Code Completion in Clang-Repl

Yuquan (Fred) Fu

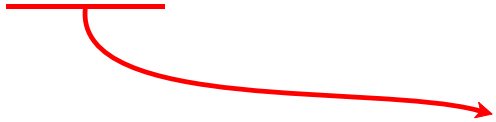
Vassil Vassilev

What is Clang-Repl?

```
clang-repl> #include <iostream>
clang-repl> std::string str = "Hello, World!";
clang-repl> std::cout << str << " has " << str.length() << "characters!\n";
Hello, World! has 13 characters!
clang-repl>
```

```
clang-repl> #include <iostream>
clang-repl> std::string str = "Hello, World!";
clang-repl> std::cout << str << " has " << str.length() << "characters!\n";
Hello, World! has 13 characters!
clang-repl> str.⇥
```

```
clang-repl> #include <iostream>
clang-repl> std::string str = "Hello, World!";
clang-repl> std::cout << str << " has " << str.length() << "characters!\n";
Hello, World! has 13 characters!
clang-repl> str.⇥
```



What have we achieved?

Basic Code Completion

```
clang-repl> class ModulePointerAndOffsetLessThanFunctionObject{ ... };
clang-repl> Mo⇥
```

Basic Code Completion

```
clang-repl> class ModulePointerAndOffsetLessThanFunctionObject{ ... };
clang-repl> ModulePointerAndOffsetLessThanFunctionObject
```

Semantic Code Completion

```
clang-repl> int number1 = 42, number2 = 84;
clang-repl> std::string name1 = "Fred", name2 = "Vassil";
clang-repl> template <typename T> T pickOne(T v1, T v2) {...};
clang-repl> pickOne(number1, ⇥
```

Semantic Code Completion

```
clang-repl> int number1 = 42, number2 = 84;
clang-repl> std::string name1 = "Fred", name2 = "Vassil";
clang-repl> template <typename T> T pickOne(T v1, T v2) {...};
clang-repl> pickOne(number1, 
number1
number2
```

Overcame challenges of reusing Sema/CodeComplete to implement code completion in REPL

Challenges

Declaration Visibility Issue

```
int num1 = 84;
int num2 = 76;
int num3 = 42;
int res = 1 + n⇥
```

```
clang-repl> int num1 = 84;
clang-repl> int num2 = 76;
clang-repl> int num3 = 42;
clang-repl> int res = 1 + n⇥
```

Declaration Visibility Issue

```
int num1 = 84;
int num2 = 76;
int num3 = 42;
int res = 1 + n
num1
num2
num3
```

```
clang-repl> int num1 = 84;
clang-repl> int num2 = 76;
clang-repl> int num3 = 42;
clang-repl> int res = 1 + n
```

Declaration Visibility Issue

Why does the code completion system fail to see previously defined declarations in REPL?

- A file is one single translation unit enclosed by one ASTContext

```
int num1 = 84;
int num2 = 76;
int num3 = 42;
int res = 1 + n⇥
```

1 ASTContext & 1 TranslationUnit

Declaration Visibility Issue

Why does the code completion system fail to see previously defined declarations in REPL?

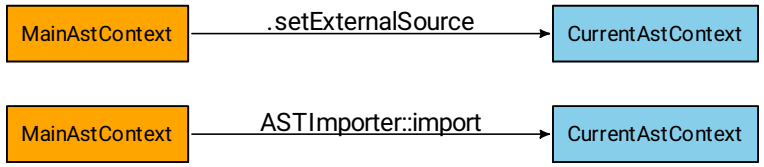
- A REPL session contains multiple partial translation units enclosed by two ASTContexts

```
clang-repl> int num1 = 84;
clang-repl> int num2 = 76;
clang-repl> int num3 = 42;
clang-repl> int res = 1 + n⇥
```

2 ASTContexts & 4 PartialTranslationUnits

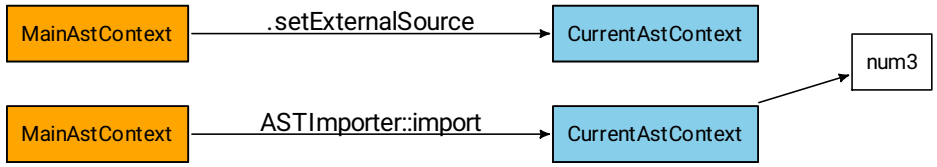
Solution to Declaration Visibility Issue

```
clang-repl> ...
clang-repl> int num1 = 84;
clang-repl> int num2 = 76;
clang-repl> int num3 = 42;
clang-repl> int res = 1 + n⇥
```



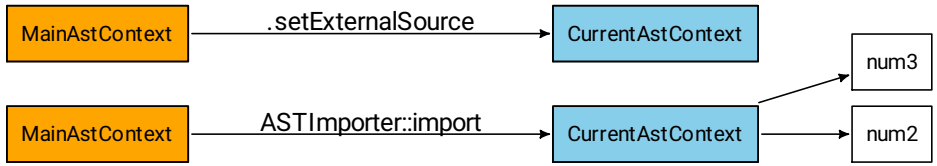
Solution to Declaration Visibility Issue

```
clang-repl> ...
clang-repl> int num1 = 84;
clang-repl> int num2 = 76;
clang-repl> int num3 = 42;
clang-repl> int res = 1 + n⇥
```



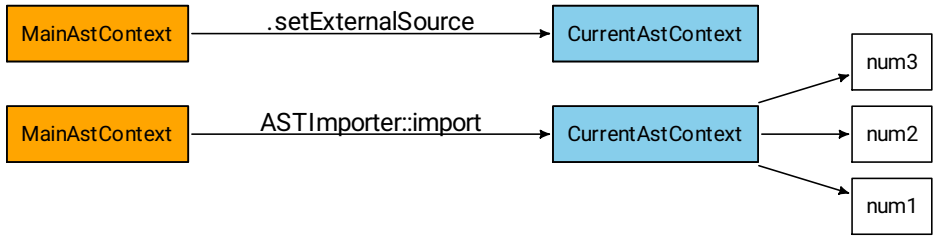
Solution to Declaration Visibility Issue

```
clang-repl> ...
clang-repl> int num1 = 84;
clang-repl> int num2 = 76;
clang-repl> int num3 = 42;
clang-repl> int res = 1 + n⇥
```



Solution to Declaration Visibility Issue

```
clang-repl> ...
clang-repl> int num1 = 84;
clang-repl> int num2 = 76;
clang-repl> int num3 = 42;
clang-repl> int res = 1 + n⇥
```



Code Completion for Top Level Expressions

- Top level expressions are syntactically invalid in a regular C++ file
- Top level expressions are bread and butter in REPL

```
clang-repl> int num = 42;
clang-repl> 1 + n⇥
```

Code Completion for Top Level Expressions

- Top level expressions are syntactically invalid in a regular C++ file
- Top level expressions are bread and butter in REPL

```
clang-repl> int num = 42;
clang-repl> 1 + n
```

Code Completion for Top Level Expressions

- Top level expressions are syntactically invalid in a regular C++ file
- Top level expressions are bread and butter in REPL

```
clang-repl> int num = 42;
clang-repl> 1 + n
CompletionContext::Kind = CCC_TopLevel
```

Code Completion for Top Level Expressions

- Top level expressions are syntactically invalid in a regular C++ file
- Top level expressions are bread and butter in REPL

```
clang-repl> int num = 42;
clang-repl> 1 + n
CompletionContext::Kind = CCC_TopLevelOrExpression
```

Code Completion for Top Level Expressions

- Top level expressions are syntactically invalid in a regular C++ file
- Top level expressions are bread and butter in REPL

```
clang-repl> int num = 42;
clang-repl> 1 + n⇥
CompletionContext::Kind = CCC_TopLevelOrExpression
```

Code Completion for Top Level Expressions

- Top level expressions are syntactically invalid in a regular C++ file
- Top level expressions are bread and butter in REPL

```
clang-repl> int num = 42;
clang-repl> 1 + num
CompletionContext::Kind = CCC_TopLevelOrExpression
```

Semantic Code Completion

What Semantic Code Completion Needs

- What context is the cursor in?

```
clang-repl> f
```

```
clang-repl> car.
```

- How to get the type w.r.t the cursor position?

```
clang-repl> pickOne(name1, )
```

Key Structure for Sematic Code Completion

`CodeCompletionContext`

- ::getKind() shows the context kind

```
clang-repl> f                    CCC_TopLevelOrExpression
clang-repl> car.                 CCC_DotMemberAccess
```

- ::getPreferedType() reveals the type w.r.t the current cursor position

```
clang-repl> pickOne(name1,       std::string
```

Key Structure for Sematic Code Completion

- CodeCompletionContext::getBaseType() returns the type of the expressions before the dot

```
clang-repl> class Car {public: int getPrice(){...} void sell(Person& p){...}}
clang-repl> Car car1
clang-repl> car1.⇥
getPrice
sell
```

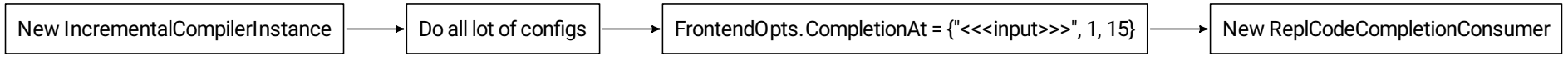
Implementation

First Attempt

```
clang-repl> int num = 42;
clang-repl> int res = 1 + n⇥
```

First Attempt

```
clang-repl> int num = 42;
clang-repl> int res = 1 + n⇥
```

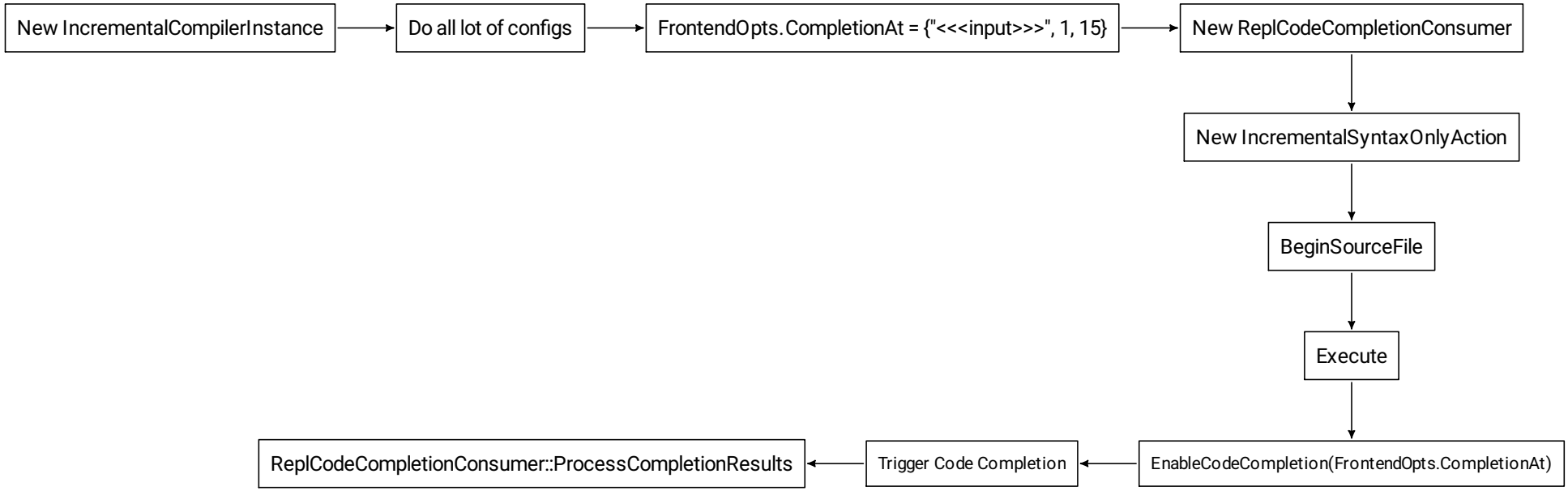


First Attempt

```
clang-repl> int num = 42;
clang-repl> int res = 1 + n⇥
```

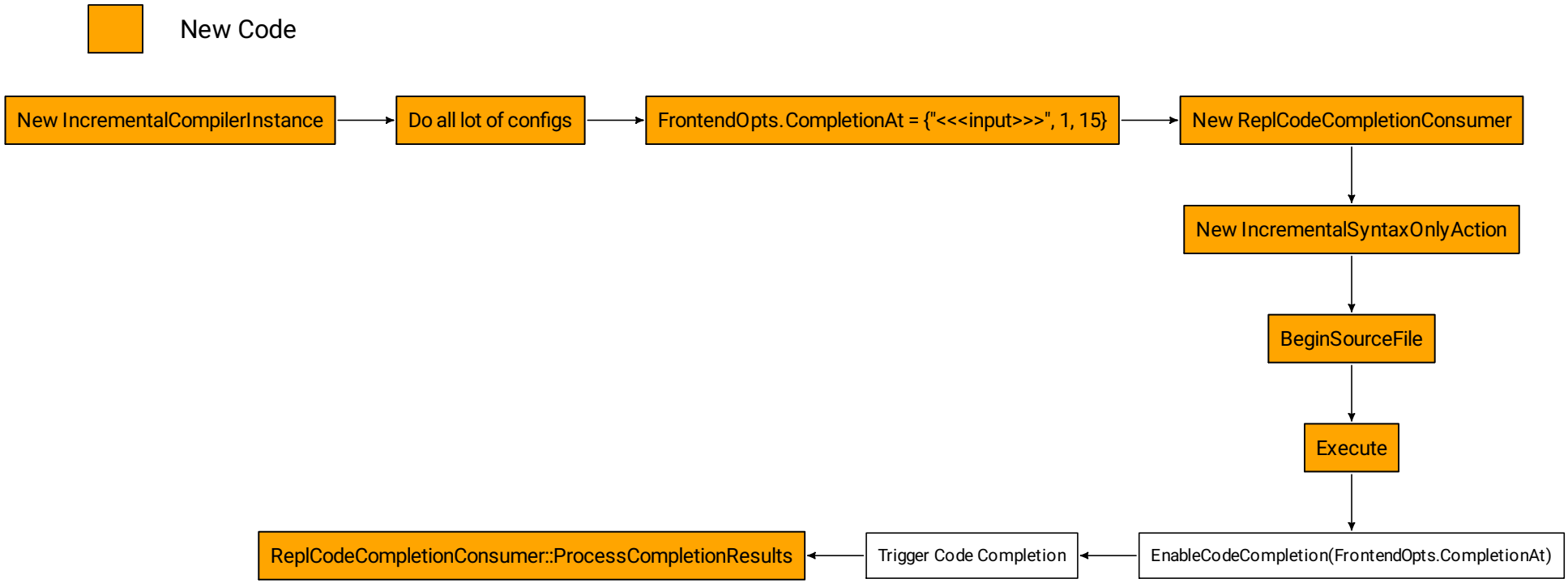
First Attempt

```
clang-repl> int num = 42;
clang-repl> int res = 1 + n⇥
```



First Attempt

```
clang-repl> int num = 42;
clang-repl> int res = 1 + n⇥
```



Using ASTUnit

```
clang-repl> int num = 42;
clang-repl> int res = 1 + n⇥
```

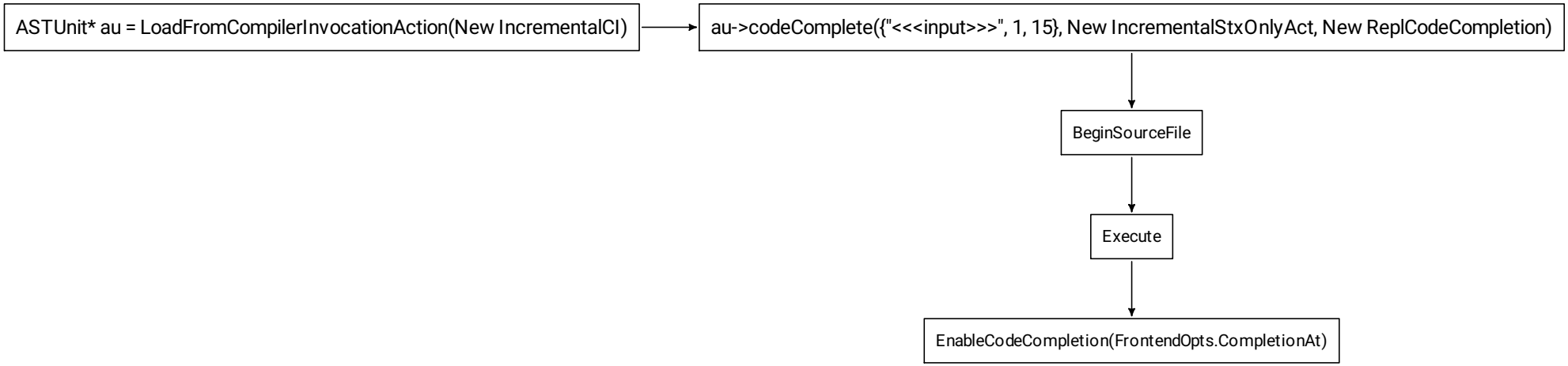
ASTUnit* au = LoadFromCompilerInvocationAction(New IncrementalCI)

Using ASTUnit

```
clang-repl> int num = 42;
clang-repl> int res = 1 + n⇥
```

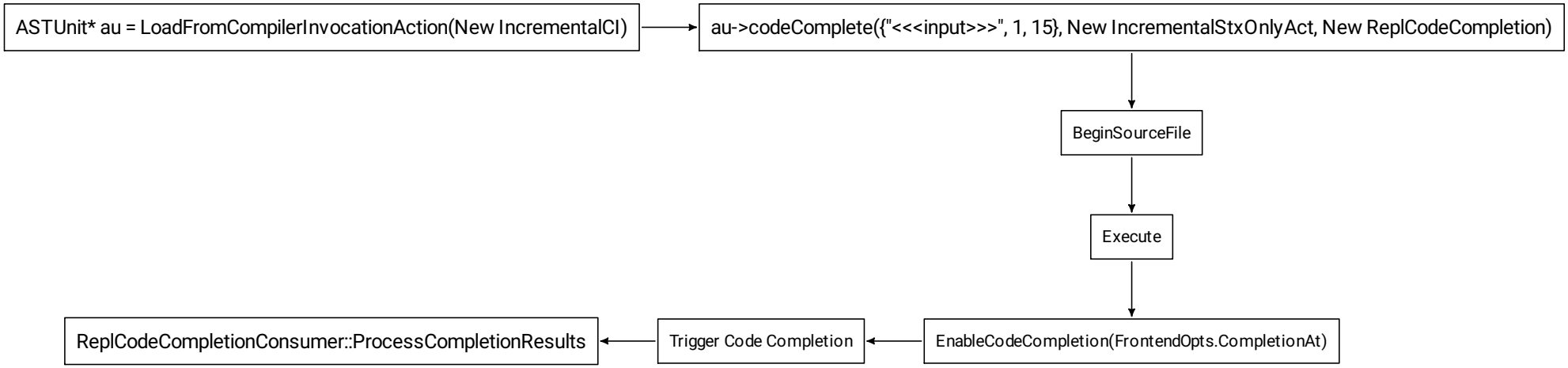
Using ASTUnit

```
clang-repl> int num = 42;
clang-repl> int res = 1 + n⇥
```



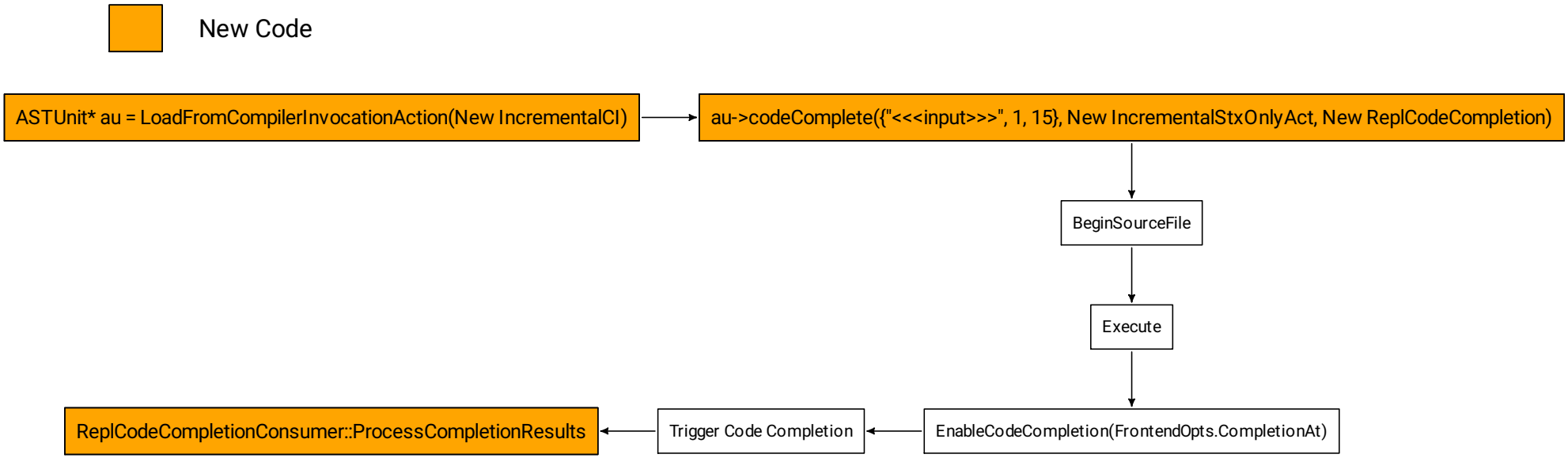
Using ASTUnit

```
clang-repl> int num = 42;
clang-repl> int res = 1 + n⇥
```



Using ASTUnit

```
clang-repl> int num = 42;
clang-repl> int res = 1 + n⇥
```



Conclusions

- Solved the visibility issue with ASTImporter and ExternalSource
- Enabled code completion in top level expressions with a new CompletionContext
- Leveraged Sema modules to achieve semantic code completions
- Concise implementation with minimal invasive changes