A Language Server for your DSL for Fun and Profit

Hannes Siebenhandl
Well-Typed LLP
BOB 2025

Introduction

Standalone Domain Specific Languages (DSLs) are prevalent and have many use-cases

- Designed to do one thing well
- Custom syntax
- Custom parser and validation

Issue: No out-of-the-box tooling support!

```
g(x) = 1 / x ;
z = 5 ;
f(h, x) =
  h(x^2) * z {- Multiplier -};

#plot for 0 < x . f(g, x) ;
```

Introduction

Standalone Domain Specific Languages (DSLs) are prevalent and have many use-cases

- Designed to do one thing well
- Custom syntax
- Custom parser and validation

Issue: No out-of-the-box tooling support!

```
g(x) = 1 / x ;
z = 5 ;
f(h, x) =
  h(x^2) * z {- Multiplier -};

#plot for 0 < x . f(g, x) ;


```

IDE Demo for L4 DSL

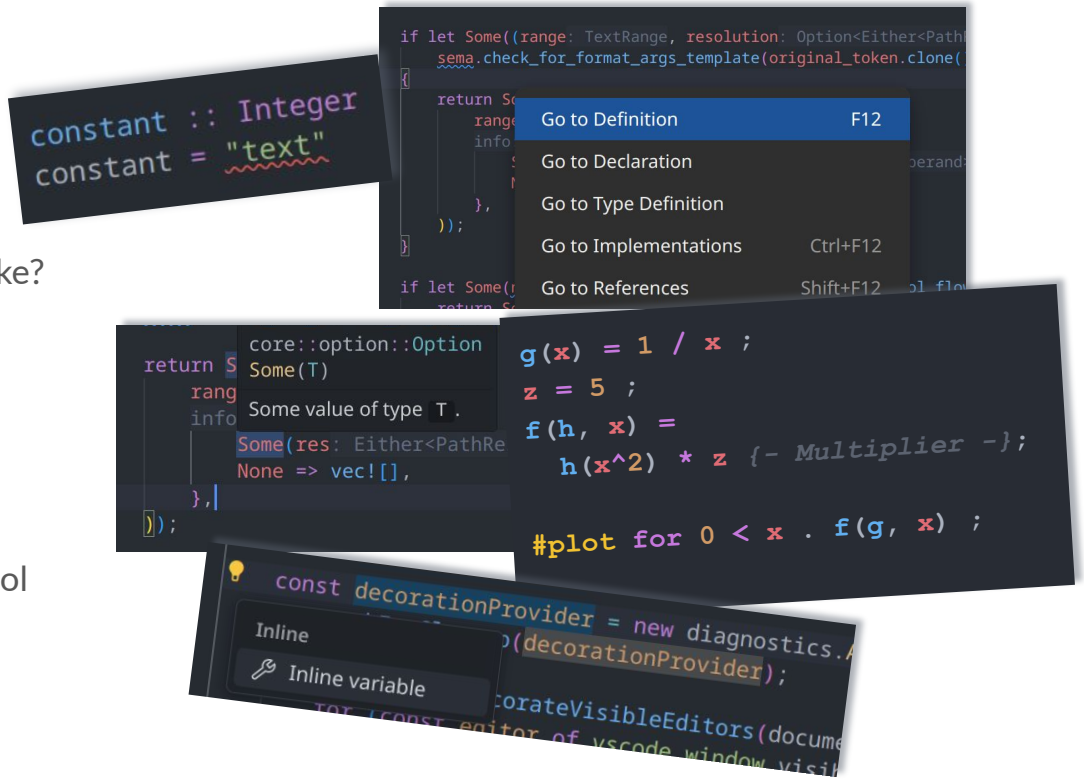
Introduction

What does a modern IDE experience look like?

- Diagnostics
- Code Navigation
- **Syntax Highlighting**
- Code Refactorings

All provided by the Language Server Protocol

➜ Let's build a Language Server!



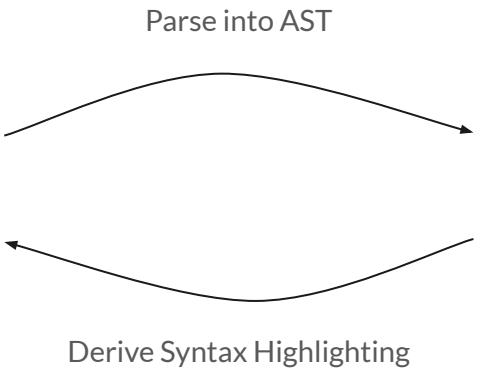
What do we want from Syntax Highlighting

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  f(h, x) =
4    h(x^2) * z {- Multiplier -};
5
6  #plot for 0 < x . f(g, x) ;
7
8
```

- Keywords, symbols, literals, and comments
  - `#plot`
  - `=, for, <, …`
  - `0, 1, 2, ...`
  - `{- Multiplier -}`
- Names
  - `z, x`
- Functions
  - `f, g, h`

Syntax Highlighting

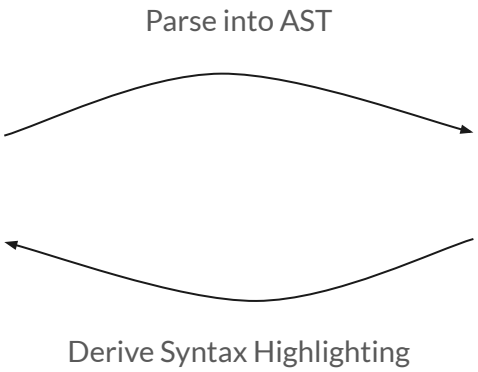
```
1  g(x) = 1 / x ;
2  z = 5 ;
3  f(h, x) =
4    h(x^2) * z {- Multiplier -};
5
6  #plot for 0 < x . f(g, x) ;
7
8
```




```
data Definition =
 MkDefinition Name Expr

data Name
 = MkConstant Var
 | MkFunc Var [Var]

data Var = MkConstant Text
data Expr
 = Mult Expr Expr
 | ...
```

Syntax Highlighting

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  f(h, x) =\n
4    h(x^2) * z {- Multiplier -};
5  \n
6  #plot for 0 < x . f(g, x) ;
7
8
```




```
data Definition =
 MkDefinition Name Expr

data Name
 = MkConstant Var
 | MkFunc Var [Var]

data Var = MkConstant Text
data Expr
 = Mult Expr Expr
 | ...
```

We are not storing enough information to provide syntax highlighting

Syntax Highlighting

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  f(h, x) =\n
4    h(x^2) * z {- Multiplier -};
5  \n
6  #plot for 0 < x . f(g, x) ;
7
8
```

Parse into AST

```
data Definition =
 MkDefinition Name Expr

data Name
 = MkConstant Var
 | MkFunc Var [Var]

data Var = MkConstant Text
data Expr
 = Mult Expr Expr
 | ...
```

- ➜ Source locations for all tokens
- ➜ Store them uniformly in the AST

Anno: **Tracking Source Locations**

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  f(h, x) =
4    h(x^2) * z {- Multiplier -};
5
6  #plot for 0 < x . f(g, x) ;
7
8
```

Parse into AST

```
data Definition =
 MkDefinition Anno Name Expr

data Name
 = MkConstant Anno Var
 | MkFunc Anno Var [Var]

data Var = MkConstant Anno Text
data Expr
 = Mult Anno Expr Expr
 | ...
```

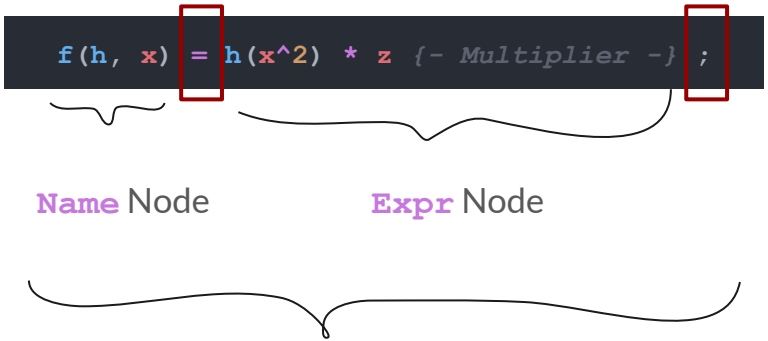
- ➜ Store Source Locations of parsed tokens in `Anno`
- ➜ Each AST node has its own `Anno`

Anno: **Tracking Source Locations**

```
f(h, x) = h(x^2) * z {- Multiplier -} ;
```

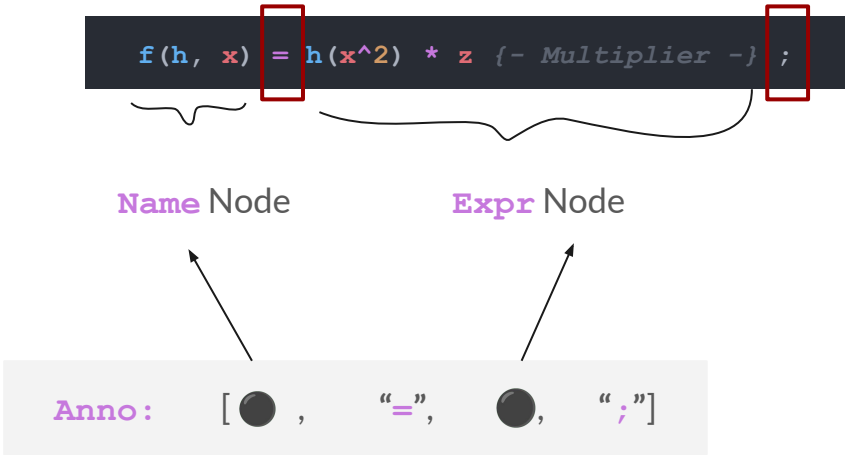
```
data Definition =
 MkDefinition Anno Name Expr

data Name
 = MkConstant Anno Var
 | MkFunc Anno Var [Var]

data Var = MkConstant Anno Text
data Expr
 = Mult Anno Expr Expr
 | ...
```

Anno: **Tracking Source Locations**



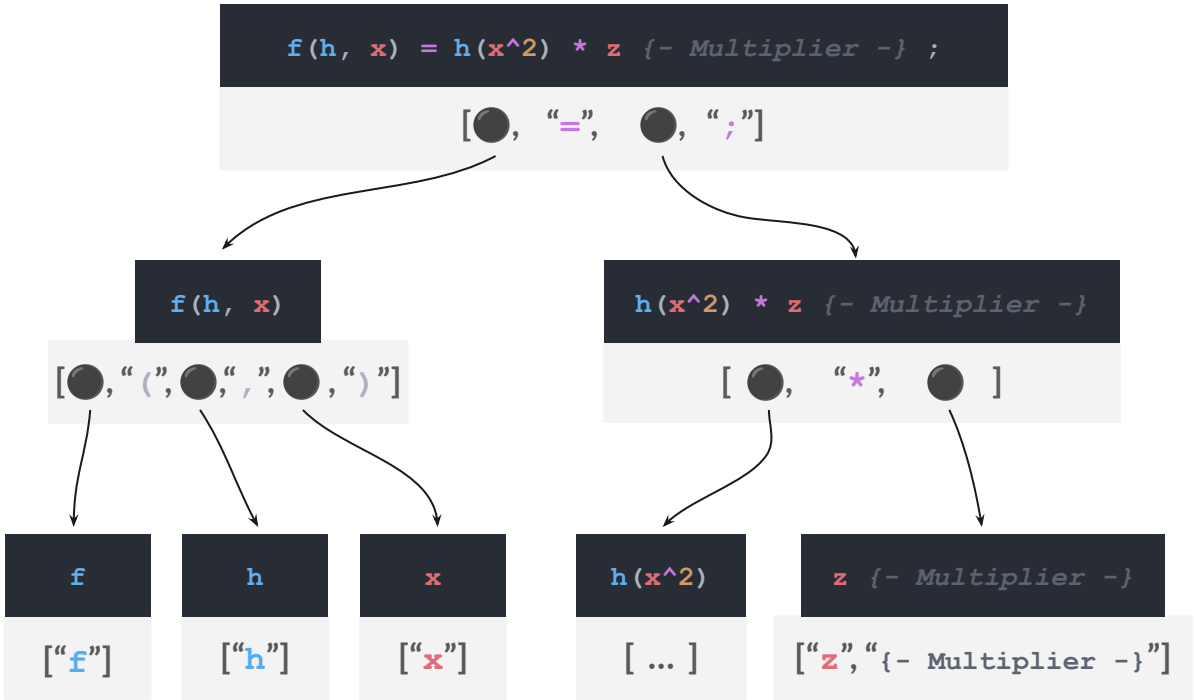

```
data Definition =
 MkDefinition Anno Name Expr

data Name
 = MkConstant Anno Var
 | MkFunc Anno Var [Var]

data Var = MkConstant Anno Text
data Expr
 = Mult Anno Expr Expr
 | ...
```

Anno: **Tracking Source Locations**




```haskell
data Definition =
 MkDefinition Anno Name Expr

data Name
 = MkConstant Anno Var
 | MkFunc Anno Var [Var]

data Var = MkConstant Anno Text
data Expr
 = Mult Anno Expr Expr
 | ...
```

Anno: **Tracking Source Locations**




```
data Definition =
 MkDefinition Anno Name Expr

data Name
 = MkConstant Anno Var
 | MkFunc Anno Var [Var]

data Var = MkConstant Anno Text
data Expr
 = Mult Anno Expr Expr
 | ...
```

Anno: **Tracking Source Locations**

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  f(h, x) =
4    h(x^2) * z {- Multiplier -};
5
6  #plot for 0 < x . f(g, x) ;
7
8
```

Parse into AST

```
data Definition =
 MkDefinition Anno Name Expr

data Name
 = MkConstant Anno Var
 | MkFunc Anno Var [Var]

data Var = MkConstant Anno Text
data Expr
 = Mult Anno Expr Expr
 | ...
```

Anno:

- ✔ Single source of truth for token locations
- ✔ Sufficient for implementing most IDE features
- → Trade Off: Flexibility vs type safety

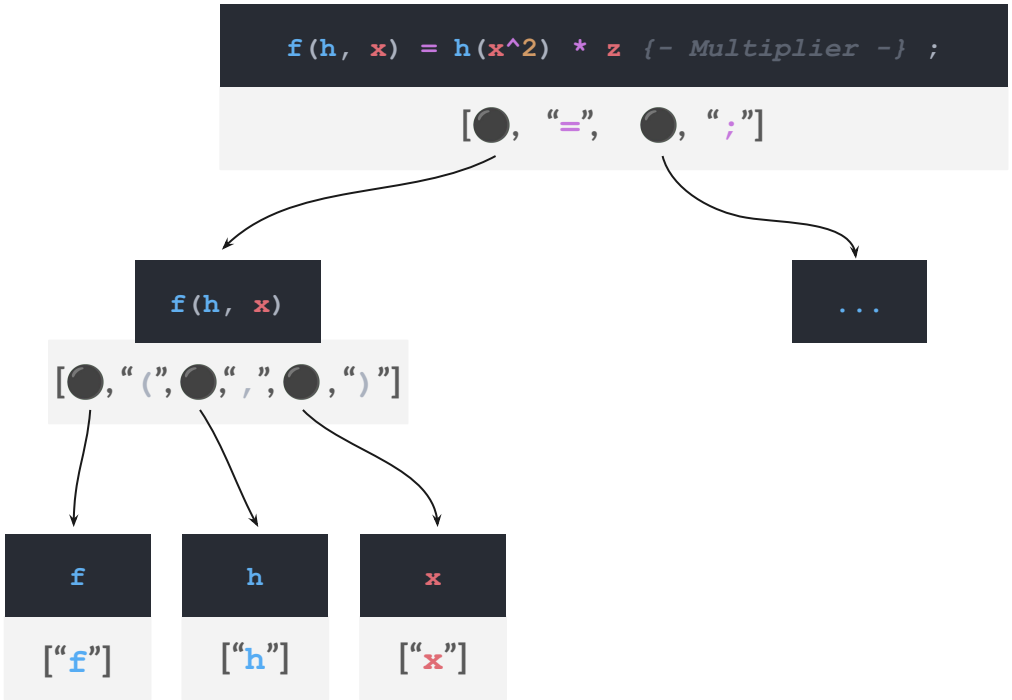
Anno: **Syntax Highlighting**

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  f(h, x) =
4    h(x^2) * z {- Multiplier -};
5
6  #plot for 0 < x . f(g, x) ;
7
8
```

Derive Syntax Highlighting

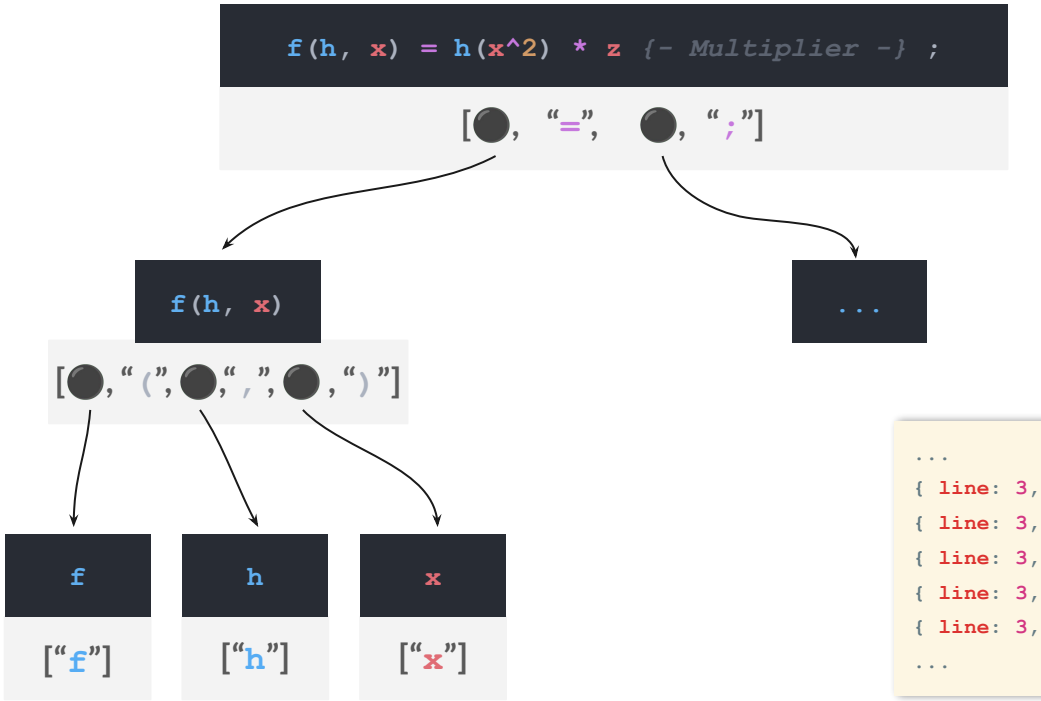
```
data Definition =
 MkDefinition Anno Name Expr

data Name
 = MkConstant Anno Var
 | MkFunc Anno Var [Var]

data Var = MkConstant Anno Text
data Expr
 = Mult Anno Expr Expr
 | ...
```

Anno: **Syntax Highlighting**




```
1  g(x) = 1 / x ;
2  z = 5 ;
3  f(h, x) =
4    h(x^2) * z {- Multiplier -};
5
6  #plot for 0 < x . f(g, x) ;
7
8
```

Anno: **Syntax Highlighting**




```
1  g(x) = 1 / x ;
2  z = 5 ;
3  f(h, x) =
4    h(x^2) * z {- Multiplier -};
5
6  #plot for 0 < x . f(g, x) ;
7
8
```

```
...
{ line: 3, startChar: 1, length: 1, tokenType: "function", ... },
{ line: 3, startChar: 2, length: 1, tokenType: "symbol",   ... },
{ line: 3, startChar: 3, length: 1, tokenType: "function", ... },
{ line: 3, startChar: 3, length: 1, tokenType: "symbol",   ... },
{ line: 3, startChar: 6, length: 1, tokenType: "variable", ... },
...
```

Syntax Highlighting

Looks great!

```
g(x) = 1 / x ;
z = 5 ;
f(h, x) =
  h(x^2) * z {- Multiplier -};

#plot for 0 < x . f(g, x) ;


```

But what happens if we **modify** the program?

Syntax Highlighting

Looks great!

But what happens if we **modify** the program?

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  incomplete =
4
5  f(h, x) =
6    h(x^2) * z {- Multiplier -};
7
8  #plot for 0 < x . f(g, x) ;
```

Syntax Highlighting

Suddenly, the syntax highlighting is gone.

Parser errors interrupt syntax highlighting.

That's unwanted and unnecessary,
can't we just reuse old results?

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  incomplete =
4
5  f(h, x) =
6    h(x^2) * z {- Multiplier -};
7
8  #plot for 0 < x . f(g, x) ;
```

Syntax Highlighting

Most of the document didn't change,
and could be highlighted just like before.

To get this, we need:

- Last value of the parsed AST
- Position Mapping
  - A function which tells us how the file has changed since the last time we parsed the program

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  incomplete =
4
5  f(h, x) =
6    h(x^2) * z {- Multiplier -};
7
8  #plot for 0 < x . f(g, x) ;
```

Building an Integrated Development Environment (IDE) on top of a Build System

The tale of a Haskell IDE

Neil Mitchell
Facebook
ndmitchell@gmail.com

Moritz Kiefer
Digital Asset
moritz.kiefer@purelyfunctional.org

Pepe Iborra
Facebook
pepeiborra@gmail.com

Luke Lau
Trinity College Dublin
luke_lau@icloud.com

Zubin Duggal
Chennai Mathematical Institute
zubin.duggal@gmail.com

Hannes Siebenhandl
TU Wien
hannes.siebenhandl@posteo.net

Javier Neira Sanchez
UNED, Spain
atreyu.bbb@gmail.com

Matthew Pickering
Well-Typed
matthewtpickering@gmail.com

Alan Zimmerman
Facebook
alan.zimm@gmail.com

Abstract

When developing a Haskell IDE we hit upon an idea – why not base an IDE on an build system? In this paper we'll explain how to go from that idea to a usable IDE, including the difficulties imposed by reusing a build system, and those imposed by technical details specific to Haskell. Our design has been successful, and hopefully provides a blue-print for others writing IDEs.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**.

ACM Reference Format:
Neil Mitchell, Moritz Kiefer, Pepe Iborra, Luke Lau, Zubin Duggal, Hannes Siebenhandl, Javier Neira Sanchez, Matthew Pickering, and Alan Zimmerman. 2020. Building an Integrated Development Environment (IDE) on top of a Build System: The tale of a Haskell IDE. In *Proceedings of Implementation and Application of Functional Languages (IFL'20).* ACM, New York, NY, USA, 12 pages.

applicable and natural for an IDE. The result is available as a project named *Haskell Language Server*[1] (HLS)[2].

In this paper we outline the core of our IDE §2, how it is fleshed out into an IDE component §3, and then how we build a complete IDE §4. We look at where the build system both helps and hurts §5. We then look at the ongoing and future work §6 before comparing to related work §7 and concluding §8.

2 Design

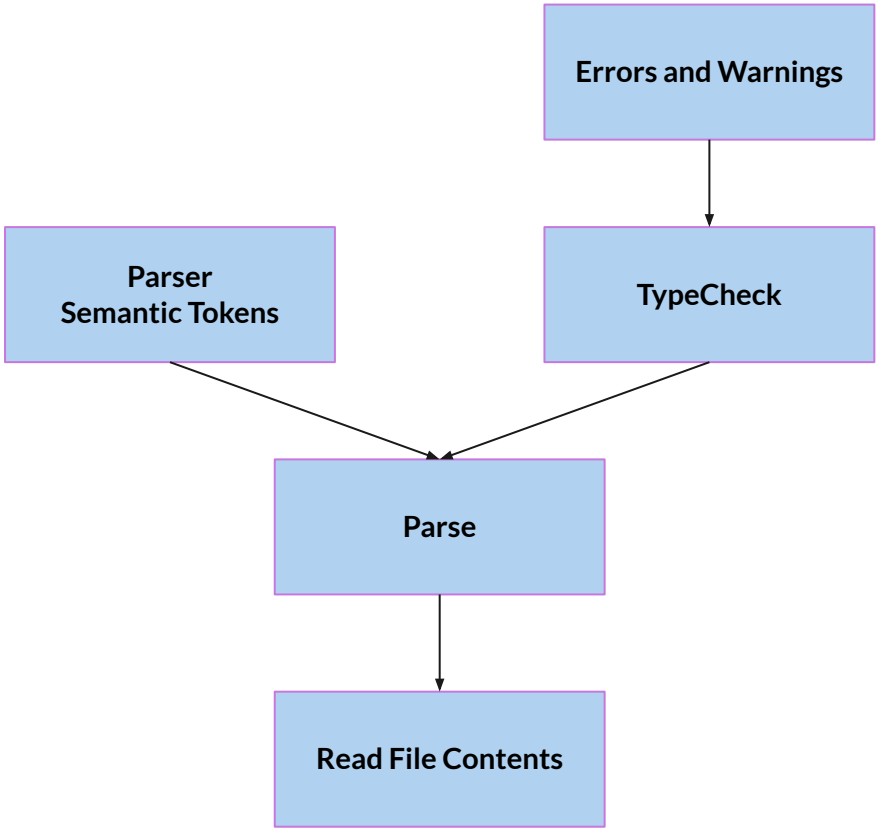
In this section we show how to implement an IDE on top of a build system. First we look at what an IDE provides, then what a build system provides, followed by how to combine the two.

2.1 Features on an IDE

To design an IDE, it is worth first reflecting on what features an IDE provides. In our view, the primary features of an

Build Graph

- Build graph natural fit for IDEs
  - Efficient caching of computations
  - Track dependencies between rules
- Rules
  - Can fail
  - Have output
- Maintain position mapping for each rule
  - Compared to last-good result



Syntax Highlighting using a Build Graph

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  f(h, x) =
4    h(x^2) * z {- Multiplier -};
5
6  #plot for 0 < x . f(g, x) ;
7
8
```

update positions

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  incomplete =
4
5  f(h, x) =
6    h(x^2) * z {- Multiplier -};
7
8  #plot for 0 < x . f(g, x) ;
```

```
...
{ line: 3, startChar: 1, length: 1, tokenType: "function", ... },
{ line: 3, startChar: 3, length: 1, tokenType: "function", ... },
{ line: 3, startChar: 6, length: 1, tokenType: "variable", ... },
...
```

```
...
{ line: 5, startChar: 1, length: 1, tokenType: "function", ... },
{ line: 5, startChar: 3, length: 1, tokenType: "function", ... },
{ line: 5, startChar: 6, length: 1, tokenType: "variable", ... },
...
```

Syntax Highlighting using a Build Graph

But still not enough!

❌ No highlighting for new code fragments.

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  incomplete =
4
5  f(h, x) =
6    h(x^2) * z {- Multiplier -};
7
8  #plot for 0 < x . f(g, x) ;
```

Syntax Highlighting using a Build Graph

But still not enough!

❌ No highlighting for new code fragments.

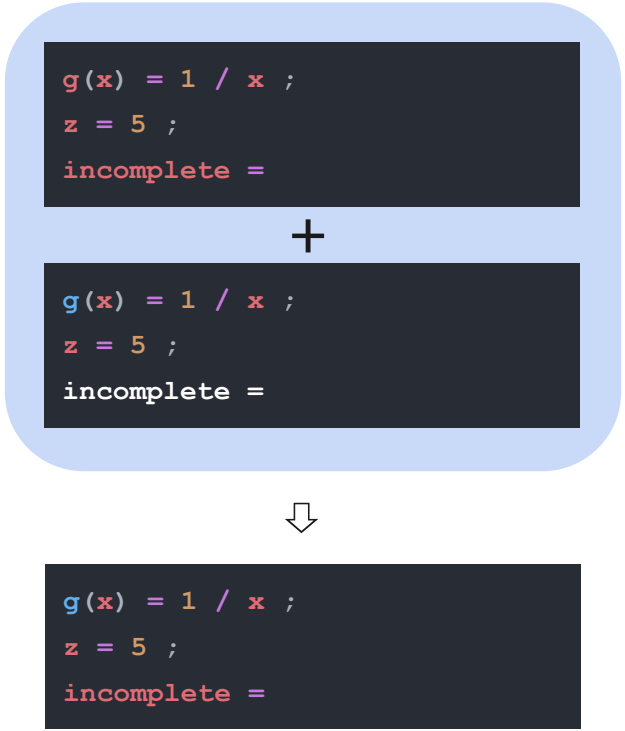
➜ Multi-phase syntax highlighting

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  incomplete =
4
5  f(h, x) =
6    h(x^2) * z {- Multiplier -};
7
8  #plot for 0 < x . f(g, x) ;
```

Syntax Highlighting using a Build Graph

We can generate semantic tokens for various phases in our compiler:

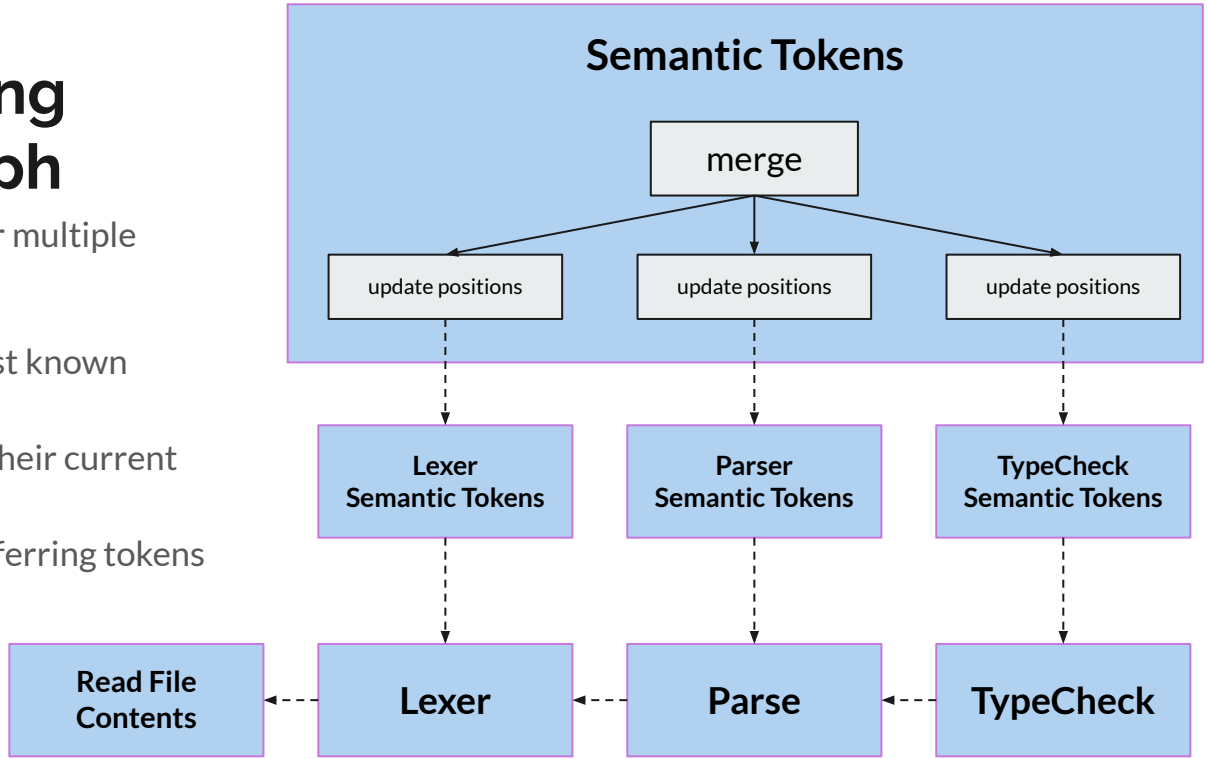
- Lexer
- Parser
- Type checking/analysis



Syntax Highlighting using a Build Graph

Combining the semantic tokens for multiple phases

1. Each phase computes the last known semantic tokens
2. Map all semantic tokens to their current location in the document
3. Merge semantic tokens, preferring tokens from later stages



Summary

LSP is a good fit for providing a modern IDE

A language server requires additional features from abstract syntax trees

- Source ranges
- Comments
- Source tokens

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  incomplete =
4
5  f(h, x) =
6    h(x^2) * z {- Multiplier -};
7
8  #plot for 0 < x . f(g, x) ;
```

Summary

Tools for implementing an IDE

- Anno
  - Stores parsed tokens that make up an AST node
  - References to child elements
  - Well-equipped to implement IDE features
- Build Graphs
  - Framework for caching computations
  - Position mapping
- Open source prototype in Haskell
  - https://github.com/smucclaw/l4-ide/
  - https://github.com/fendor/bobkonf

```
1  g(x) = 1 / x ;
2  z = 5 ;
3  incomplete =
4
5  f(h, x) =
6    h(x^2) * z {- Multiplier -};
7
8  #plot for 0 < x . f(g, x) ;
```