if description $== ""
        then todos
        else (description, false) :: todos in
    let remove: (Int, [Todo]) -> [Todo]=
      fun (index, todos) ->
        List.filteri(fun i, _ -> i!= index, todos) in
    let toggle: (Int, [Todo]) -> [Todo]=
      fun (index, todos) ->
        List.mapi(
          fun i, (description, done) ->
            (description, if i == index then !done else done),
          todos) in
    case action
    | AddTodo => (input, add(input, todos))
    | RemoveTodo(i) => (input, remove(i, todos))
    | ToggleTodo(i) => (input, toggle(i, todos))
    | UpdateBuffer(s) => (s, todos) end in
```

- We evaluated completions via 12 held-out unit tests.
- We experimented by holding out the major constituents of the prompt, performing 10 trials each at default temperature



INSTRUCTIONS

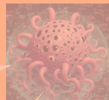
SKETCH

TYPES

COMPLETION

ERRORS

CORRECTION



System Message: Preliminary Results

Sketch

```
let update: (Model, Action) -> Model =
  fun (input, todos), action ->
    ?? in
```

Reference Solution

```
let update: (Model, Action) -> Model =
  fun ((input: String, todos: [Todo]), action) ->
    let add: Model -> [Todo] =
      fun (description, todos) ->
        if description $== ""
        then todos
        else (description, false) :: todos in
    let remove: (Int, [Todo]) -> [Todo]=
      fun (index, todos) ->
        List.filteri(fun i, _ -> i!= index, todos) in
    let toggle: (Int, [Todo]) -> [Todo]=
      fun (index, todos) ->
        List.mapi(
          fun i, (description, done) ->
            (description, if i == index then !done else done),
            todos) in
    case action
      | AddTodo => (input, add(input, todos))
      | RemoveTodo(i) => (input, remove(i, todos))
      | ToggleTodo(i) => (input, toggle(i, todos))
      | UpdateBuffer(s) => (s, todos) end in
```

- **All parts included**

  10/10 parsed, **4.2/12** tests passing (σ 1.9)

- **No Input/Output Examples**

  1/10 parsed, **0** tests passing

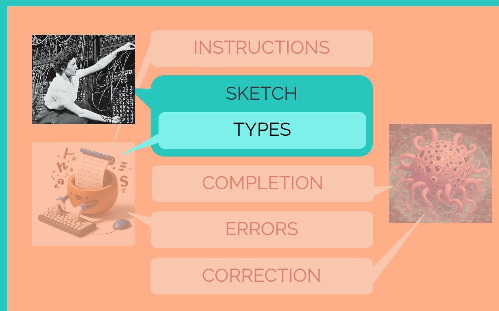
- **No Syntax Specification**

  4/10 parsed, **0** tests passing

- **No Task Description**

  7/10 parsed, **2.7/12** tests passing (σ 3.6)

Type-Directed Prompting In Hazel



INSTRUCTIONS

SKETCH

TYPES

COMPLETION

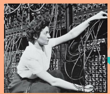
ERRORS

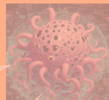
CORRECTION

Programmer

Language Server (Varies per prompt)

- Program Sketch (around Cursor)
- Expected Type (at Cursor)
- Typing Context (at Cursor)
- Extracted Relevant Definitions

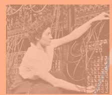




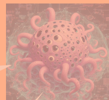
Language Server (Varies per prompt)

- **Program Sketch (around Cursor)**
- Expected Type (at Cursor)
- Typing Context (at Cursor)
- Extracted Relevant Definitions

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  fun (input, todos), action ->
    ?? in
```

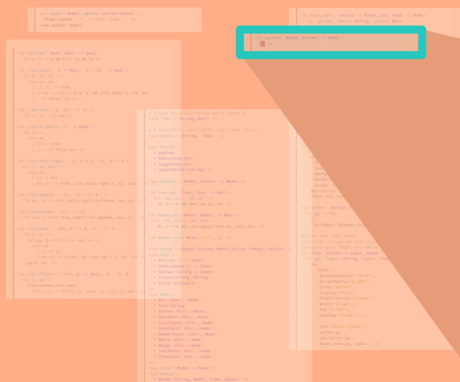


INSTRUCTIONS
SKETCH
TYPES
COMPLETION
ERRORS
CORRECTION



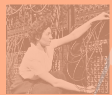
Language Server

- Program Sketch (around Cursor)
- **Expected Type (at Cursor)**
- Typing Context (at Cursor)
- Extracted Relevant Definitions



```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  fun (input, todos), action ->
    ?? in
```

Γ EXP ? Empty expression hole
Fillable by any expression of type `Model`



INSTRUCTIONS

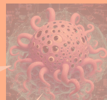
SKETCH

TYPES

COMPLETION

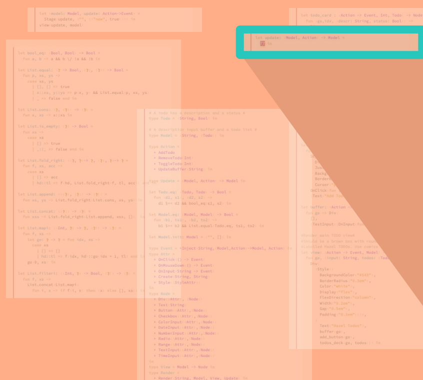
ERRORS

CORRECTION



Language Server

- Program Sketch (around Cursor)
- Expected Type (at Cursor)
- **Typing Context (at Cursor)**
- Extracted Relevant Definitions

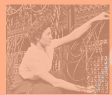


```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  fun (input, todos), action ->
   ?? in
```

Γ EXP ? Empty expression hole

Fillable by any expression of type Model

```
toggle : (Int, [Todo]) -> [Todo]
input : String
todos : [Todo]
action : Action
```



INSTRUCTIONS

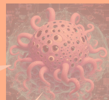
SKETCH

TYPES

COMPLETION

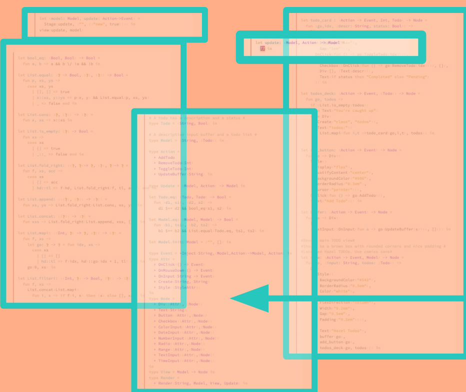
ERRORS

CORRECTION



Language Server

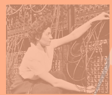
- Program Sketch (around Cursor)
- Expected Type (at Cursor)
- Typing Context (at Cursor)
- **Extracted Relevant Definitions**



Γ EXP ? Empty expression hole

Fillable by any expression of type `Model`

```
toggle : (Int, [Todo]) -> [Todo]
input : String
todos : [Todo]
action : Action
```



INSTRUCTIONS

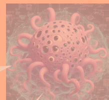
SKETCH

TYPES

COMPLETION

ERRORS

CORRECTION



Language Server

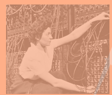
- Program Sketch (around Cursor)
- Expected Type (at Cursor)
- Typing Context (at Cursor)
- **Extracted Relevant Definitions**

```
# A todo has a description and a status #
type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Model = (String, [Todo]) in

type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in

type Update = (Model, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
  fun (d1, s1), (d2, s2) ->
    d1 $== d2 && bool_eq(s1, s2) in

let Model.eq: (Model, Model) -> Bool =
  fun (b1, ts1), (b2, ts2) ->
    b1 $== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in
type Attr =
  + OnClick(() -> Event)
  + OnMouseDown(() -> Event)
  + OnInput(String -> Event)
  + Create(String, String)
  + Style([StyleAttr])
in
type Node =
  + Div([Attr], [Node])
  + Text(String)
  + Button([Attr],[Node])
  + Checkbox([Attr],[Node])
  + ColorInput([Attr],[Node])
  + DateInput([Attr],[Node])
  + NumberInput([Attr],[Node])
  + Radio([Attr],[Node])
  + Range([Attr],[Node])
  + TextInput([Attr],[Node])
  + TimeInput([Attr],[Node])
```

Γ EXP ? Empty expression hole

Fillable by any expression of type `Model`

```
toggle : (Int, [Todo]) -> [Todo]
input : String
todos : [Todo]
action : Action
```



INSTRUCTIONS

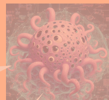
SKETCH

TYPES

COMPLETION

ERRORS

CORRECTION



Language Server

- Program Sketch (around Cursor)
- Expected Type (at Cursor)
- Typing Context (at Cursor)
- **Extracted Relevant Definitions**

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  fun (input, todos), action ->
    ?? in
```

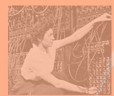
```
type Model = (String, [Todo]) in
```

```
type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in
```

Γ EXP ? Empty expression hole

Fillable by any expression of type Model

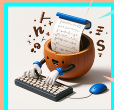
```
toggle : (Int, [Todo]) -> [Todo]
input : String
todos : [Todo]
action : Action
```

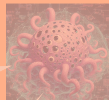


INSTRUCTIONS

SKETCH

TYPES





COMPLETION

ERRORS

CORRECTION

Language Server

- Program Sketch (around Cursor)
- Expected Type (at Cursor)
- Typing Context (at Cursor)
- **Extracted Relevant Definitions**

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  fun (input, todos), action ->
    ?? in
```

```
type Todo = (String, Bool) in
```

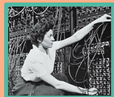
```
type Model = (String, [Todo]) in
```

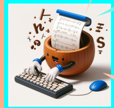
```
type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in
```

Γ EXP ? Empty expression hole

Fillable by any expression of type Model

```
toggle : (Int, [Todo]) -> [Todo]
input : String
todos : [Todo]
action : Action
```





INSTRUCTIONS

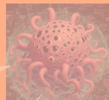
SKETCH

TYPES

COMPLETION

ERRORS

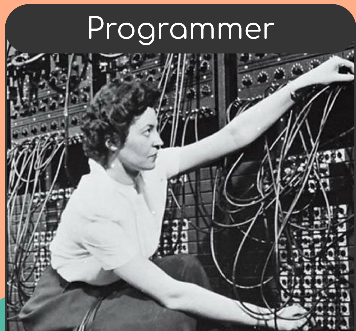
CORRECTION



Language Server

- Program Sketch (around Cursor)
- Expected Type (at Cursor)
- Typing Context (at Cursor)
- Extracted Relevant Definitions

Programmer



Language Server

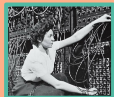
```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  fun (input, todos), action ->
    ?? in
```

```
type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in

type Todo = (String, Bool) in

type Model = (String, [Todo]) in
```

Γ EXP ? Empty expression hole

Fillable by any expression of type Model

```
toggle : (Int, [Todo]) -> [Todo]
input : String
todos : [Todo]
action : Action
```



INSTRUCTIONS

SKETCH

TYPES

COMPLETION

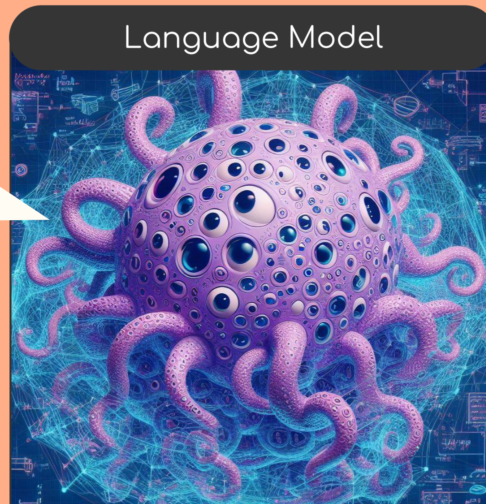
ERRORS

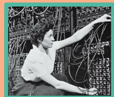
CORRECTION

And after a time...

```
let update: (Model, Action) -> Model =
  fun (input, todos), action ->
    case action
      | AddTodo =>
      let newTodo = (buffer, false) in
      ("", newTodo::todos)
      | RemoveTodo(index) =>
      let updatedTodos = List.filteri((fun i, _ -> i!= index), todos) in
      (buffer, updatedTodos)
      | ToggleTodo(index) =>
      let updatedTodos = List.mapi((fun i, (desc, done) ->
        if i == index then (desc, not done) else (desc, done)), todos) in
      (buffer, updatedTodos)
      | UpdateBuffer(newBuffer) =>
      (newBuffer, todos)
    end
  end
```

Language Model





INSTRUCTIONS

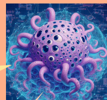
SKETCH

TYPES

COMPLETION

ERRORS

CORRECTION



Typed Prompt: Prelim Results

Completing the update function (complex version)

Simple version

```
let add: Model -> [Todo] =
  fun (description, todos) ->
    if description $== ""
    then todos
    else (description, false) :: todos in
let remove: (Int, [Todo]) -> [Todo]=
  fun (index, todos) ->
    List.filteri(fun i, _ -> i!= index, todos) in
let toggle: (Int, [Todo]) -> [Todo]=
  fun (index, todos) ->
    List.mapi(
      fun i, (description, done) ->
        (description, if i == index then !done else done),
        todos) in

let update: (Model, Action) -> Model =
  fun ((input: String, todos: [Todo]), action) ->
    case action
    | AddTodo => (input, add(input, todos))
    | RemoveTodo(i) => (input, remove(i, todos))
    | ToggleTodo(i) => (input, toggle(i, todos))
    | UpdateBuffer(s) => (s, todos) end in
```

- **Simple version:**

with types	10/10 parsed,	**8.0/12** tests passing (σ 4.1)
w/o types	9/10 parsed,	**0/12** tests passing

- **Complex version:**

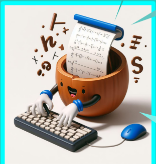
with types	10/10 parsed,	**4.2/12** tests passing (σ 1.9)
w/o types	10/10 parsed,	**0/12** tests passing

Error Correction

INSTRUCTIONS

SKETCH

TYPES



COMPLETION

ERRORS

CORRECTION

—

Error Correction

The language server intercepts responses and sends any errors back to the LLM

Error Correction

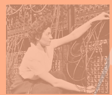
The language server intercepts responses and sends any errors back to the LLM

Interestingly, this provided us with impetus to write better, more human readable error messages for hazel

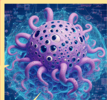
Error Correction

The language server intercepts responses and sends any errors back to the LLM

Interestingly, this provided us with impetus to write better, more human readable error messages for hazel

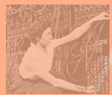


INSTRUCTIONS
SKETCH
TYPES
COMPLETION
ERRORS
CORRECTION



Error correction round

```
let update: (Model, Action) -> Model =
  fun (model, action) ->
    case action
      | AddTodo => (model.1, add(model))
      | RemoveTodo(i) => (model.1, remove(i, model.2))
      | ToggleTodo(i) => (model.1, toggle(i, model.2))
      | UpdateBuffer(s) => (s, model.2) end in
```



INSTRUCTIONS

SKETCH

TYPES

COMPLETION

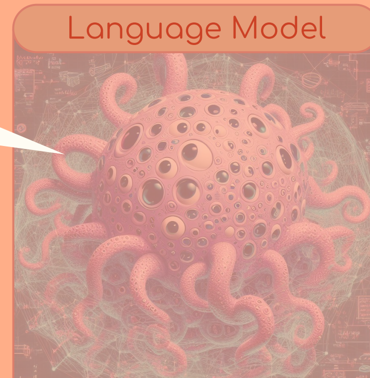
ERRORS

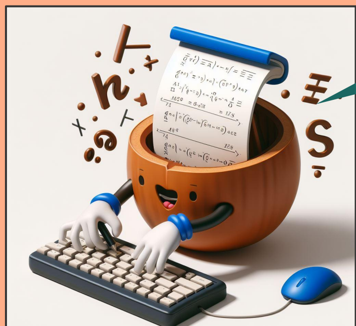
CORRECTION

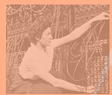
Error correction round

```
let update: (Model, Action) -> Model =
  fun (model, action) ->
   case action
    | AddTodo => (model.1, add(model))
    | RemoveTodo(i) => (model.1, remove(i, model.2))
    | ToggleTodo(i) => (model.1, toggle(i, model.2))
    | UpdateBuffer(s) => (s, model.2) end in
```


```
{"error_report": [
  "StaticErrors", [
    "Error in term: \"model.2\" Nature of error: \"model.2\" isn't a valid token",
    "Error in term: \"model.2\" Nature of error: \"model.2\" isn't a valid token",
    "Error in term: \"model.1\" Nature of error: \"model.1\" isn't a valid token",
    "Error in term: \"model.2\" Nature of error: \"model.2\" isn't a valid token",
    "Error in term: \"model.1\" Nature of error: \"model.1\" isn't a valid token",
    "Error in term: \"model.1\" Nature of error: \"model.1\" isn't a valid token"]]}
```





INSTRUCTIONS

SKETCH

TYPES

COMPLETION

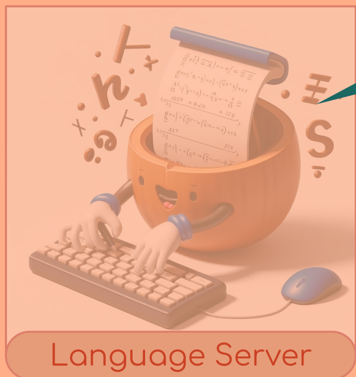
ERRORS

CORRECTION

Error correction round

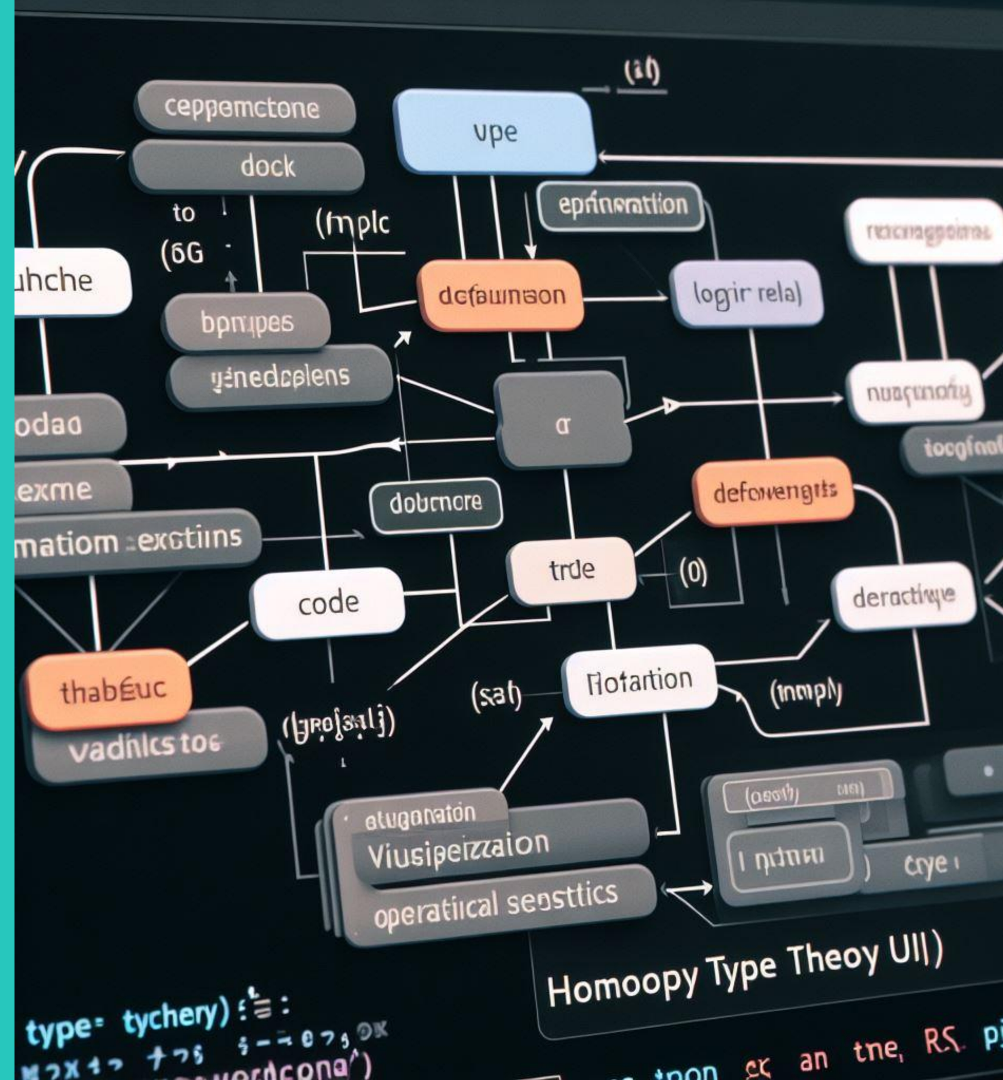
```
let update: (Model, Action) -> Model =
  fun (model, action) ->
   case action
    | AddTodo => (model.1, add(model))
    | RemoveTodo(i) => (model.1, remove(i, model.2))
    | ToggleTodo(i) => (model.1, toggle(i, model.2))
    | UpdateBuffer(s) => (s, model.2) end in
```

```
{"error_report": [
  "StaticErrors", [
   "Error in term: \"model.2\" Nature of error: \"model.2\" isn't a valid token",
   "Error in term: \"model.2\" Nature of error: \"model.2\" isn't a valid token",
   "Error in term: \"model.1\" Nature of error: \"model.1\" isn't a valid token",
   "Error in term: \"model.2\" Nature of error: \"model.2\" isn't a valid token",
   "Error in term: \"model.1\" Nature of error: \"model.1\" isn't a valid token",
   "Error in term: \"model.1\" Nature of error: \"model.1\" isn't a valid token"]]}
```


```
let update: (Model, Action) -> Model =
  fun (model, action) ->
   let (description, todos) = model in
   case action
    | AddTodo => (description, add(model))
    | RemoveTodo(i) => (description, remove(i, todos))
    | ToggleTodo(i) => (description, toggle(i, todos))
    | UpdateBuffer(s) => (s, todos) end in
```

Ongoing & Future Directions



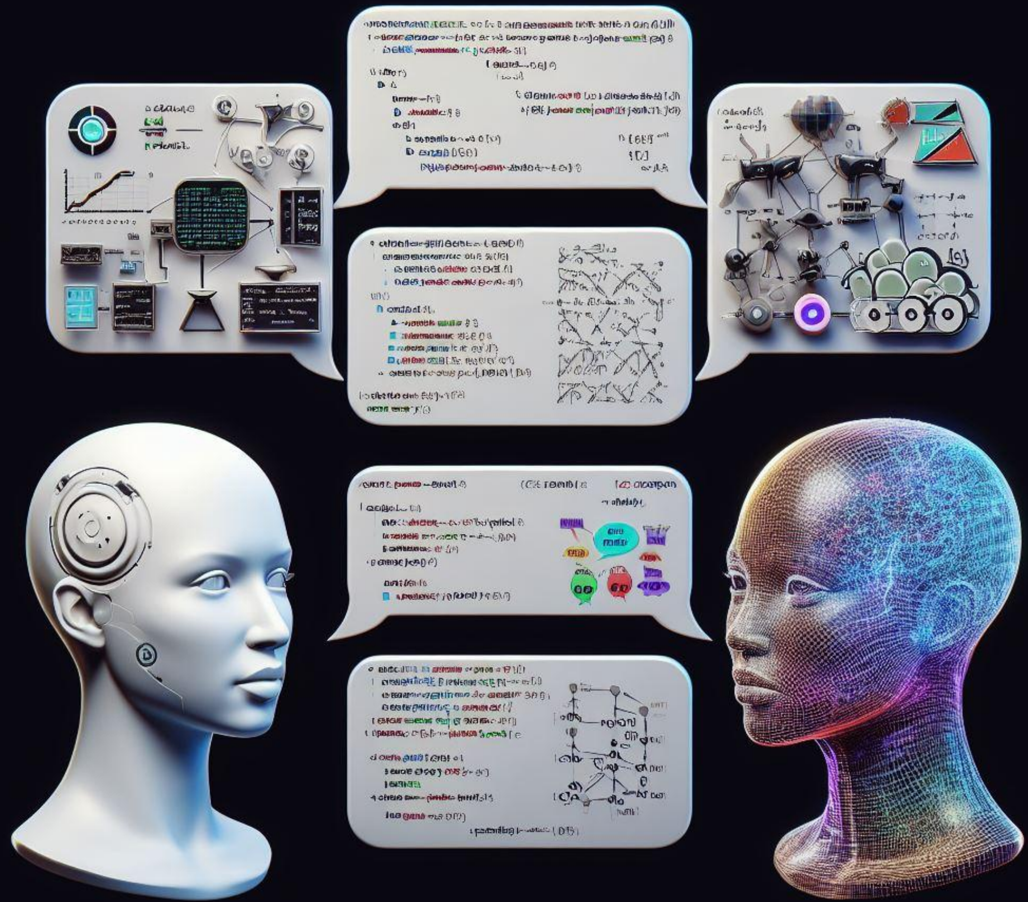
Future Directions

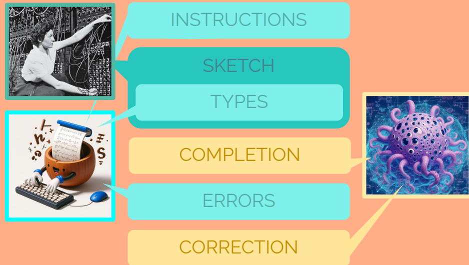
- Language Server Tweaks:
  - Replace fixed examples with type-specific ones when available
  - Move towards integrating lightweight program synthesis: Follow arrow types in context to find functions linking available input types and desired output types

- Per-token type correctness by construction:
  - Why we are using hazel: Always-available type-directed completion
  - Synergies with Token masking techniques; (Jsonformer, CFGs in llama.cpp)
  - See: Monitor-Guided Decoding (Agrawal et al 2023) & Repilot (Wei et al 2023)

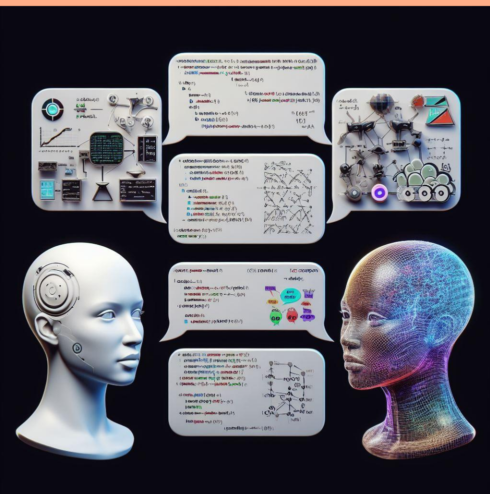
- Dynamic information
  - Smyth (Lubin et al) - Type & example directed synthesis via backwards unevaluation: Feed hole-specific value specs to the model, not just types

Dynamic Strategies & Collaboration



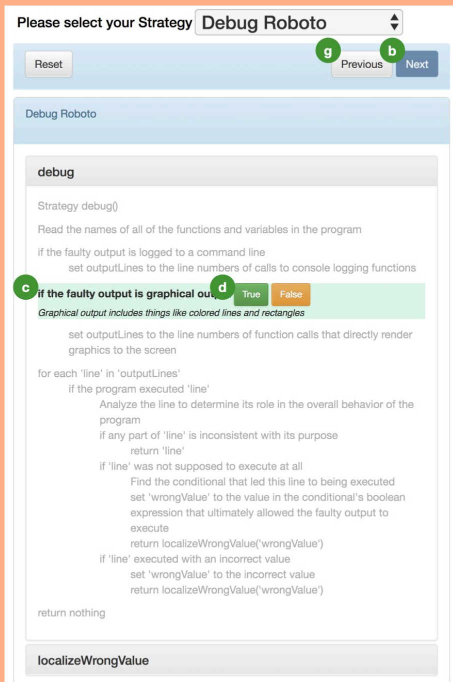


Dialogue with Language Server



- Our system so far consists of a statically-structured dialogue between user, LLM, and language server
- But what if the LLM had the ability to be more agentic? What if our dialogs could be dynamic?
- To carry out multi-stage plans to accomplish more than just code completion, we need to find ways to get models to query users and language servers

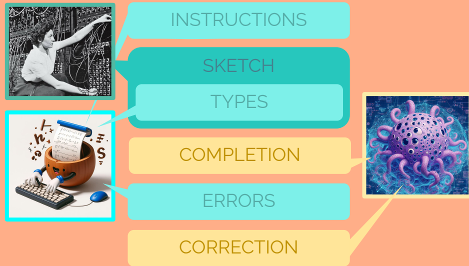
Programming Strategies



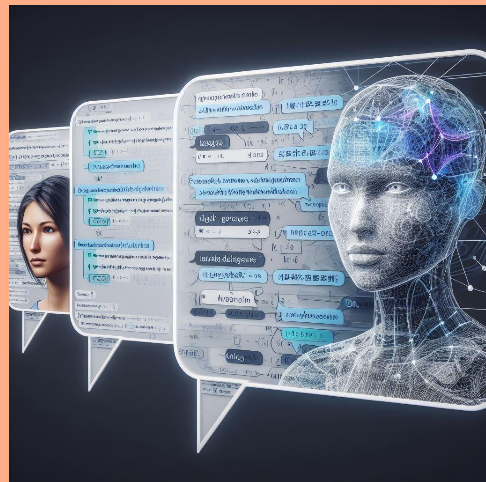
Explicit Programming Strategies
Thomas D. LaToza, Maryam Arab, Dastyni Loksa, Amy J. Ko

- LaToza et al (2020) have developed a language, Roboto to express semi-formal programming strategies based on real programmer practices

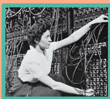
- Roboto is executable, but instead of bottoming out at machine instructions, it executes down to a trace of natural-language instructions to programmers, as a aid to help keep on task



Dialogue with Language Server



- What if our language server had a library of type-directed strategies a model could try to match to a given task?
- The LS would manage execution, which would bottom out to a trace of prompts to the LLM, to which it would be expected to return one or more edit actions to attempt accomplish that goal.



Add a priority to each TODO item

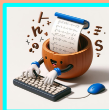
Maybe try the following generic strategy:
- Find the appropriate type definition
- Change the definition to support the change
- For all type errors created
  - Attempt a fix
  - Ask the language server if the fix is correct
    - If so, move on to the next type error
    - Otherwise, ask the user for clarification

I will try the following concrete strategy:
- I have found the type definition **Todo**
- For my change, Execute: **Command (Diff)**

  Type todo = (string, bool)

  Type todo = (string, bool, num)

Request(@LanguageServer): I need a list of all type errors



Command(Diff) has been executed successfully.

There are now 12 type errors.

The first error is:

```
let Todo.eq: (Todo, Todo) -> Bool =
  fun (d1, s1), (d2, s2) ->
```

(?, ?) inconsistent with expected type Todo

Please submit a fix

… and so on

Takeaways

- Possible with prompt engineering to get decent LLM completions in at least one niche language

- Type information provides effective guide rails to collate program sketches in some situations

- There is so much to do; let's talk!

Questions?