Compiler Construction: Practical Introduction

Case Study: Some remarks about compiler design

Eugene Zouev

Fall Semester 2025

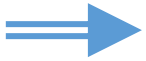
Innopolis University

How to implement **parser**
(syntax analyzer) in your projects –

My recommendations

Recursive descent parser: example

```
Expr ->
  Term { +|- Term }
```



```
Term ->
 Factor { *|/ Factor }
```

```
Factor -> Id
Factor -> ( Expr )
```

```
Tree parseExpr()
{
    Tree left = parseTerm();
    while ( tk=get(), tk==tkPlus||tk==tkMinus )
        left = mkBinTree(tk,left,parseTerm());
    return left;
}

Tree parseTerm()
{
    Tree left = parseFactor();
    while ( tk=get(), tk==tkStar||tk==tkSlash )
        left = mkBinTree(tk,left,parseFactor());
    return left;
}
Tree parseFactor()
{
    Tree res;
    if ( tk=get(), tk==tkLParen )
    {
        res = parseExpr();
        get(); // skip ')'
    }
    else
        res = mkUnaryTree(parseId());
    return res;
}
```

Tree

```
class Node
{
    ...
}

class Program: Node
{
    List<Function> functions;
}

class Statement: Node
{
    ...
}

class If: Statement
{
    Expression condition;
    Statement thenPart;
    Statement elsePart;
}

class Expression: Node
{
    ...
}
...
```

Structure only

Parser

```
class Parser
{
    Program parseProgram()
    {
        ...
        return Tree for program
    }

    Tree parseExpr()
    {
        ...
       return Tree for expression;
    }
    Tree parseTerm()
    {
        ...
        return Tree for term;
    }
    Tree parseFactor()
    {
        ...
        return Tree for factor;
    }
    ...
}
```

Functionality only

Compiler

```
class Compiler
{
  int Main()
  {
    ...
    var p = new Parser(…);
    p.parseProgram();

    var v = new Validator(p);
    v.validate();

    var g = new Generator(p);
    g.generate();
    ...
  }
}
```

```
abstract class Entity
{
    // Validation
    public abstract bool validate();
    // Generation
    public abstract void generate();
    // Reporting
    public abstract void report();
    ...
}
```

```
abstract class Statement: Entity
{
    // (Almost) empty
    ...
}
```

```
abstract class Declaration: Entity
{
    string name;
    ...
}
```

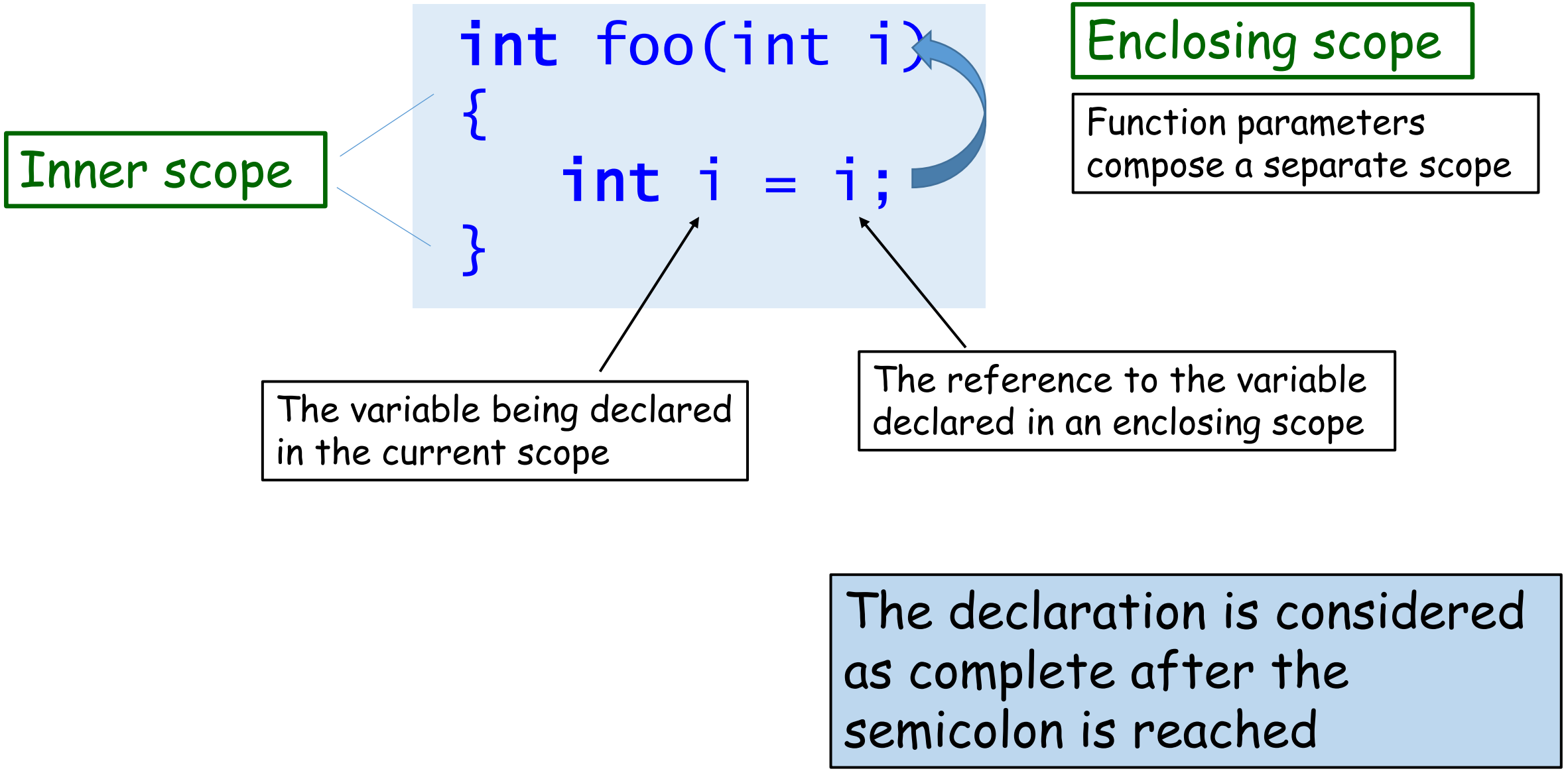
```
class If: Statement
{
    // Structure
    Expression condition;
    Statement thenPart;
    Statement elsePart;

    // Creation
    public If(...) { ... }

    // Parsing
    public static If parse()
    {
        ...
    }

    // Validation
    public override bool validate()
    {
        ...
    }

    // Generation
    public override void generate()
    {
        ...
    }
}
```

Structure & and all kinds of functionality together

```
class Entity
{
    ...
}

class Program: Entity
{
    List<Function> functions;

    public static Program parse() {
        while ( end-of-source )
            Function.parse();
}

class Statement: Entity
{
    public static Statement parse()
    {
        ...
        token = get();
        switch ( token.code )
        {
            case TokenCode.While: While.parse(); break;
            case TokenCode.For  : For.parse(); break;
            case TokenCode.Return: Return.parse(); break;
            ...
        }
    }
}
```

```
class Compiler
{
    int Main()
    {
        ...
        var tree = Program.parse();
        tree.validate();
        tree.generate();
        ...
    }
}
```

Declarations & Scopes


int foo(int i)
{
int i = i;
}
Inner scope
Enclosing scope
Function parameters compose a separate scope
The variable being declared in the current scope
The reference to the variable declared in an enclosing scope
The declaration is considered as complete after the semicolon is reached

```
interface iScope
{
    iScope enclosing();
    Declaration findInScope(string name);
    Declaration find(string name);
}

...
class Block: Statement, iScope
{
    List<Function> declarations;

    iScope enclosing() return ref-to enclosing

    Declaration findInScope(string name) { ... }
    Declaration find(string name) { ... }
}
```

Returns the node representing an enclosing scope

Returns the node with the given name in the current scope – for checking duplications

Returns the node with the given name in the current and all enclosing scopes