

# **Lecture 3: Syntactic Editor Services**

**CS4200 Compiler Construction**

**Eelco Visser**

**TU Delft**

**September 2018**

# This Lecture

## Lexical syntax

- defining the syntax of tokens / terminals including layout
- making lexical syntax and layout explicit

## Syntactic editor services

- more interpretations of syntax definitions

## Formatting specification

- how to map (abstract syntax) trees to text

## Syntactic completion

- proposing valid syntactic completions in an editor

# Reading Material

The inverse of parsing is unparsing or pretty-printing or formatting, i.e. mapping a tree representation of a program to a textual representation. A plain context-free grammar can be used as specification of an unparser. However, then it is unclear where the whitespace should go.

This paper extends context-free grammars with templates that provide hints for layout of program text when formatting.

Based on master's thesis project of Tobi Vollebregt

<https://doi.org/10.1145/2427048.2427056>

## Declarative Specification of Template-Based Textual Editors

Tobi Vollebregt  
tobivollebregt@gmail.com

Lennart C. L. Kats  
l.c.l.kats@tudelft.nl

Eelco Visser  
visser@acm.org

Software Engineering Research Group  
Delft University of Technology  
The Netherlands

### ABSTRACT

Syntax discoverability has been a crucial advantage of structure editors for new users of a language. Despite this advantage, structure editors have not been widely adopted. Based on immediate parsing and analyses, modern textual code editors are also increasingly syntax-aware: structure and textual editors are converging into a new editing paradigm that combines text and templates. Current text-based language workbenches require redundant specification of the ingredients for a template-based editor, which is detrimental to the quality of syntactic completion, as consistency and completeness of the definition cannot be guaranteed.

In this paper we describe the design and implementation of a specification language for syntax definition based on templates. It unifies the specification of parsers, unparsers and template-based editors. We evaluate the template language by application to two domain-specific languages used for tax benefits and mobile applications.

### Categories and Subject Descriptors

D.2.3 [Software Engineering]: Coding Tools and Techniques—*pretty printers, program editors*; D.3.1 [Programming Languages]: Processors—*parsing, code generation*; D.3.1 [Programming Languages]: Formal Definitions and Theory—*syntax*; D.2.6 [Software Engineering]: Programming Environments; D.2.11 [Software Architectures]: Languages

### 1. INTRODUCTION

Language-aware structure editors provide a template-based paradigm for editing programs. They allow composing programs by selecting a template and filling in the placeholders, which can again be extended using templates. A crucial advantage of structure editors is syntax discoverability, helping new users to learn a language by presenting possible syntactic completions in a menu. Structure editors can be automatically generated from a syntax definition. Notable projects aiming at automatic generation of structure editors include MPS [23] and the Intentional Domain Work-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LDTA 2012 Tallinn, Estonia

Copyright 2012 ACM 978-1-4503-1536-4 ...\$15.00.

bench [19]. Structure editors can be used for general-purpose languages or for domain-specific languages (DSLs). A modern example of the former is the structure editor in MPS [23], for extensible languages based on Java. An example of the latter category is the DSL for modeling tax-benefit rules developed by IT services company Capgemini using the Cheetah system. Cheetah's facilities for discoverability and the use of templates are particularly effective to aid a small audience of domain expert programmers manage the verbose syntax based on legal texts.

Despite their good support for discoverability, structure editors have not been widely adopted. Pure structure editors tend to introduce an increased learning curve for basic editing operations. For example, they only support copy-pasting operations that maintain well-formedness of the tree and require small, yet non-trivial “refactoring” operations for editing existing code, e.g. when converting an *if* statement to an *if-else* statement. They also lack integration with other tools and expose the user to vendor lock-in. Transferring code across tools requires a shared representation that is generally not available. With software engineering tools such as issue trackers, forums, search, and version control being based on text, a textual representation is preferable, but requires the use of a parser and a parseable language syntax. This forces tools based on structure editors to find new solutions to problems long solved in the text domain.

To alleviate the problems of structure editors, there has been a long history of hybrid structure editors that introduce textual editing features to structure editors [24, 18, 11]. Conversely, modern textual code editors such as those in Eclipse and Visual Studio are increasingly syntax-aware, based on parsers that run while a program is edited. Over time, they have acquired features ranging from code folding to syntactic completions allowing programmers to fill in textual templates. Indeed, structure and textual editors are converging into a new editing paradigm that combines text and templates.

In order to provide the advantages of text editing to the tax-benefit DSL of Capgemini, we converted the language from the Cheetah system, which uses a structure editor, to the parser-based Spoofax language workbench [10]. We quickly realized that it would be impossible for a user to write new models in such a verbose language in the textual editor of Spoofax, without accurate and complete syntax discovery. In syntax-aware text editors this discovery is provided in the form of syntactic completion. Accurate and complete syntactic completion depends critically on two features of a language workbench: first, the syntactic completion proposals presented to the user must be relevant and complete, and second, it must be feasible to create and maintain the specification necessary to make the editor aware of these completion proposals.

Syntax definitions cannot be used just for parsing, but for many other operations. This paper shows how syntactic completion can be provided generically given a syntax definition.

## Part of PhD thesis work Eduardo Amorim

<https://doi.org/10.1145/2427048.2427056>

## Principled Syntactic Code Completion using Placeholders



Luís Eduardo de Souza Amorim Sebastian Erdweg Guido Wachsmuth Eelco Visser  
Delft University of Technology, Netherlands  
l.e.desouzaamorim-1@tudelft.nl, s.t.erdweg@tudelft.nl, guwac@acm.org, e.visser@tudelft.nl

### Abstract

Principled syntactic code completion enables developers to change source code by inserting code templates, thus increasing developer efficiency and supporting language exploration. However, existing code completion systems are ad-hoc and neither complete nor sound. They are not complete and only provide few code templates for selected programming languages. They also are not sound and propose code templates that yield invalid programs when inserted. This paper presents a generic framework that automatically derives complete and sound syntactic code completion from the syntax definition of arbitrary languages. A key insight of our work is to provide an explicit syntactic representation for incomplete programs using placeholders. This enables us to address the following challenges for code completion separately: (i) completing incomplete programs by replacing placeholders with code templates, (ii) injecting placeholders into complete programs to make them incomplete, and (iii) introducing lexemes and placeholders into incorrect programs through error-recovery parsing to make them correct so we can apply one of the previous strategies. We formalize our framework and provide an implementation in Spooftax.

**Categories and Subject Descriptors** D.2.6 [Software Engineering]: Programming Environments

**Keywords** Code Completion, Language Workbenches, IDEs

### 1. Introduction

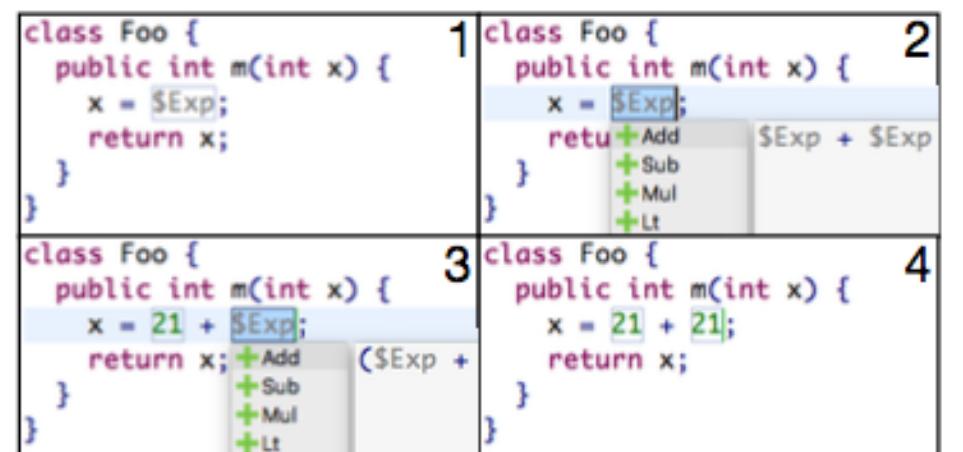
*Code completion*, also known as content completion or content assist, is an editor service that proposes and performs expansion of the program text. Code completion helps the programmer to avoid misspellings and acts as a guide to discover



This work is licensed under a Creative Commons Attribution International 4.0 License.

Copyright is held by the owner/author(s).

SLE'16, October 31 – November 1, 2016, Amsterdam, Netherlands  
ACM, 978-1-4503-4447-0/16/10..\$15.00  
<http://dx.doi.org/10.1145/2997364.2997374>



**Figure 1.** (1) Incomplete program with explicit placeholders. (2) Triggering completion for a placeholder. (3) After selecting a proposal, showing completions for nested placeholders. (4) Completing a nested placeholder by typing.

language features and APIs. Most mainstream integrated development environments (IDEs) provide some form of code completion and industrial studies indicate that code completion is one of the most frequently used IDE services [1].

There are two classes of code completion: syntactic and semantic. Syntactic code completion considers the syntactic context at the cursor position and proposes code templates for syntactic structures of the language. For example, most IDEs for Java support syntactic code completion with class and method templates. Semantic code completion also uses the cursor position to propose templates, but by applying semantic analysis to the program, the IDE can propose code templates or identifiers that do not violate the language's name binding or typing rules. For example, in this case IDEs for Java may suggest variables or methods that are visible in the current scope and have the expected type at the cursor position.

In this paper, we focus on *syntactic code completion*. Even for mainstream languages in mainstream IDEs, syntactic code completion is often ad-hoc and unreliable. Specifically, most existing services for syntactic code completion are incomplete and only propose code templates for selected language constructs of a few supported languages, thus inhibiting exploring the language's syntax. Moreover, most existing services are unsound and propose code templates that yield syntax errors when inserted at the cursor position.

The SDF3 syntax definition formalism is documented at the [metaborg.org](http://www.metaborg.org) website.

The screenshot shows the Spoofax Language Workbench documentation interface. At the top, there's a header with the logo and the word "latest". Below it is a search bar labeled "Search docs". The main content area has a sidebar on the left with several sections: "The Spoofax Language Workbench", "Examples", "Publications", "TUTORIALS" (which is highlighted in blue), "REFERENCE MANUAL", and "Transformation with Stratego". Under "TUTORIALS", there are links to "Installing Spoofax", "Creating a Language Project", "Using the API", and "Getting Support". Under "REFERENCE MANUAL", there are links to "Language Definition with Spoofax", "Abstract Syntax with ATerms", and "Syntax Definition with SDF3". The "Syntax Definition with SDF3" section is expanded, showing a numbered list from 1 to 7: "1. SDF3 Overview", "2. SDF3 Reference Manual", "3. SDF3 Examples", "4. SDF3 Configuration", "5. Migrating SDF2 grammars to SDF3 grammars", "6. Generating Scala case classes from SDF3 grammars", and "7. SDF3 Bibliography".

[Docs](#) » Syntax Definition with SDF3

[Edit on GitHub](#)

## Syntax Definition with SDF3

The definition of a textual (programming) language starts with its syntax. A grammar describes the well-formed sentences of a language. When written in the grammar language of a parser generator, such a grammar does not just provide such a description as documentation, but serves to generate an implementation of a parser that recognizes sentences in the language and constructs a parse tree or abstract syntax tree for each valid text in the language. **SDF3** is a *syntax definition formalism* that goes much further than the typical grammar languages. It covers all syntactic concerns of language definitions, including the following features: support for the full class of context-free grammars by means of generalized LR parsing; integration of lexical and context-free syntax through scannerless parsing; safe and complete disambiguation using priority and associativity declarations; an automatic mapping from parse trees to abstract syntax trees through integrated constructor declarations; automatic generation of formatters based on template productions; and syntactic completion proposals in editors.

### Table of Contents

- [1. SDF3 Overview](#)
- [2. SDF3 Reference Manual](#)
- [3. SDF3 Examples](#)
- [4. SDF3 Configuration](#)
- [5. Migrating SDF2 grammars to SDF3 grammars](#)
- [6. Generating Scala case classes from SDF3 grammars](#)
- [7. SDF3 Bibliography](#)

# Tiger Lexical Syntax

# Tiger Lexical Syntax: Identifiers

module Identifiers

lexical syntax

$\text{Id} = [\text{a-zA-Z}] [\text{a-zA-Z0-9}\backslash\_]^*$

lexical restrictions

# Tiger Lexical Syntax: Number Literals

```
module Numbers

lexical syntax

IntConst = [0-9]++

lexical syntax

RealConst.RealConstNoExp = IntConst "." IntConst
RealConst.RealConst = IntConst "." IntConst "e" Sign IntConst
Sign = "+"
Sign = "-"

context-free syntax

Exp.Int = IntConst
```

# Tiger Lexical Syntax: String Literals

```
module Strings

sorts StrConst

lexical syntax

StrConst = "\""" StrChar* "\""""
StrChar  = ~[\\\"\\n]
StrChar  = [\\n] [n]
StrChar  = [\\n] [t]
StrChar  = [\\n] [\^] [A-Z]
StrChar  = [\\n] [0-9] [0-9] [0-9]
StrChar  = [\\n] ["]
StrChar  = [\\n] [\\n]
StrChar  = [\\n] [\ \t\n]+ [\\n]

context-free syntax // records

Exp.String = StrConst
```

# Tiger Lexical Syntax: Whitespace

```
module Whitespace

lexical syntax

  LAYOUT = [\ \t\n\r]

context-free restrictions

  // Ensure greedy matching for comments

  LAYOUT? -/- [\ \t\n\r]
  LAYOUT? -/- [V].[V]
  LAYOUT? -/- [V].[\*]
```

```
syntax

  LAYOUT-CF = LAYOUT-LEX
  LAYOUT-CF = LAYOUT-CF LAYOUT-CF {left}
```

Implicit composition of layout

# Tiger Lexical Syntax: Comment

## lexical syntax

```
CommentChar      = [\*]
LAYOUT          = /* InsideComment* */
InsideComment    = ~[\*]
InsideComment    = CommentChar
```

## lexical restrictions

```
CommentChar  -/- [\\]
```

## context-free restrictions

```
LAYOUT? -/- [\\].[\*]
```

## lexical syntax

```
LAYOUT          = SingleLineComment
SingleLineComment = // ~[\n\r]* NewLineEOF
NewLineEOF       = [\n\r]
NewLineEOF       = EOF
EOF              =
```

## lexical restrictions

```
EOF    -/- ~[]
```

## context-free restrictions

```
LAYOUT? -/- [\\].[\\]
```

# Desugaring Lexical Syntax

# Explication of Lexical Syntax

## Core language

- context-free grammar productions
- with constructors
- only character classes as terminals
- explicit definition of layout

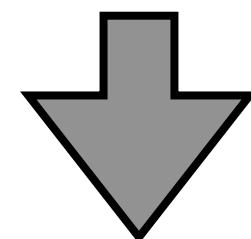
## Desugaring

- express lexical syntax in terms of character classes
- explicate layout between context-free syntax symbols
- separate lexical and context-free syntax non-terminals

# Explication of Layout by Transformation

context-free syntax

```
Exp.Int      = IntConst
Exp.Uminus   = "-" Exp
Exp.Times    = Exp "*" Exp {left}
Exp.Divide   = Exp "/" Exp {left}
Exp.Plus     = Exp "+" Exp {left}
```



Symbols in context-free syntax are implicitly separated by optional layout

syntax

```
Exp-CF.Int    = IntConst-CF
Exp-CF.Uminus = "-" LAYOUT?-CF Exp-CF
Exp-CF.Times   = Exp-CF LAYOUT?-CF "*" LAYOUT?-CF Exp-CF {left}
Exp-CF.Divide  = Exp-CF LAYOUT?-CF "/" LAYOUT?-CF Exp-CF {left}
Exp-CF.Plus    = Exp-CF LAYOUT?-CF "+" LAYOUT?-CF Exp-CF {left}
```

# Separation of Lexical and Context-free Syntax

lexical syntax

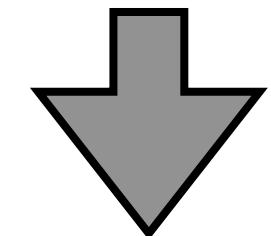
Id = [a-zA-Z] [a-zA-Z0-9\\_]\*

Id = "if" {reject}

Id = "then" {reject}

context-free syntax

Exp.Var = Id



syntax

Id-LEX = [\u0065-\u0090\u0097-\u00122] [\u0048-\u0057\u0065-\u0090\u0095\u0097-\u00122]\*-LEX

Id-LEX = "if" {reject}

Id-LEX = "then" {reject}

Id-CF = Id-LEX

Exp-CF.Var = Id-CF

# Why Separation of Lexical and Context-Free Syntax?

lexical syntax

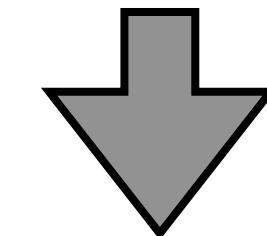
Id = [a-zA-Z] [a-zA-Z0-9\\_]\*

Id = "if" {reject}

Id = "then" {reject}

context-free syntax

Exp.Var = Id



syntax

Id = [\u0065-\u0090\u0097-\u00122] [\u0048-\u0057\u0065-\u0090\u0095\u0097-\u00122]\*

Id = "if" {reject}

Id = "then" {reject}

Exp.Var = Id

Homework: what would go wrong if we not do this?

## syntax

```
"if"  = [\105] [\102]
"then" = [\116] [\104] [\101] [\110]
```

character classes  
as only terminals

```
[\48-\57\65-\90\95\97-\122]+-LEX = [\48-\57\65-\90\95\97-\122]
[\48-\57\65-\90\95\97-\122]+-LEX = [\48-\57\65-\90\95\97-\122]+-LEX [\48-\57\65-\90\95\97-\122]
[\48-\57\65-\90\95\97-\122]*-LEX =
[\48-\57\65-\90\95\97-\122]*-LEX = [\48-\57\65-\90\95\97-\122]+-LEX
```

```
Id-LEX = [\65-\90\97-\122] [\48-\57\65-\90\95\97-\122]*-LEX
```

```
Id-LEX = "if" {reject}
```

```
Id-LEX = "then" {reject}
```

```
Id-CF = Id-LEX
```

separate lexical and  
context-free syntax

```
Exp-CF.Var = Id-CF
```

```
Exp-CF.Call = Exp-CF LAYOUT?-CF Exp-CF {left}
```

```
Exp-CF.IfThen = "if" LAYOUT?-CF Exp-CF LAYOUT?-CF "then" LAYOUT?-CF Exp-CF
```

```
LAYOUT-CF = LAYOUT-CF LAYOUT-CF {left}
```

```
LAYOUT?-CF = LAYOUT-CF
```

```
LAYOUT?-CF =
```

## restrictions

```
Id-LEX -/- [\48-\57\65-\90\95\97-\122]
"if" -/- [\48-\57\65-\90\95\97-\122]
"then" -/- [\48-\57\65-\90\95\97-\122]
```

separate context-  
free symbols by  
optional layout

## priorities

```
Exp-CF.Call left Exp-CF.Call,
```

```
LAYOUT-CF = LAYOUT-CF LAYOUT-CF left LAYOUT-CF = LAYOUT-CF LAYOUT-CF
```

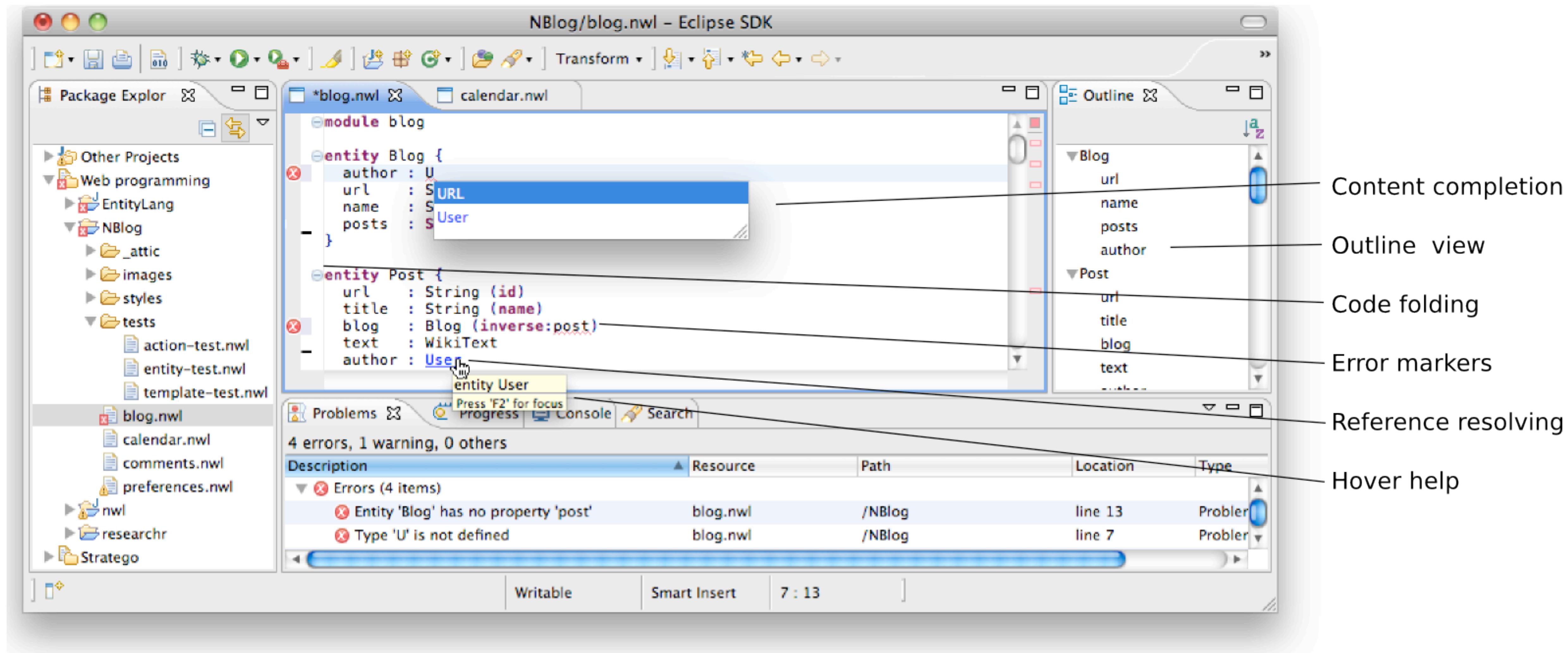
lexical syntax  
 $\text{Id} = [\text{a-zA-Z}][\text{a-zA-Z0-9}\_]^*$   
 $\text{Id} = \text{"if"} \{\text{reject}\}$   
 $\text{Id} = \text{"then"} \{\text{reject}\}$

lexical restrictions  
 $\text{Id} \sim [\text{a-zA-Z0-9}\_]$   
 $\text{"if"} \text{ "then"} \sim [\text{a-zA-Z0-9}\_]$

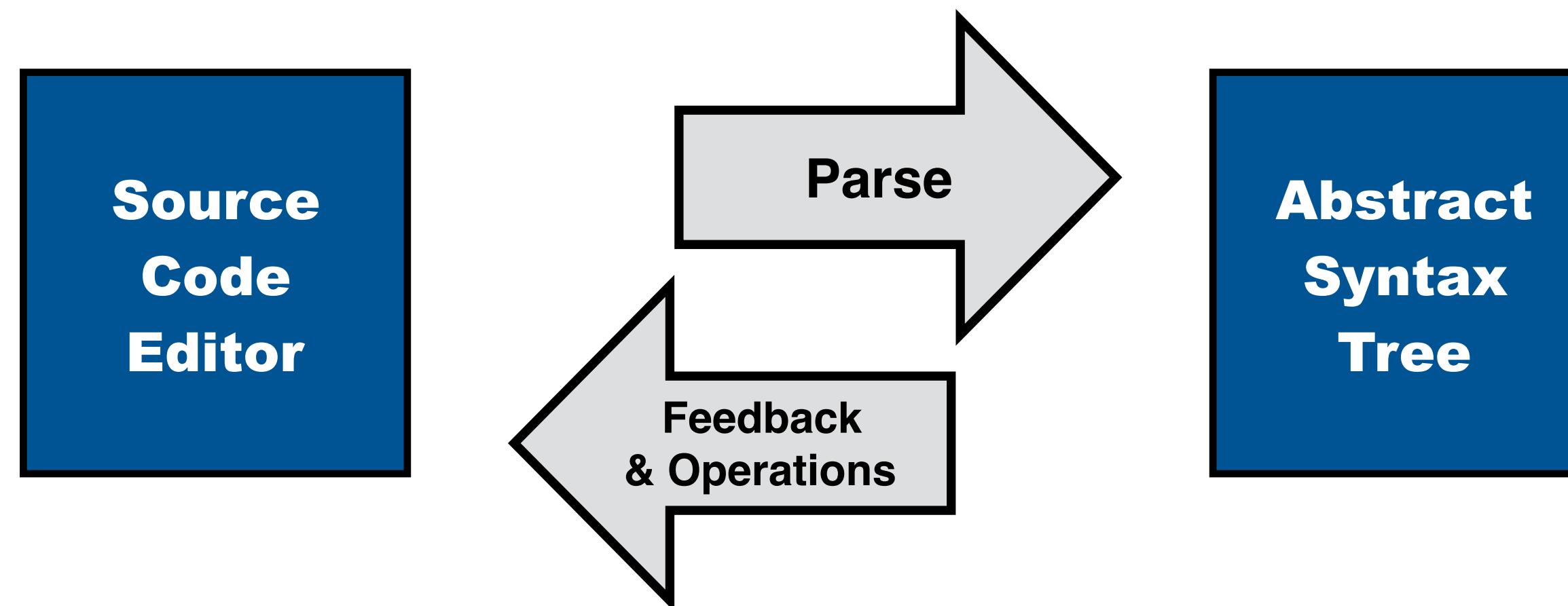
context-free syntax  
 $\text{Exp.Var} = \text{Id}$   
 $\text{Exp.Call} = \text{Exp Exp} \{\text{left}\}$   
 $\text{Exp.IfThen} = \text{"if"} \text{ Exp} \text{ "then"} \text{ Exp}$

# **Syntactic Editor Services**

# Editor Services



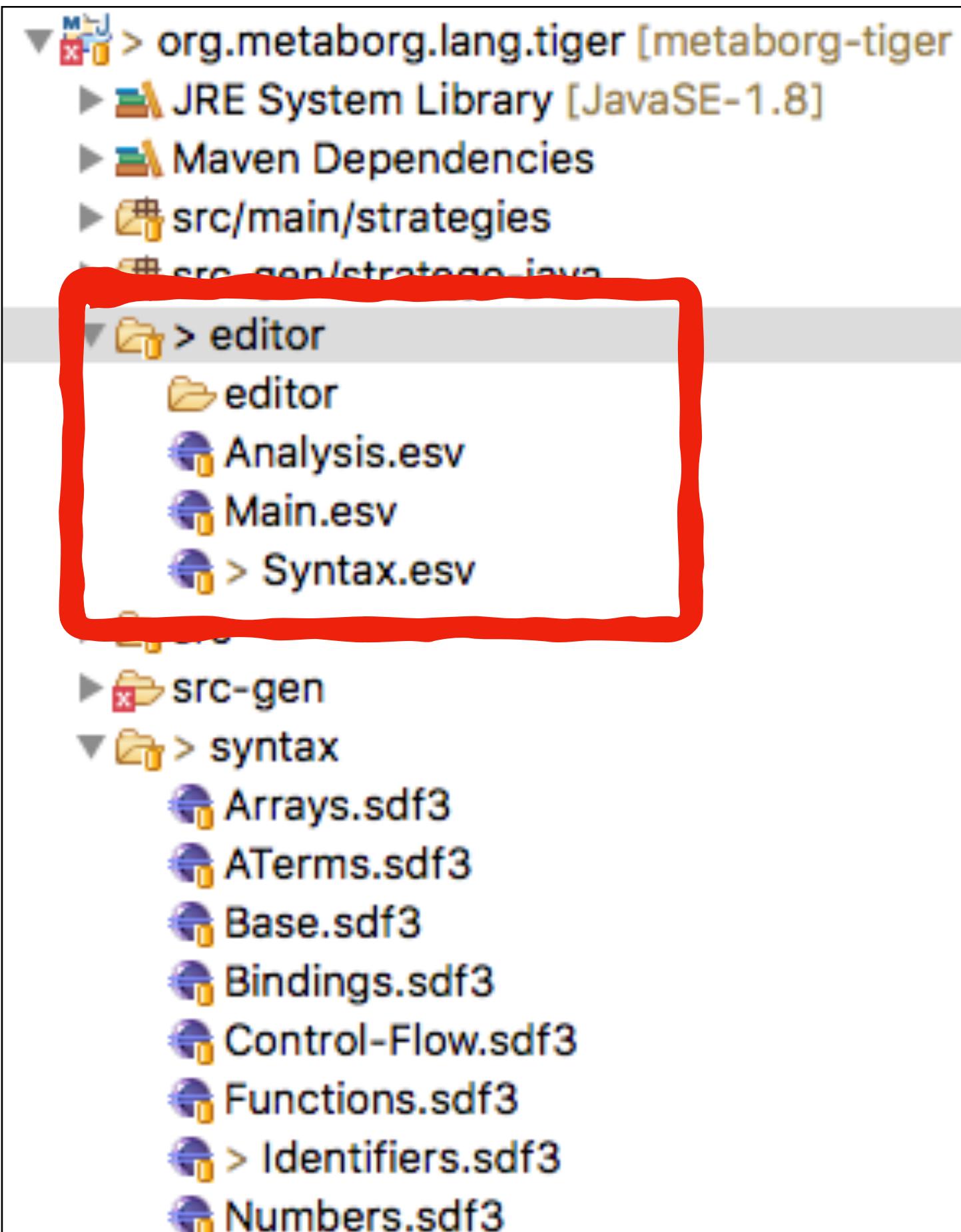
# Editor Services



**Feedback**  
- syntax coloring  
- syntax checking  
- outline view

**Operations**  
- syntactic completion  
- formatting  
- abstract syntax tree

# Language Project Configuration (ESV)



```
module Main

imports

    Syntax
    Analysis

language

extensions : tig

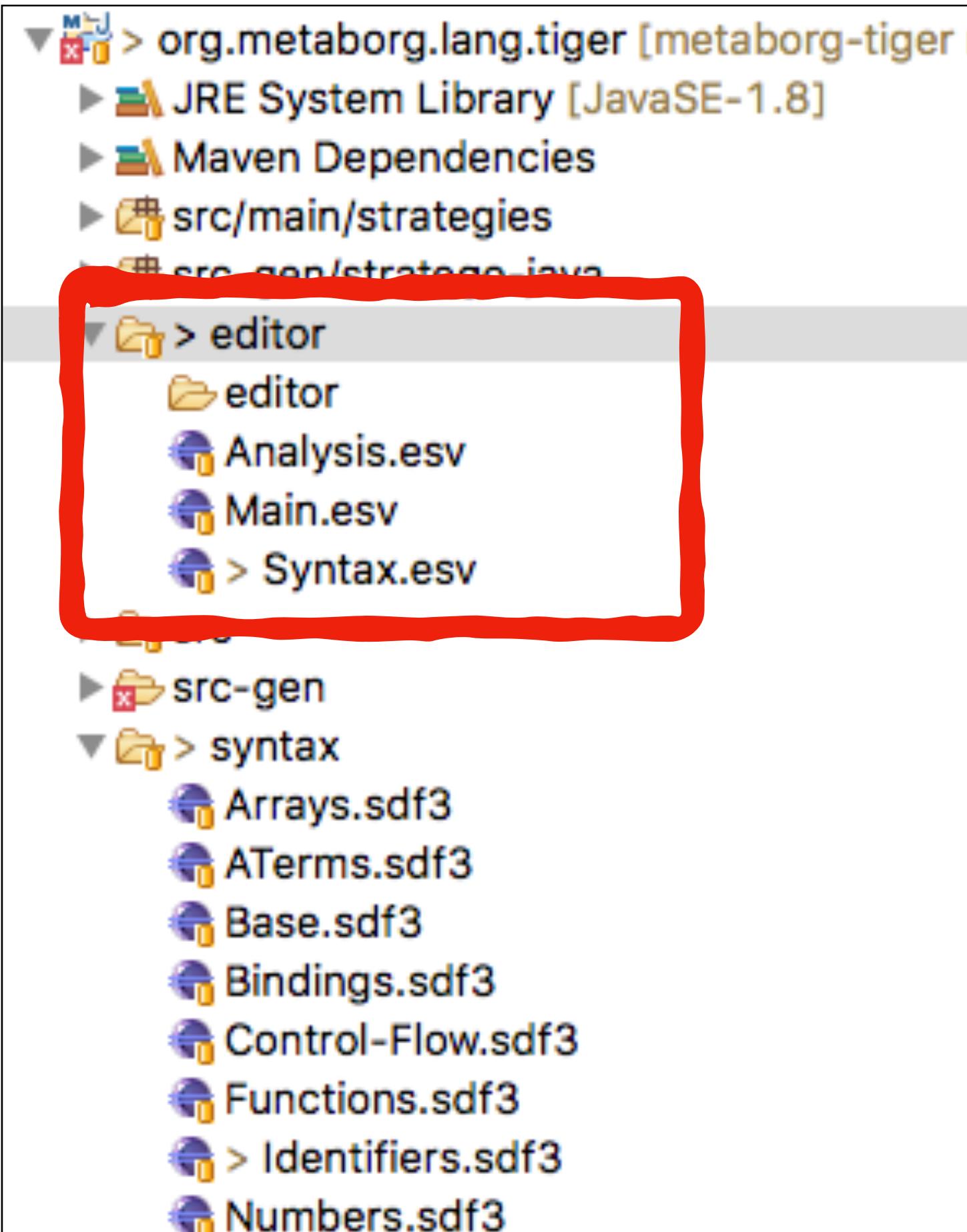
//provider : target/metaborg/stratego.ctree
provider : target/metaborg/stratego.jar
provider : target/metaborg/stratego-javastrat.jar

menus

menu: "Transform" (openeditor) (realtime)

action: "Desugar"      = editor-desugar (source)
action: "Desugar AST" = editor-desugar-ast (source)
```

# Configuration of Syntactic Services (ESV)



```
module Syntax

imports

    libspofax/color/default
    completion/colorer/Tiger-cc-esv

language

    table          : target/metaborg/sdf-new.tbl
    //table        : target/metaborg/sdf.tbl
    start symbols : Module

    line comment  : "//"
    block comment : "/*" * "*/"
    fences        : [ ] ( ) { }

menus

menu: "Syntax" (openeditor)

    action: "Format"           = editor-format (source)
    action: "Show parsed AST" = debug-show-aterm (source)

views

outline view: editor-outline (source)
expand to level: 3
```

# Syntax Coloring

# Generated Syntax Coloring

```
module libspofax/color/default

imports

    libspofax/color/colors

colorer // Default, token-based
highlighting

keyword      : 127 0 85 bold
identifier   : default
string       : blue
number       : darkgreen
var          : 139 69 19 italic
operator     : 0 0 128
layout       : 63 127 95 italic
```

```
// compute the n-th fibonacci number
let function fib(n: int): int =
    if n <= 1 then 1
    else fib(n - 1) + fib(n - 2)
in fib(10)
end
```

# Customized Syntax Coloring

```
module Tiger-Colorer
```

```
colorer
```

```
red    = 255 0 0
```

```
green  = 0 255 0
```

```
blue   = 0 0 255
```

```
TUDlavender = 123 160 201
```

```
colorer token-based highlighting
```

```
keyword : red
```

```
Id      : TUDlavender
```

```
StrConst : darkgreen
```

```
TypeId  : blue
```

```
layout   : green
```

```
// compute the n-th fibonacci number
let function fib(n: int): int =
    if n <= 1 then 1
    else fib(n - 1) + fib(n - 2)
in fib(10)
end
```

# **From Unparsers to Pretty-Printers with Templates**

The inverse of parsing is unparsing or pretty-printing or formatting, i.e. mapping a tree representation of a program to a textual representation. A plain context-free grammar can be used as specification of an unparser. However, then it is unclear where the whitespace should go.

This paper extends context-free grammars with templates that provide hints for layout of program text when formatting.

Based on master's thesis project of Tobi Vollebregt

<https://doi.org/10.1145/2427048.2427056>

## Declarative Specification of Template-Based Textual Editors

Tobi Vollebregt  
tobivollebregt@gmail.com

Lennart C. L. Kats  
l.c.l.kats@tudelft.nl

Eelco Visser  
visser@acm.org

Software Engineering Research Group  
Delft University of Technology  
The Netherlands

### ABSTRACT

Syntax discoverability has been a crucial advantage of structure editors for new users of a language. Despite this advantage, structure editors have not been widely adopted. Based on immediate parsing and analyses, modern textual code editors are also increasingly syntax-aware: structure and textual editors are converging into a new editing paradigm that combines text and templates. Current text-based language workbenches require redundant specification of the ingredients for a template-based editor, which is detrimental to the quality of syntactic completion, as consistency and completeness of the definition cannot be guaranteed.

In this paper we describe the design and implementation of a specification language for syntax definition based on templates. It unifies the specification of parsers, unparsers and template-based editors. We evaluate the template language by application to two domain-specific languages used for tax benefits and mobile applications.

### Categories and Subject Descriptors

D.2.3 [Software Engineering]: Coding Tools and Techniques—*pretty printers, program editors*; D.3.1 [Programming Languages]: Processors—*parsing, code generation*; D.3.1 [Programming Languages]: Formal Definitions and Theory—*syntax*; D.2.6 [Software Engineering]: Programming Environments; D.2.11 [Software Architectures]: Languages

### 1. INTRODUCTION

Language-aware structure editors provide a template-based paradigm for editing programs. They allow composing programs by selecting a template and filling in the placeholders, which can again be extended using templates. A crucial advantage of structure editors is syntax discoverability, helping new users to learn a language by presenting possible syntactic completions in a menu. Structure editors can be automatically generated from a syntax definition. Notable projects aiming at automatic generation of structure editors include MPS [23] and the Intentional Domain Work-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LDTA 2012 Tallinn, Estonia

Copyright 2012 ACM 978-1-4503-1536-4 ...\$15.00.

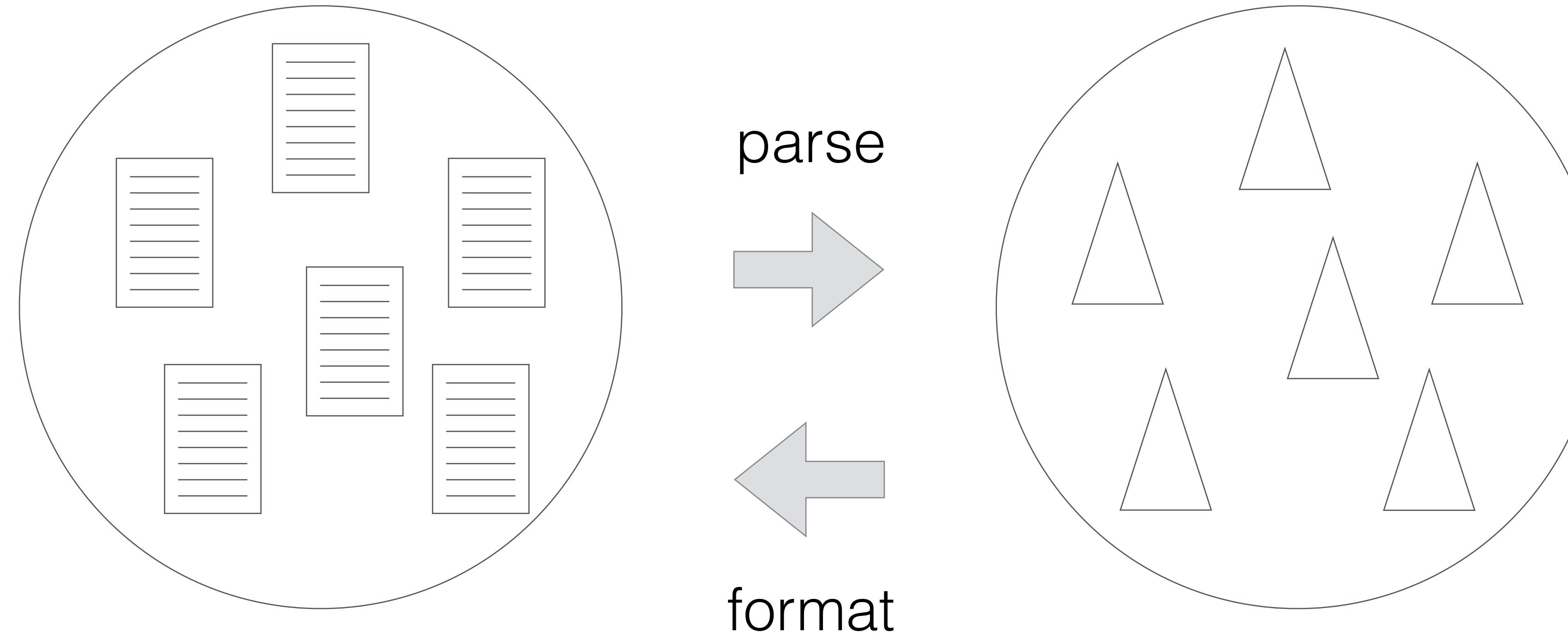
bench [19]. Structure editors can be used for general-purpose languages or for domain-specific languages (DSLs). A modern example of the former is the structure editor in MPS [23], for extensible languages based on Java. An example of the latter category is the DSL for modeling tax-benefit rules developed by IT services company Capgemini using the Cheetah system. Cheetah's facilities for discoverability and the use of templates are particularly effective to aid a small audience of domain expert programmers manage the verbose syntax based on legal texts.

Despite their good support for discoverability, structure editors have not been widely adopted. Pure structure editors tend to introduce an increased learning curve for basic editing operations. For example, they only support copy-pasting operations that maintain well-formedness of the tree and require small, yet non-trivial “refactoring” operations for editing existing code, e.g. when converting an *if* statement to an *if-else* statement. They also lack integration with other tools and expose the user to vendor lock-in. Transferring code across tools requires a shared representation that is generally not available. With software engineering tools such as issue trackers, forums, search, and version control being based on text, a textual representation is preferable, but requires the use of a parser and a parseable language syntax. This forces tools based on structure editors to find new solutions to problems long solved in the text domain.

To alleviate the problems of structure editors, there has been a long history of hybrid structure editors that introduce textual editing features to structure editors [24, 18, 11]. Conversely, modern textual code editors such as those in Eclipse and Visual Studio are increasingly syntax-aware, based on parsers that run while a program is edited. Over time, they have acquired features ranging from code folding to syntactic completions allowing programmers to fill in textual templates. Indeed, structure and textual editors are converging into a new editing paradigm that combines text and templates.

In order to provide the advantages of text editing to the tax-benefit DSL of Capgemini, we converted the language from the Cheetah system, which uses a structure editor, to the parser-based Spoofax language workbench [10]. We quickly realized that it would be impossible for a user to write new models in such a verbose language in the textual editor of Spoofax, without accurate and complete syntax discovery. In syntax-aware text editors this discovery is provided in the form of syntactic completion. Accurate and complete syntactic completion depends critically on two features of a language workbench: first, the syntactic completion proposals presented to the user must be relevant and complete, and second, it must be feasible to create and maintain the specification necessary to make the editor aware of these completion proposals.

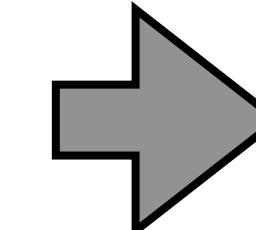
# Converting between Text and Tree Representations



# Unparsing: From Abstract Syntax Term to Text

context-free syntax

```
Exp.Int    = IntConst  
Exp.Times = Exp "*" Exp  
Exp.Plus   = Exp "+" Exp
```



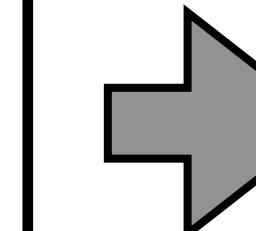
rules

```
unparse :  
  Int(x) -> x
```

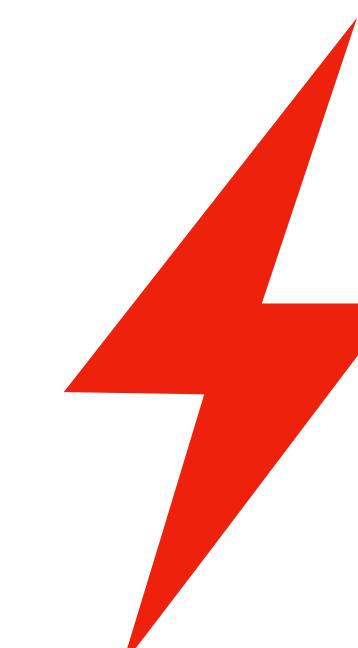
```
unparse :  
  Plus(e1, e2) -> ${[<unparse> e1] + [<unparse> e2]}  
unparse :  
  Times(e1, e2) -> ${[<unparse> e1] * [<unparse> e2]}
```

Mod(

```
Plus(  
  Int("1")  
, Times(  
  Int("2")  
, Plus(Int("3"), Int("4"))))
```



```
1 + 2 * 3 + 4
```



take priorities into account!

# Pretty-Printing

## From ASTs to text

- insert keywords
- insert layout: spaces, line breaks, indentation
- insert parentheses to preserve tree structure

## Unparser

- derive transformation rules from context-free grammar
- keywords, literals defined in grammar productions
- parentheses determined by priority, associativity rules
- separate all symbols by a space => not pretty, or even readable

## Pretty-printer

- introduce spaces, line breaks, and indentation to produce readable text
- doing that manually is tedious

# Specifying Formatting Layout with Templates

## context-free syntax

```
Exp.Seq = <  
  (  
    <{Exp ";"\n}*>  
  )  
>  
  
Exp.If = <  
  if <Exp> then  
    <Exp>  
  else  
    <Exp>  
>  
  
Exp.IfThen = <  
  if <Exp> then  
    <Exp>  
>  
  
Exp.While = <  
  while <Exp> do  
    <Exp>  
>
```

## Inverse quotation

- template quotes literal text with <>
- anti-quotations insert non-terminals with <>

## Layout directives

- whitespace (linebreaks, indentation, spaces) in template guides formatting
- is interpreted as LAYOUT? for parsing

## Formatter generation

- generate rules for mapping AST to text (via box expressions)

## Applications

- code generation; pretty-printing generated AST
- syntactic completions
- formatting

# Templates for Tiger: Binary Expressions

context-free syntax

Exp.Int	= IntConst	
Exp.Uminus	= [- [Exp]]	
Exp.Times	= [[Exp] * [Exp]]	{left}
Exp.Divide	= [[Exp] / [Exp]]	{left}
Exp.Plus	= [[Exp] + [Exp]]	{left}
Exp.Minus	= [[Exp] - [Exp]]	{left}
Exp.Eq	= [[Exp] = [Exp]]	{non-assoc}
Exp.Neq	= [[Exp] <> [Exp]]	{non-assoc}
Exp.Gt	= [[Exp] > [Exp]]	{non-assoc}
Exp.Lt	= [[Exp] < [Exp]]	{non-assoc}
Exp.Geq	= [[Exp] >= [Exp]]	{non-assoc}
Exp.Leq	= [[Exp] <= [Exp]]	{non-assoc}
Exp.And	= [[Exp] & [Exp]]	{left}
Exp.Or	= [[Exp]   [Exp]]	{left}

Use [] quotes instead of  
<> to avoid clash with  
comparison operators

# Templates for Tiger: Functions

Indent body  
of function

context-free syntax

```
Dec.FunDecls = <<{FunDec "\n"}+>> {longest-match}

FunDec.ProcDec = <
  function <Id>(<{FArg ", "}*>) =
    <Exp>
>

FunDec.FunDec = <
  function <Id>(<{FArg ", "}*>) : <Type> =
    <Exp>
>

FArg.FArg = <<Id> : <Type>>

Exp.Call = <<Id>(<{Exp ", "}*>)>
```

Function declarations  
separated by newline

Space after comma!

No space after function name in call

# Templates for Tiger: Bindings and Records

context-free syntax

```
Exp.Let = <  
  let  
    <{Dec "\n"}*>  
    in  
    <{Exp ";"\n}*>  
  end  
>
```

context-free syntax // records

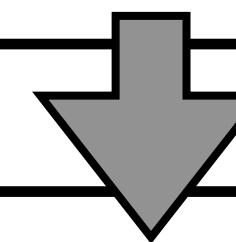
```
Type.RecordTy = <  
  {  
    <{Field ", "\n"}*>  
  }  
>  
  
Field.Field = <<Id> : <TypeId>>  
  
Exp.NilExp = <nil>  
  
Exp.Record = <<TypeId>{ <{InitField ", "}*> }>  
  
InitField.InitField = <<Id> = <Exp>>  
  
LValue.FieldVar = <<LValue>.<Id>>
```

Note spacing / layout in separators

# Generating Pretty-Print Rules from Template Productions

context-free syntax

```
FunDec.FunDec = <  
  function <Id>(<{FArg ", "}*>) : <Type> =  
  <Exp>  
>
```



rules

```
prettyprint-Tiger-FunDec :  
  ProcDec(t1__, t2__, t3__) -> [ H(  
    [S0pt(HS(), "0")]  
    , [ S("function ")  
    , t1__  
    , S("(")  
    , t2__'  
    , S(") =")  
    ]  
    )  
    , t3__'  
  ]  
  
with t1__' := <pp-one-Z(prettyprint-Tiger-Id) <+ pp-one-Z(prettyprint-completion-aux)> t1__  
with t2__' := <pp-H-list(prettyprint-Tiger-FArg1", ")  
           <+ pp-one-Z(prettyprint-completion-aux)> t2__  
with t3__' := <pp-indent(1"2")> [ <pp-one-Z(prettyprint-Tiger-Exp) <+ pp-one-Z(prettyprint-completion-aux)> t3__ ]
```

Separation of concerns:

- generated formatter transforms AST to Box
- Box formatter produces text

# Boxes for Formatting

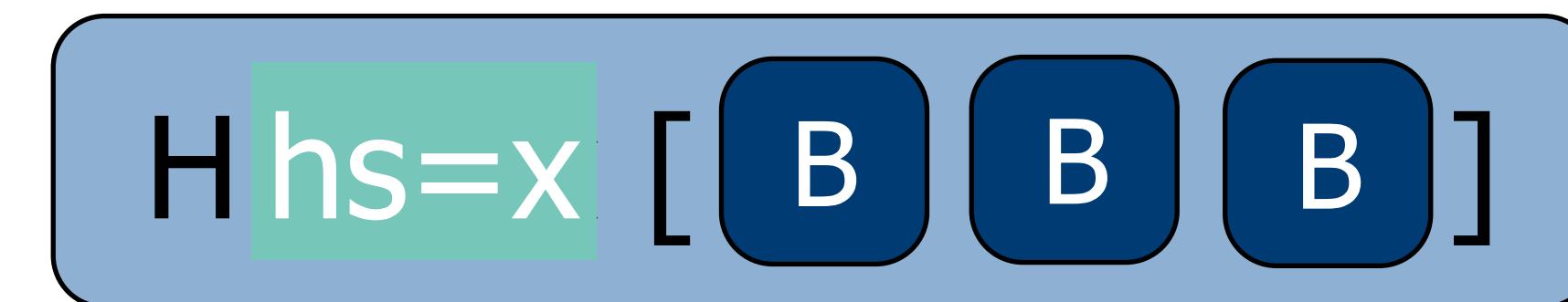
“foo”

KW [ “foo” ]

\_1

literal text, keywords, parameters

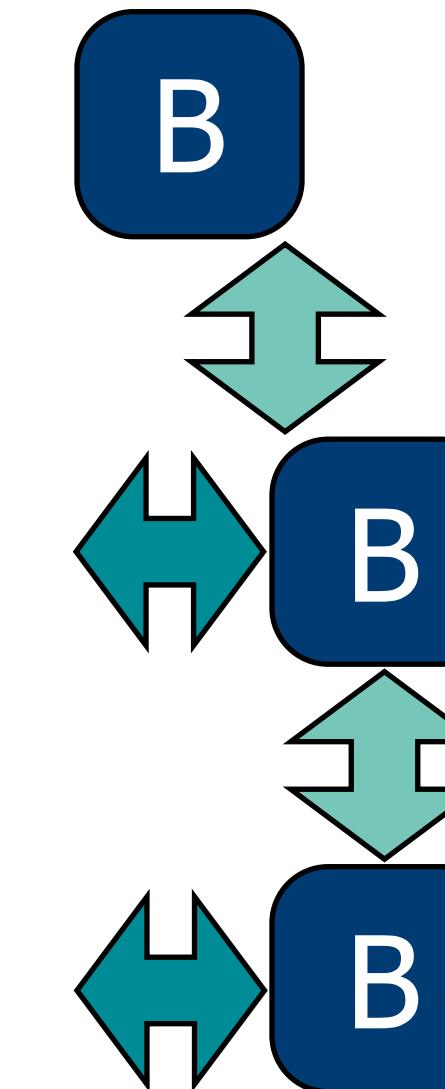
# Horizontal Layout



hs: horizontal space between boxes

# Vertical Layout

V vs=y is=i [ B B B ]



vs: vertical space between boxes; is: indentation space

# **Tiger Syntax Definition with Templates**

# Tiger Syntax: Composition

```
module Tiger

imports Whitespace
imports Comments
imports Types
imports Identifiers
imports Bindings
imports Variables
imports Functions
imports Numbers
imports Strings
imports Records
imports Arrays
imports Control-Flow

context-free start-symbols Module

context-free syntax

Module.Mod = Exp

context-free priorities

Exp.Or > Exp.Array > Exp.Assign ,
{Exp.Uminus LValue.FieldVar LValue.Subscript}
> {left : Exp.Times Exp.Divide}
```

# Tiger Syntax: Identifiers and Strings

```
module Identifiers
```

```
lexical syntax
```

```
Id = [a-zA-Z] [a-zA-Z0-9\_]*
```

```
lexical restrictions
```

```
Id --/- [a-zA-Z0-9\_]
```

```
lexical syntax
```

```
Id = "nil" {reject}  
Id = "let" {reject}  
Id = ... {reject}
```

```
module Strings
```

```
sorts StrConst
```

```
lexical syntax
```

```
StrConst = "\\" StrChar* "\"  
StrChar = ~[\\"\\n]  
StrChar = [\n] [n]  
StrChar = [\t] [t]  
StrChar = [\v] [\^] [A-Z]  
StrChar = [\v] [0-9] [0-9] [0-9]  
StrChar = [\v] ["]  
StrChar = [\v] [\v]  
StrChar = [\v] [\v \t\n]+ [\v]
```

```
context-free syntax // records
```

```
Exp.String = StrConst
```

# Tiger Syntax: Whitespace & Comments

```
module Whitespace
```

```
lexical syntax
```

```
LAYOUT = [\ \t\n\r]
```

```
context-free restrictions
```

```
LAYOUT? -/- [\ \t\n\r]
```

```
module Comments
```

```
lexical syntax // multiline comments
```

```
CommentChar = [\*]
```

```
LAYOUT = /* InsideComment* */
```

```
InsideComment = ~[\*]
```

```
InsideComment = CommentChar
```

```
lexical restrictions
```

```
CommentChar -/- [\v]
```

```
context-free restrictions
```

```
LAYOUT? -/- [\v].[ \v]
```

```
lexical syntax // single line comments
```

```
LAYOUT = // ~[\n\r]* NewLineEOF
```

```
NewLineEOF = [\n\r]
```

```
NewLineEOF = EOF
```

```
EOF =
```

```
// end of file since it cannot be followed by any character
```

```
// avoids the need for a newline to close a single line comment
```

```
// at the last line of a file
```

```
lexical restrictions
```

```
EOF -/- ~[]
```

```
context-free restrictions
```

```
LAYOUT? -/- [\v].[ \*]
```

# Tiger Syntax: Numbers

```
module Numbers
```

```
lexical syntax
```

```
IntConst = [0-9]+
```

```
context-free syntax
```

```
Exp.Int      = IntConst
```

```
Exp.Uminus   = [- [Exp]]
```

```
Exp.Times    = [[Exp] * [Exp]] {left}
```

```
Exp.Divide   = [[Exp] / [Exp]] {left}
```

```
Exp.Plus     = [[Exp] + [Exp]] {left}
```

```
Exp.Minus   = [[Exp] - [Exp]] {left}
```

```
Exp.Eq       = [[Exp] = [Exp]] {non-assoc}
```

```
Exp.Neq      = [[Exp] <> [Exp]] {non-assoc}
```

```
Exp.Gt       = [[Exp] > [Exp]] {non-assoc}
```

```
Exp.Lt       = [[Exp] < [Exp]] {non-assoc}
```

```
Exp.Geq      = [[Exp] >= [Exp]] {non-assoc}
```

```
Exp.Leq      = [[Exp] <= [Exp]] {non-assoc}
```

```
Exp.And     = [[Exp] & [Exp]] {left}
```

```
Exp.Or      = [[Exp] | [Exp]] {left}
```

context-free  
priorities

```
{Exp.Uminus}
```

```
> {left :  
  Exp.Times  
  Exp.Divide}
```

```
> {left :  
  Exp.Plus  
  Exp.Minus}
```

```
> {non-assoc :  
  Exp.Eq  
  Exp.Neq  
  Exp.Gt  
  Exp.Lt  
  Exp.Geq  
  Exp.Leq}
```

```
> Exp.And  
> Exp.Or
```

# Tiger Syntax: Variables and Functions

```
module Bindings

imports Control-Flow
imports Identifiers
imports Types
imports Functions
imports Variables

sorts Declarations

context-free syntax

Exp.Let = <
  let
    <{Dec "\n"}*>
    in
    <{Exp ";"\n"}*>
  end
>

Declarations.Declarations = <
  declarations <{Dec "\n"}*>
>
```

```
module Variables

imports Identifiers
imports Types

sorts Var

context-free syntax

Dec.VarDec = <var <Id> : <Type> := <Exp>>

Dec.VarDecNoType = <var <Id> := <Exp>>
Var.Var = Id

LValue = Var

Exp = LValue

Exp.Assign = <<LValue> := <Exp>>
```

```
module Functions

imports Identifiers
imports Types

context-free syntax

Dec.FunDecls = <<{FunDec "\n"}+>> {longest-match}

FunDec.ProcDec = <
  function <Id>(<{FArg ", "}*>) =
  <Exp>
>

FunDec.FunDec = <
  function <Id>(<{FArg ", "}*>) : <Type> =
  <Exp>
>

FArg.FArg = <<Id> : <Type>>

Exp.Call = <<Id>(<{Exp ", "}*>)>
```

# Tiger Syntax: Records, Arrays, Types

```
module Records
imports Base
imports Identifiers
imports Types
context-free syntax // records

Type.RecordTy = <
  {
    <{Field ", \n"}*>
  }
>

Field.Field = <<Id> : <TypeId>>

Exp.NilExp = <nil>

Exp.Record = <<TypeId>{ <{InitField ", "}*> }>

InitField.InitField = <<Id> = <Exp>>

LValue.FieldVar = <<LValue>.<Id>>
```

```
module Arrays
imports Types
context-free syntax // arrays

Type.ArrayTy = <array of <TypeId>>

Exp.Array = <<TypeId>[<Exp>] of <Exp>>

LValue.Subscript = <<LValue>[<Index>]>

Index = Exp
```

```
module Types
imports Identifiers
imports Bindings
sorts Type
context-free syntax // type declarations
Dec.TypeDecls = <<{TypeDec "\n"}+>> {longest-match}
TypeDec.TypeDec = <type <Id> = <Type>>
context-free syntax // type expressions
Type = TypeId
TypeId.Tid = Id
sorts Ty
context-free syntax // semantic types
Ty.INT      = <INT>
Ty.STRING   = <STRING>
Ty.NIL      = <NIL>
Ty.UNIT     = <UNIT>
Ty.NAME     = <NAME <Id>>
Ty.RECORD   = <RECORD <Id>>
Ty.ARRAY    = <ARRAY <Ty> <Id>>
Ty.FUN      = <FUN ( <{Ty ","}*> ) <Ty>>
```

# Syntactic Completion

Syntax definitions cannot be used just for parsing, but for many other operations. This paper shows how syntactic completion can be provided generically given a syntax definition.

## Part of PhD thesis work Eduardo Amorim

<https://doi.org/10.1145/2427048.2427056>

## Principled Syntactic Code Completion using Placeholders



Luís Eduardo de Souza Amorim Sebastian Erdweg Guido Wachsmuth Eelco Visser  
Delft University of Technology, Netherlands  
l.e.desouzaamorim-1@tudelft.nl, s.t.erdweg@tudelft.nl, guwac@acm.org, e.visser@tudelft.nl

### Abstract

Principled syntactic code completion enables developers to change source code by inserting code templates, thus increasing developer efficiency and supporting language exploration. However, existing code completion systems are ad-hoc and neither complete nor sound. They are not complete and only provide few code templates for selected programming languages. They also are not sound and propose code templates that yield invalid programs when inserted. This paper presents a generic framework that automatically derives complete and sound syntactic code completion from the syntax definition of arbitrary languages. A key insight of our work is to provide an explicit syntactic representation for incomplete programs using placeholders. This enables us to address the following challenges for code completion separately: (i) completing incomplete programs by replacing placeholders with code templates, (ii) injecting placeholders into complete programs to make them incomplete, and (iii) introducing lexemes and placeholders into incorrect programs through error-recovery parsing to make them correct so we can apply one of the previous strategies. We formalize our framework and provide an implementation in Spooftax.

**Categories and Subject Descriptors** D.2.6 [Software Engineering]: Programming Environments

**Keywords** Code Completion, Language Workbenches, IDEs

### 1. Introduction

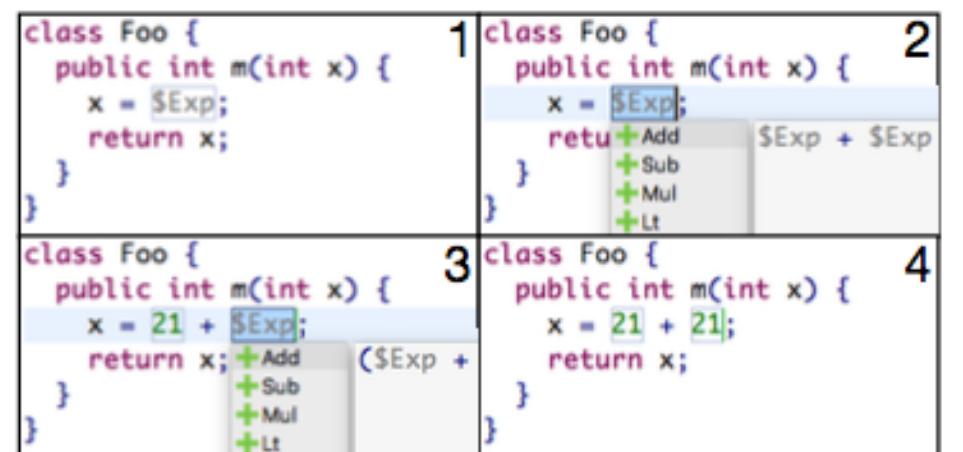
*Code completion*, also known as content completion or content assist, is an editor service that proposes and performs expansion of the program text. Code completion helps the programmer to avoid misspellings and acts as a guide to discover



This work is licensed under a Creative Commons Attribution International 4.0 License.

Copyright is held by the owner/author(s).

SLE'16, October 31 – November 1, 2016, Amsterdam, Netherlands  
ACM, 978-1-4503-4447-0/16/10..\$15.00  
<http://dx.doi.org/10.1145/2997364.2997374>

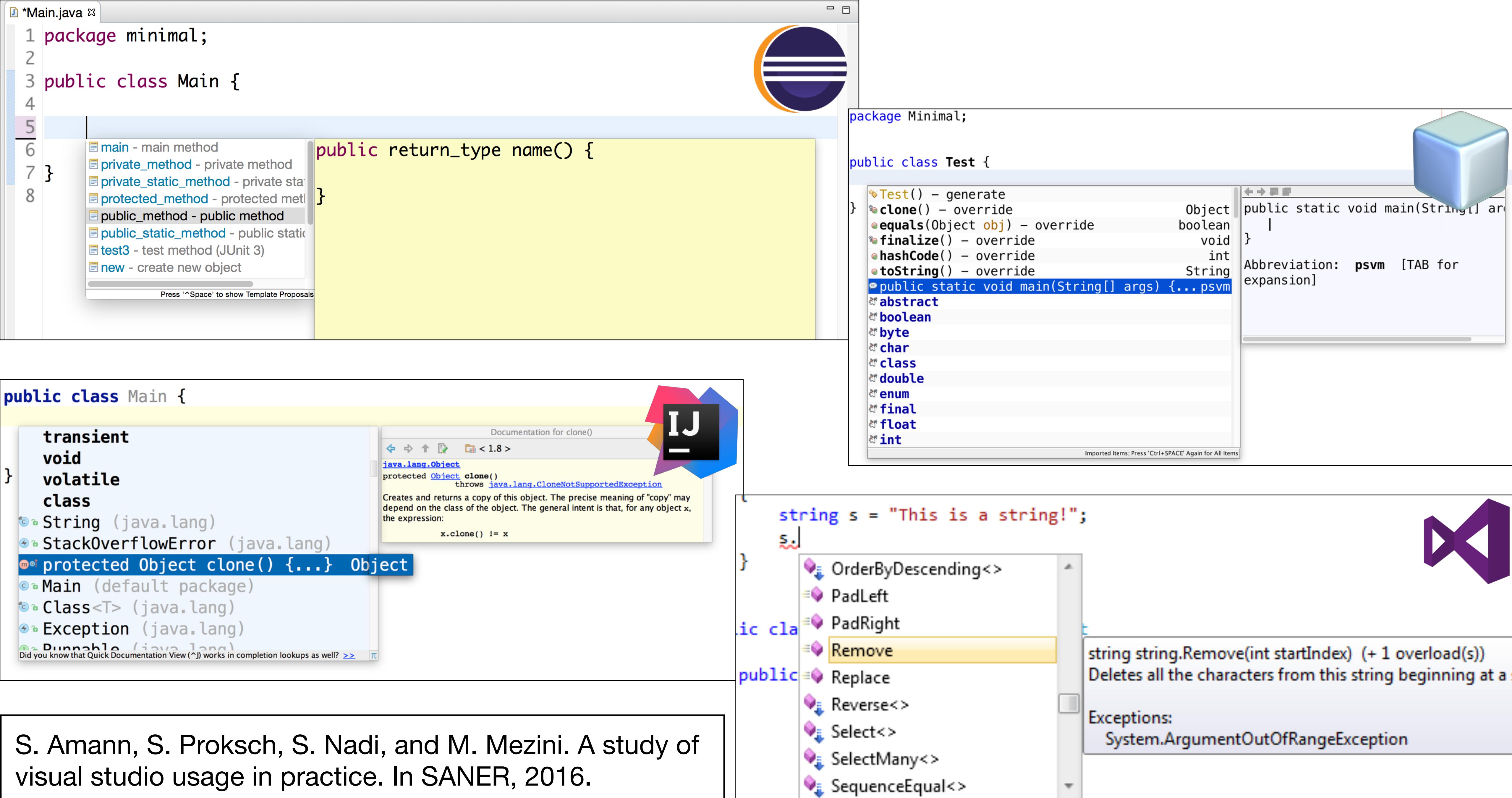


**Figure 1.** (1) Incomplete program with explicit placeholders. (2) Triggering completion for a placeholder. (3) After selecting a proposal, showing completions for nested placeholders. (4) Completing a nested placeholder by typing.

language features and APIs. Most mainstream integrated development environments (IDEs) provide some form of code completion and industrial studies indicate that code completion is one of the most frequently used IDE services [1].

There are two classes of code completion: syntactic and semantic. Syntactic code completion considers the syntactic context at the cursor position and proposes code templates for syntactic structures of the language. For example, most IDEs for Java support syntactic code completion with class and method templates. Semantic code completion also uses the cursor position to propose templates, but by applying semantic analysis to the program, the IDE can propose code templates or identifiers that do not violate the language's name binding or typing rules. For example, in this case IDEs for Java may suggest variables or methods that are visible in the current scope and have the expected type at the cursor position.

In this paper, we focus on *syntactic code completion*. Even for mainstream languages in mainstream IDEs, syntactic code completion is often ad-hoc and unreliable. Specifically, most existing services for syntactic code completion are incomplete and only propose code templates for selected language constructs of a few supported languages, thus inhibiting exploring the language's syntax. Moreover, most existing services are unsound and propose code templates that yield syntax errors when inserted at the cursor position.



The image shows four screenshots of IDE interfaces illustrating various features:

- Top Left:** Shows code completion in IntelliJ IDEA. A tooltip for the method `name()` lists options like `main`, `private_method`, and `protected_method`. A sidebar shows template proposals.
- Top Right:** Shows code completion in PyCharm. It displays a list of Java keywords such as `Object`, `boolean`, `void`, etc., with an abbreviation key `psvm`.
- Bottom Left:** Shows code completion in IntelliJ IDEA. A tooltip for `clone()` from `Object` class provides documentation and examples.
- Bottom Right:** Shows code completion in Visual Studio Code. A tooltip for `Remove` from the `string` class provides documentation and examples.

S. Amann, S. Proksch, S. Nadi, and M. Mezini. A study of visual studio usage in practice. In SANER, 2016.

```
public class Main {  
    private_method - private method  
    private_static_method - private static method  
    protected_method - protected method  
    public_method - public method  
    public_static_method - public static method  
    main - main method  
  
    Press '^Space' to show SWT Template Proposals
```

## Syntactic Completion

```
public class Fibonacci {  
    static long fibo(int n) {  
        return (n < 2) ? n : fibo(n - 1) + fibo(n - 2);  
    }  
  
    public static void main(String[] args){  
        for (int i = 0; i < 30; i++){  
            System.out.print("(" + i + "):" + Fibonacci.| + "\t");  
        }  
    }  
}
```

Completion suggestions:

- fibo(int n) : long - Fibonacci
- class : Class<slepresentation>
- main(String[] args) : void - Fi

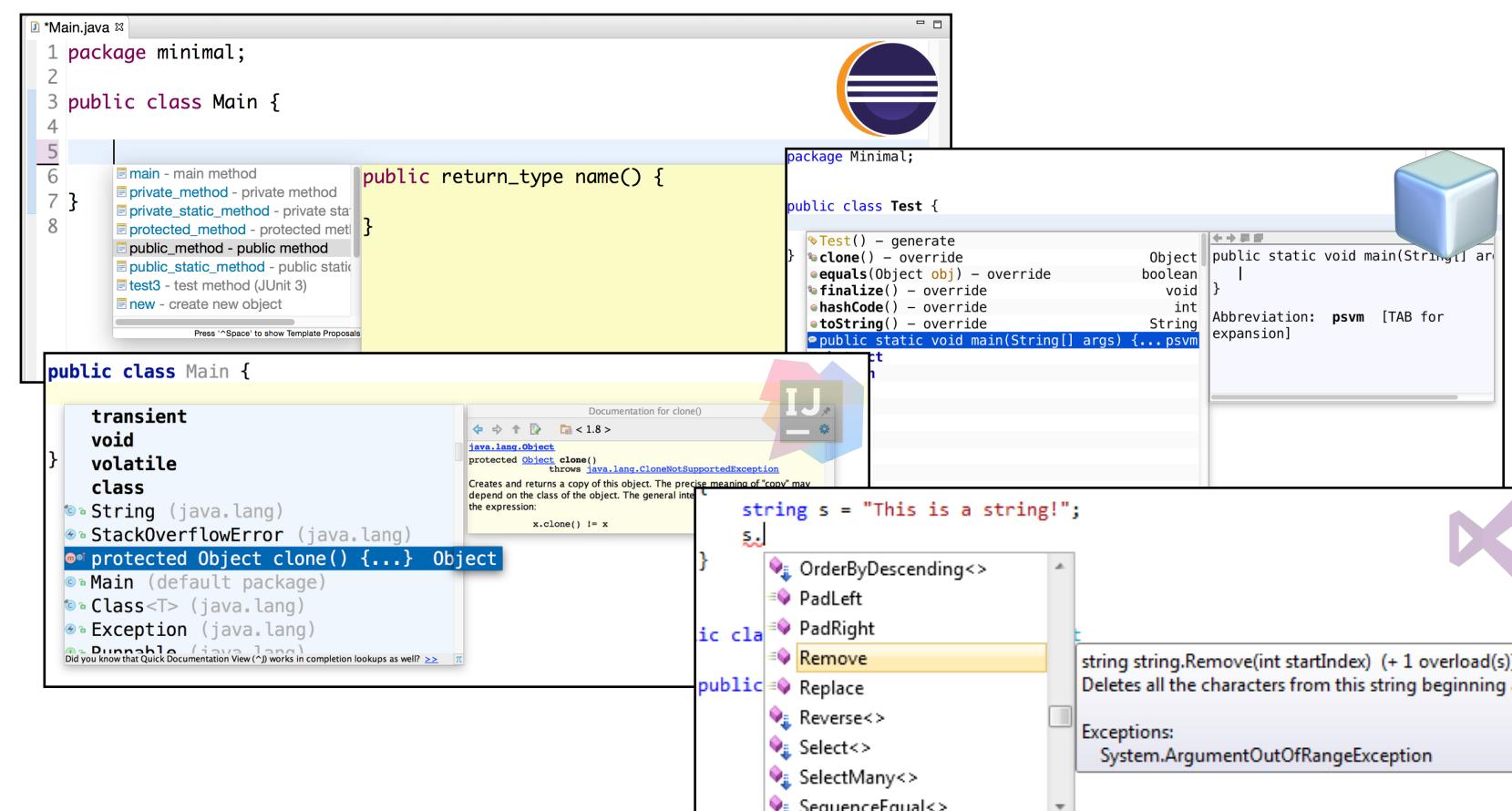
## Semantic Completion

# Problems

```
public class Main {  
    public static void main(String[] args) {  
        } }  
        arrayadd - add an element  
        arraymerge - merge two ar  
        cast - dynamic cast  
        catch - catch block  
        do - do while statement  
        else - else block  
        ...  
        Press '^Space' to show SWT Template Proposals
```

```
public class Main {  
    public static void main(String[] args) {  
        } }  
        arrayadd - add an element  
        arraymerge - merge two ar  
        cast - dynamic cast  
        catch - catch block  
        do - do while statement  
        else - else block  
        ...  
        Press '^Space' to show SWT Template Proposals
```

Unsound: propose invalid constructs



Incomplete: not all programs reachable

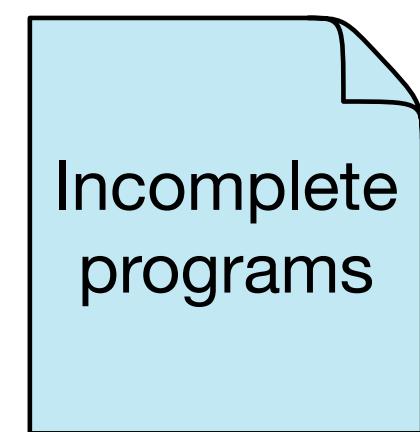
Ad-hoc: re-implement for each language / IDE

# **Sound** and **Complete** Syntactic Code Completion from **Syntax Definition**

# Explicit Placeholders



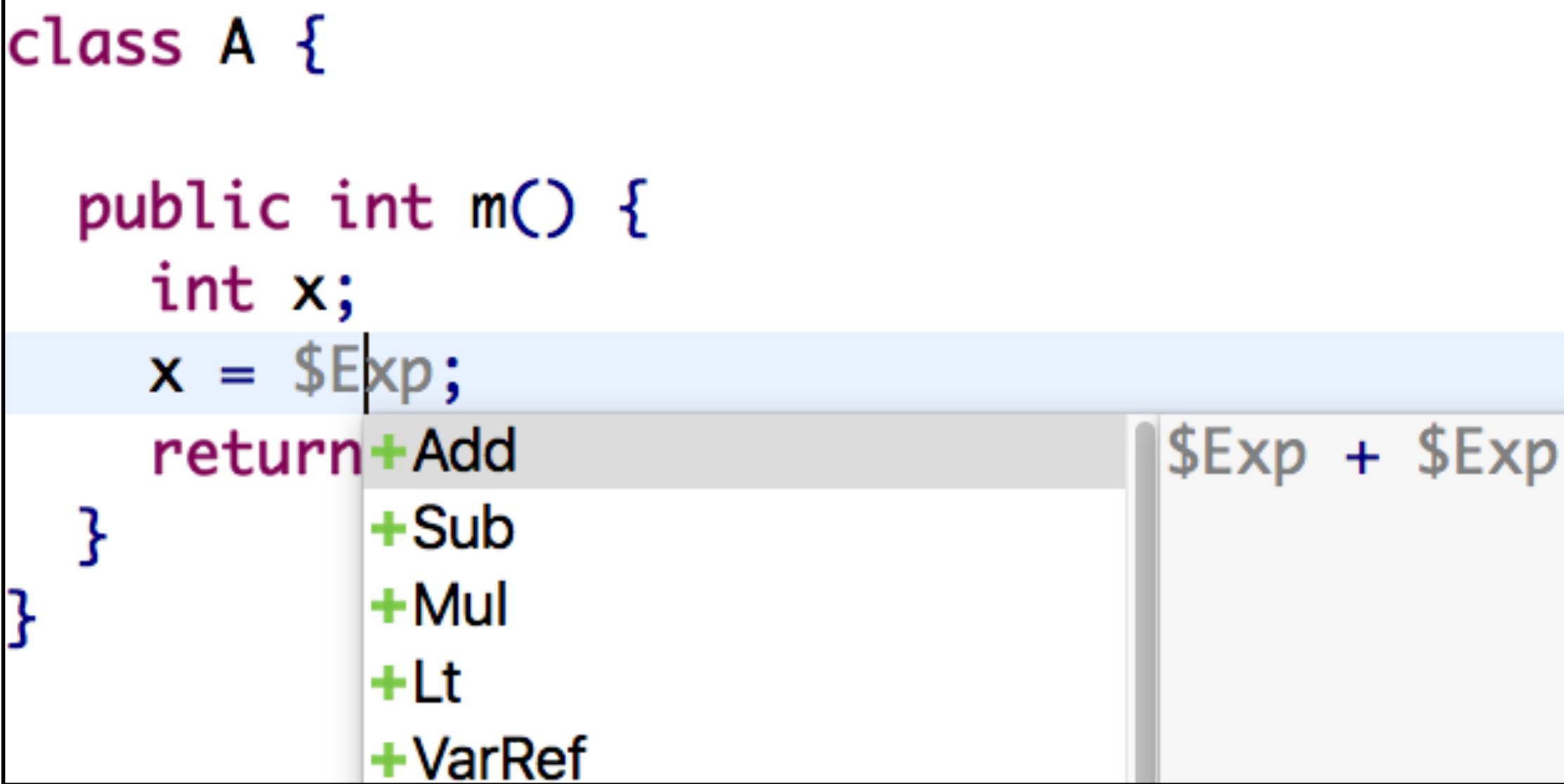
```
class A {  
  
    public int m() {  
        int x;  
..... x = ;  
        return x;  
    }  
}
```



```
class A {  
  
    public int m() {  
        int x;  
        x = $Exp;  
        return x;  
    }  
}
```

Make incompleteness explicit using placeholders!

```
class A {  
  
    public int m() {  
        int x;  
        x = $Exp;  
        return +Add  
    }  
}
```



```
class A {  
  
    public int m() {  
        int x;  
        x = $Exp + $Exp;  
        return $Exp + $Exp;  
    }  
}
```

```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + $Exp;  
        return x; }  
    }  
  
    +Add  
    +Sub  
    +Mul  
    +Lt  
    +VarRef
```

```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + 21;  
        return x;  
    }  
}
```

1

```
class A {  
  
    public int m() {  
        int x;  
        x = $Exp;  
        return +Add  
              +Sub  
              +Mul  
              +Lt  
              +VarRef  
    }  
}
```

2

```
class A {  
  
    public int m() {  
        int x;  
        x = $Exp + $Exp;  
        return +Add  
              +Sub  
              +Mul  
              +Lt  
              +VarRef  
    }  
}
```

3

```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + $Exp;  
        return x; +Add  
              +Sub  
              +Mul  
              +Lt  
              +VarRef  
    }  
}
```

4

```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + 21;  
        return x;  
    }  
}
```

# Deriving Syntactic Completion from Syntax Definition

```
context-free syntax // regular production  
  
Statement.If = "if" "(" Exp ")" Statement "else" Statement
```

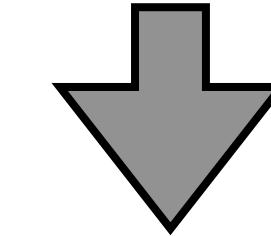
# Placeholders as Language Constructs

```
context-free syntax // regular production
```

```
Statement.If = "if" "(" Exp ")" Statement "else" Statement
```

```
context-free syntax // placeholder rule
```

```
Statement.Statement-Plhdr = "$Statement"
```



```
class A {  
  
    public int m() {  
        int x;  
        if(x < 0) x = 0;  
        else $Statement  
        return x;  
    }  
}
```

# Calculate Placeholder Expansions

```
context-free syntax // regular production
```

```
Statement.If = "if" "(" Exp ")" Statement "else" Statement
```

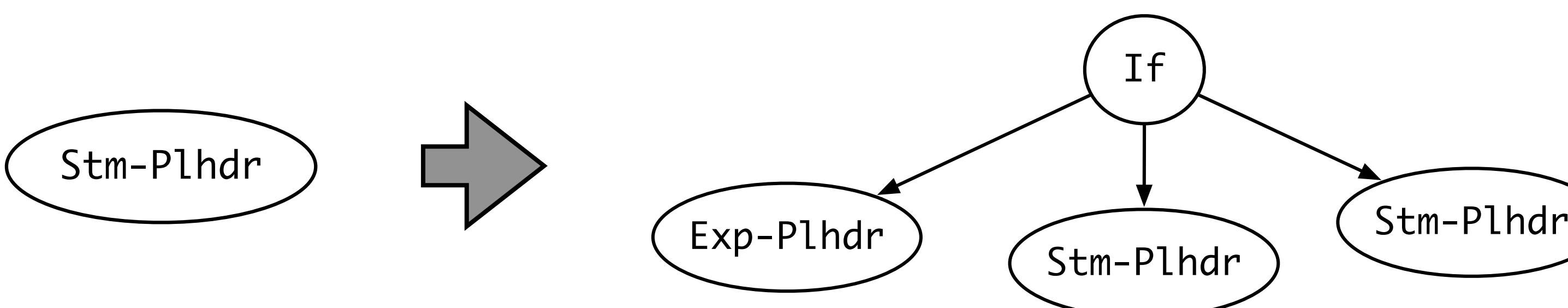
```
context-free syntax // placeholder rule
```

```
Statement.Statement-Plhdr = "$Statement"
```

```
rules
```

```
rewrite-placeholder:
```

```
Statement-Plhdr() -> If(Exp-Plhdr(), Statement-Plhdr(),  
Statement-Plhdr())
```



# Calculate Placeholder Expansions

```
context-free syntax // regular production
```

```
Statement.If = "if" "(" Exp ")" Statement "else" Statement
```

```
context-free syntax // placeholder rule
```

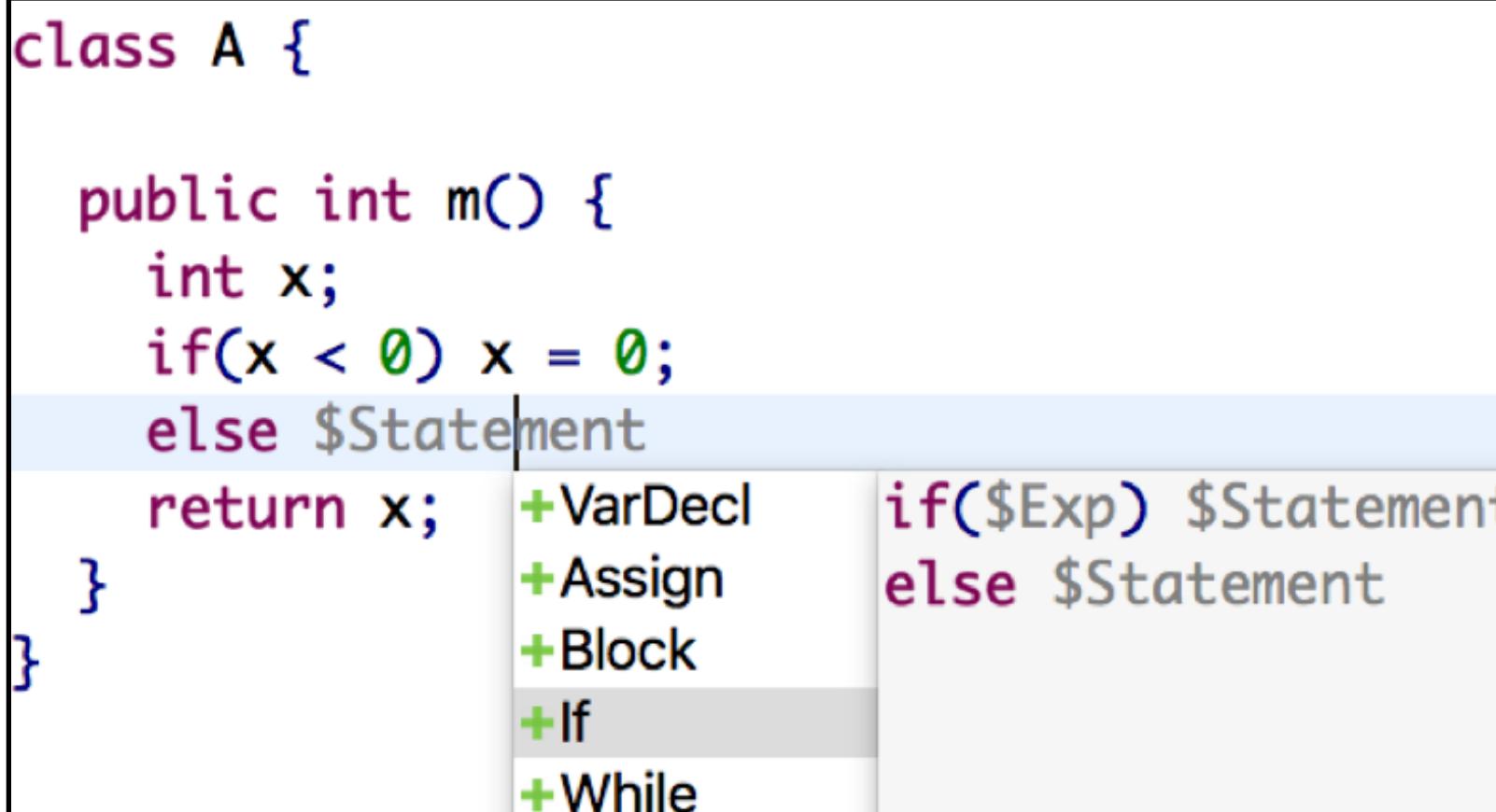
```
Statement.Statement-Plhdr = "$Statement"
```

```
rules
```

```
rewrite-placeholder:
```

```
Statement-Plhdr() -> If(Exp-Plhdr(), Statement-Plhdr(),  
Statement-Plhdr())
```

```
class A {  
  
    public int m() {  
        int x;  
        if(x < 0) x = 0;  
        else $Statement  
        return x;  
    }  
}
```



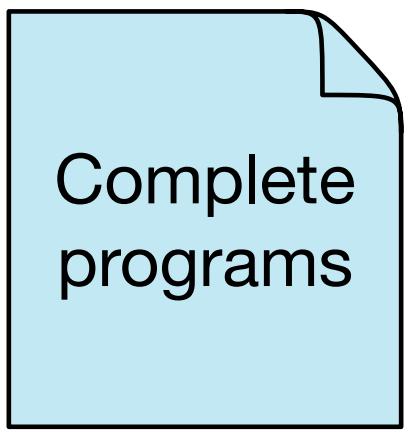
Correct  
programs

Complete  
programs

Expand  
placeholder

Incomplete  
programs

Expand/overwrite  
placeholders



```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + 21;  
        return x;  
    }  
}
```

How to expand a complete program?

```
class A {
    public int m() {
        int x;
        x = 21 + 21;
        return x;
    }
}
```

How to expand a complete program?

```
class A $ParentDecl {
    +Parent extends $ID
    public int m() {
        int x;
        x = 21 + 21;
        return x;
    }
}
```

Insert a placeholder?

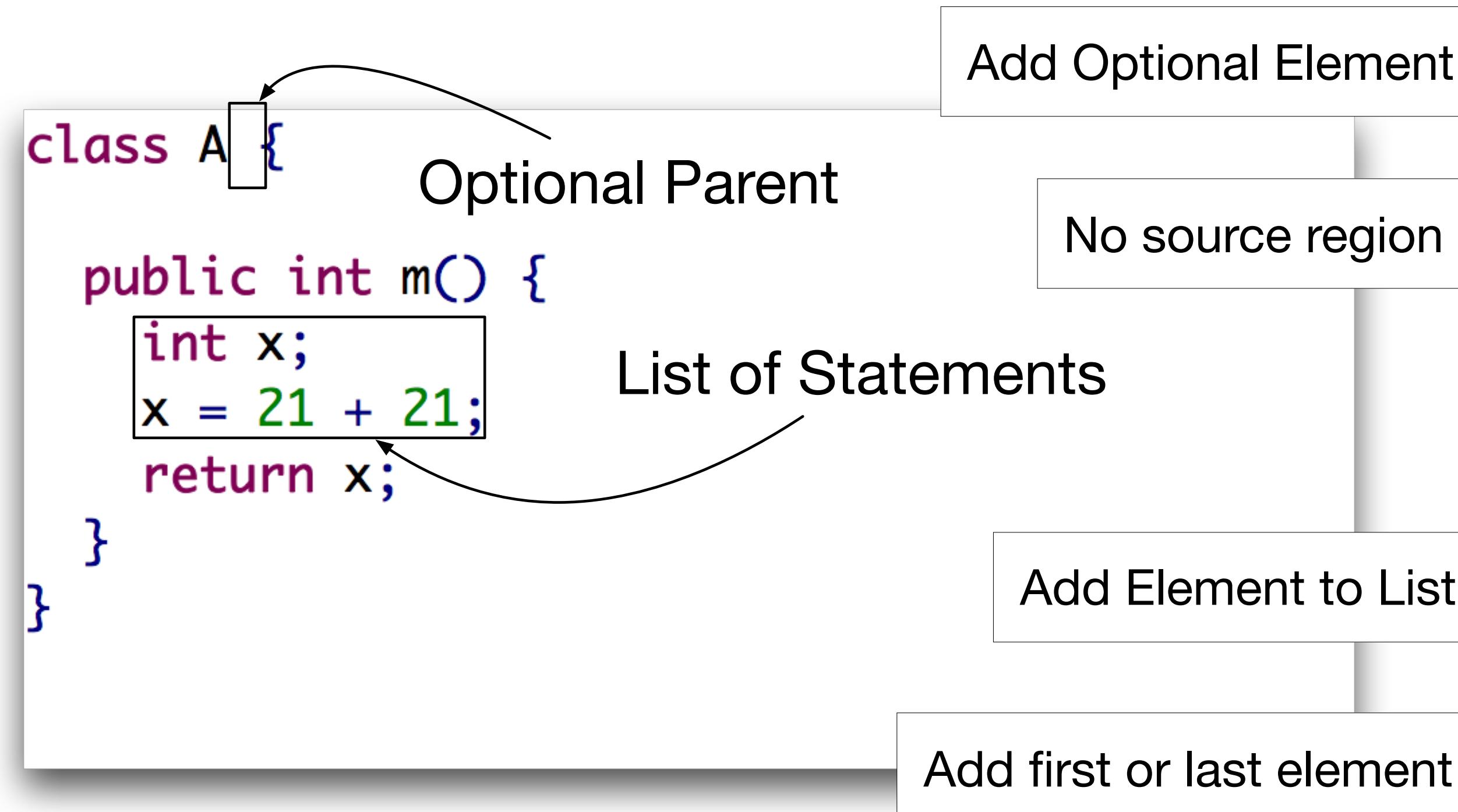
```
class A extends $ID {
    public int m() {
        int x;
        x = 21 + 21;
        return x;
    }
}
```

```
class A {
    public int m() {
        $Statement
        int+$VarDecl $Type $ID;
        x =+$Assign
        ret+$Block
    }
}
```

```
class A {
    public int m() {
        $Type $ID;
        +Int int
        +Bool
        +ClassType
    }
}
```

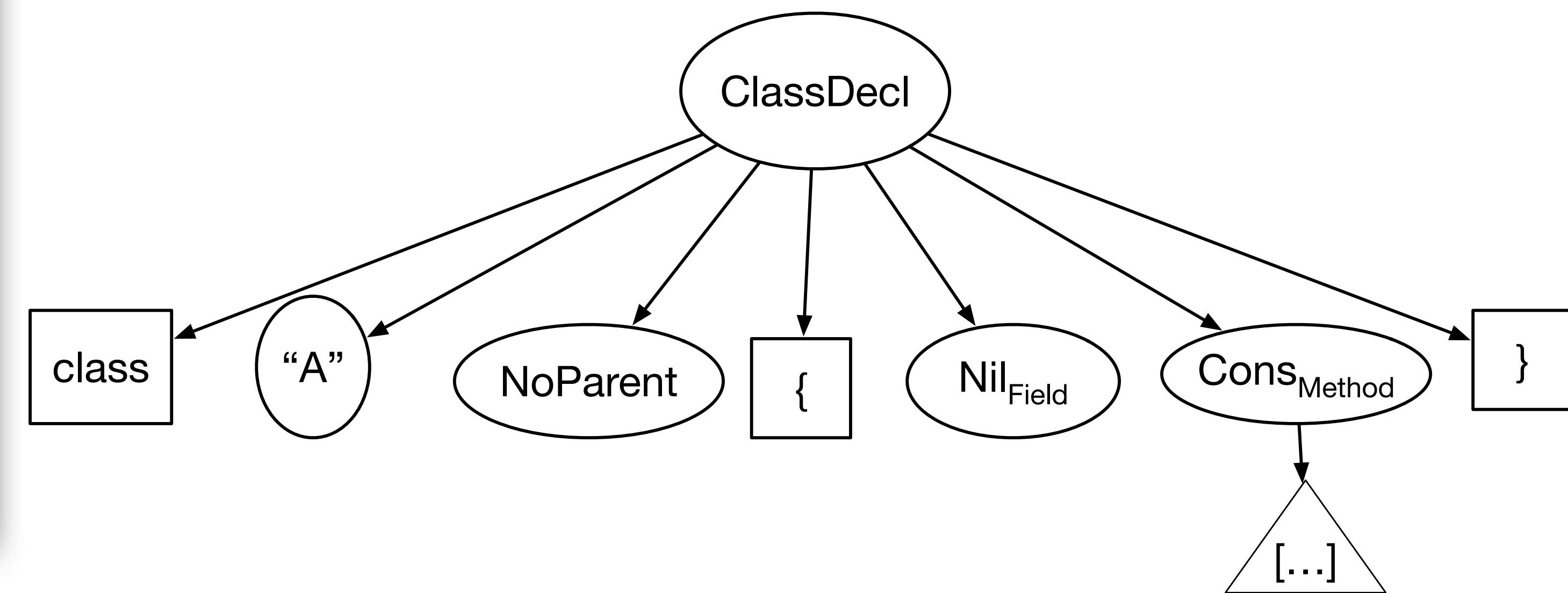
# Inferring Placeholders

# Placeholder Inference



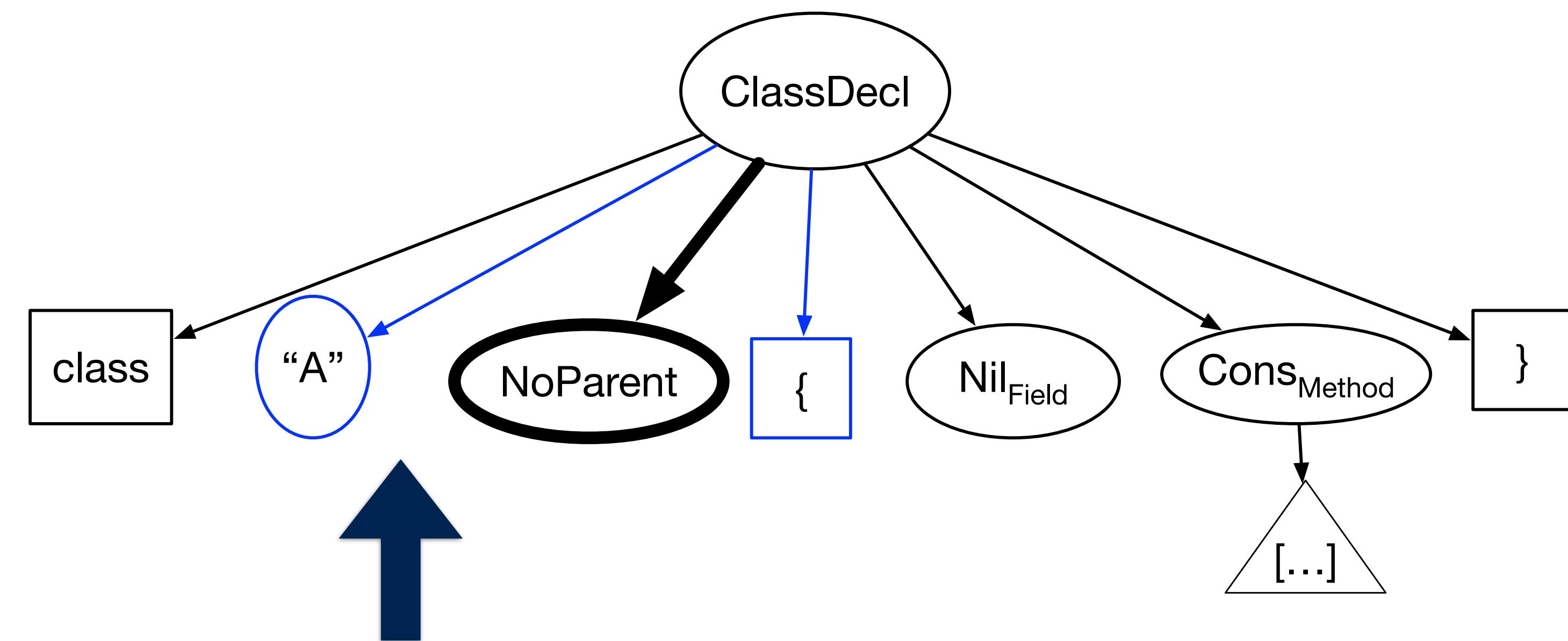
# Placeholder Inference: Optional

```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + 21;  
        return x;  
    }  
}
```



# Placeholder Inference: Optional

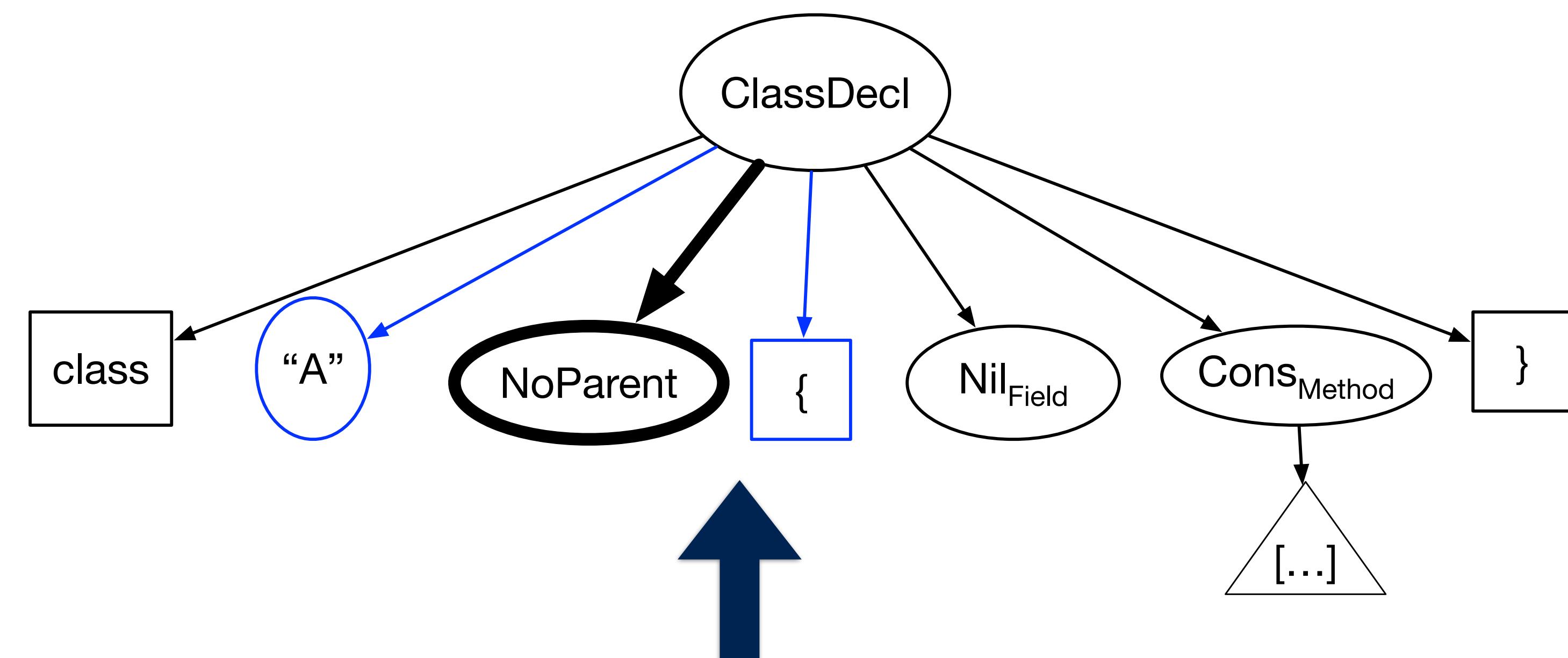
```
class A {  
    public int m() {  
        int x;  
        x = 21 + 21;  
        return x;  
    }  
}
```



# Placeholder Inference: Optional

```
class A {  
    public int m() {  
        int x;  
        x = 21 + 21;  
        return x;  
    }  
}
```

```
class A {  
    public int m() {  
        int x;  
        x = 21 + 21;  
        return x;  
    }  
}
```



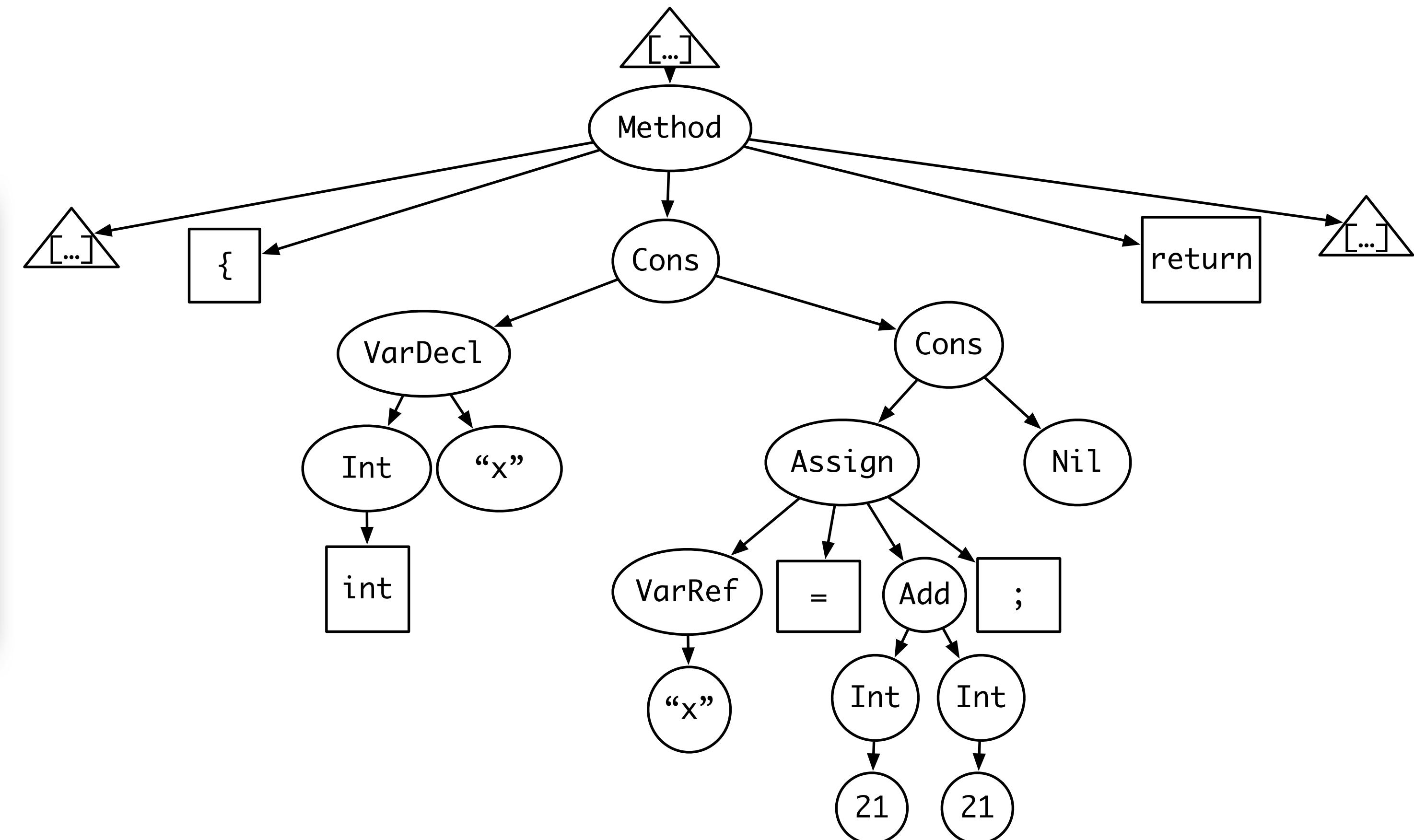
# Placeholder Inference: Optional

```
class A {  
    +Parent  
    public  
        int  
        x =  
        retu  
    }  
}
```

```
class A extends $ID {  
  
    public int m() {  
        int x;  
        x = 21 + 21;  
        return x;  
    }  
}
```

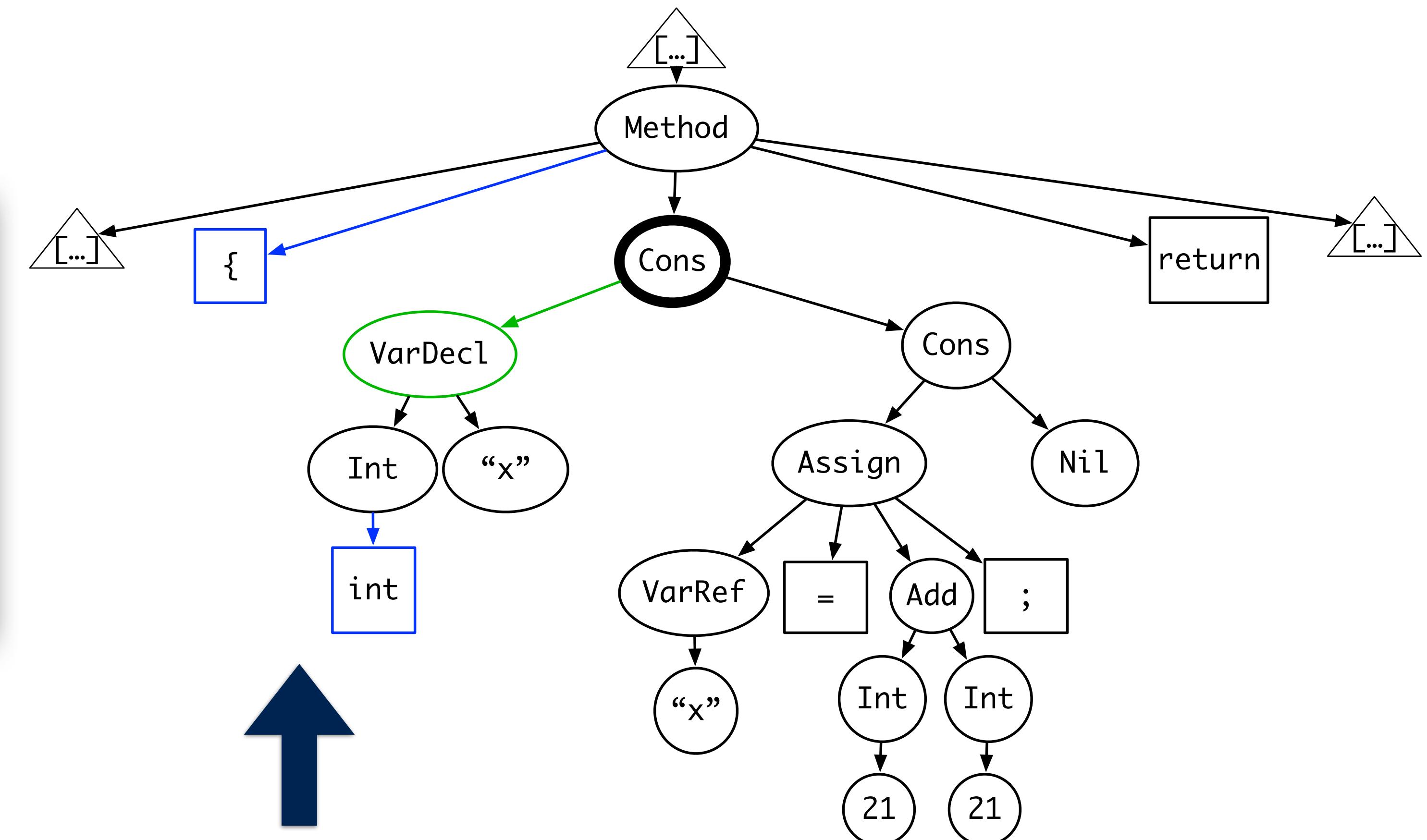
# Placeholder Inference - Lists

```
class A {  
    public int m() {  
        int x;  
        x = 21 + 21;  
        return x;  
    }  
}
```



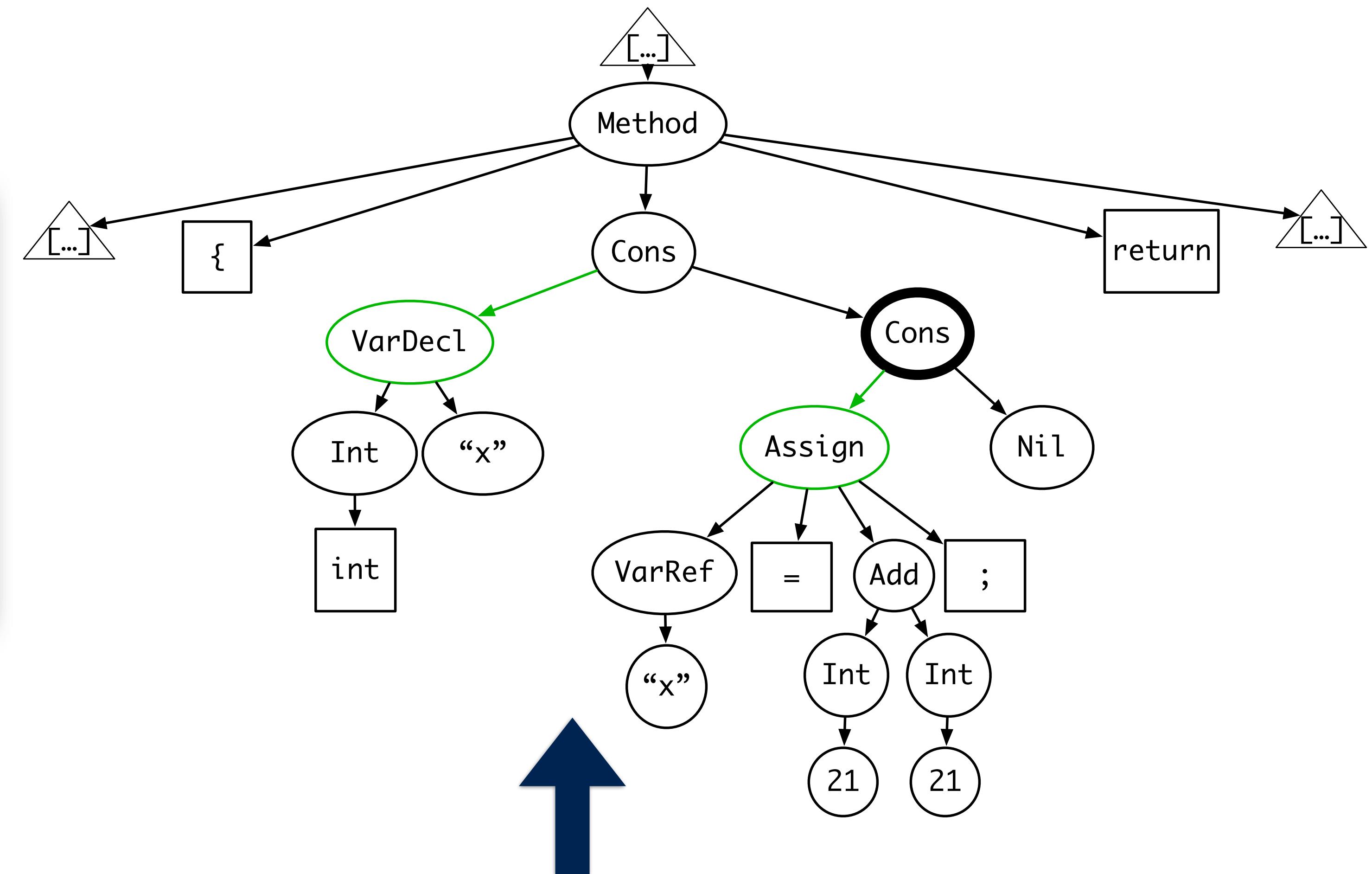
# Placeholder Inference - Lists

```
class A {  
  
    public int m() {  
        int x; // Placeholder  
        x = 21 + 21;  
        return x;  
    }  
}
```



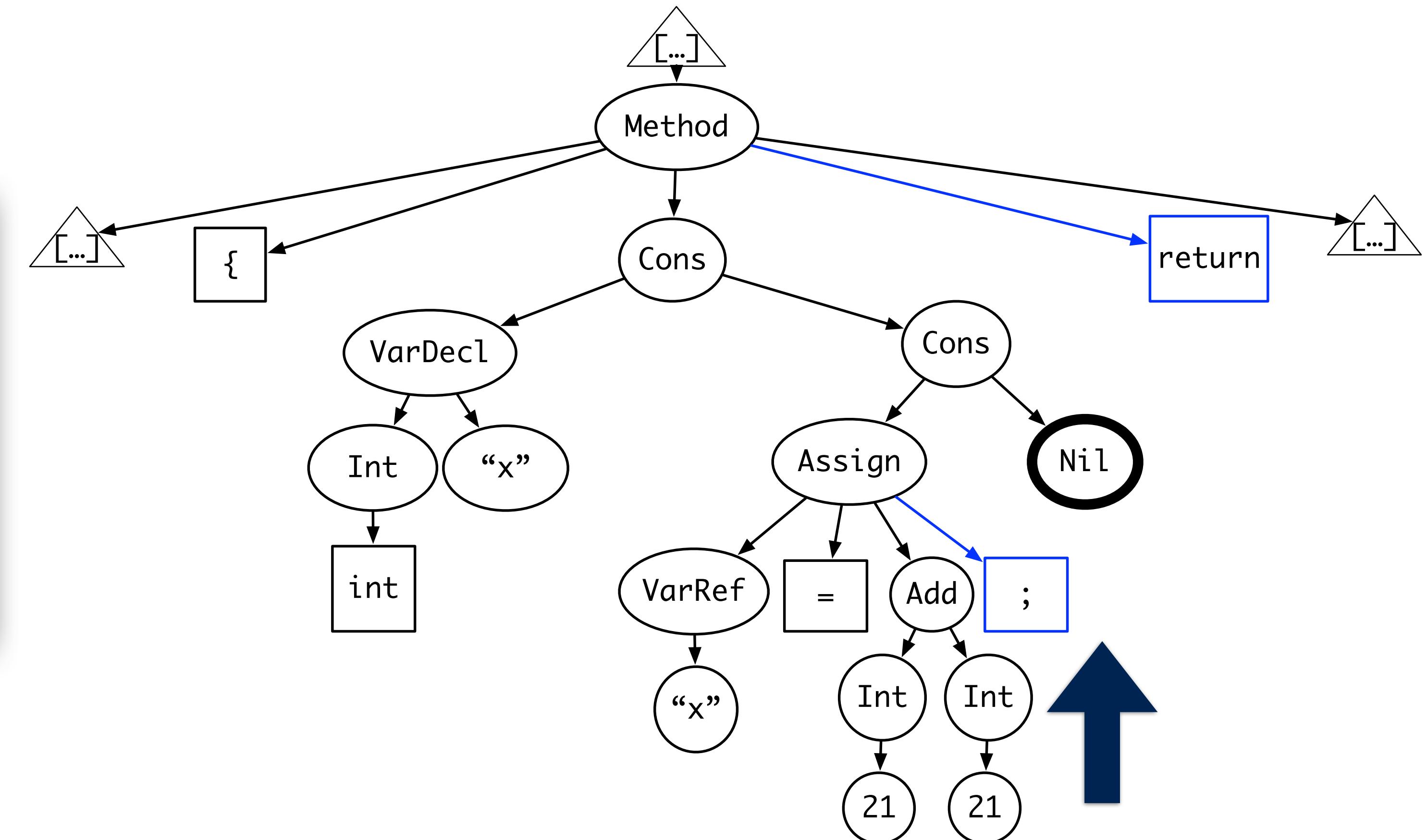
# Placeholder Inference - Lists

```
class A {  
    public int m() {  
        int x; |  
        x = 21 + 21;  
        return  
    }  
}
```



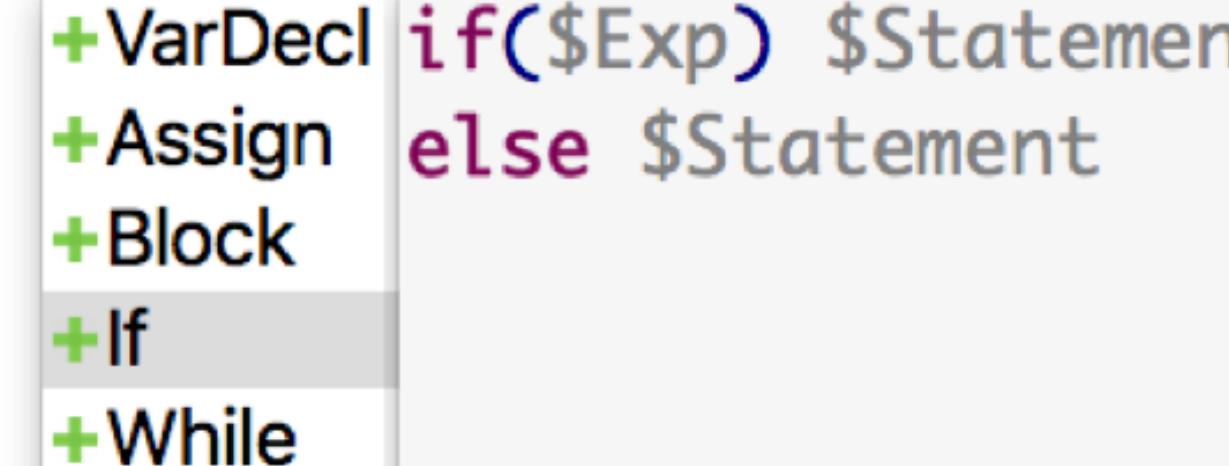
# Placeholder Inference - Lists

```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + 21;  
        return x;  
    }  
}
```



# Completion Proposals for Inferred Placeholders

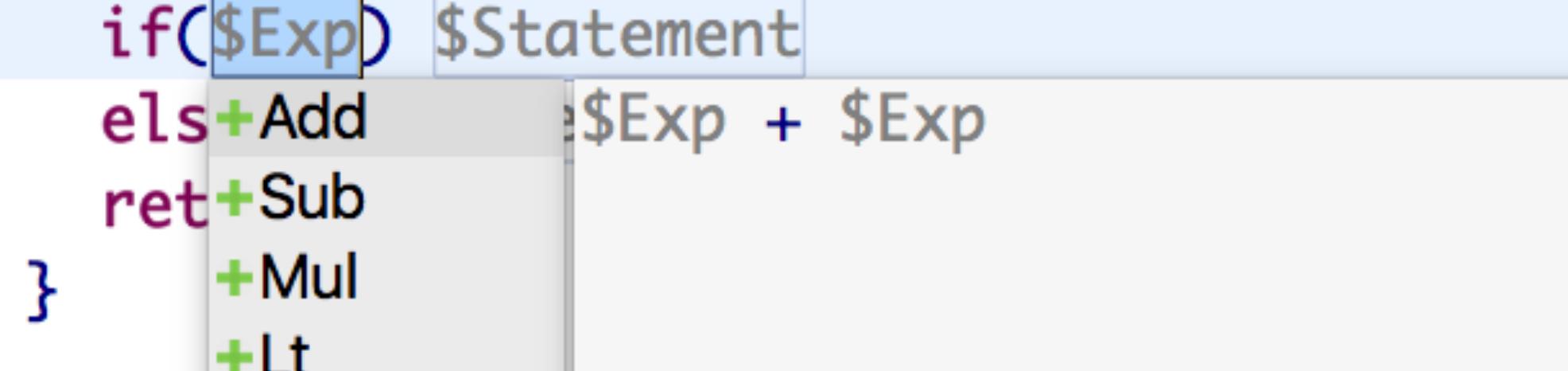
```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + 21; |  
        return x;  
    }  
}
```



A code editor interface showing completion proposals for an inferred placeholder variable. The placeholder '\$Exp' is highlighted in blue. The proposals are listed in a dropdown menu:

- +VarDecl
- if(\$Exp) \$Statement
- +Assign
- else \$Statement
- +Block
- +If
- +While

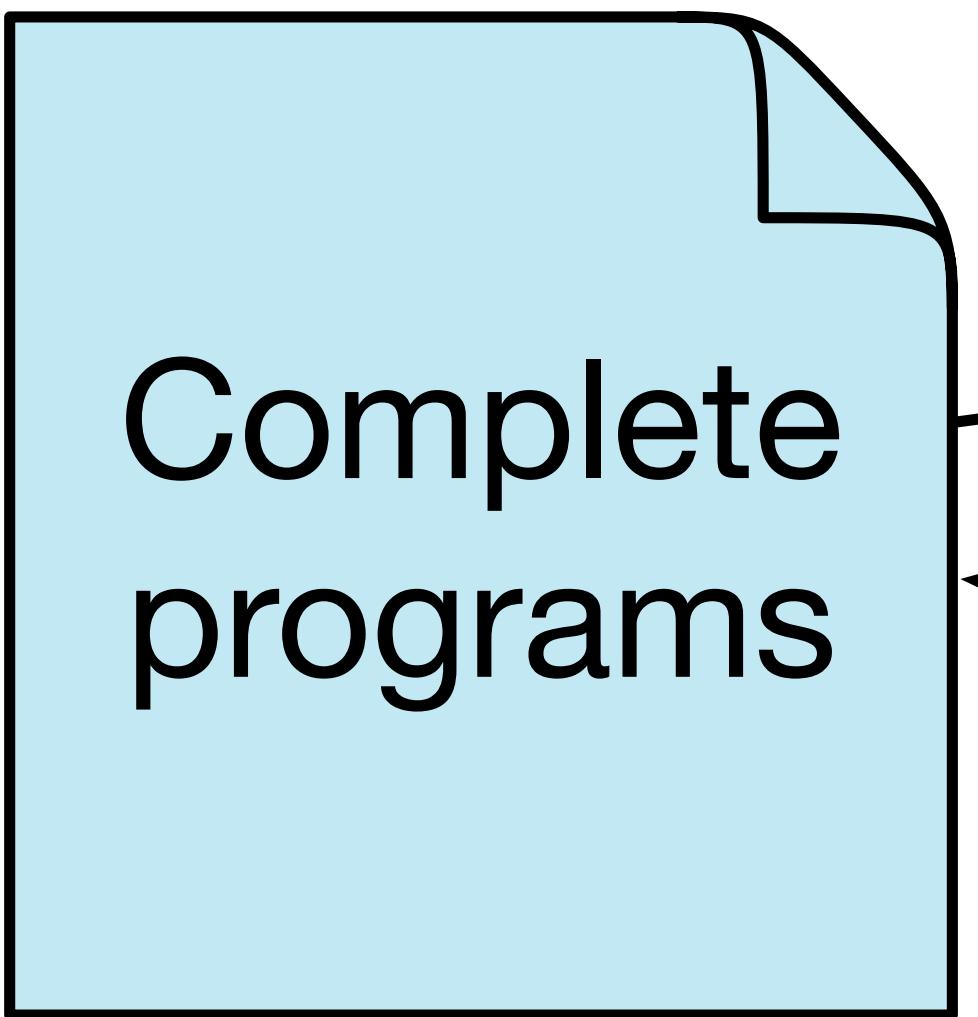
```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + 21; |  
        if($Exp) $Statement  
        else $Exp + $Exp  
        ret$Sub  
    }  
}
```



A code editor interface showing completion proposals for an inferred placeholder variable. The placeholder '\$Exp' is highlighted in blue. The proposals are listed in a dropdown menu:

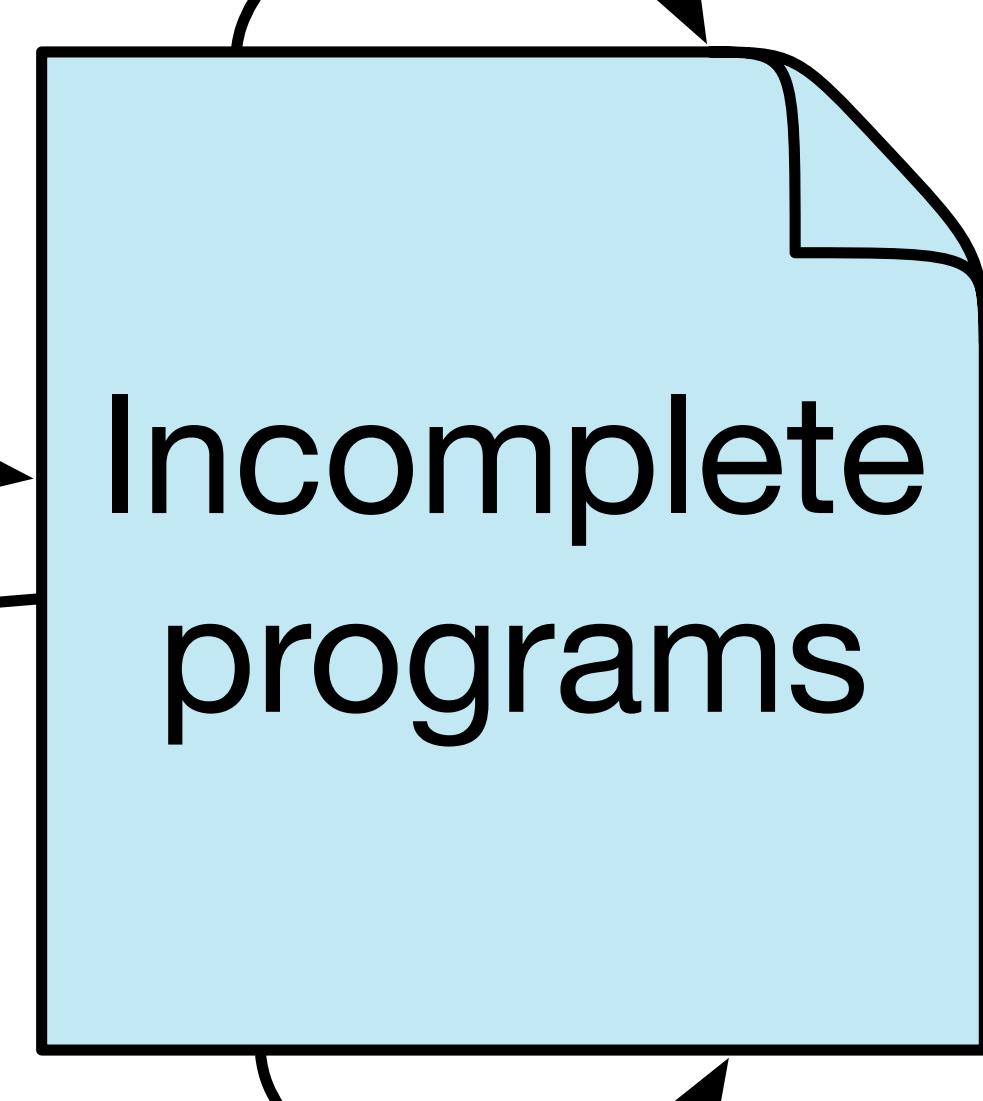
- if(\$Exp) \$Statement
- else \$Exp + \$Exp
- ret\$Sub

Correct  
programs



Infer  
placeholder

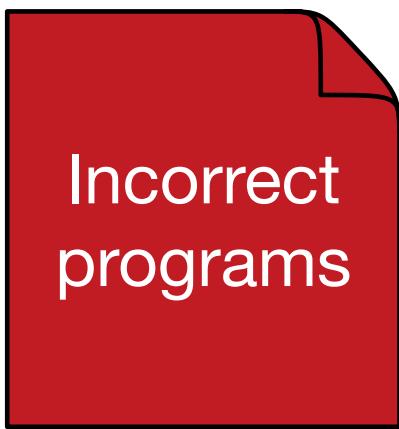
Expand  
placeholder



Expand/overwrite  
placeholders

Infer  
placeholder





```
class A {  
  
    public int m() {  
        int x;  
        x | Syntax error, not expected here: 'x'  
        return x;  
    }  
}
```

# Error Recovery

```
class A {  
  
    public int m() {  
        int x;  
        x  
        re✓VarDecl  
    }    ✓Assign  
}
```

```
class A {  
  
    public int m() {  
        int x;  
        x $ID;  
        return x;  
    }  
}
```

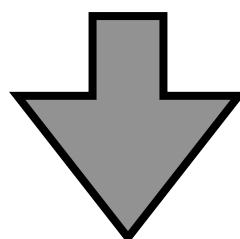
```
class A {  
  
    public int m() {  
        int x;  
        x  
        re✓VarDecl  
    }    ✓Assign  
}
```

```
class A {  
  
    public int m() {  
        int x;  
        x = $Exp;  
        retu✓Add  
    }    $Exp + $Exp  
        ✓Sub  
        ✓Mul  
        ✓Lt  
        ✓VarRef
```

# Insertion Rules

context-free syntax //regular syntax rules

```
Statement.VarDecl = <<Type> <ID>;>
Statement.Assign  = <<VarRef> = <Exp>;>
```



// derived insertion rules for placeholders  
context-free syntax

```
Type.Type-Plhdr          = {symbol-insertion}
ID.ID-Plhdr              = {symbol-insertion}
Statement.Statement-Plhdr = {symbol-insertion}
VarRef.VarRef-Plhdr      = {symbol-insertion}
Exp.Exp-Plhdr            = {symbol-insertion}
```

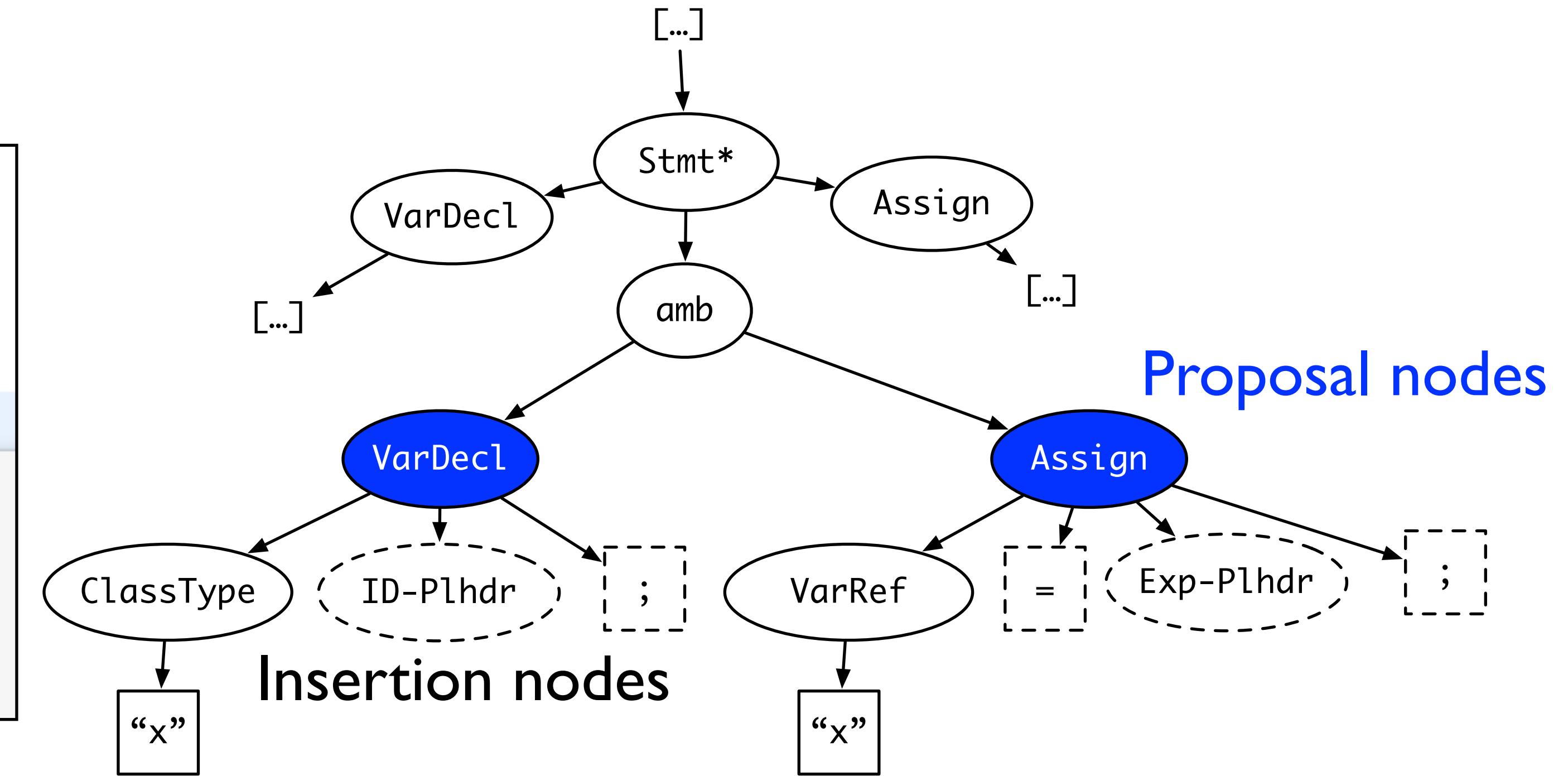
// derived insertion rules for literals  
lexical syntax

```
"=" = {symbol-completion}
";" = {symbol-completion}
```

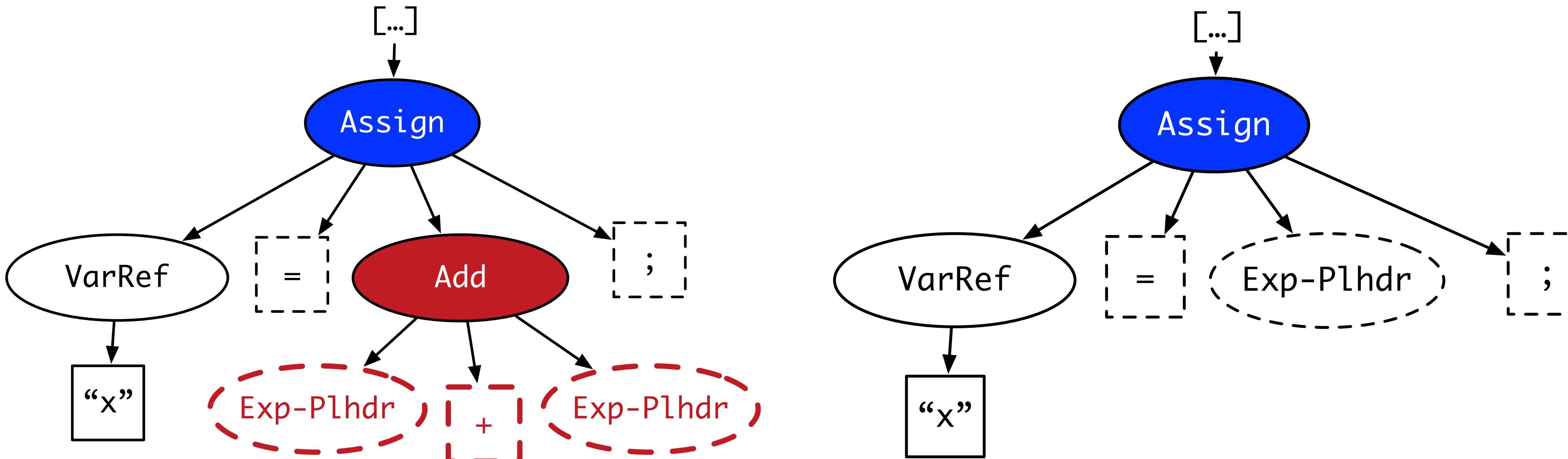
Empty productions

# Apply Insertion Rules at Cursor

```
class A {  
    public int m() {  
        int x;  
        x |  
        re✓ VarDecl  
        ✓ Assign  
    }  
}
```



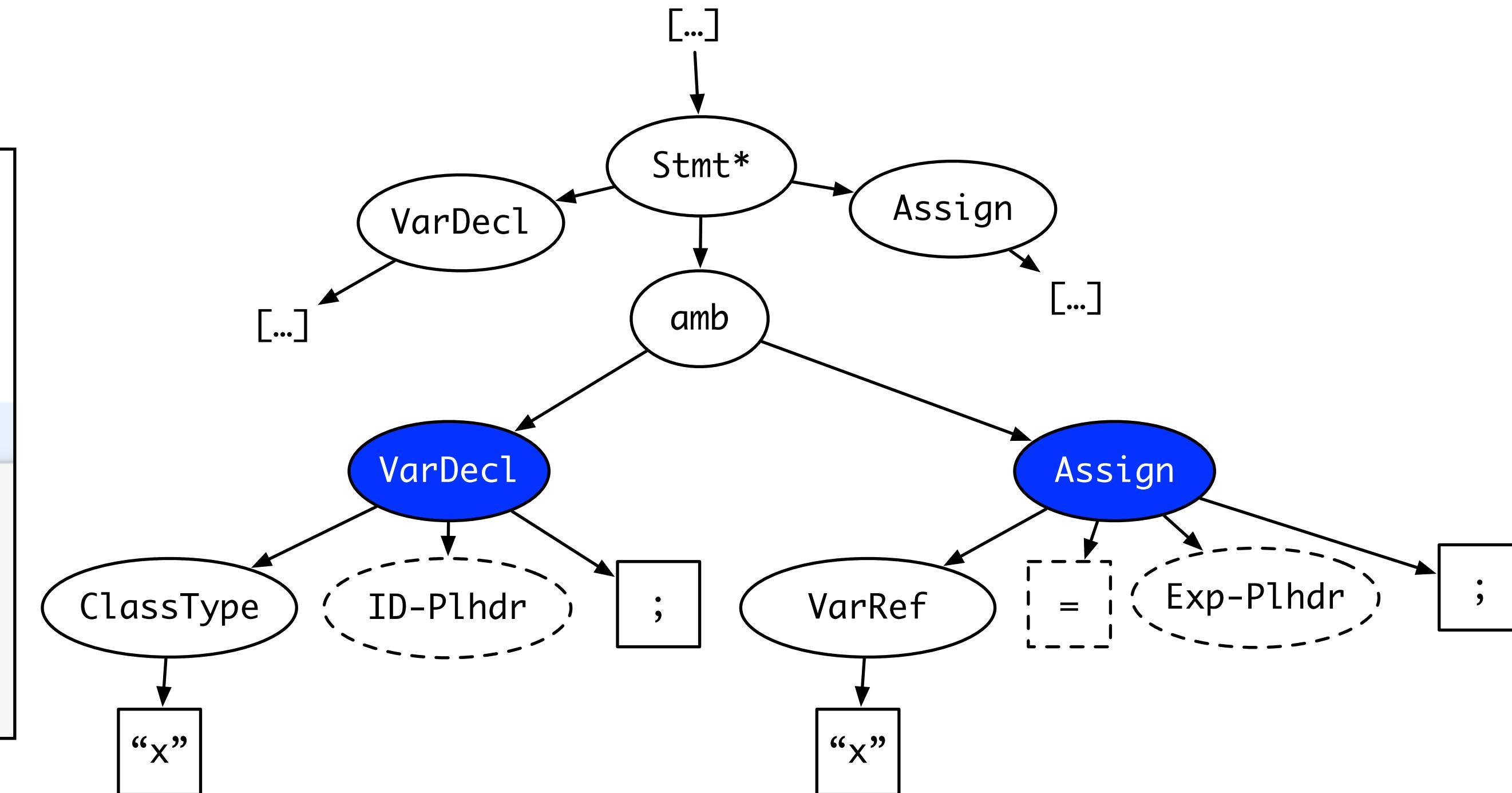
# Limit Search Space



Use the simplest possible expansions

# Greedy Recovery

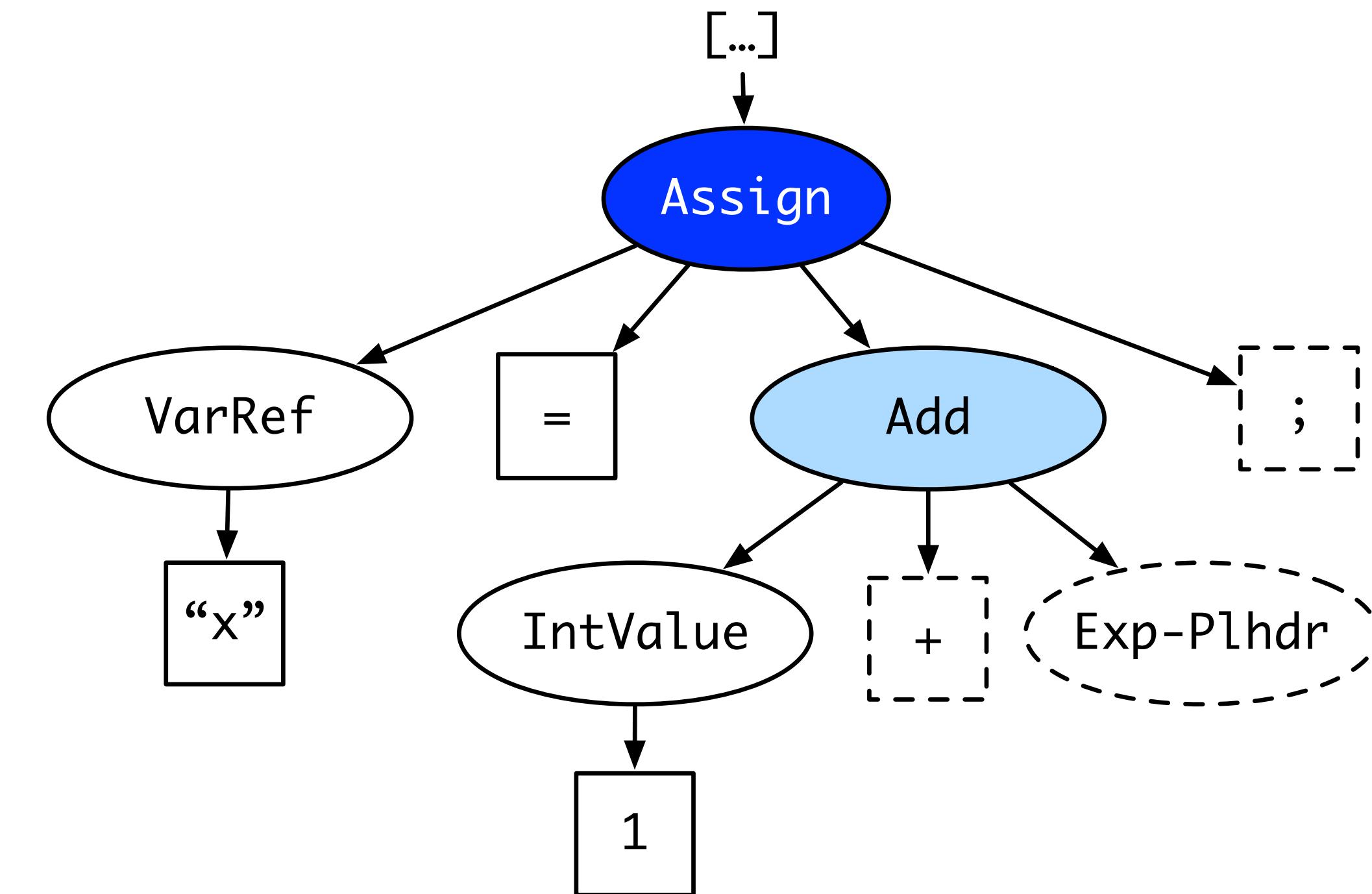
```
class A {  
  
    public int m() {  
        int x;  
        x ;  
        re+ID-Plhdr x = $Exp;  
    }  
}
```



Include postfix in recovery proposal

# Nested Proposal Nodes

```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + |  
        return x; ✓ Assign-Add x = 21 + $Exp;  
    }  
}
```

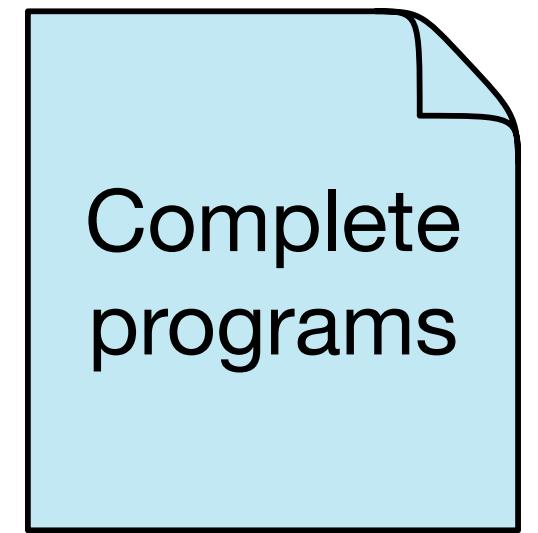


```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + |  
        return x; ✓ Assign-Add | x = 21 + $Exp;  
    }  
}
```

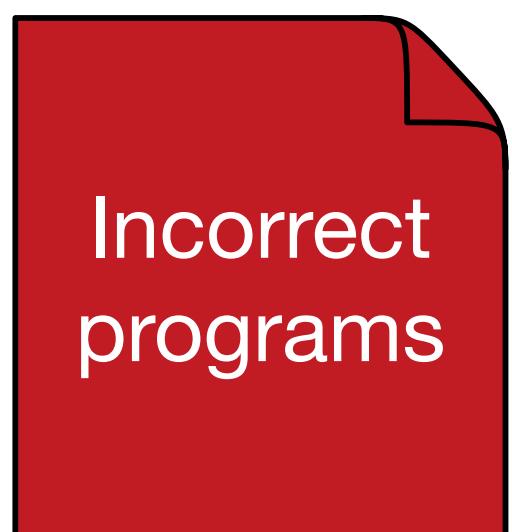
```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + $Exp; |  
        return x; + Add | ($Exp + $Exp)  
    }  
}
```

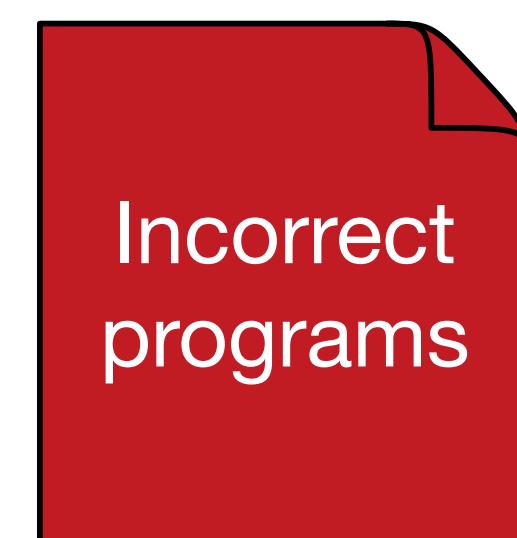
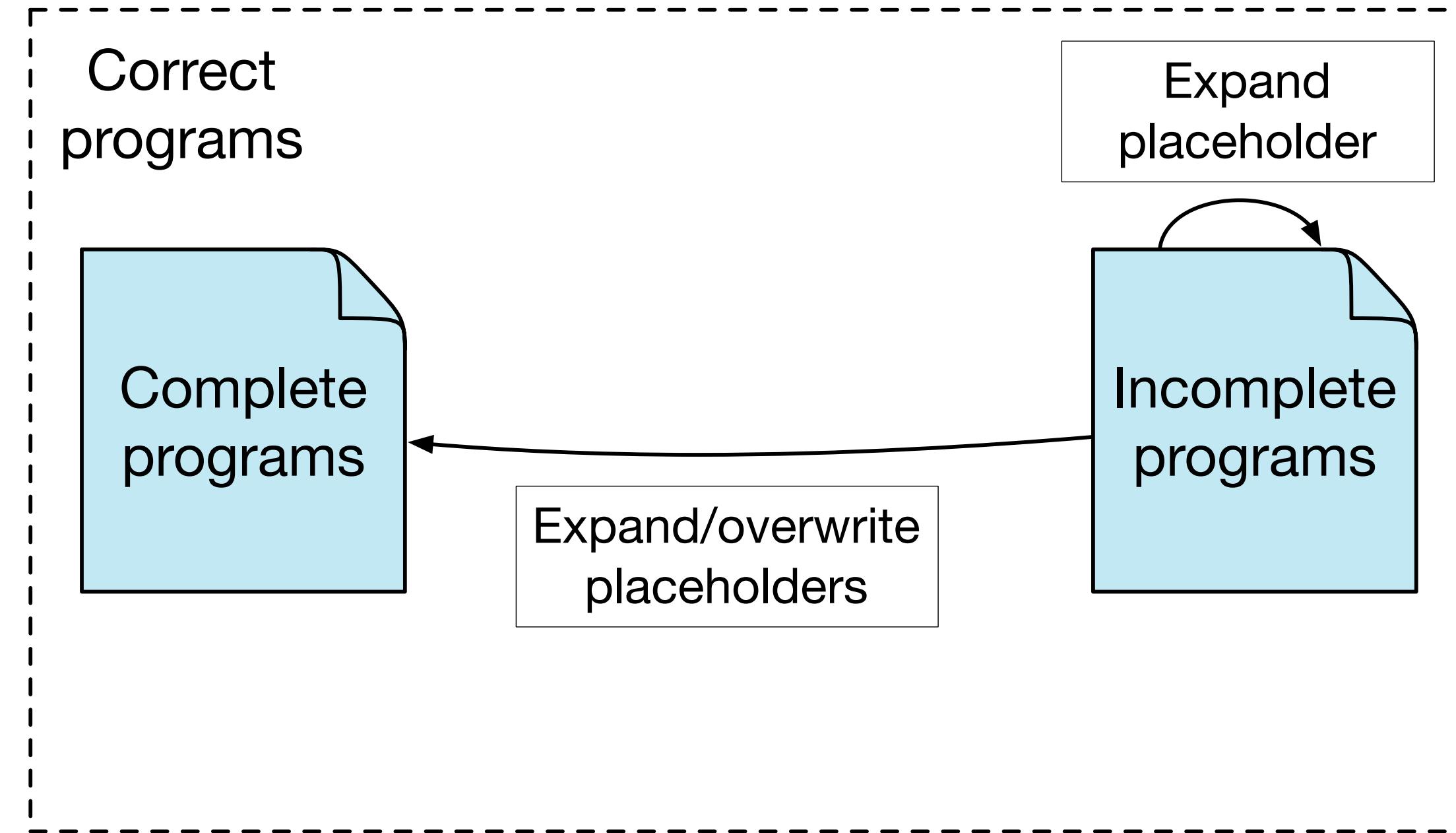
# Syntactic Completion

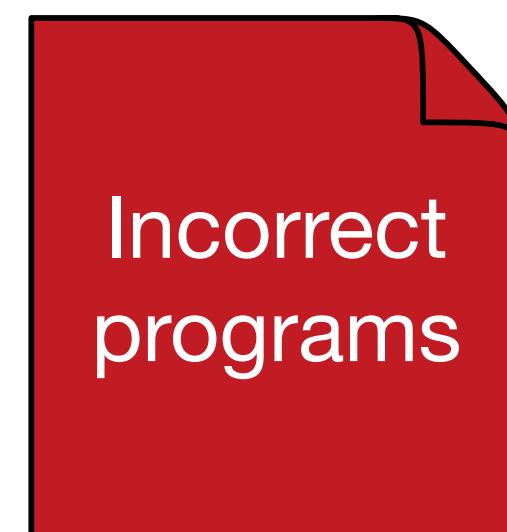
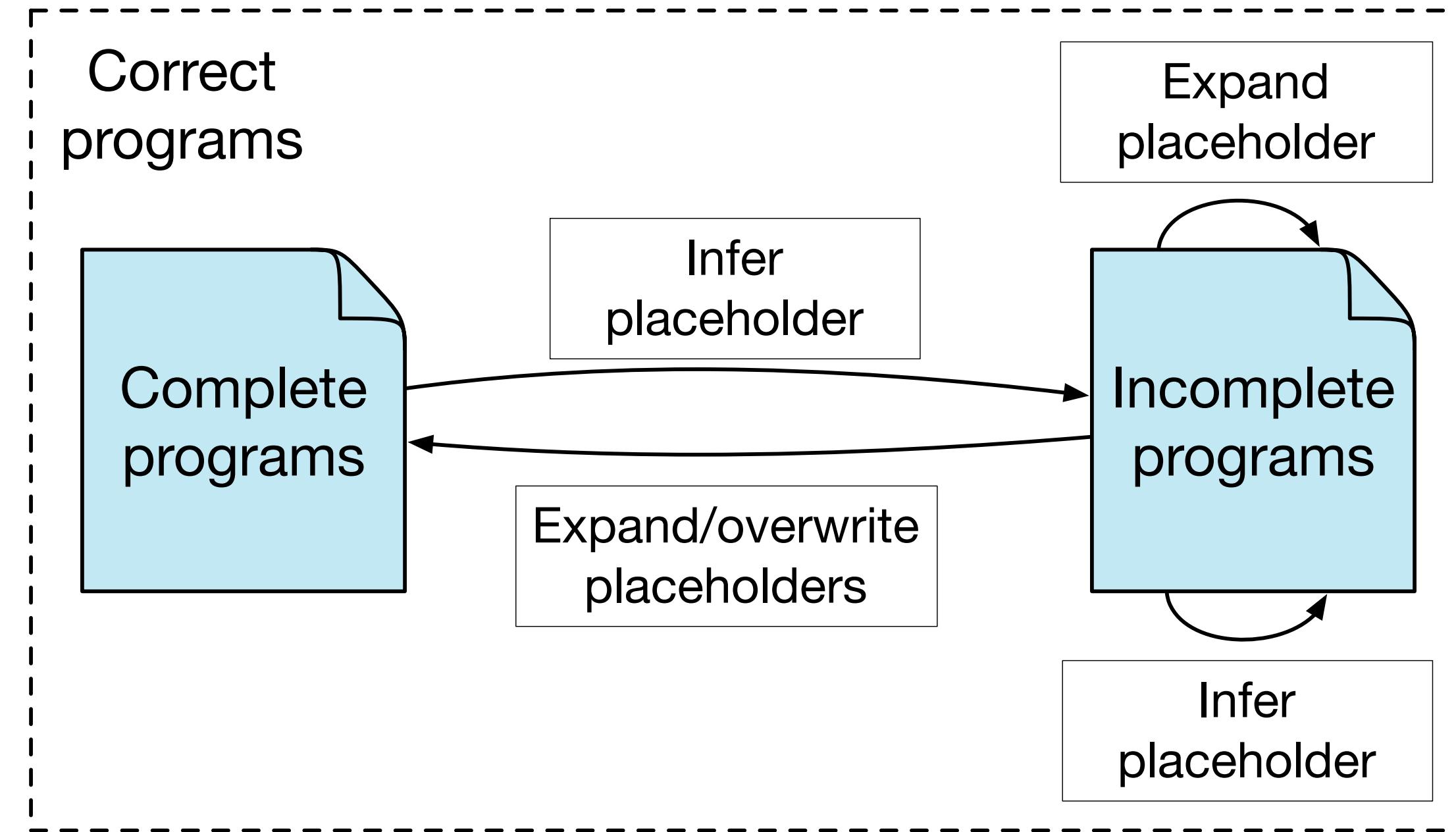
Correct  
programs

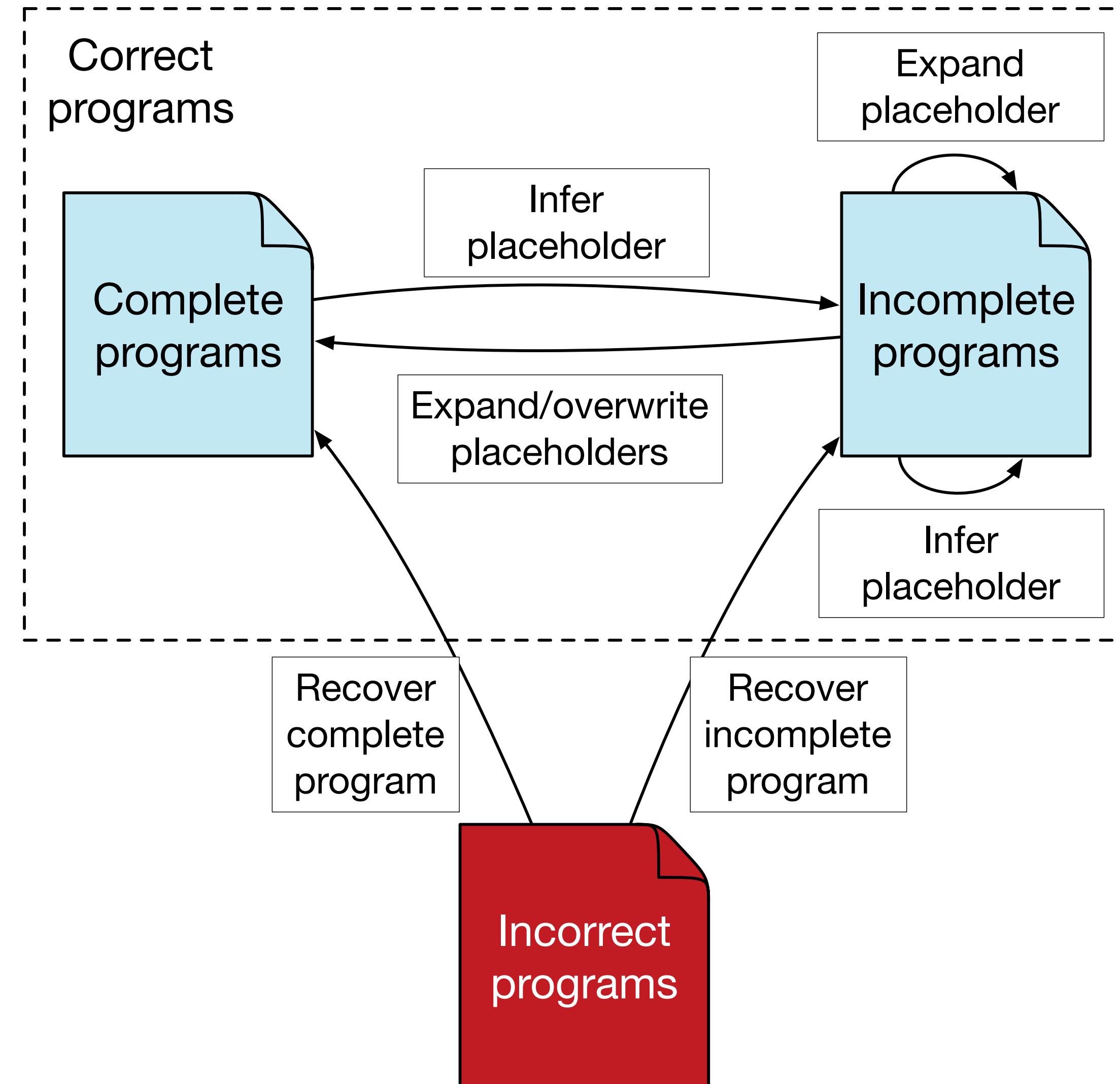


Incomplete  
programs









# **Syntactic Services: Summary**

# Syntactic Services from Syntax Definition

## Syntax definition = CFG++

- Concrete syntax: notation
- Constructors: abstract syntax / structure
- Associativity & priority: disambiguation
- Templates: formatting

## Parsing

- Mapping text to abstract syntax
- Syntax checking
- Permissive grammars: error recovery

# Syntactic Services from Syntax Definition

## Syntax coloring

- ESV: mapping token sorts to colors

## Formatting

- Unparsing: mapping from abstract syntax to concrete syntax
- Pretty-printing: derived from templates
- Parenthesization: derived from disambiguation declarations

## Completion

- Make incompleteness explicit, part of the structure
- Generating proposals: derived from structure
- Pretty-printing proposals: derived from templates

# Next: Parsing

Classical compiler textbook

Chapter 4: Syntax Analysis

## **Compilers: Principles, Techniques, and Tools, 2nd Edition**

Alfred V. Aho, Columbia University

Monica S. Lam, Stanford University

Ravi Sethi, Avaya Labs

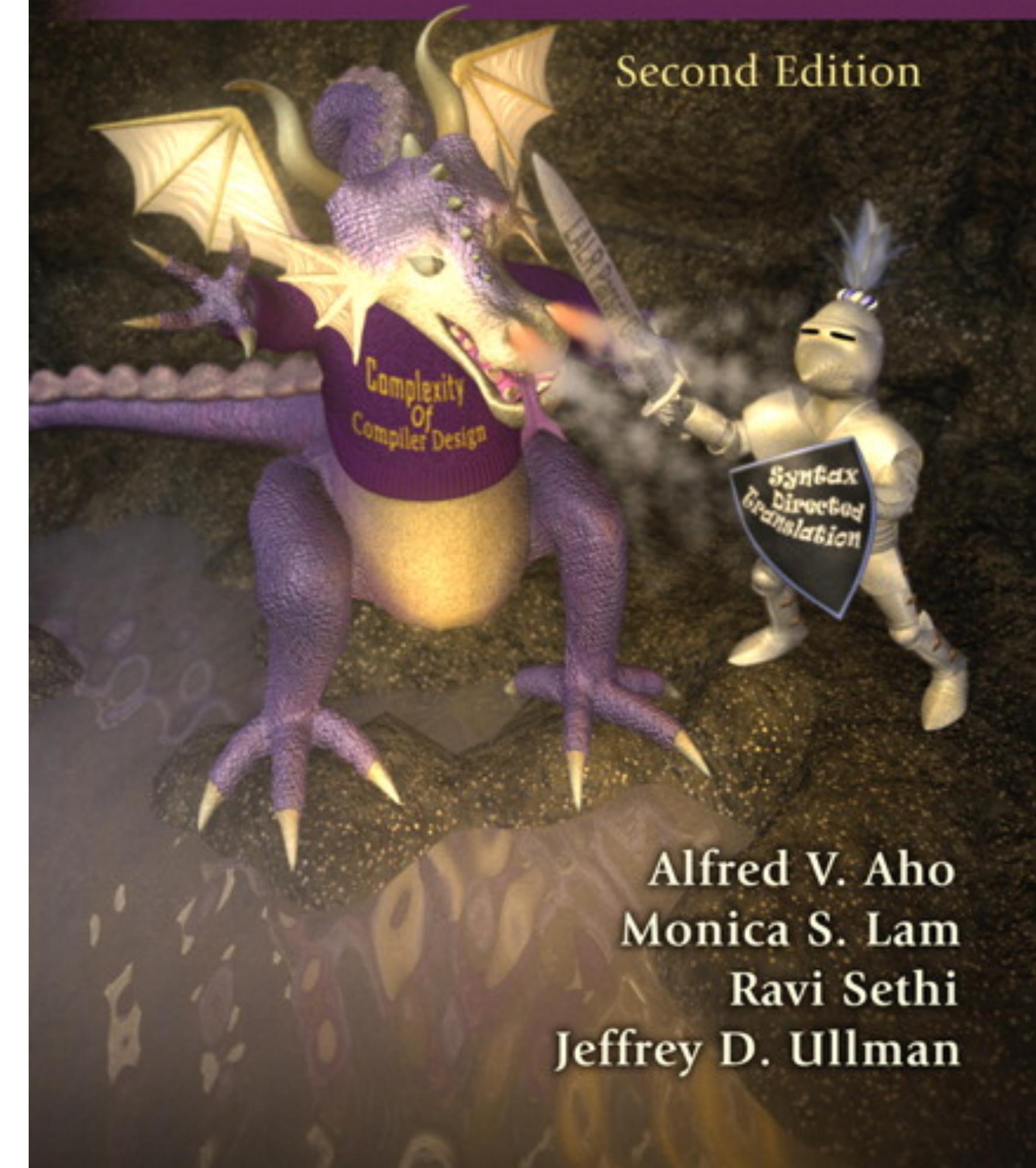
Jeffrey D. Ullman, Stanford University

2007 | Pearson

# **Compilers**

*Principles, Techniques, & Tools*

Second Edition



Alfred V. Aho  
Monica S. Lam  
Ravi Sethi

Jeffrey D. Ullman

Except where otherwise noted, this work is licensed under

