How to build your own programming language?

Prepared by Sam Zhou

Getting Started

git clone https://github.com/SamChou19815/dti-lang

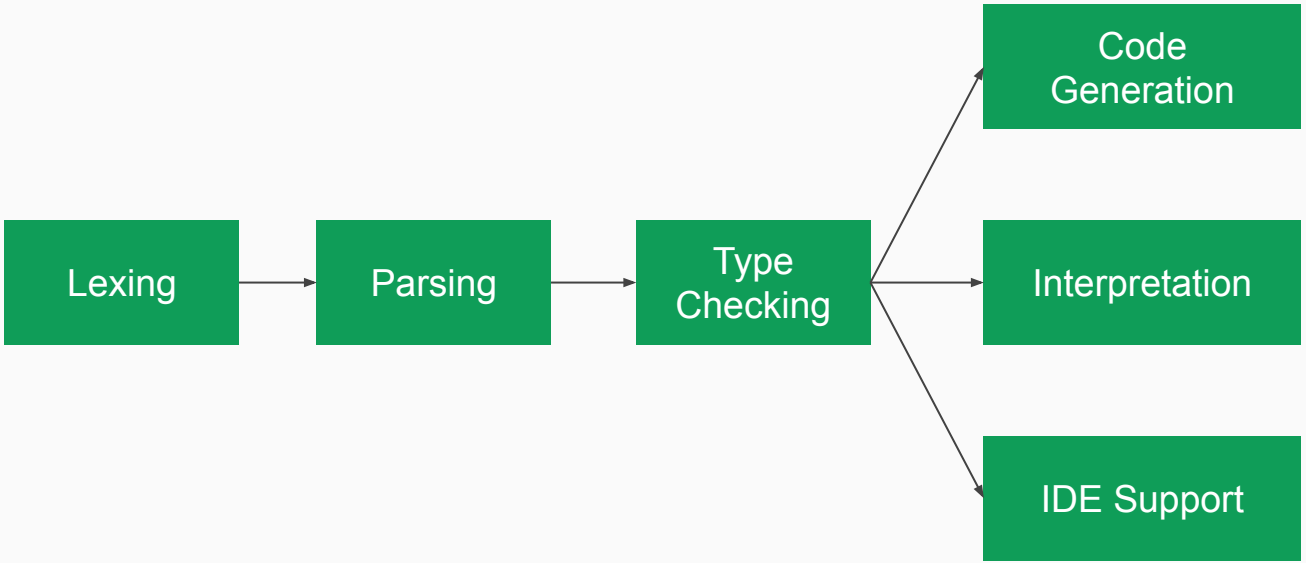
cd dti-lang

yarn

or

npm install

Background: Compiler Pipeline



Example: Lexing

```
console.log('Hello World');

// Comments to be ignored
```

⇒

IDENTIFIER(console)

DOT

IDENTIFIER(log)

LPAREN

STRING_LITERAL(Hello World)

RPAREN

SEMI

Example: Lexing

```
console ‘Hello World’ () . log;

// It can still be lexed!
```

⇒

IDENTIFIER(console)

STRING_LITERAL(Hello World)

LPAREN

RPAREN

DOT

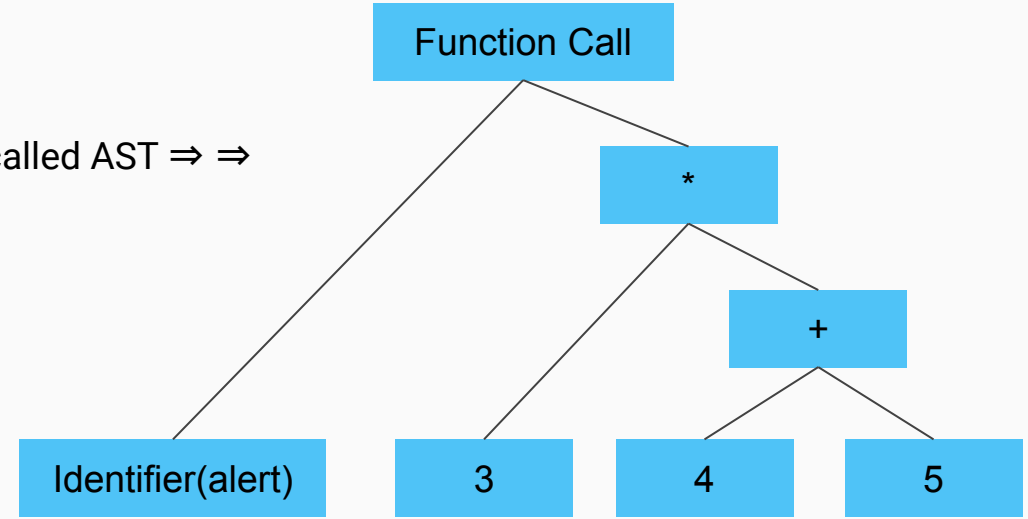
IDENTIFIER(log)

SEMI

Example: Parsing

alert(3 * (4 + 5)); ⇒

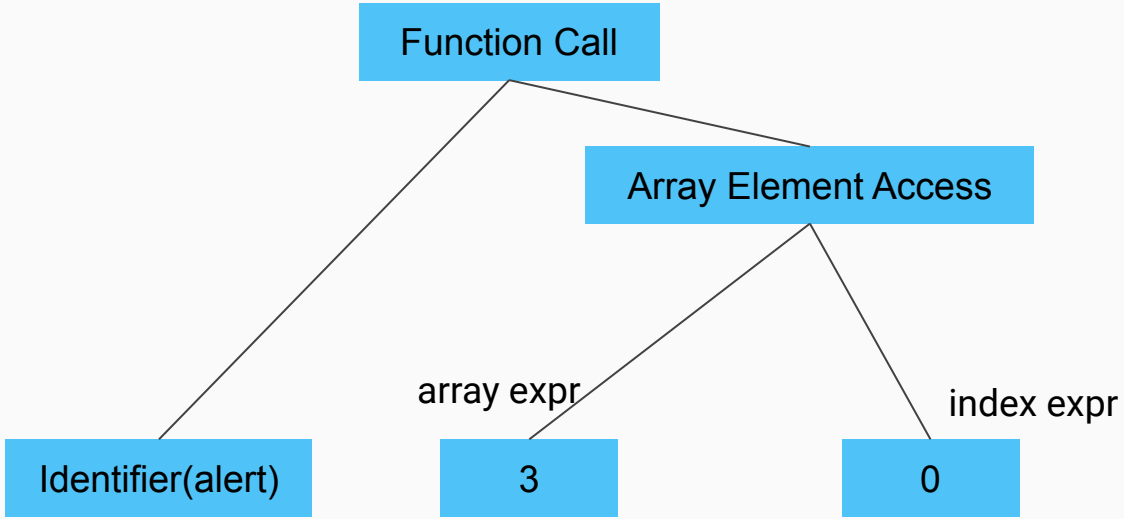
This is called AST ⇒ ⇒ ⇒



Example: Parsing

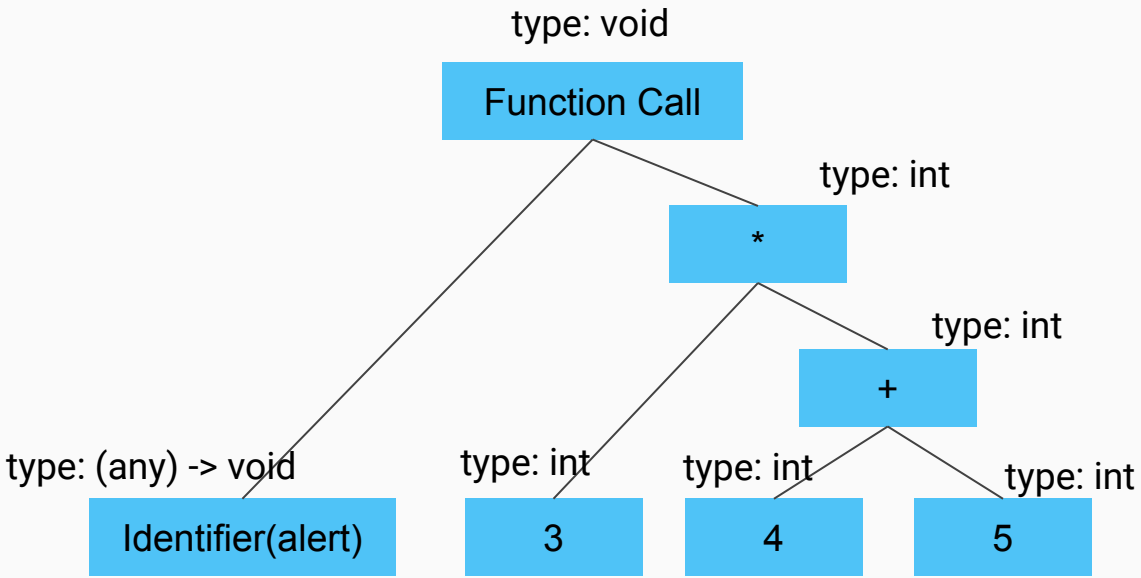
```
alert(3[0]);

// It can still be parsed!
```

⇒



Example: Type Checking

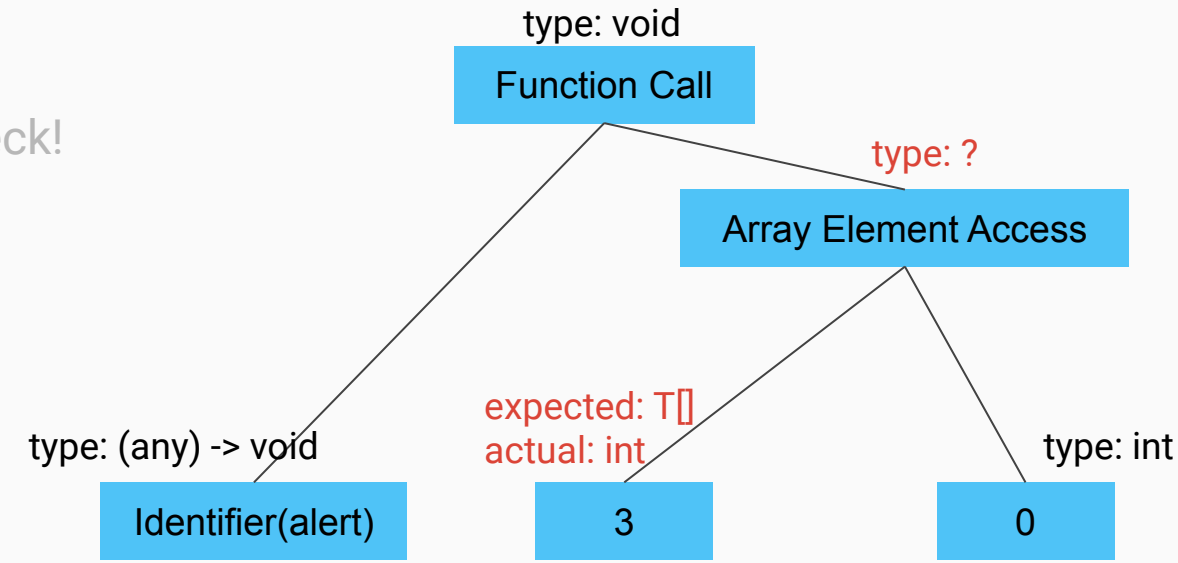
alert(3 * (4 + 5)); ⇒



Example: Type Checking

```
alert(3[0]);

// It now fails to type check!
```

⇒



Decorated AST

We usually want to add some extra stuff to AST beyond its program structure.

Examples:

- Line and column number
- Type information

Background: Lambda Calculus

Only three types of language constructs:

- Variable (x)
- Abstraction (λx.x)
  - (Think of arrow function in JS, lambda in Java)
  - Only allow one parameter! Currying can take care of the rest.
- Application ((λx.x) ((λx.x)))
  - (Think of function call)

Background: DTI-Lang

Extension of lambda calculus

Added support for int literal and add

Use our slack workspace emojis rather than normal English words

Lexing

Lexing: Lexer Generator

- Represent rules by regular expressions
  - e.g. a stream of characters that matches `[1-9][0-9]*` ⇒ int literal
- DFA ⇆ NFA ⇆ Regex
  - We can encode regex as DFA for more efficient tokenization
- Longest sequence wins
  - What if language has a keyword if and allows iffffff to be a variable name
  - What happens when we see iffffff?
  - It's classified as variable, longest sequence wins

Lexing: Lexer Generator

It looks complicated…

But you don’t write the code by hand.

You write Regex and let lexer generators generate code for you.

Lexer generators also give you line and column numbers of tokens.

Lexer Generator Example (ANTLR4)

```
FUN : ':octocat:';
INT: ':1e10:';
COMMA : ',';
LPAREN : '(';
RPAREN : ')';
COLON : ':';
PLUS : ':portalparrot:';
ARROW : ':dti:';

Identifier : Letter (Letter | Digit)*;
fragment Letter
  : ':evan-ooos:'
  | ':neha-dies-inside:'
  | ':devsam:'
  | ':pikachu-laura:'
  | ':emily-bakes:'
  | ':megan-disapproves:'
  ;
DecimalLiteral : ZeroDigit | NonZeroDigit (Digit)*;
fragment Digit : NonZeroDigit | ZeroDigit;
fragment NonZeroDigit : ':one:' | ':two:' | ':three:' | ':four:' | ':five:' | ':six:'
fragment ZeroDigit : ':zero:';

COMMENT : '/*' .*? '*/' -> channel(HIDDEN); // match anything between /* and */
WS : [ \r\t\u000C\n]+ -> channel(HIDDEN); // white space
LINE_COMMENT : '//' ~[\r\n]* '\r'? '\n' -> channel(HIDDEN);
```

```
// @Override
public get channelNames(): string[] { return PLLexer.channelNames; }

// @Override
public get modeNames(): string[] { return PLLexer.modeNames; }

public static readonly _serializedATN: string =
  "\x03\uC91D\uCABA\u058D\uAFBA\u4F53\u0607\uEA8B\uC241\x02\x0F\u0123\b\x01" +
  "\x04\x02\t\x02\x04\x03\t\x03\x04\x04\t\x04\x04\x05\t\x05\x04\x06\t\x06" +
  "\x04\x07\t\x07\x04\b\t\b\x04\t\t\t\x04\n\t\n\x04\v\t\v\x04\f\t\f\x04\r" +
  "\t\r\x04\x0E\t\x0E\x04\x0F\t\x0F\x04\x10\t\x10\x04\x11\t\x11\x04\x12\t" +
  "\x12\x03\x02\x03\x02\x03\x02\x03\x02\x03\x02\x03\x02\x03\x02\x03\x02\x03" +
  "\x02\x03\x02\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03\x03" +
  "\x04\x03\x04\x03\x05\x03\x05\x03\x06\x03\x06\x03\x07\x03\x07\x03\b\x03" +
  "\b\x03\b\x03\b\x03\b\x03\b\x03\b\x03\b\x03\b\x03\b\x03\b\x03\b\x03" +
  "\b\x03\b\x03\t\x03\t\x03\t\x03\t\x03\t\x03\t\x03\n\x03\n\x03\n\x07\nW" +
  "\n\n\f\n\x0E\nZ\v\n\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03" +
  "\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03" +
  "\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03" +
  "\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03" +
  "\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03" +
  "\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03" +
  "\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03\v\x03" +
  "\v\x03\v\x03\v\x03\v\x05\v\xB0\n\v\x03\f\x03\f\x03\f\x07\f\xB5\n\f\f\f" +
  "\x0E\f\xB8\v\f\x05\f\xBA\n\f\x03\r\x03\r\x05\r\xBE\n\r\x03\x0E\x03\x0E" +
  "\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E" +
  "\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E" +
  "\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E" +
  "\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E" +
  "\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E" +
  "\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x03\x0E\x05\x0E\xF6\n" +
  "\x0E\x03\x0F\x03\x0F\x03\x0F\x03\x0F\x03\x0F\x03\x0F\x03\x0F\x03\x10\x03" +
  "\x10\x03\x10\x03\x10\x07\x10\u0103\n\x10\f\x10\x0E\x10\u0106\v\x10\x03" +
  "\x10\x03\x10\x03\x10\x03\x10\x03\x10\x03\x11\x06\x11\u010E\n\x11\r\x11" +
  "\x0E\x11\u010F\x03\x11\x03\x11\x03\x12\x03\x12\x03\x12\x03\x12\x07\x12" +
  "\u0118\n\x12\f\x12\x0E\x12\u011B\v\x12\x03\x12\x05\x12\u011E\n\x12\x03" +
  "\x12\x03\x12\x03\x12\x03\u0104\x02\x02\x13\x03\x02\x03\x05\x02" +
  "\x04\x07\x02\x05\t\x02\x06\v\x02\x07\r\x02\b\x0F\x02\t\x11\x02\n\x13\x02" +
  "\v\x15\x02\x02\x17\x02\f\x19\x02\x02\x1B\x02\x02\x1D\x02\x02\x1F\x02\r" +
  "!\x02\x0E#\x02\x0F\x03\x02\x04\x05\x02\v\f\x0E\x0F\"\"\x04\x02\f\f\x0F" +
  "\x0F\x02\u0134\x02\x03\x03\x02\x02\x02\x02\x05\x03\x02\x02\x02\x02\x07" +
  "\x03\x02\x02\x02\x02\t\x03\x02\x02\x02\x02\v\x03\x02\x02\x02\x02\r\x03" +
  "\x02\x02\x02\x02\x0F\x03\x02\x02\x02\x02\x11\x03\x02\x02\x02\x02\x13\x03" +
  "\x02\x02\x02\x02\x17\x03\x02\x02\x02\x02\x1F\x03\x02\x02\x02\x02!\x03" +
  "\x02\x02\x02\x02#\x03\x02\x02\x02\x03%\x03\x02\x02\x02\x05/\x03\x02\x02" +
  "\x02\x076\x03\x02\x02\x02\t8\x03\x02\x02\x02\v:\x03\x02\x02\x02\r<\x03" +
  "\x02\x02\x02\x0F>\x03\x02\x02\x02\x11M\x03\x02\x02\x02\x13S\x03\x02\x02" +
  "\x02\x15\xAF\x03\x02\x02\x02\x17\xB9\x03\x02\x02\x02\x19\xBD\x03\x02\x02" +
  "\x02\x1B\xF5\x03\x02\x02\x02\x1D\xF7\x03\x02\x02\x02\x1F\xFE\x03\x02\x02" +
  "\x02!\u010D\x03\x02\x02\x02#\u0113\x03\x02\x02\x02%&\x07<\x02\x02&\'\x07" +
```

What you wrote (left) vs what's generated (right)

Exercise: add more emojis to language

Modify PLLexerPart.g4

Install VSCode extension “ANTLR4 grammar syntax support” for better dev experience

Run `yarn` or `npm install` again to regenerate lexer code.

Parsing

Parsing: Parser Generator

Top-down vs bottom up

Theory behind parsing is difficult. Take compiler if you are interested.

You don’t write parsing code by hand.

Parser Generator Example (ANTLR4)

```
grammar PL;

import PLLexerPart;

topLevel : expression EOF;

expression
  : LPAREN expression RPAREN # NestedExpression
  | Identifier # IdentifierExpression
  | DecimalLiteral # NumberLiteralExpression
  | expression PLUS expression # PlusExpression
  | FUN LPAREN Identifier COLON type RPAREN ARROW expression # LambdaExpression
  | expression LPAREN expression RPAREN # FunctionApplicationExpression
  ;

type
  : LPAREN type RPAREN # NestedType
  | INT # IntType
  | type ARROW type # FunctionType
  ;
```

```
case PLParser.DecimalLiteral:
  {
  _localctx = new NumberLiteralExpressionContext(_localctx);
  this._ctx = _localctx;
  _prevctx = _localctx;
  this.state = 15;
  this.match(PLParser.DecimalLiteral);
  }
  break;
case PLParser.FUN:
  {
  _localctx = new LambdaExpressionContext(_localctx);
  this._ctx = _localctx;
  _prevctx = _localctx;
  this.state = 16;
  this.match(PLParser.FUN);
  this.state = 17;
  this.match(PLParser.LPAREN);
  this.state = 18;
  this.match(PLParser.Identifier);
  this.state = 19;
  this.match(PLParser.COLON);
  this.state = 20;
  this.type(0);
  this.state = 21;
  this.match(PLParser.RPAREN);
  this.state = 22;
  this.match(PLParser.ARROW);
  this.state = 23;
  this.expression(2);
  }
  break;
default:
  throw new NoViableAltException(this);
}
this._ctx._stop = this._input.tryLT(-1);
this.state = 37;
this._errHandler.sync(this);
_alt = this.interpreter.adaptivePredict(this._input, 2, this._ctx);
while (_alt !== 2 && _alt !== ATN.INVALID_ALT_NUMBER) {
  if (_alt === 1) {
    if (this._parseListeners != null) {
      this.triggerExitRuleEvent();
    }
    _prevctx = _localctx;
    {
    this.state = 35;
    this._errHandler.sync(this);
    switch ( this.interpreter.adaptivePredict(this._input, 1, this._ctx) ) {
    case 1:
      {
      _localctx = new PlusExpressionContext(new ExpressionContext(_parentctx, _parentState));
      this.pushNewRecursionContext(_localctx, _startState, PLParser.RULE_expression);
      this.state = 27;
```

What you wrote (left) vs what's generated (right)

Parsing: Creating AST

ANTLR4 gives you parse tree.

Parse tree is awfully hard to work with.

We need to convert it into AST.

See src/ast.ts and src/parser.ts.

Exercise: support multiply expression

Modify PLLexerPart.g4 and PL.g4 to update lexer and parser.

Modify src/ast.ts to update AST.

Modify src/parser.ts to update parser adapter.

Modify src/checker.ts and src/interpreter.ts to handle new case.

Run `yarn` or `npm install` again to regenerate lexer code.

Type Checking and Interpretation

Interpretation

Making local computations based on AST node and value context.

Very recursive.

- Pattern matching in OCaml
- Visitor pattern in Java

See src/interpreter.ts.

Type Checking

Making local decisions based on AST node and typing context.

Very recursive.

Very similar to interpretation (for simple languages).

See src/checker.ts.

Advanced PL Topics

Recovery Parsing

Recovery Parsing: Rationale

We don’t want a syntax error to abort the entire pipeline.

Some IDE language services might work on unparsable ASTs.

Recovery Parsing: Implementation

```
class TypeVisitor extends AbstractParseTreeVisitor<ExpressionType>
  implements PLVisitor<ExpressionType> {
  defaultResult = (): ExpressionType => throwParserError();
```

Very easy in ANTLR4.

Replace the block with code that returns dummy AST node.

Type Inference

Type Inference: Introduction

Can we infer types even if user supplies zero type annotation?

Yes, we can.

However, it’s difficult.

Type Inference: Data Structure

Type inference is constraint solving.

Keeping track of type equality constraints in union-find data structure.

Type Inference: Example

if a(b + 1) then b else c

unknown_a

unknown_b+1

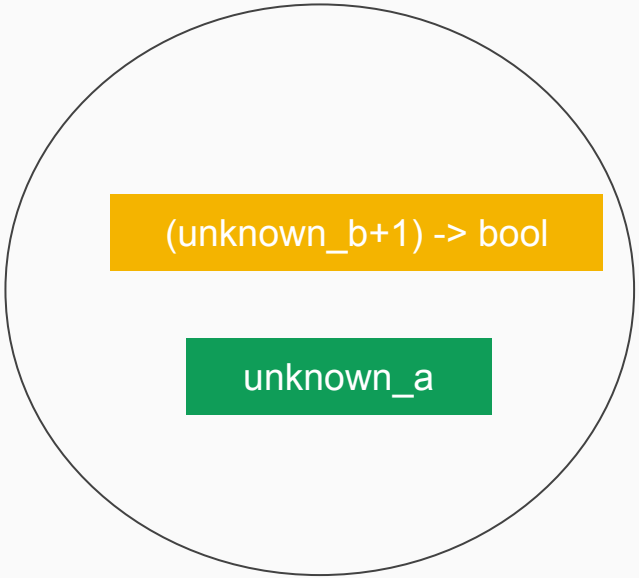
unknown_b

unknown_c

unknown_entire_exp

Type Inference: Example

if a(b + 1) then b else c



unknown_b+1

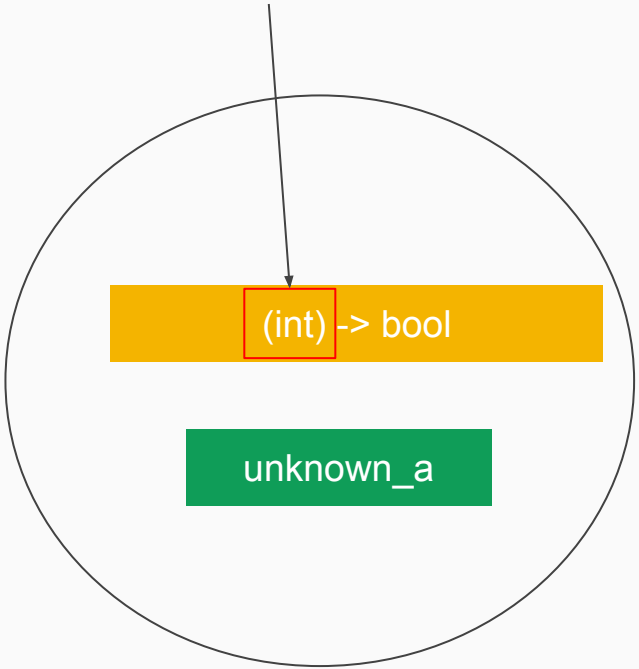
unknown_b

unknown_c

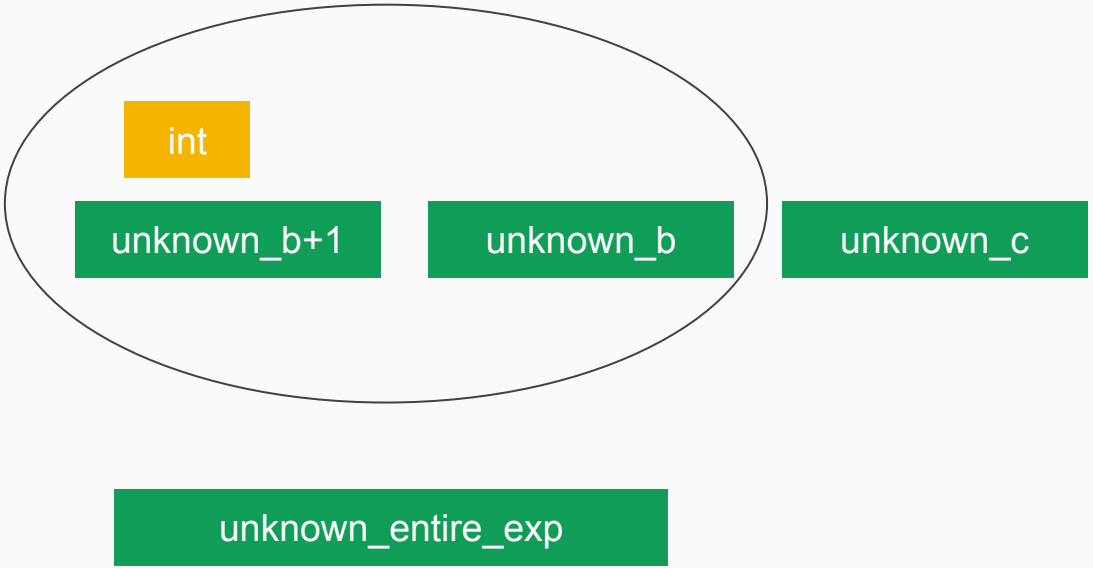
unknown_entire_exp

Type Inference: Example

unknown_b+1 is now resolved!

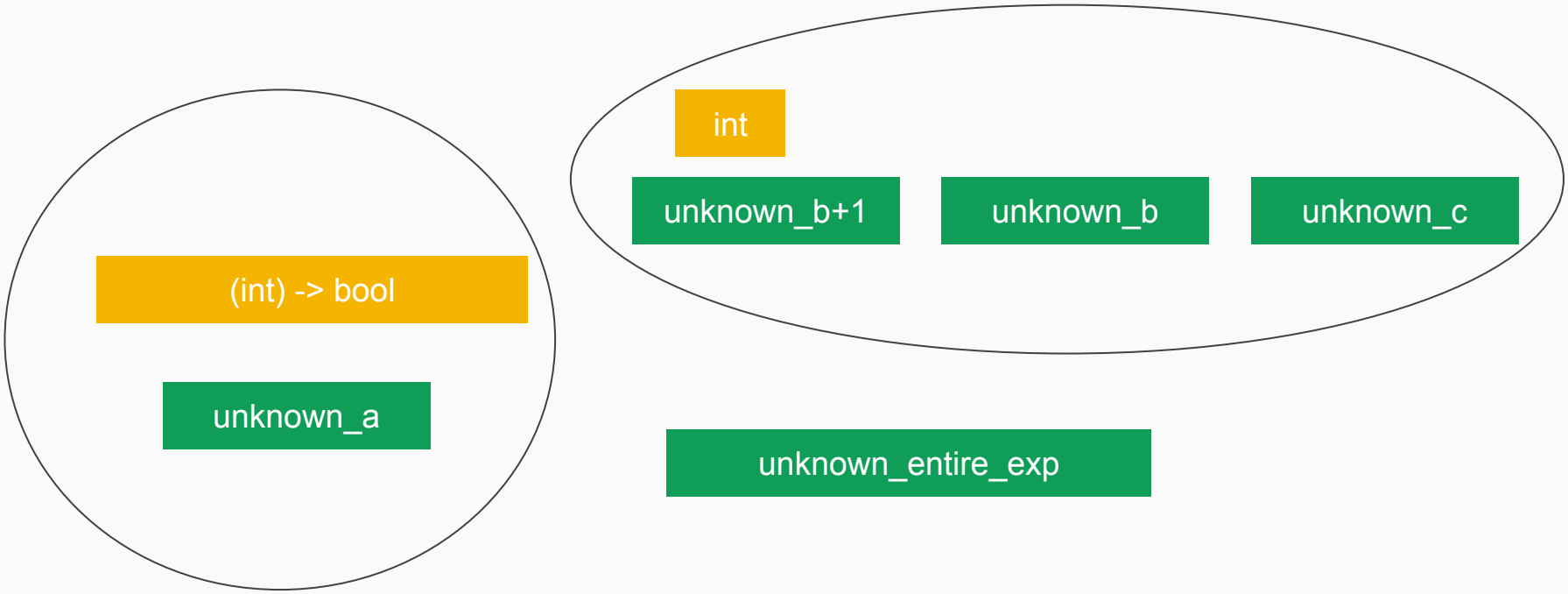


if a(b + 1) then b else c



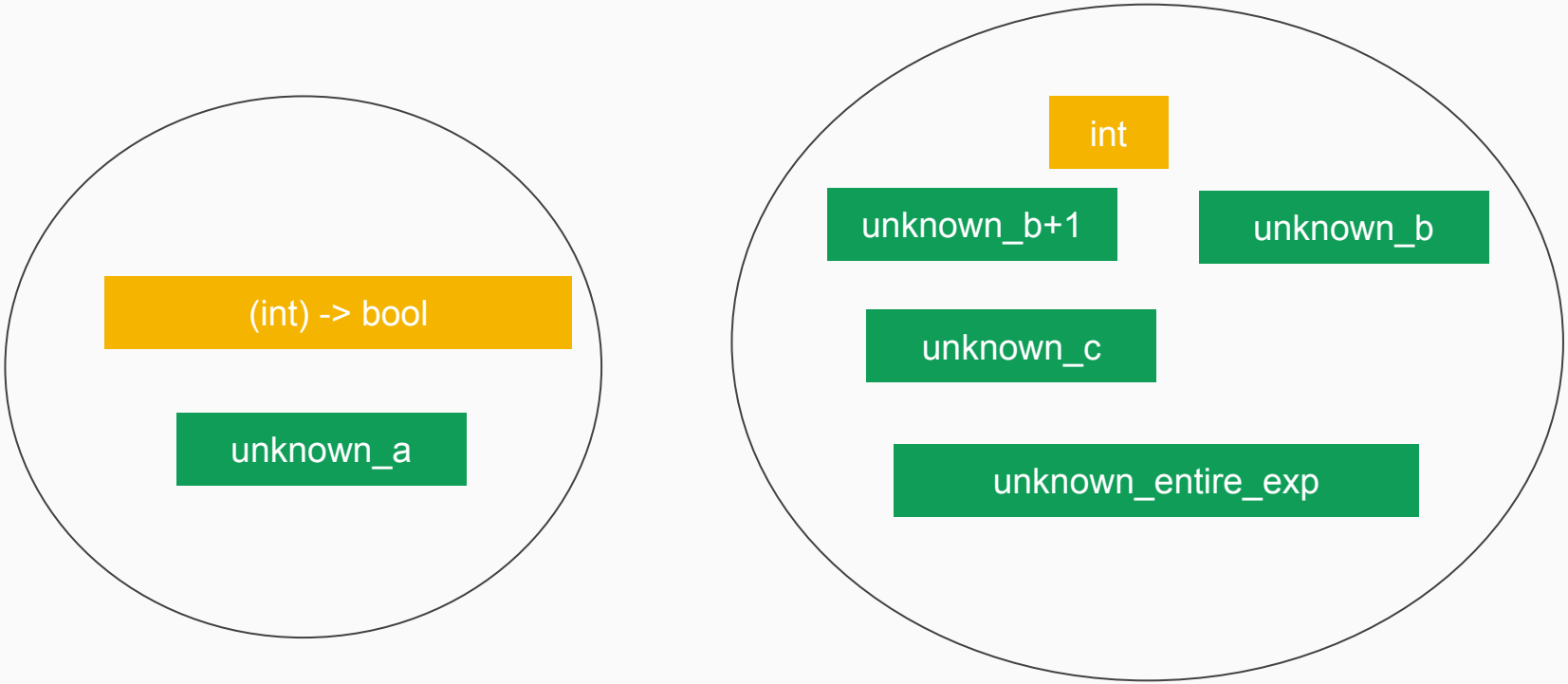
Type Inference: Example

if a(b + 1) then b else c



Type Inference: Example

```
if a(b + 1) then b else c
```



IDE and LSP

Definitions

Why LSP?

LSP creates the opportunity to reduce the *m-times-n* complexity problem of providing a high level of support for any programming language in any editor, IDE, or client endpoint to a simpler *m-plus-n* problem.

For example, instead of the traditional practice of building a Python plugin for VSCode, a Python plugin for Sublime Text, a Python plugin for Vim, a Python plugin for Sourcegraph, and so on, for every language, LSP allows language communities to concentrate their efforts on a single, high performing language server that can provide code completion, hover tooltips, jump-to-definition, find-references, and more, while editor and client communities can concentrate on building a single, high performing, intuitive and idiomatic extension that can communicate with *any* language server to instantly provide deep language support.

The problem: "The Matrix"

	Go	Java	TypeScript	...
Emacs				
Vim				
VSCode				
...				

The solution: lang servers and clients

Go	✔
Java	✔
TypeScript	✔
...	

Emacs	✔
Vim	✔
VSCode	✔
...	

State of LSP

VSCode 😎

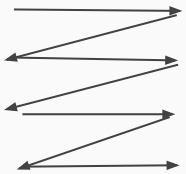
Atom 👍

JetBrains IDEs 😱😜

LSP: Server and Client

Client

- Run some command to start server.
- Send some text document updates
- Send server type query request
- Send server autocompletion request



Server

- Tell client what it can do in a json.
- Save updates in memory
- Send back type query response
- Send back completion items

VSCode Extension

- Define some language syntax
- Define some basic language constructs (keywords, comments, etc)
- Provide a language client
  - VSCode has a good library for language client, don’t need to write a lot
  - In the client code, tell VSCode how to start a server

VSCode Extension: Example

Open the project in vscode

NPM or Yarn install vsce globally

Run yarn package or npm run package.

Run F5 in VSCode

Play with it

Implement Type Query & Autocomplete

https://blog.developersam.com/2020/01/09/implement-autocomplete

Poll

https://forms.gle/8RZr4jvXSVL67Mn9A

That’s it.