Linters

What are they good for?

What is a linter?

- Depends on who you ask and what you are looking for
  - A tool to help you improve your code
  - A tool to identify problems with code
  - A static code analysis tool
- Linters analyze your code and provide feedback on it
- Many, MANY different linters out there

History of Linters

- Originally developed by Stephen Johnson at Bell Labs in 1978
- Named after “lint”, the stuff you find in your dryer
- Original reasoning for developing a linter was to debug language grammars he was working on
- His goal was to create a distinct program that aided with development by catching errors early – still used for that exact same purpose today

Linter vs Compiler

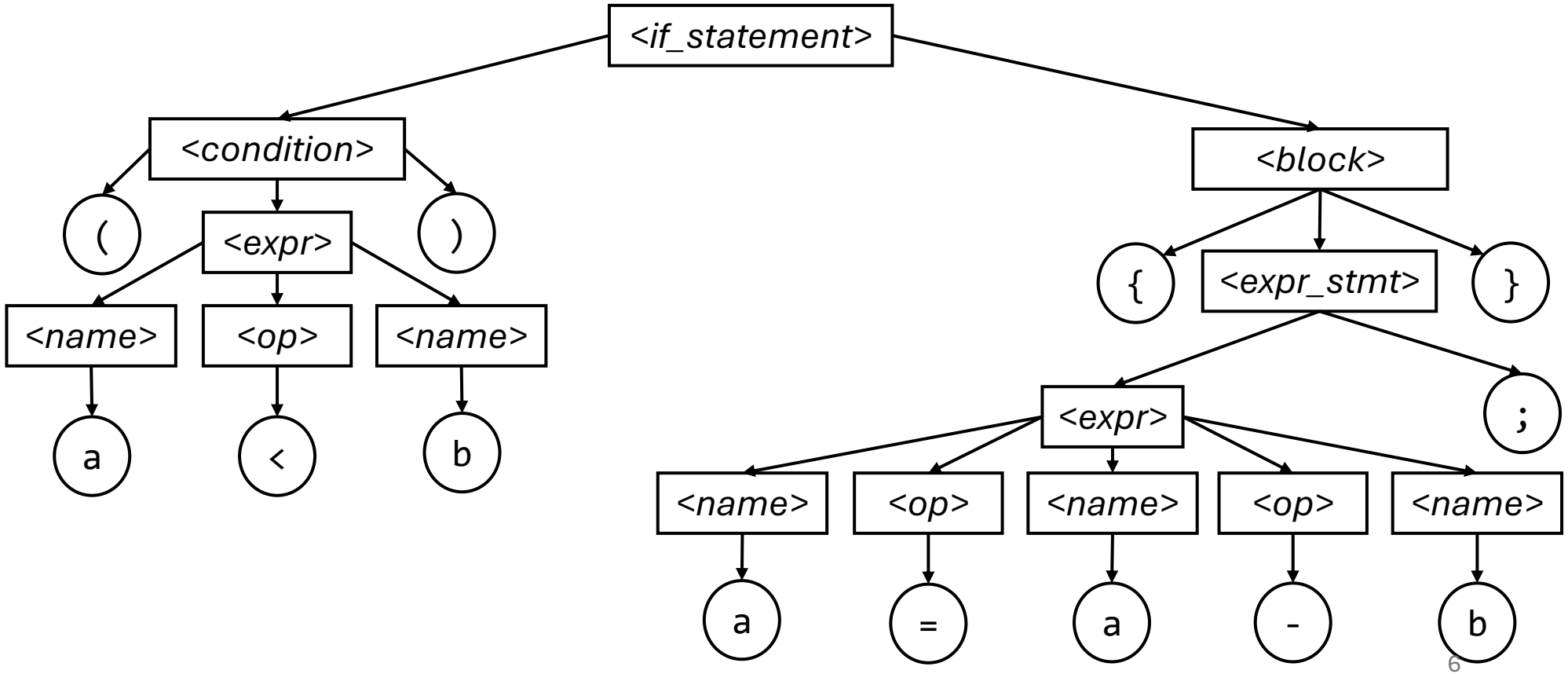
- Both Linters and Compilers will give you errors – why bother with two? What's the difference?
- Compilers take the written code, attempt to convert it to runnable machine code, and compile an executable file.
  - Compiler errors happen when the compiler runs into issues trying to generate this runnable machine code – bad syntax, bad typing, etc.
- Linters perform "static program analysis", which attempts to analyze the code purely in its textual format
  - Raw code files are examined and analyzed – no conversion to machine code is every performed.
  - Typically, much faster and can be done on the fly

Simple Types of Analysis

Abstract Syntax Tree (AST)

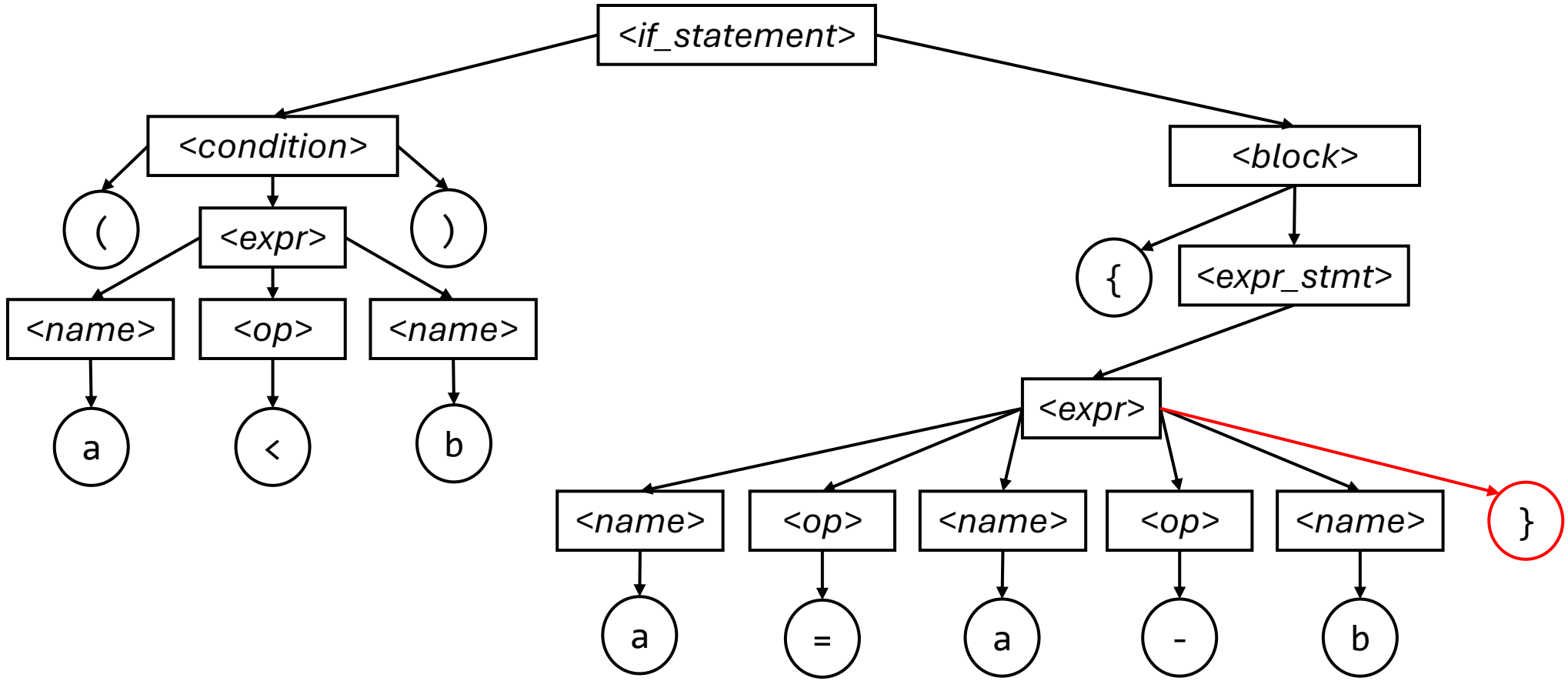
- Tree data structure that represents the structure and syntax of code.

```
if(a < b) {
  a = a – b;
}
```



Syntax Error Correction – Missing Semicolon

```
if(a < b) {
  a = a - b
}
```



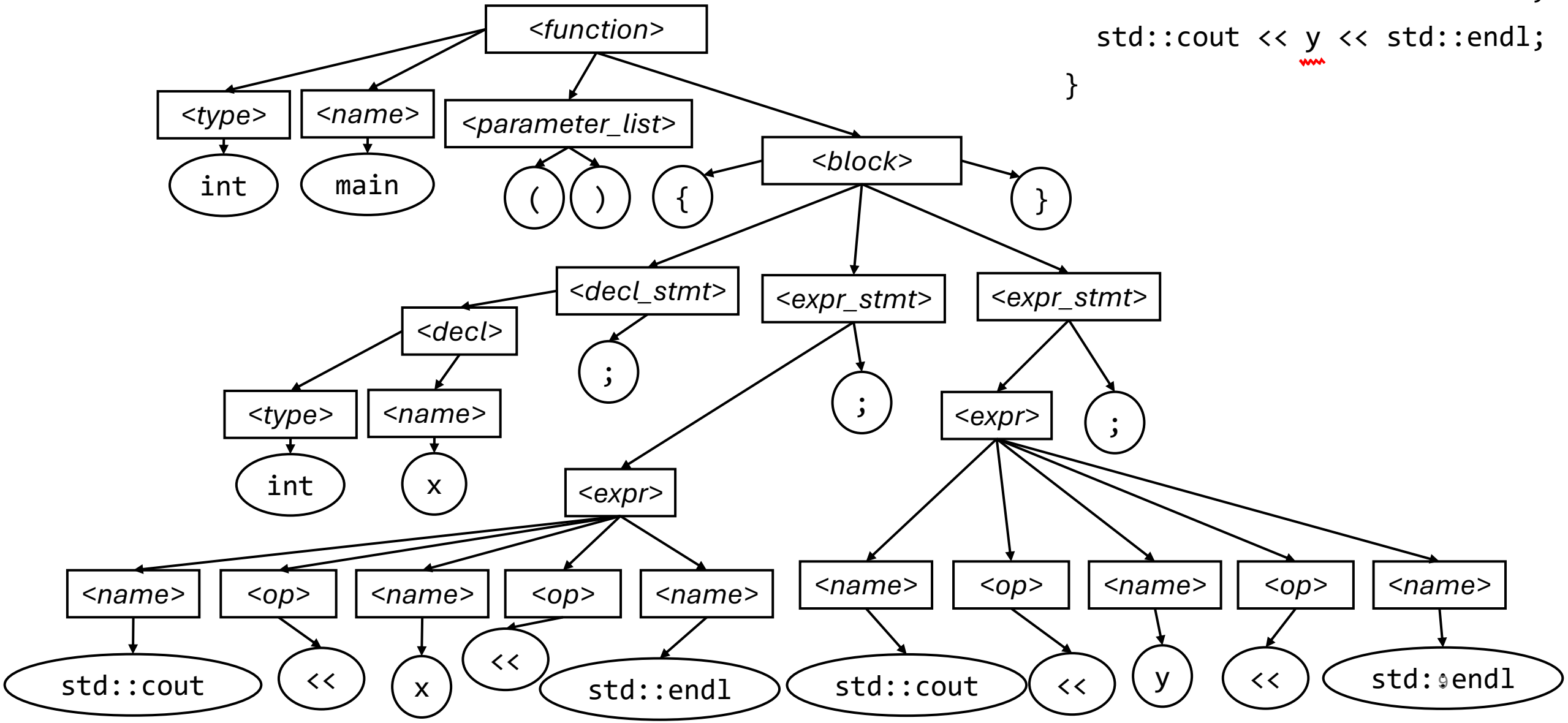
Undefined Names

```
int main() {
  int x = 10;
  std::cout << x << std::endl;
  std::cout << y << std::endl;
}
```

- Syntactically correct, but the code won’t compile
- Harder to detect with ASTs
  - Still possible, but requires more work

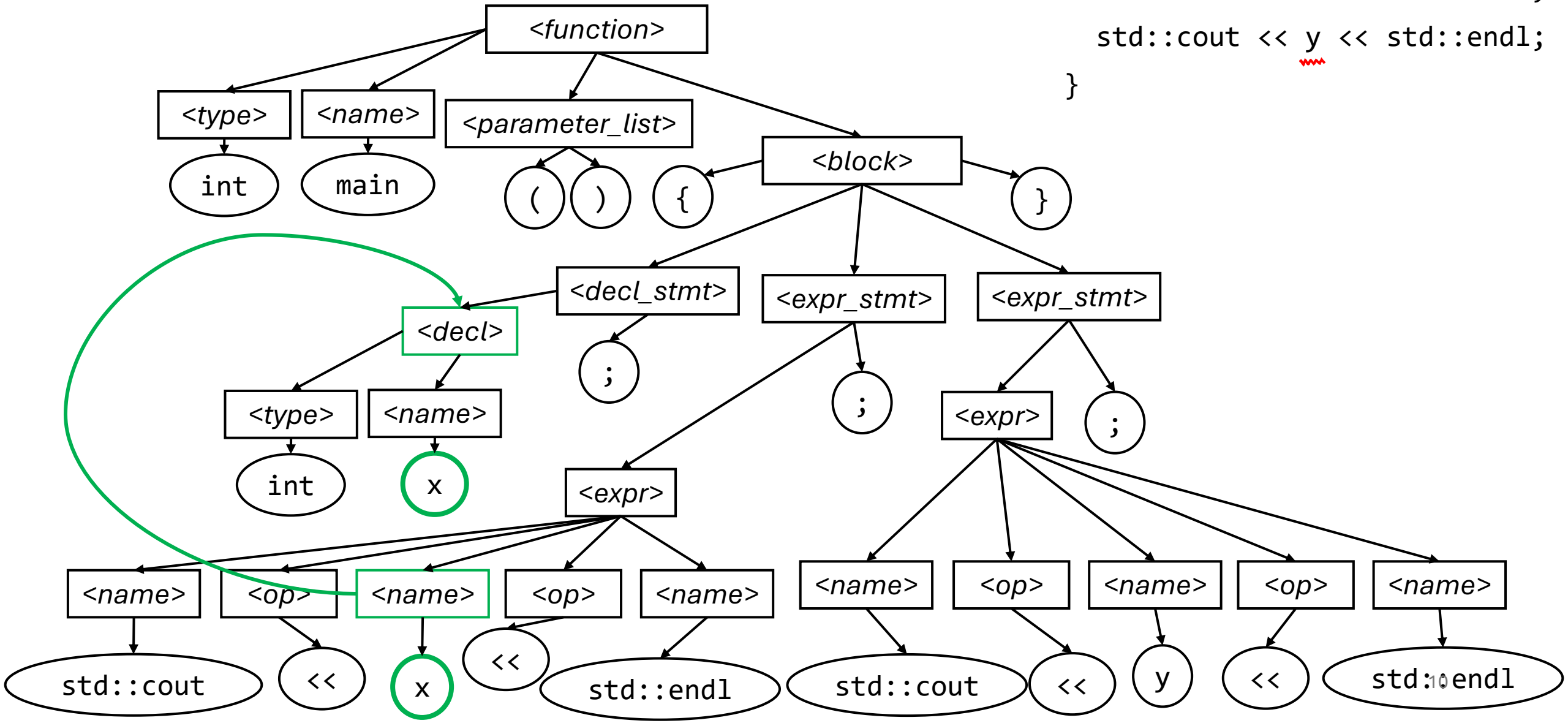
Undefined Names

```
int main() {
  int x = 10;
  std::cout << x << std::endl;
  std::cout << y << std::endl;
}
```



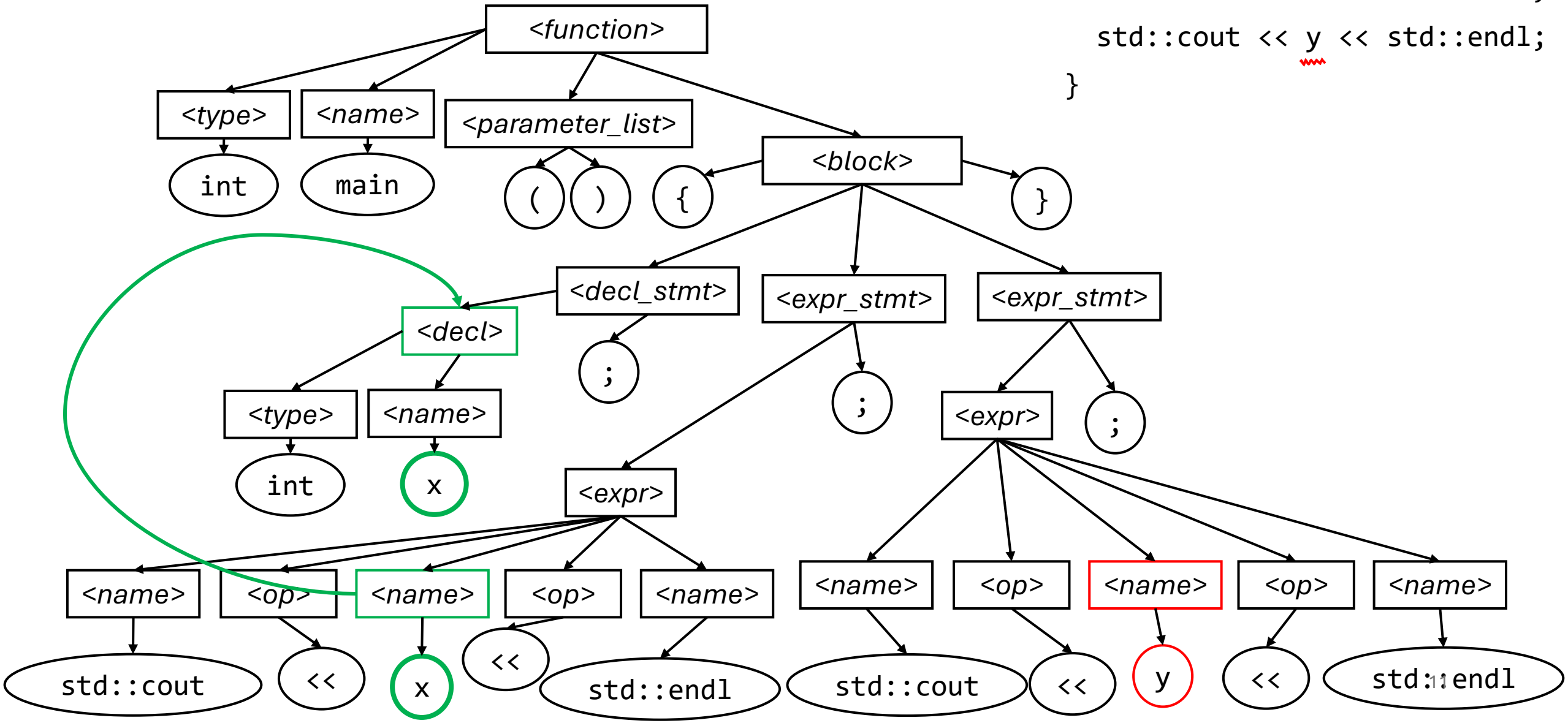
Undefined Names

```
int main() {
  int x = 10;
  std::cout << x << std::endl;
  std::cout << y << std::endl;
}
```



Undefined Names

```
int main() {
  int x = 10;
  std::cout << x << std::endl;
  std::cout << y << std::endl;
}
```



Unused Variables Warning

```
#include <iostream>

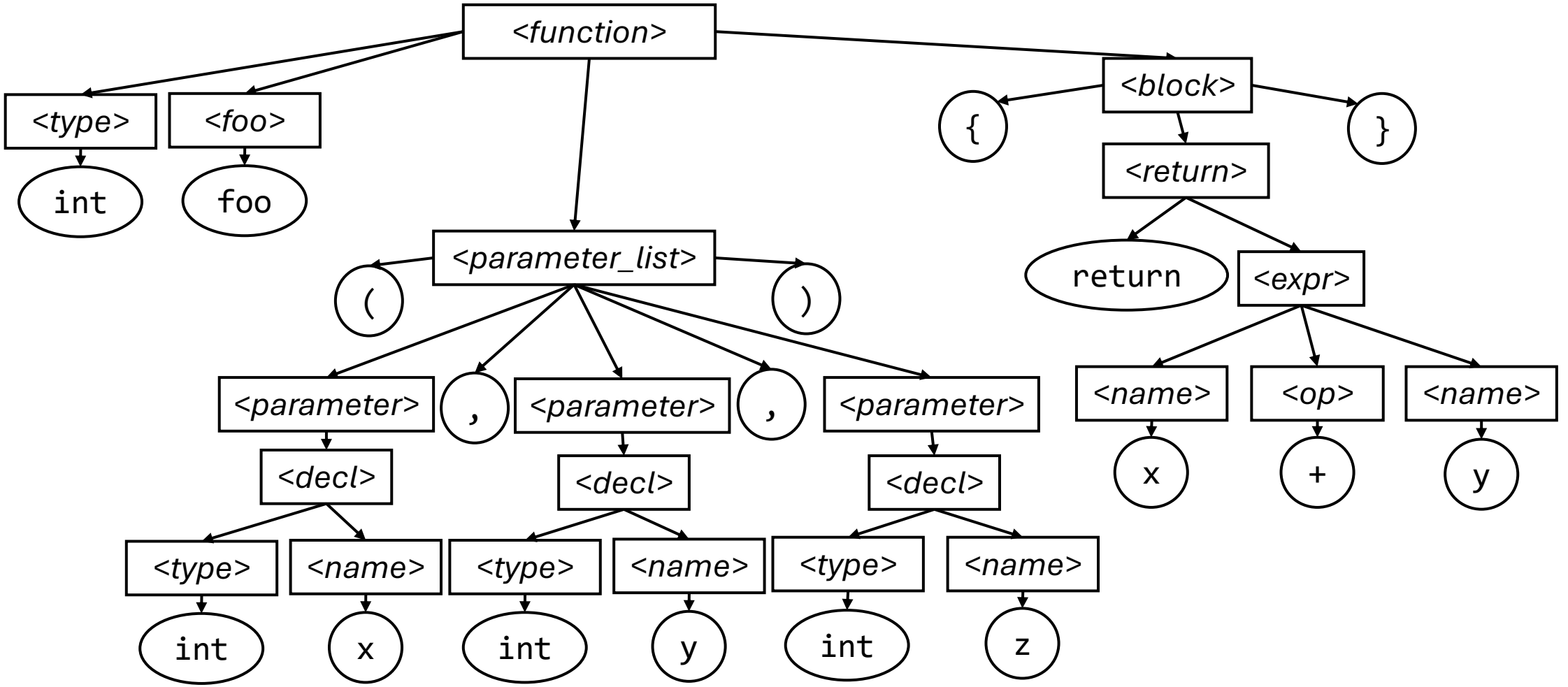
int foo(int x, int y, int z) {
  return x + y;
}

int main() {
  foo(1,3,5);
}
```

- Syntactically correct AND compilable – how do we detect?

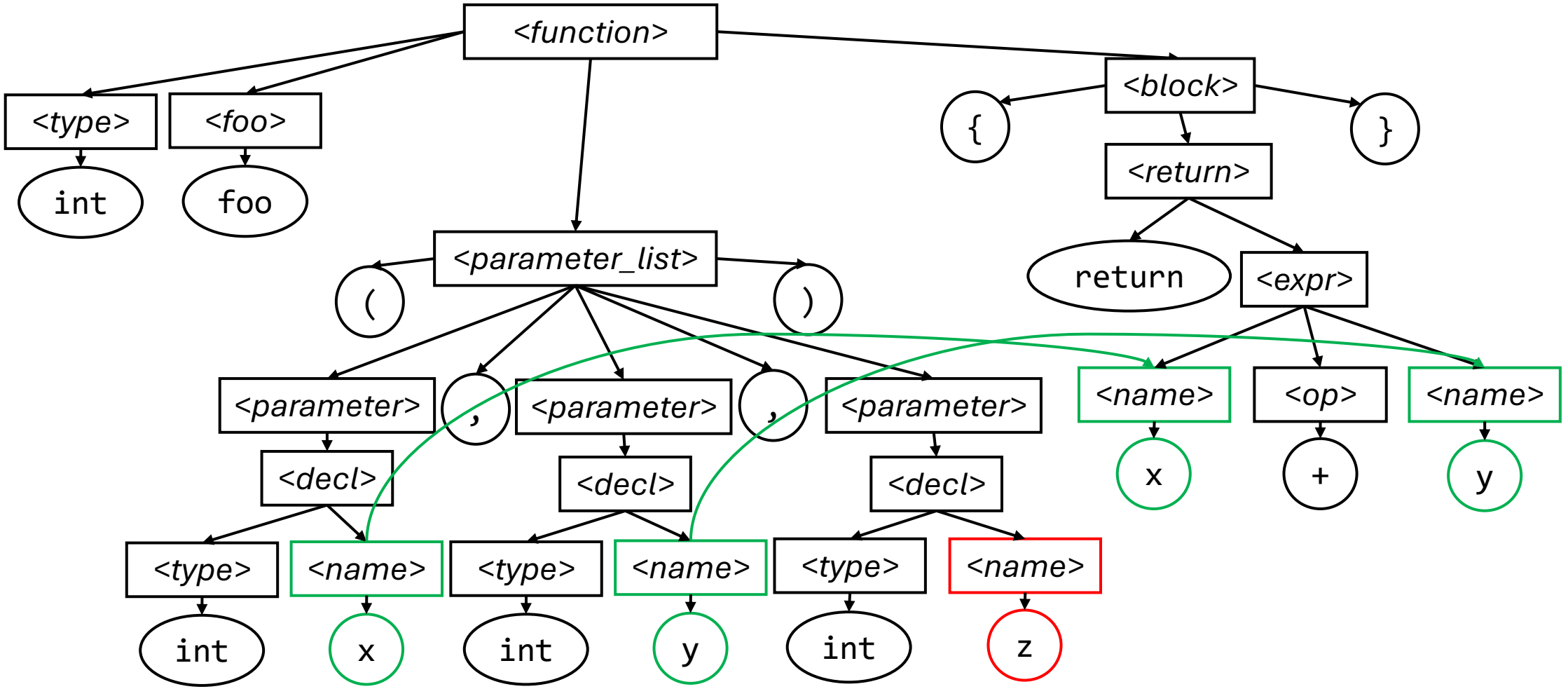
Unused Variables

```
int foo(int x, int y, int z) {
  return x + y;
}
```



Unused Variables

```
int foo(int x, int y, int z) {
  return x + y;
}
```



Code Style

- Many different coding styles exist
- Curly Brace styles:

```
if(a) { b; }
```

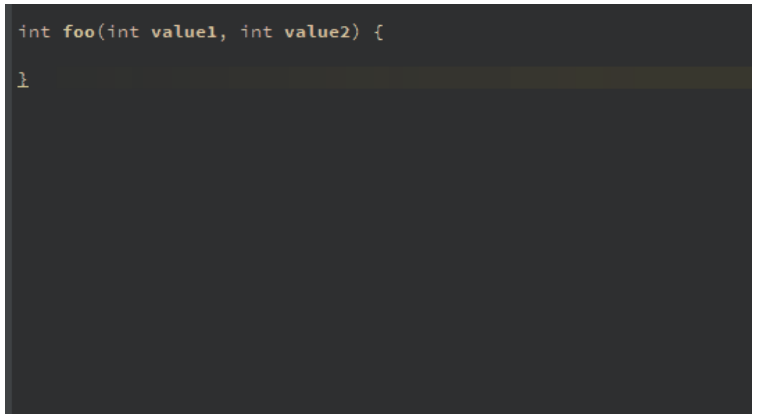
```
if(a) {
  b;
}
```

```
if(a)
{
  b;
}
```

- Name styles:
  - camelCase
  - snake_case
  - PascalCase
- Others

Code Style

- Linters can help enforce coding styles
- Warn users if multiple naming styles are used
  - Give user a warning if 1 variable is snake_case while the others are camelCase
- Automatically format curly braces
  - Some coding environments will automatically place and position curly braces when the left curly brace is inserted.
- Autocomplete
  - Autocomplete common names in a file



Advanced Types of Analysis

Naming Rules

- Current research is focused on developing good guidelines for how to name variables, functions, classes, etc.
- Using analysis like natural language processing, part of speech tagging, and name splitting, we can evaluate the quality of names
- Role of linters – getting the names from the source code, applying rules, and then displaying warnings.

Naming Rules

- `int allMyLittleDucksSwimming;` - Name too long
- `int Result_Count;` - Wrong naming style for variables
- `class Compile {}` – Class names shouldn’t be verbs
- `void printer() {}` – Function names shouldn’t be nouns

Higher Level Code Analysis

- Beyond just looking at individual elements of the code, larger portions of code can be analyzed.
- Now instead of just looking at names or missing syntax, whole code functionality can be analyzed.
  - Measuring the flow of the code
  - Analyzing classes and functions for features

Measuring Idiomatic Compliance

- What are idioms?
  - A code fragment that has a semantic role in programming
  - Typically considered the “right” way to do something in a language, whether for code clarity, execution speed, or just general community acceptance

```
array = []
for i in range(10):
  array.append(i)
```

VS

```
array = [i for i in range(10)]
```

  - Idioms are vital for the development of languages and for becoming fluent in a language

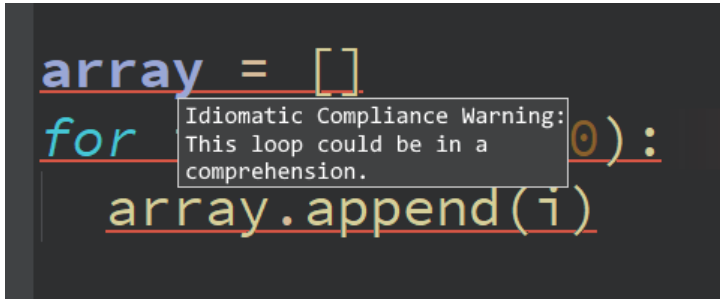
Measuring Idiomatic Compliance

- Code is **in** compliance if it uses an idiom where it can
- Code is **out** of compliance if it uses code that isn't idiomatic.

- Code is not necessarily *wrong* if it isn't in compliance
  - Code can work perfectly without using any idioms
- However, idioms are preferred to make code more consistent and readable for others

Measuring Idiomatic Compliance

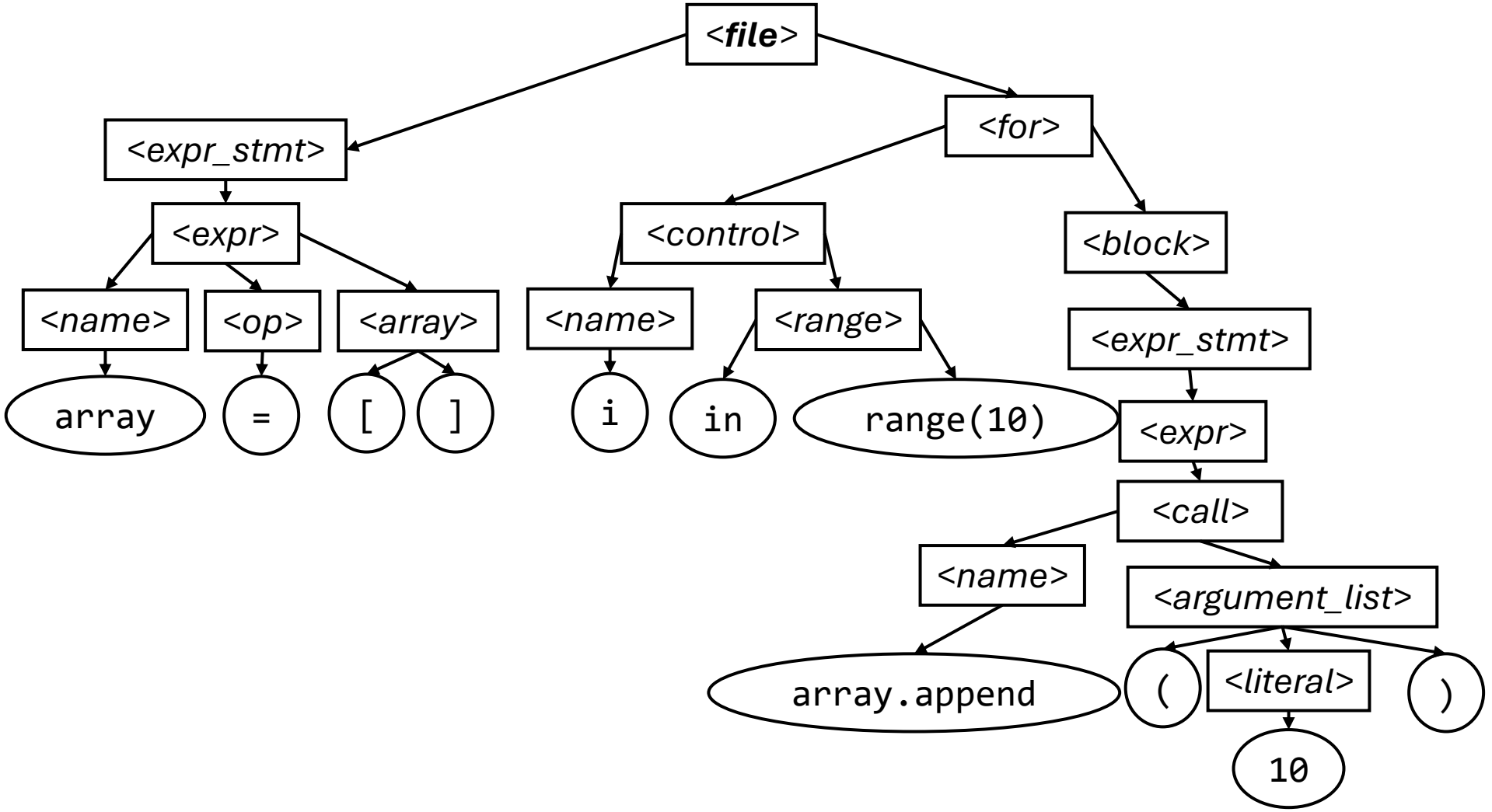
- Linters can be used alongside idioms to annotate code with warnings



- Non-idiomatic code could be refactored
- Using for loop example – by manipulating the AST, code can be refactored

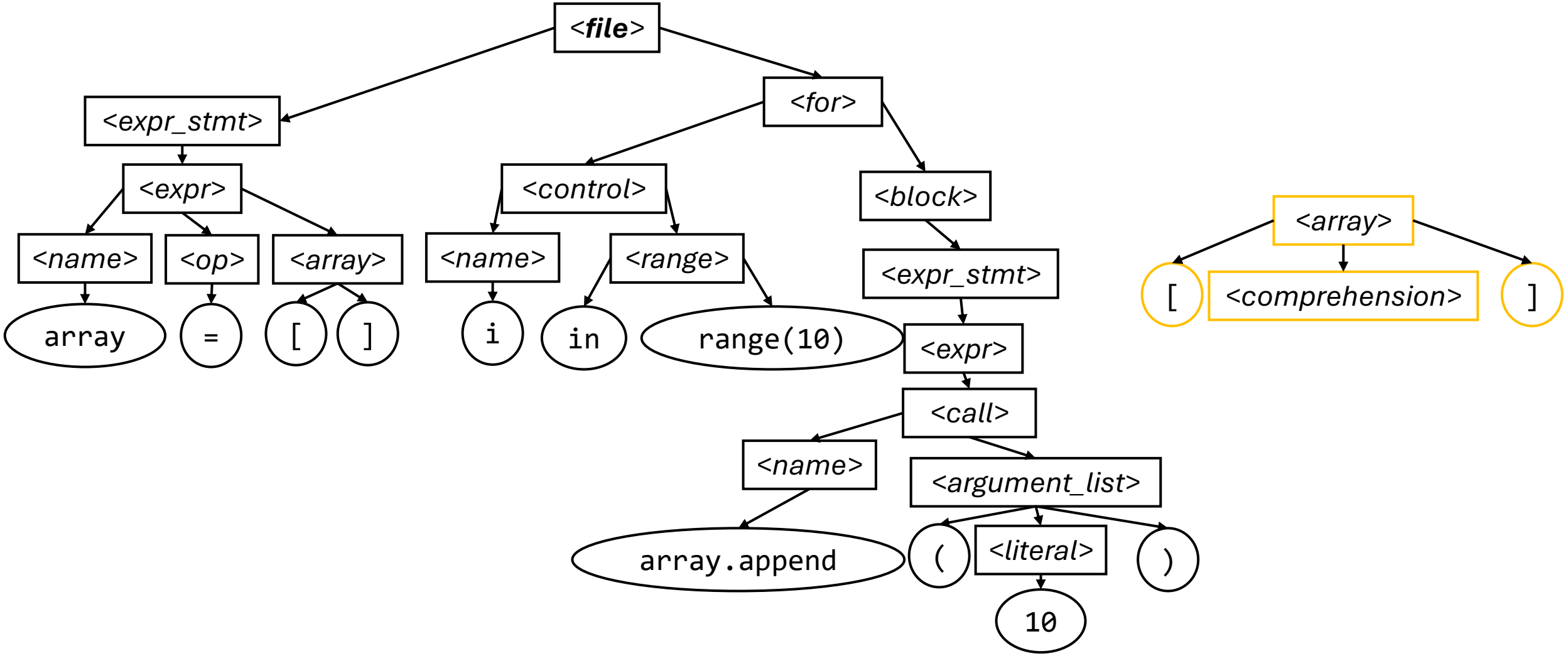
Refactoring for Idiomatic Compliance

```
array = []
for i in range(10):
  array.append(i)
```



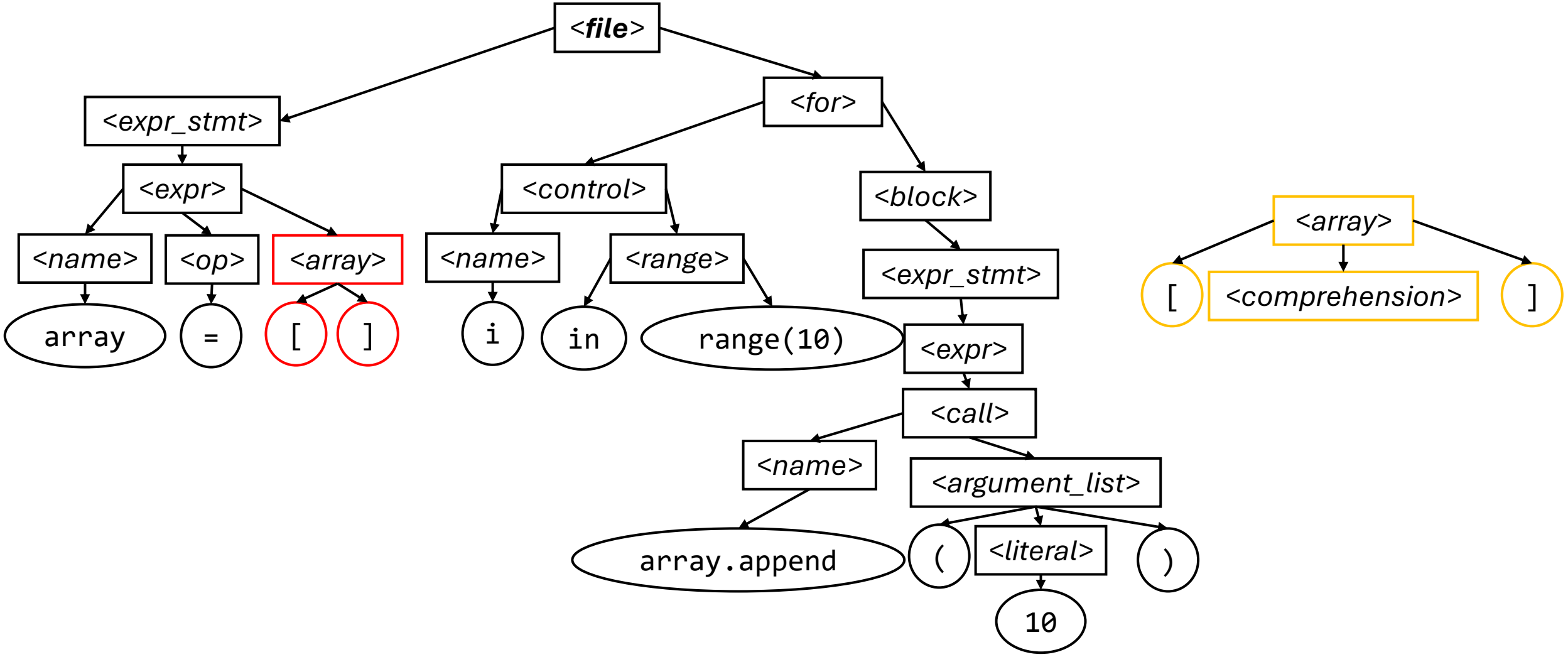
Refactoring for Idiomatic Compliance

```
array = []
for i in range(10):
  array.append(i)
```



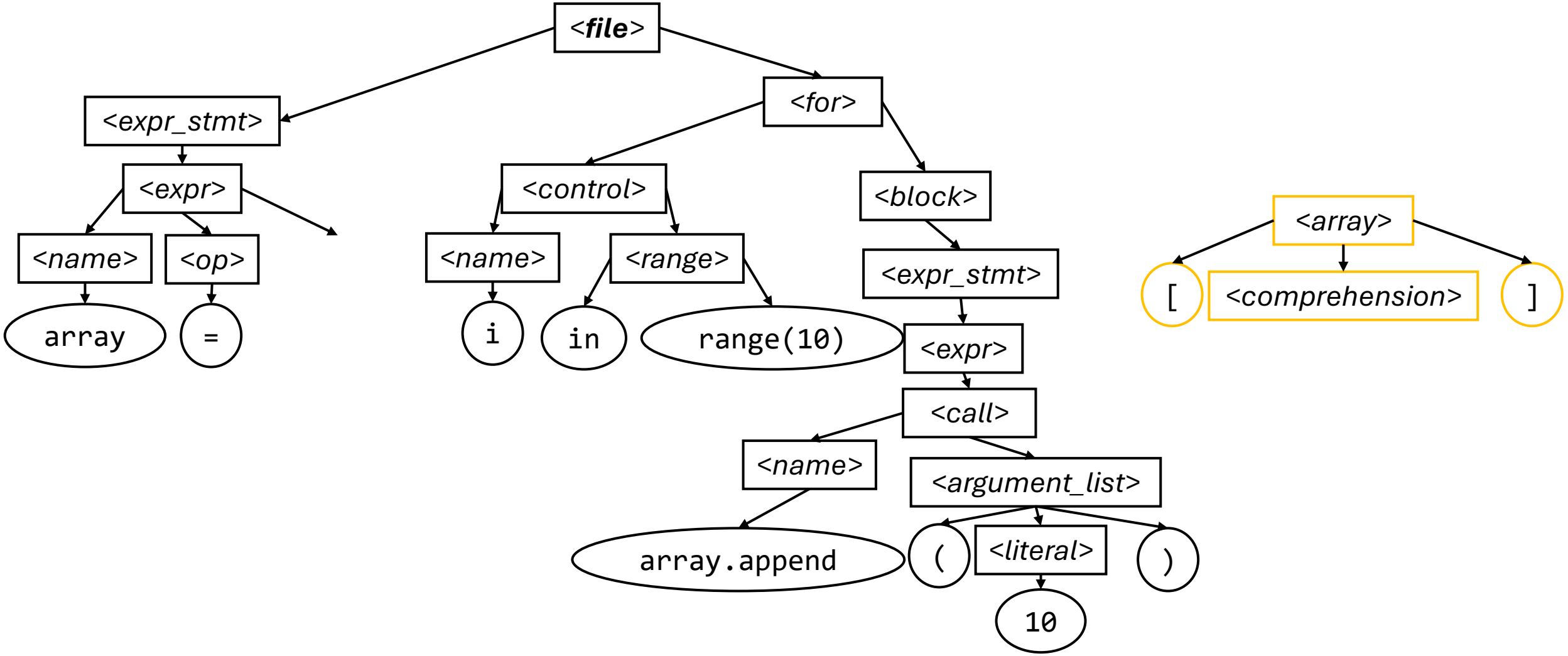
Refactoring for Idiomatic Compliance

```
array = []
for i in range(10):
  array.append(i)
```



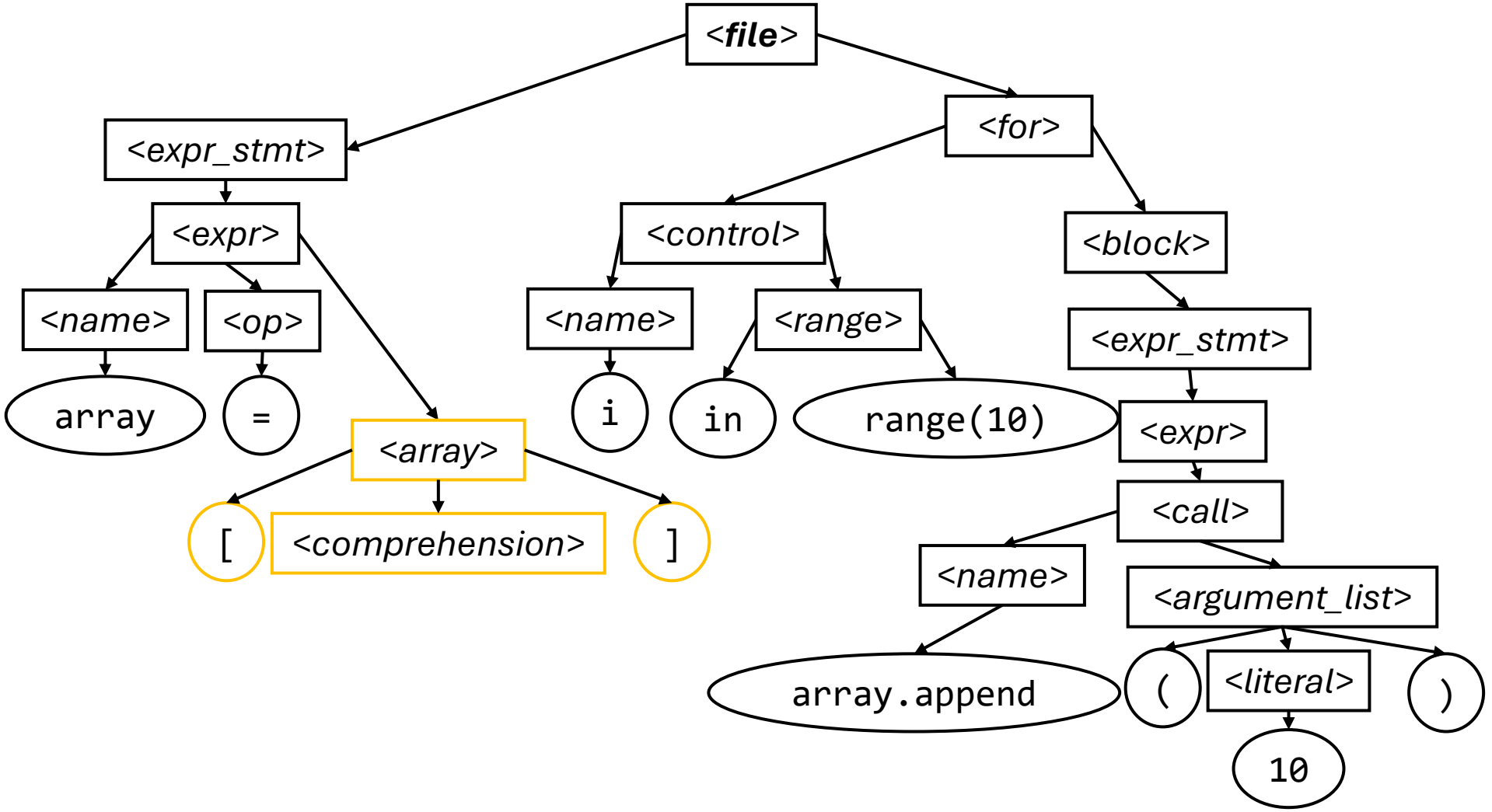
Refactoring for Idiomatic Compliance

```
array =
for i in range(10):
  array.append(i)
```



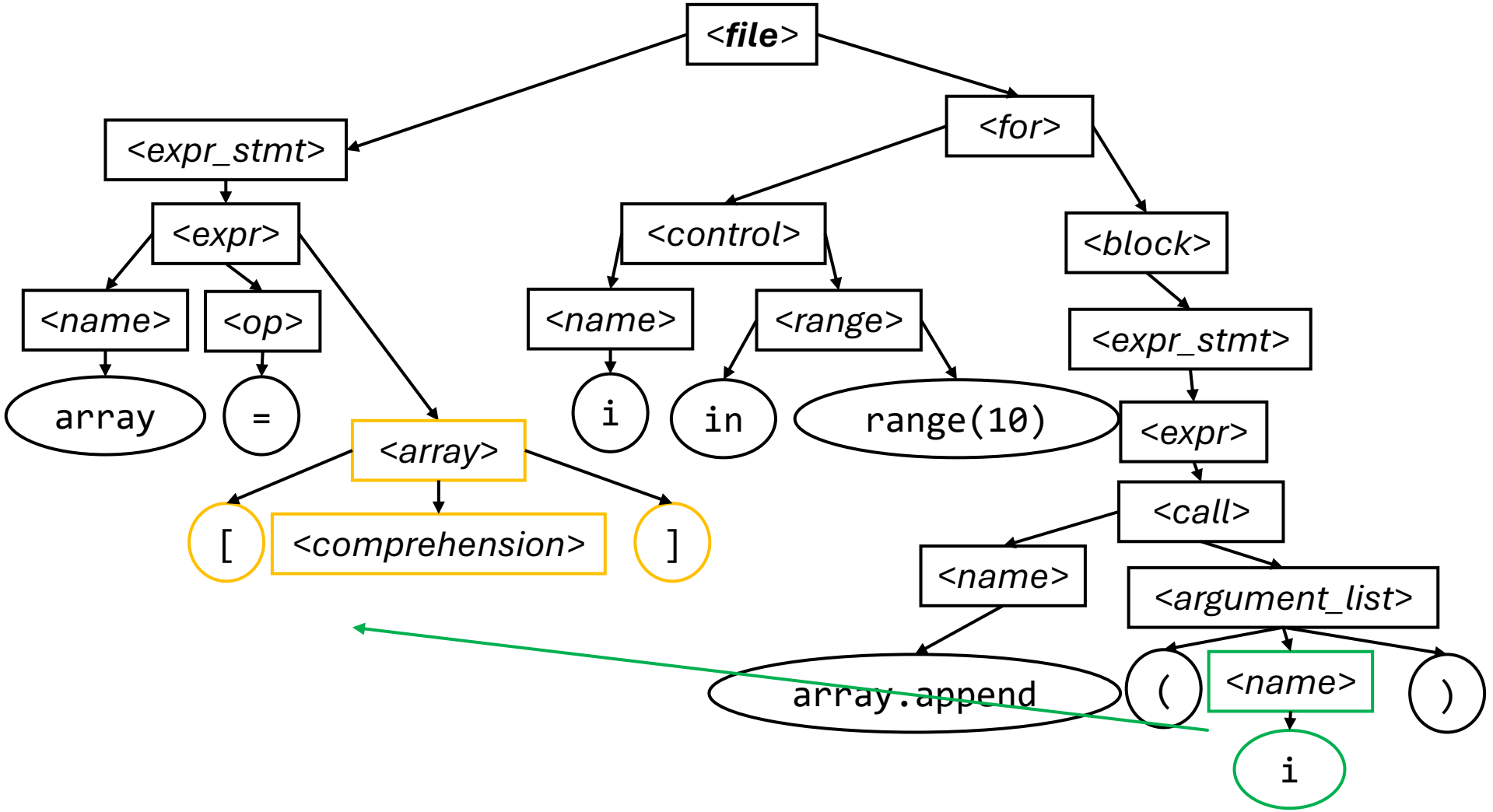
Refactoring for Idiomatic Compliance

```
array = [ ]
for i in range(10):
  array.append(i)
```



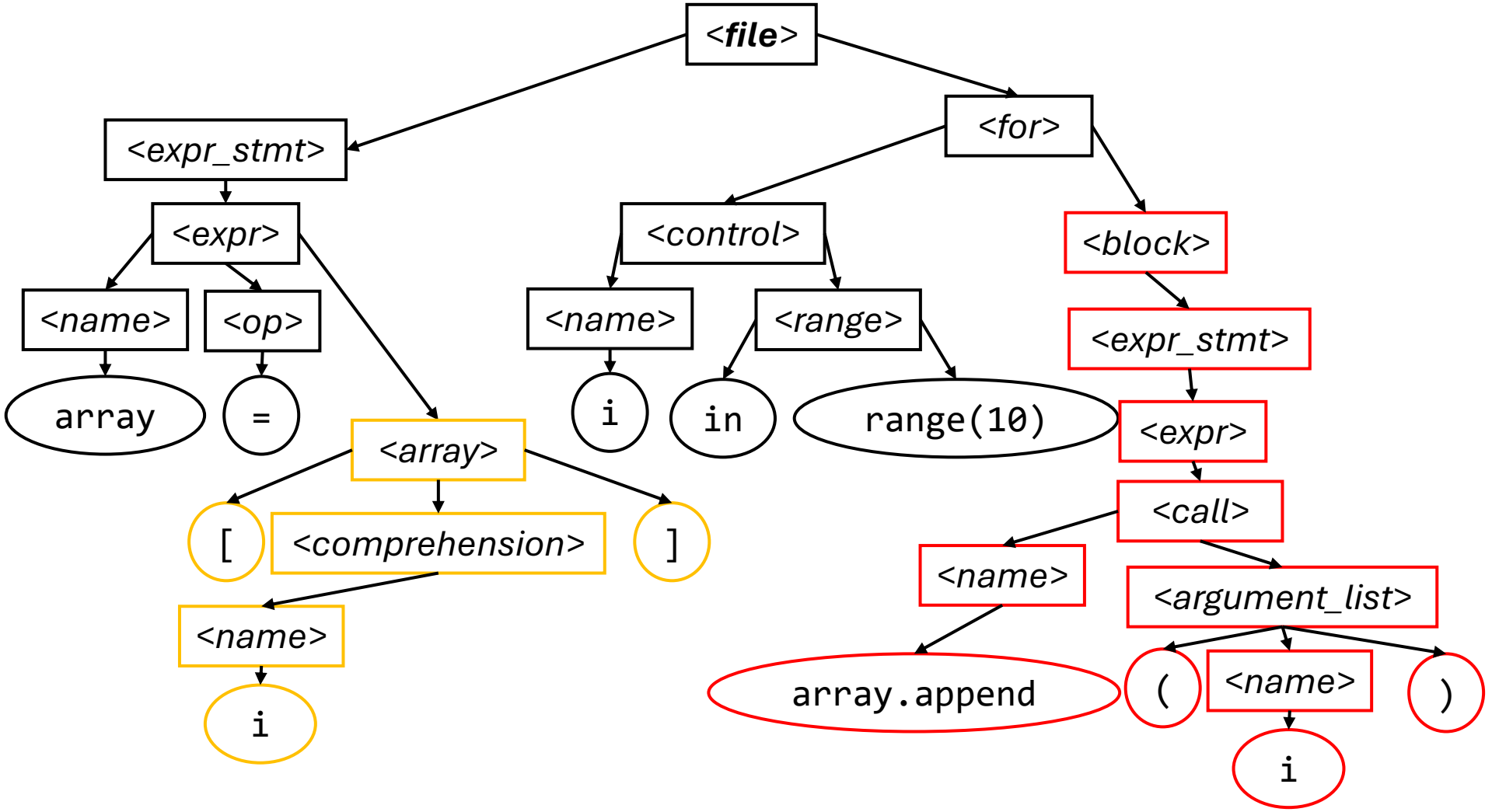
Refactoring for Idiomatic Compliance

```
array = [ ]
for i in range(10):
  array.append(i)
```



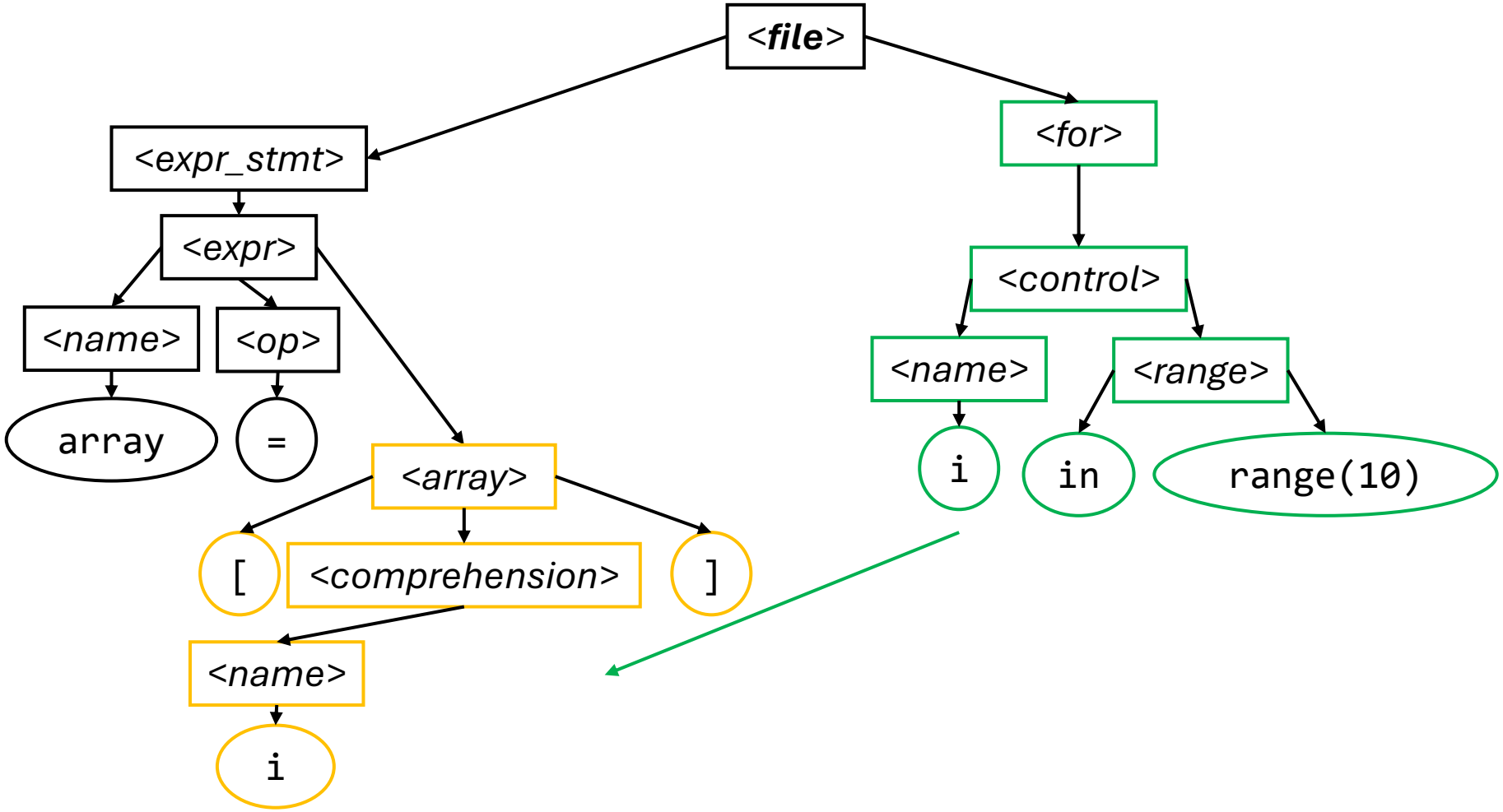
Refactoring for Idiomatic Compliance

```
array = [i ]
for i in range(10):
  array.append(i)
```

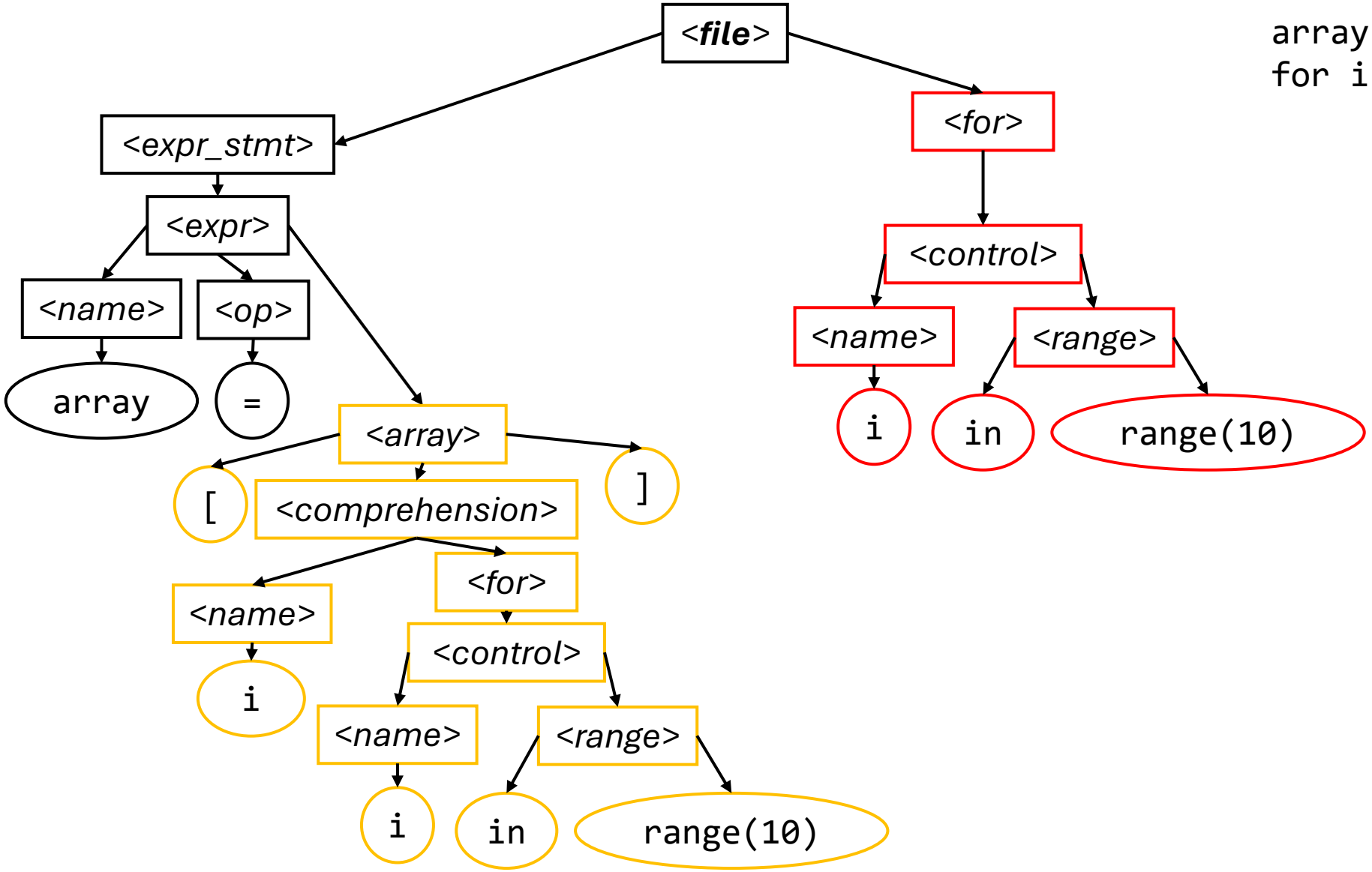


Refactoring for Idiomatic Compliance

```
array = [i ]
for i in range(10)
```



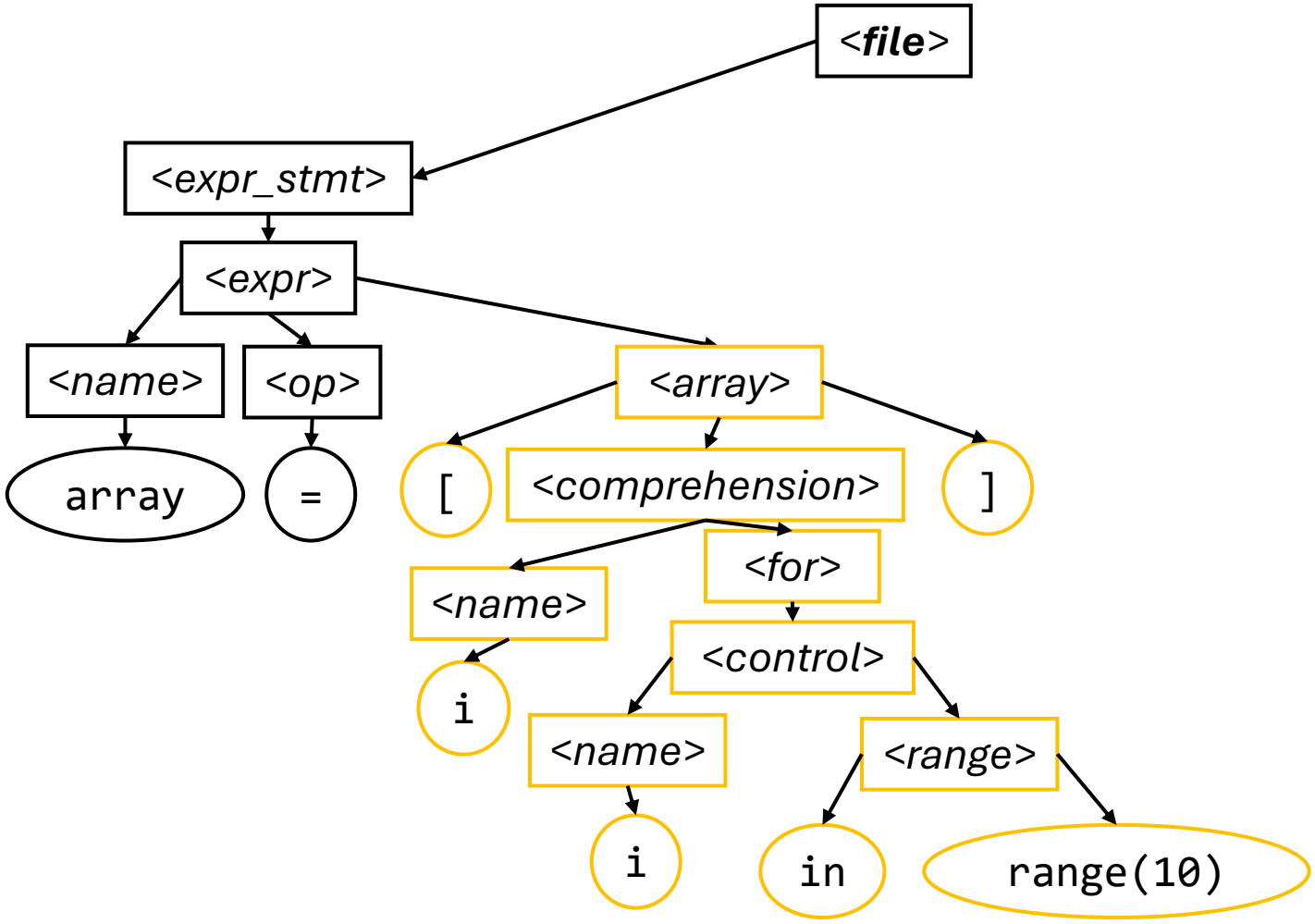
Refactoring for Idiomatic Compliance




```
array = [i for i in range(10)]
for i in range(10)
```

Refactoring for Idiomatic Compliance

```
array = [i for i in range(10)]
```



Modern Software Development

- Software systems are becoming increasingly complex
- As developers, we want to understand these systems, to develop and maintain them effectively. But how can we understand such complexity?
- **Abstractions** are fundamental building blocks in software development that promote good design by capturing and simplifying underlying ideas to their purest form
- *Design patterns* are one type of abstraction
  - Provide reusable solutions to common Object-Oriented design problems
  - Example: *Singleton*, which restricts a class to a single instance and provides a global access point to it.

What are Stereotypes?

- ***Stereotypes*** are another form of ***abstractions*** that capture high-level design and behavior of software

- Think of stereotypes as labels or tags that we attach to different parts of a software program to quickly understand what they do

- Two types:
  - **Method stereotype:** a tag for a method in your code. It tells you what the method is supposed to do
  - **Class stereotype**: a label for a whole class in your code. It gives you a big picture idea of what the class is responsible for

Method Stereotypes

Stereotype Category	Stereotype	Description
Structural Accessor	get	Returns a data member
	predicate	Returns a Boolean value that is not a data member
Structural Mutators	set	Modifies a data member
	command	Performs a complex change to the object's state
Creational	constructor	Creates and/or destroys objects
	destructor	
	factory	
Collaborational	collaborator	Works with objects belonging to classes other than itself
Degenerate	stateless	Does not read or change an object's state
	empty	Has no statements

Examples of Method Stereotypes

```
// @stereotype get
double AxisModelBase::getFirstTick() const {
  return m_first_tick;
}

// @stereotype set
void AxisModelBase::setAutoRanging ( bool flag ) {
  m_auto_range = flag;
}

// @stereotype command
void AxisModelBase::setScaleFactor ( double sf ){
  m_scale_factor = sf;
  m_scaling_on = true;
  ...
}
```

Class Stereotypes

Class Stereotype	Description
entity	Encapsulates data and behavior. Keeper of data model and/or business logic
command	Consists mostly of mutator methods
boundary	Communicator with a large percentage of collaborational methods
factory	Creator of objects
control	Provides functionality to control external objects
degenerate	Consists mostly of methods that do not read/write to the object's state
empty	Has no methods

Example of Class Stereotypes

```
// @Stereotype boundary degenerate
public class ResumeController : Controller {

  // @Stereotype command
  public async void NoResume();

  // @Stereotype controller stateless
  private bool TryGetRangeHeader(out string rangeHeader);

 // @Stereotype wrapper
  private void SetResumeResponseHeaders();

 // @Stereotype command collaborator
 public async void Index();

 ...
}
```

Linters and Stereotypes - The Connection

- Both linters and stereotypes help in promoting good coding practices and improving the overall quality of your code.
- Stereotypes provide a more holistic view of your code.
- While linters analyze individual lines or blocks of code, stereotypes can give you insights about the overall behavior of your software.
- Stereotypes can enhance the capabilities of linters. They can help linters detect bad coding practices and potential design issues, often referred to as “code smells”.

Code Smells

- Code smells are indicators of potential problems in your code.
- They suggest that some part of your code may need to be refactored or redesigned.
- For instance, if a method is labeled with a stereotype that suggests it should not modify any class fields, but it does, this could be a sign of a code smell.

Code Smell Example

```
// @stereotype get property collaborator
const std::string & DataRep::getTitle ( ) const {

  if ( m_title == "%t" ) {

    return m_projector -> getTitle ();

  }

  return m_title;
}
```

Static or Non-Static?

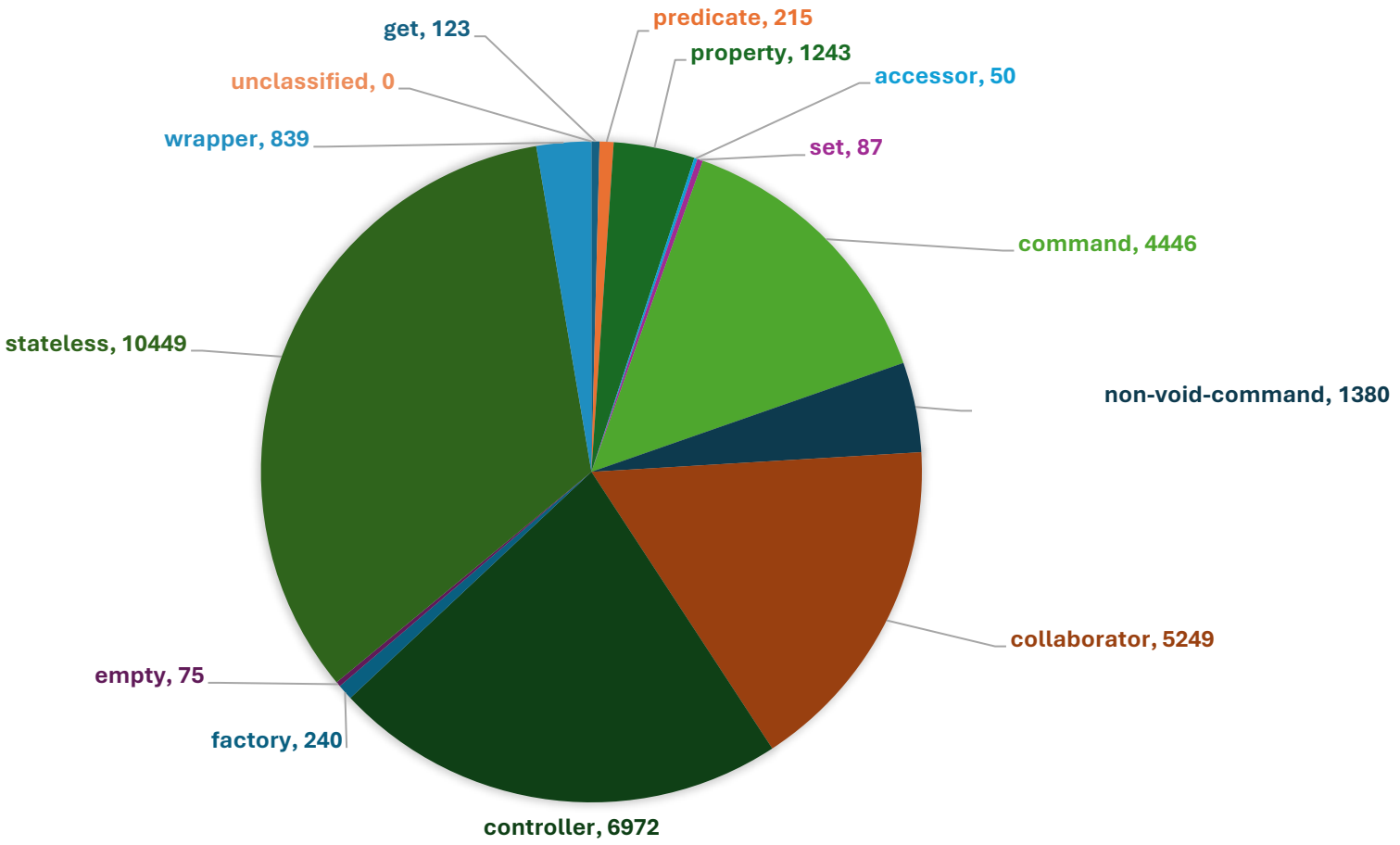
```
// @stereotype controller stateless
void ScreenshotFrame::OnSeeScreenshots(wxCommandEvent& WXUNUSED(event)) {

    wxString defaultDir = AutoCaptureMechanism::GetDefaultDirectoryAbsPath();

    if (wxFileName::DirExists(defaultDir))
        wxLaunchDefaultBrowser(defaultDir);

    else
        wxMessageBox(_("There isn't any screenshots yet."));
}
```

Bad Design?



RxJava

Unnecessary Class Function

```
class DocumentViewer {
public:
  DocumentViewer(const std::string& filePath) {
    if (!documentExists(filePath)) { cerr << "File could not be found!"; return; }
    defaultDir = filePath;
  }
  // @stereotype controller stateless
  bool documentExists(const std::string& filePath) {
      return std::filesystem::exists(filePath);
  }
private:
  std::string docName;
  std::string docType;
};
```

Better Free Function

```
bool documentExists(const std::string& filePath) {
  return std::filesystem::exists(filePath);
}

class DocumentViewer {
public:
  DocumentViewer(const std::string& filePath) {
    if (!documentExists(filePath)) { cerr << "File could not be found!"; return; }
    defaultDir = filePath;
  }
private:
  std::string docName;
  std::string docType;
};
```