

# 50.051 Programming Language Concepts

## W12-S1 Semantic Analysis and Intermediate Representations

Matthieu De Mari



SINGAPORE UNIVERSITY OF  
TECHNOLOGY AND DESIGN

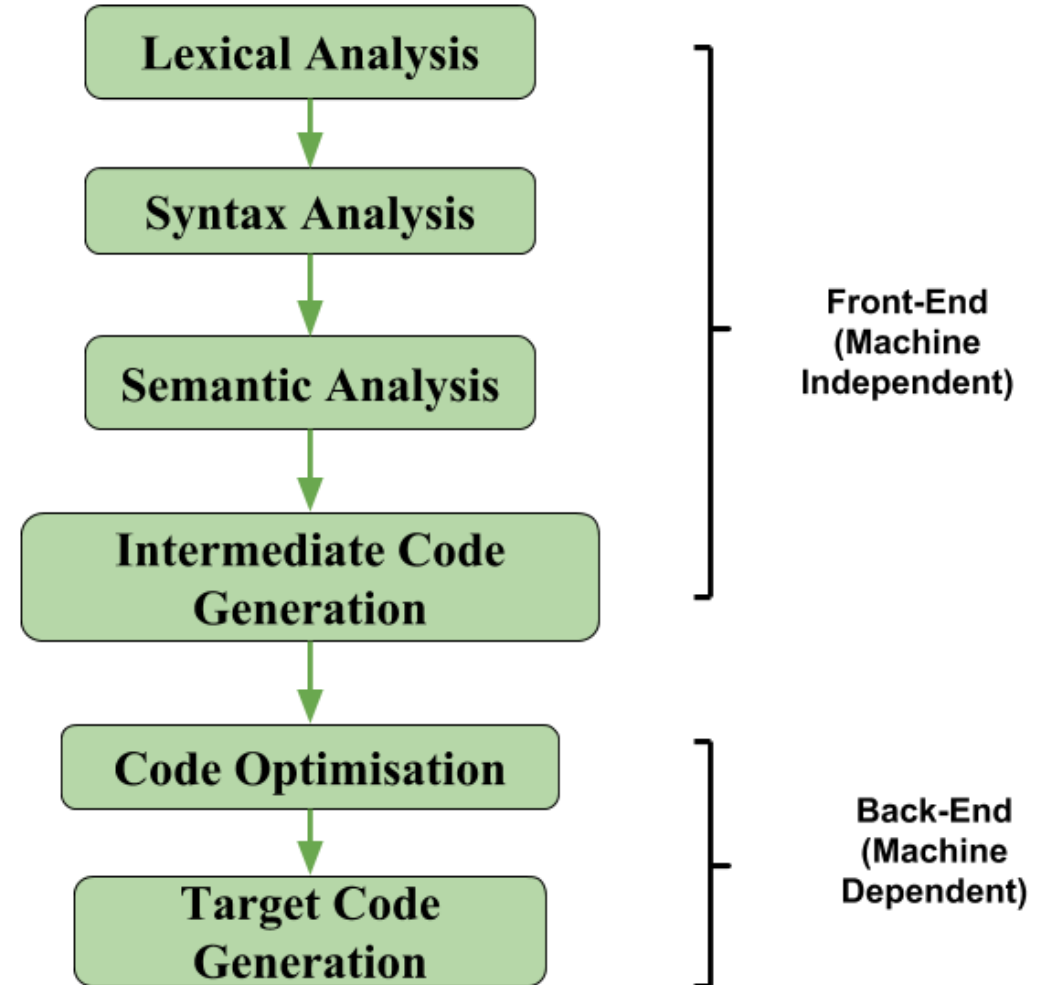
# The front-end of a compiler

## Definition (The front-end part of a compiler):

The **front-end of a compiler** is responsible for analysing the source code, and converting it into a form that can be used by the rest of the compiler.

It involves tasks, such as:

- **Lexical analysis,**
- **Syntax analysis,**
- and **Semantic analysis.**



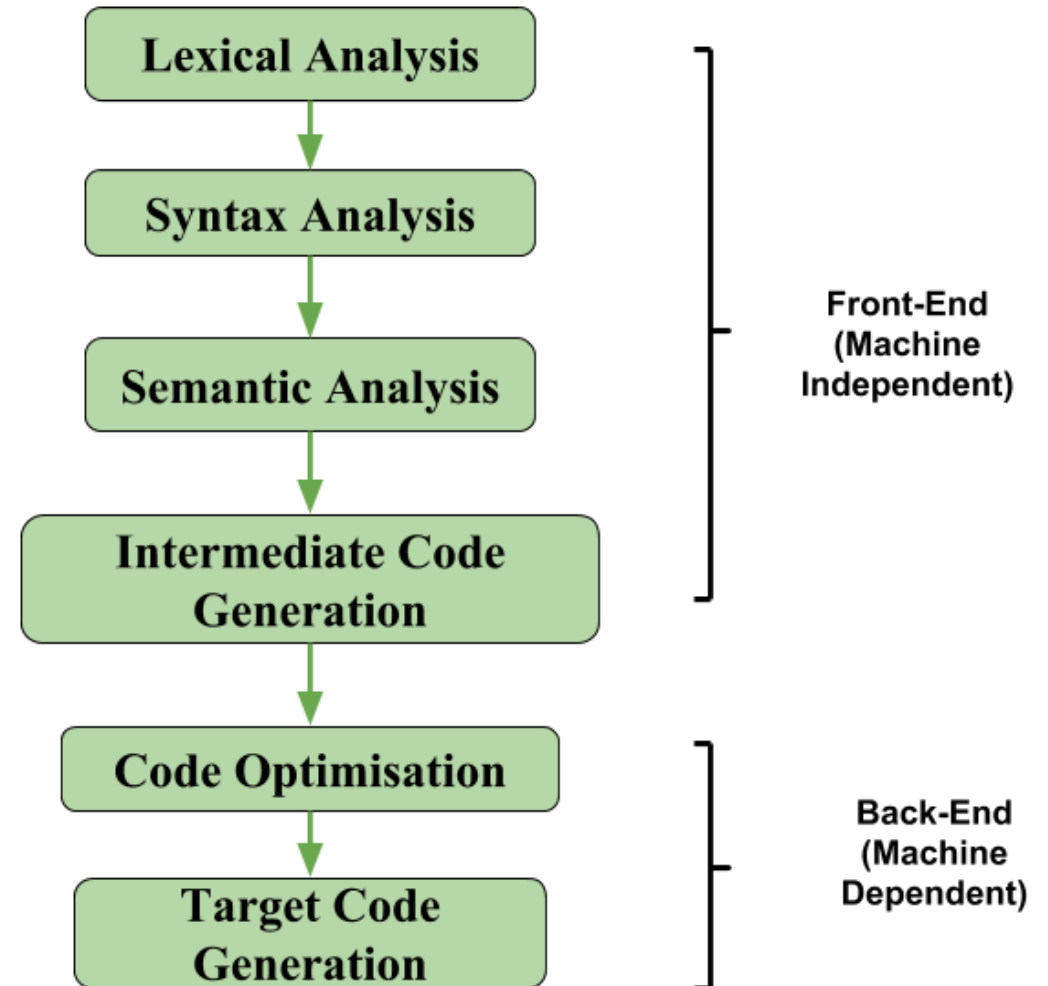
# The front-end of a compiler

## Lexical Analysis

- Detects illegal tokens in source code (e.g. incorrect use of identifiers, literals, etc.)

## Syntax Analysis

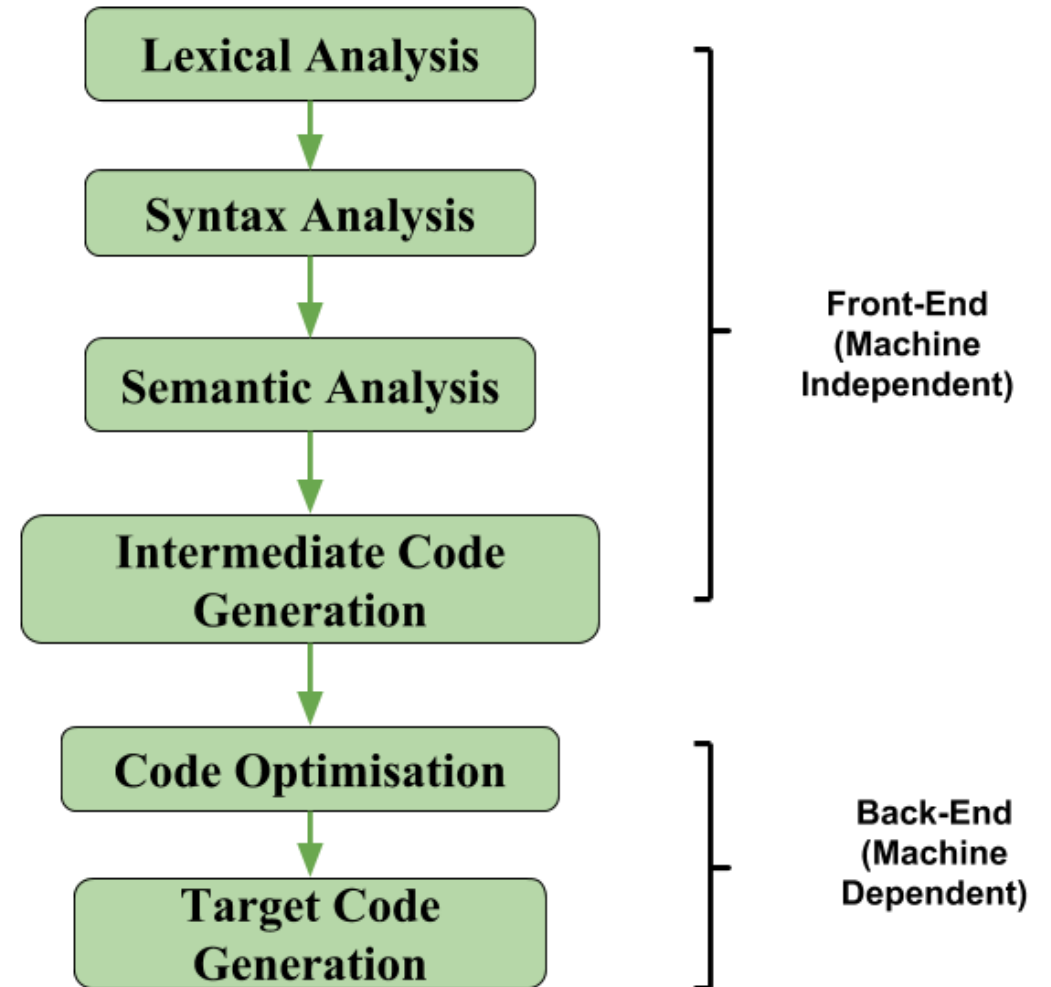
- Detects inputs that violate the syntax rules of the language
- Will typically result in impossible-to-form or ill-formed parsing trees.



# The front-end of a compiler

## Semantic Analysis

- Final step in the “front-end” phase, whose job is to check that the **source code is legal**.
- Ensures that the program has a **well-defined meaning**.
- Should catch **all remaining errors** that lexical analysis and syntax analysis could not catch.
- Errors that require **contextual information** about the code.



# Examples of remaining errors

## Scope problems

- A variable is defined after it is called for the first time.
- A function is defined after it is called for the first time and no function prototypes were mentioned.
- A variable defined in a function is called outside of the function.
- A variable name is defined twice.
- Etc.

<pre>#include &lt;stdio.h&gt;  int main() {     int y;     y = x + 7;     int x = 3;     return 0; }</pre>	<pre>#include &lt;stdio.h&gt;  int main() {     int x = 3;     int x = 7;     return 0; }</pre>
--	---

```
#include <stdio.h>

// Subfunction2 is defined first, but it calls subfunction1
int subfunction2(int x) {
    return subfunction1(x) + 2;
}

// Subfunction1 is defined later, but it is called by subfunction2
int subfunction1(int x) {
    return x * 3;
}

int main() {
    int x = 5;
    printf("Result from subfunction1: %d\n", subfunction1(x));
    return 0;
}
```

# Examples of remaining errors

## Why cannot we just catch these errors during parsing?

- Try writing a CFG that will be able to prevent duplicate variable definitions?
- For most programming languages, using a CFG to do that is **provably impossible**.
- Proof using the pumping lemma for context-free languages, or Ogden's lemma.

<pre>#include &lt;stdio.h&gt;  int main() {     int y;     y = x + 7;     int x = 3;     return 0; }</pre>	<pre>#include &lt;stdio.h&gt;  int main() {     int x = 3;     int x = 7;     return 0; }</pre>
--	---

```
#include <stdio.h>

// Subfunction2 is defined first, but it calls subfunction1
int subfunction2(int x) {
    return subfunction1(x) + 2;
}

// Subfunction1 is defined later, but it is called by subfunction2
int subfunction1(int x) {
    return x * 3;
}

int main() {
    int x = 5;
    printf("Result from subfunction1: %d\n", subfunction1(x));
    return 0;
}
```

# Examples of remaining errors

## Type problems

- An operation attempts to combine two invalid types together (e.g. summing a string and a float together).
- An operation attempts to convert a variable of a certain type in another type that is not feasible.
- Etc.

```
#include <stdio.h>

int main() {
    char* x = "hello";
    float y = x + 7.5;
    printf("%f", y);
    return 0;
}
```

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

int main() {
    struct Point p = {3, 4};
    int z;

    // The following line tries to convert a struct
    // (Point) to an int, which is not feasible.
    z = (int)p;

    printf("Point: (%d, %d)\n", p.x, p.y);
    printf("Converted to int: %d\n", z);

    return 0;
}
```

# Examples of remaining errors

## And more problems that have to do with classes!

- A class inherits from another class that is not defined yet.
- A child class attempts to use a method that was neither defined in the child class itself, nor the parent class it inherited from.
- Operators on custom classes being used incorrectly.
- Etc.

```
#include <stdio.h>

int main() {
    char* x = "hello";
    float y = x + 7.5;
    printf("%f", y);
    return 0;
}
```

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

int main() {
    struct Point p = {3, 4};
    int z;

    // The following line tries to convert a struct
    // (Point) to an int, which is not feasible.
    z = (int)p;

    printf("Point: (%d, %d)\n", p.x, p.y);
    printf("Converted to int: %d\n", z);

    return 0;
}
```



# Examples of remaining errors

**Important fact:** These errors cannot be checked during the lexical analysis or the syntax analysis, as they require some **context**.

- Cannot be checked by a context-free grammar.
- Often depends on values and context, not syntax.
- May sometimes require and involve computation.

## **What to do then?**

- Instead, will need to **define a set of semantic rules for scoping and types** explicitly.
- And then, **perform additional checks after parsing**.
- These checks will be implemented in the form of “if-else” checks.

# Where are we at now?

At the end of the lexical analysis and parsing phase.

- We have successfully **tokenized our source code and produced a stream of Tokens objects** (containing information about token type, lexeme, value, line number/position, etc.).
- We have established a **derivation for the language CFG and the given stream of tokens**.
- We have produced a **parse tree** following from that derivation.
- Using **Syntax Directed Translation**, we could define a **list of elementary operations** to be executed, matching each production rule we used and eventually obtain an **Abstract Syntax Tree**.

# Implementing an SDT for Arithmetic CFG

Consider the CFG below, modified with the following SDT rules.

$$E \rightarrow E + T \{E.val = E1.val + T.val\}$$

$$E \rightarrow T \{E.val = T.val\}$$

$$T \rightarrow T * F \{T.val = T1.val * F.val\}$$

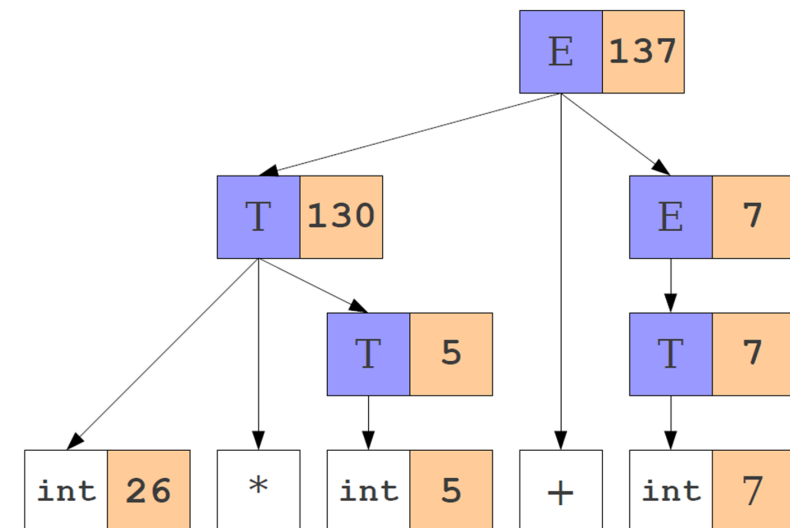
$$T \rightarrow F \{T.val = F.val\}$$

$$F \rightarrow (E) \{F.val = E.val\}$$

$$F \rightarrow num \{F.val = num.val\}$$

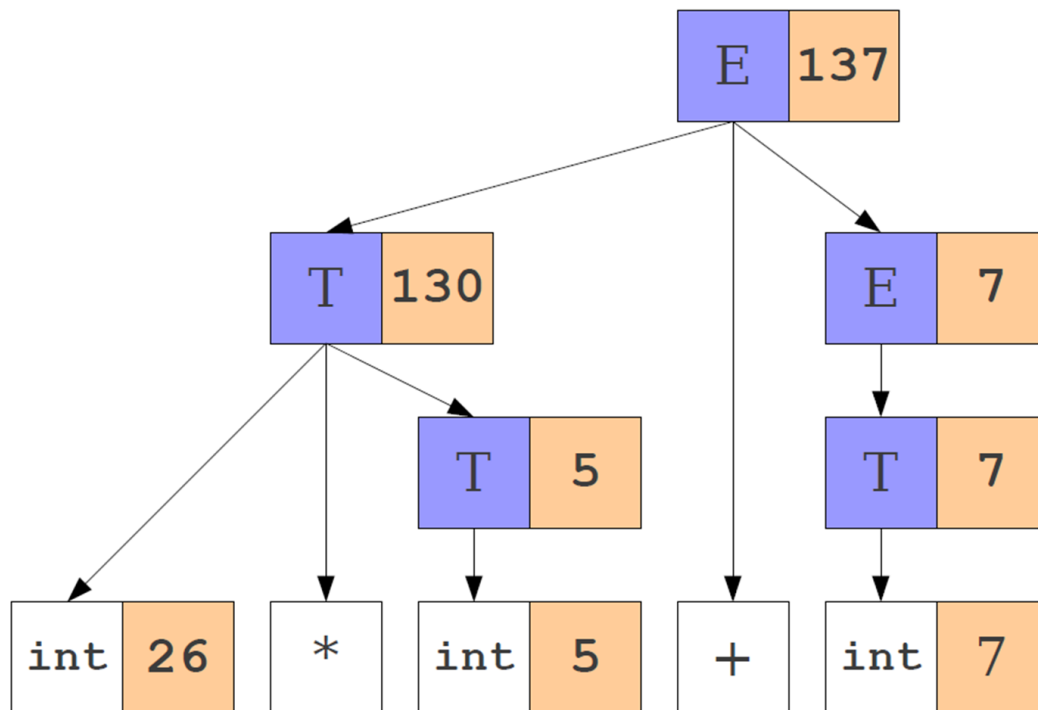
Using this CFG/SDT on  $26 * 5 + 7$  gives the following derivation and parse tree.

Sequence of operations in parse tree = bottom-up reading = DFS!



# Implementing an SDT for Arithmetic CFG

Using these steps, on the string “26\*5 + 7”, then gives the following Abstract Syntax Tree and list of SDT rules...



And the list of operations below:

I1.val = 26

I2.val = 5

T1.val = i2.val

T2.val = T1.val\*i1.val

I3.val = 7

T3.val = i3.val

E1.val = T3.val

E.val = T2.val + E1.val

# Semantic analysis and ASTs

**Objective:** The answer to the semantic problems

- Scope problems,
- Type problems,
- And more,...

And the checks to be conducted, both rely on the **Abstract Syntax Tree** produced by the syntax analysis step.

# Problem #1: Scope

## Scope problems

- A variable is defined after it is called for the first time.
- A function is defined after it is called for the first time and no function prototypes were mentioned.
- A variable defined in a function is called outside of the function.
- A variable name is defined twice.
- Etc.

```
#include <stdio.h>

int main() {
    int y;
    y = x + 7;
    int x = 3;
    return 0;
}
```

```
#include <stdio.h>

int main() {
    int x = 3;
    int x = 7;
    return 0;
}
```

```
#include <stdio.h>

// Subfunction2 is defined first, but it calls subfunction1
int subfunction2(int x) {
    return subfunction1(x) + 2;
}

// Subfunction1 is defined later, but it is called by subfunction2
int subfunction1(int x) {
    return x * 3;
}

int main() {
    int x = 5;
    printf("Result from subfunction1: %d\n", subfunction1(x));
    return 0;
}
```

# First things first

## Definition (**Scope**):

The **scope** of an identifier (variable/function name, etc.) is the **portion of the program in which this identifier is accessible**.

- Decides if a variable can be called outside of a function,
- Resolves ambiguity about identifiers with similar names used in different functions,
- Etc.

```
#include <stdio.h>

void function1() {
    int x = 10;
}

int main() {
    function1();
    printf("The value of x is %d\n", x);
    return 0;
}
```

```
#include <stdio.h>

void function1() {
    int x = 10;
    printf("The value of x in function1 is %d\n", x);
}

void function2() {
    int x = 20;
    printf("The value of x in function2 is %d\n", x);
}

int main() {
    function1();
    function2();
    return 0;
}
```

# First things first

## Definition (**Static and Dynamic Scope**):

Most languages have a **static scope**.

- A **static scope depends only on the program code, not its runtime behaviour**.
- Typically C and Java.

Some languages are **dynamically scoped**.

- A **dynamic scope may depend on the execution of the program and is decided during runtime**.
- For instance, JavaScript (?) and LaTeX.

**Is static better than dynamic?** Not a kind of debate I want to participate in!



# Static scoping

Let us start with **static scoping**.

Some questions that need to be answered for static scoping are:

- How can we tell what object a particular identifier refers to?
- How do we store this information?

Need a **symbol table** to keep track of symbols/identifiers!

*(**Note:** Dynamic scoping follows roughly the same ideas, just not happening during compile time, but during runtime instead!)*

# Symbol table

## Definition (**symbol table** in a compiler):

A **symbol table** contains information about identifiers, that might prove useful to the compiler. It simply represents a mapping from a name to the thing that name refers to. This information typically consists of

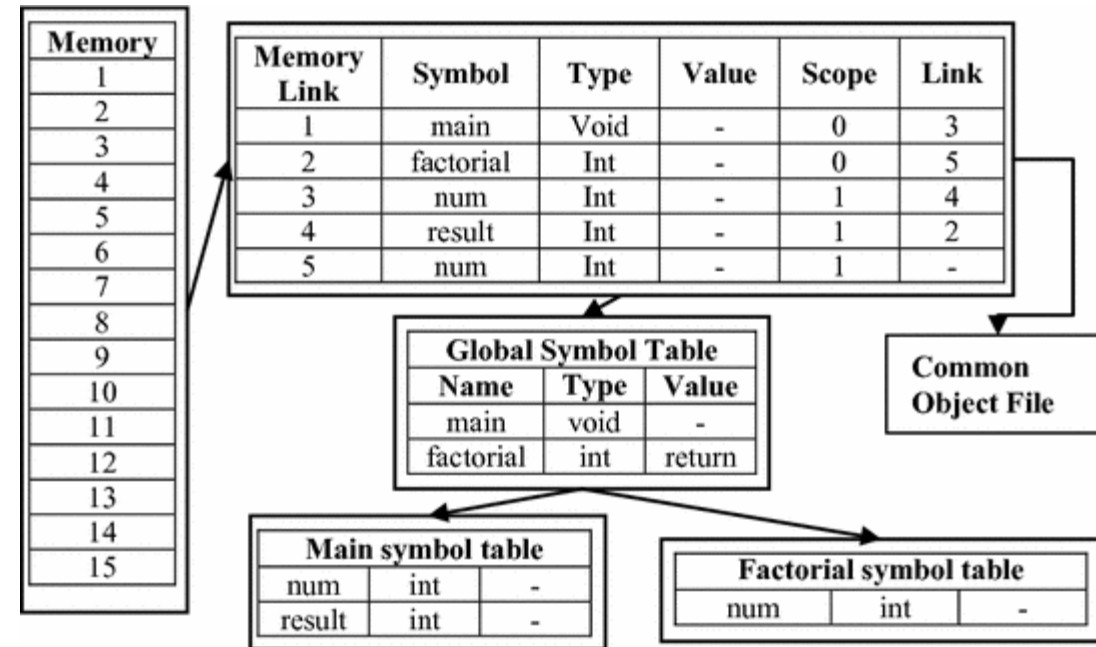
- Textual name,
- Data type,
- Declaration procedure,
- If array, number and size of dimensions,
- If procedure, number and type of parameters, location for function definition and procedure to be run by function when called,
- Etc.

# Implementing a symbol table

**Simple solution:** Define our symbol table as a **hash table** of some sort.

As we run our semantic analysis, continuously update the symbol table with information about what is in scope.

- What does this look like in practice?
- What operations need to be defined on it?
- How do we implement it?



**Forgot about hash tables?**

Time to go back to the contents of [50.004 Algorithms!](#)

# Implementing a symbol table

**For simplicity:** can be implemented as a **stack of maps**.

- Each map corresponds to a particular scope.
- Stack allows for easy “enter” and “exit” operations.

Symbol table operations are

- **Push scope:** Enter a new scope.
- **Pop scope:** Leave a scope, discarding all declarations in it.
- **Insert symbol:** Add a new entry to the current scope.
- **Lookup symbol:** Find what an identifier name corresponds to.
- **Remove symbol:** Remove entry from the current scope.

# Implementing an SDT for Arithmetic CFG

Consider the CFG below, modified with the following SDT rules.

$$E \rightarrow E + T \{E.val = E1.val + T.val\}$$

$$E \rightarrow T \{E.val = T.val\}$$

$$T \rightarrow T * F \{T.val = T1.val * F.val\}$$

$$T \rightarrow F \{T.val = F.val\}$$

$$F \rightarrow (E) \{F.val = E.val\}$$

$$F \rightarrow num \{F.val = num.val\}$$

In practice, the symbol table actions could be added to our Syntax-Driven Translation rules!

We would then have **two SDT instructions per CFG rule**:

- One that defines the elementary operation to be used (in a language close to the 3-address format),
- And one describing the symbol table updates.

# Implementing a symbol table

**Before demo:** the idea is to process each portion of the program that creates a scope independently (e.g. block statements, function calls, classes, etc.):

- **Start by entering a new scope.**
- **Add all variable declarations to the symbol table.**
- **Process the body of the block/function/class.**
- **Exit the scope.**


Most of the semantic analysis is defined in terms of recursive AST traversals like this.

# Implementing a symbol table

Consider the code below.

Global scope

During execution, the symbol table will be created as follows.



```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```

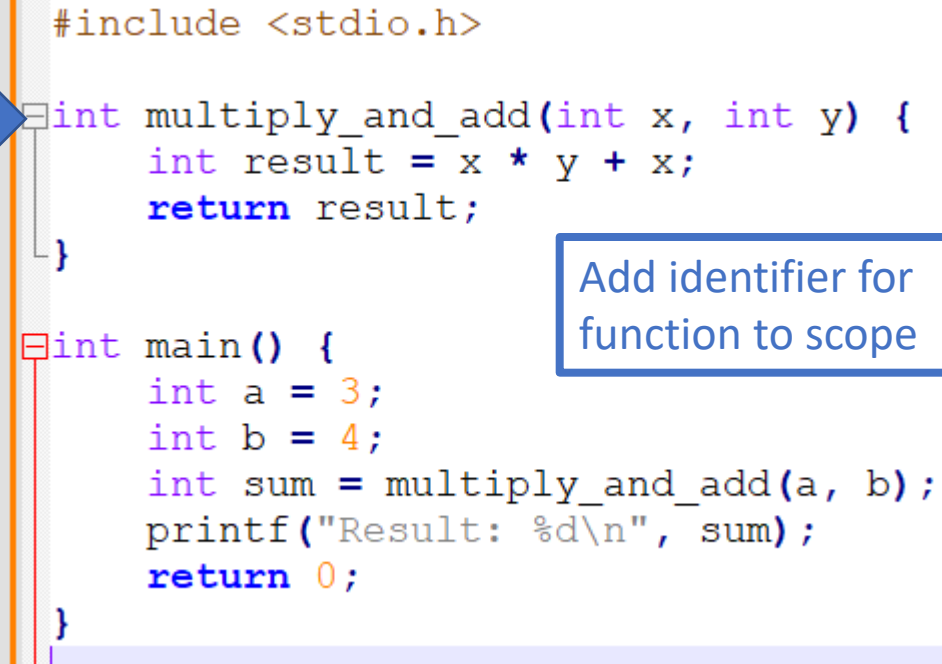
# Implementing a symbol table

Consider the code below.

During execution, the symbol table will be created as follows.

Global scope

└ multiply\_and\_add (function)



```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```

A blue arrow points to the opening curly brace of the `multiply_and_add` function. A blue box with the text "Add identifier for function to scope" is positioned next to the function definition.



# Implementing a symbol table

Consider the code below.

During execution, the symbol table will be created as follows.

Global scope

└ multiply\_and\_add (function)  
└ main (func. scope)

Skipped  
for now

```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```

Open new scope  
with name main()

# Implementing a symbol table

Consider the code below.

During execution, the symbol table will be created as follows.

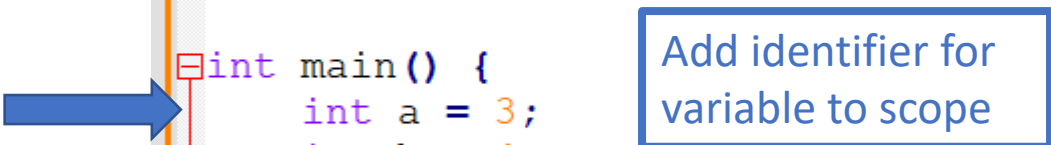
Global scope

└ multiply\_and\_add (function)  
└ main (func. scope)  
└ a (int)

```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```



# Implementing a symbol table

Consider the code below.

During execution, the symbol table will be created as follows.

Global scope

└ multiply\_and\_add (function)

└ main (func. scope)


└ a (int)

└ b (int)

```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```



Add identifier for  
variable to scope

# Implementing a symbol table

Consider the code below.

During execution, the symbol table will be created as follows.


```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```

Add identifier for variable to scope

Also, open a new scope for function call



Global scope

└ multiply\_and\_add (function)

└ main (func. scope)

└ a (int)

└ b (int)

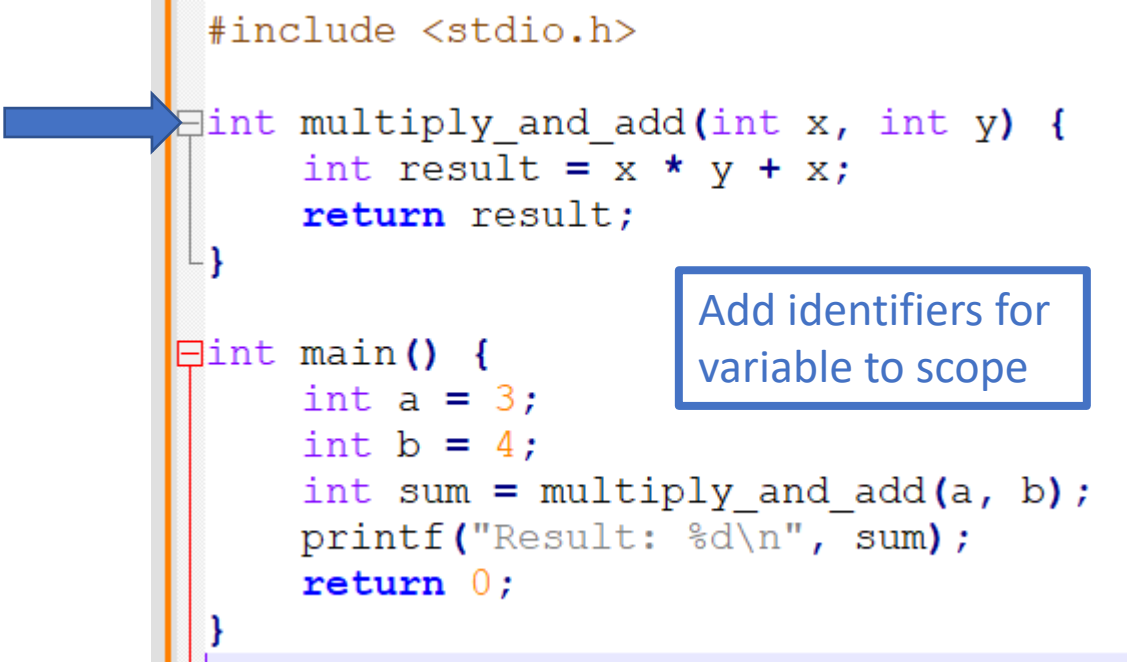
└ sum (int)

└ multiply\_and\_add (func. scope)

# Implementing a symbol table

Consider the code below.

During execution, the symbol table will be created as follows.



```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```

Add identifiers for variable to scope

Global scope

└ multiply\_and\_add (function)

└ main (func. scope)

└ a (int)

└ b (int)

└ sum (int)

└ multiply\_and\_add (func. scope)

└ x (int, parameter)

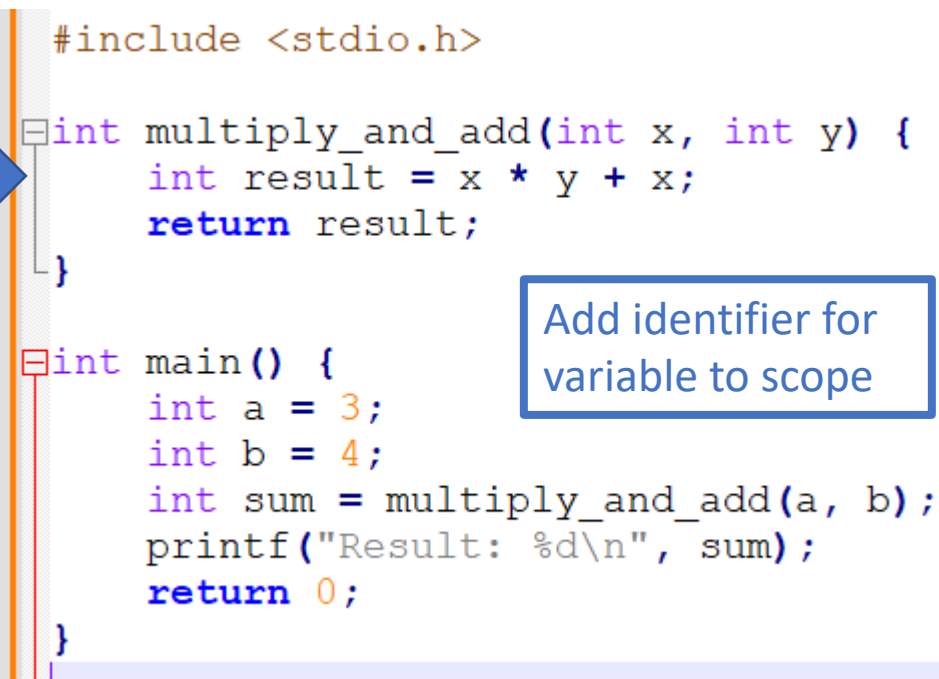
└ y (int, parameter)

└ result (int)

# Implementing a symbol table

Consider the code below.

During execution, the symbol table will be created as follows.



```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```

Add identifier for variable to scope

Global scope

└ multiply\_and\_add (function)

└ main (func. scope)

└ a (int)

└ b (int)

└ sum (int)

└ multiply\_and\_add (func. scope)

└ x (int, parameter)

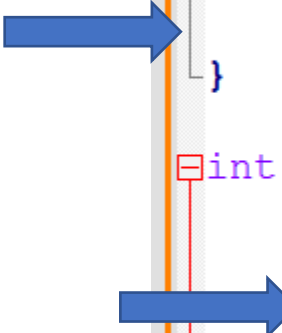
└ y (int, parameter)

└ result (int)

# Implementing a symbol table

Consider the code below.

During execution, the symbol table will be created as follows.



```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```

Return = transfer value of identifier result to identifier sum and close function scope

Global scope

- └ multiply\_and\_add (function)
- └ main (func. scope)

- └ a (int)

- └ b (int)

- └ sum (int)

- └ multiply\_and\_add (func. scope)

- └ x (int, parameter)

- └ y (int, parameter)

- └ result (int)

# Implementing a symbol table

Consider the code below.

During execution, the symbol table will be created as follows.

Global scope

└ multiply\_and\_add (function)

└ main (func. scope)

└ a (int)

└ b (int)

└ sum (int)

```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```

Return = transfer value of  
identifier result to identifier sum  
and close function scope



# Implementing a symbol table

Consider the code below.

During execution, the symbol table will be created as follows.

Global scope

└ multiply\_and\_add (function)

└ main (func. scope)

└ a (int)

└ b (int)

└ sum (int)

```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```

Print fetches identifier sum from  
symbol table

# Implementing a symbol table

Consider the code below.

During execution, the symbol table will be created as follows.

Global scope

└ multiply\_and\_add (function)

└ main (func. scope)

└ a (int)

└ b (int)

└ sum (int)

```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```

Return 0 and close function  
scope main()

# Implementing a symbol table

Consider the code below.

During execution, the symbol table will be created as follows.

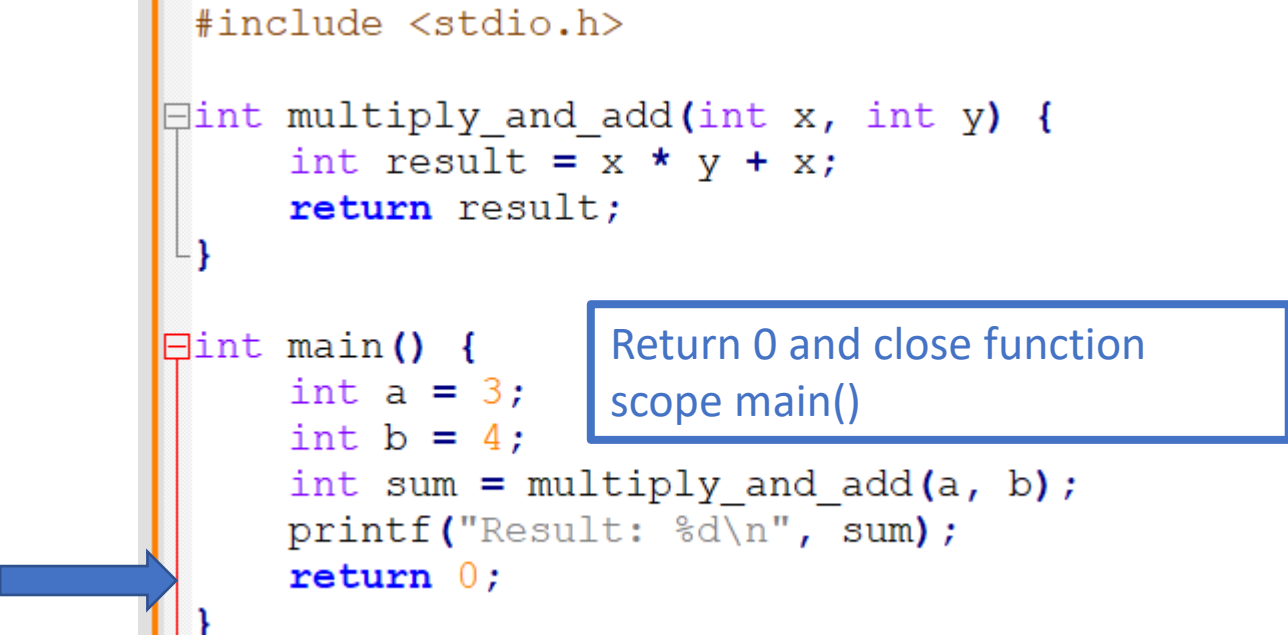
Global scope

└ multiply\_and\_add (function)

```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```



# Implementing a symbol table

Consider the code below.

Global scope

During execution, the symbol table will be created as follows.

```
#include <stdio.h>

int multiply_and_add(int x, int y) {
    int result = x * y + x;
    return result;
}

int main() {
    int a = 3;
    int b = 4;
    int sum = multiply_and_add(a, b);
    printf("Result: %d\n", sum);
    return 0;
}
```

In fact, close global scope and end

# Representing the symbol table as a stack

Technically, two ways to represent the symbol table below.

- **Simple stack**
- **Spaghetti stack**

Global scope

└ multiply\_and\_add (function)

└ main (func. scope)

└ a (int)

└ b (int)

└ sum (int)

└ multiply\_and\_add (func. scope)

└ x (int, parameter)

└ y (int, parameter)

└ result (int)

# Representing the symbol table as a stack

Technically, two ways to represent the symbol table below.

- **Simple stack**
- **Spaghetti stack**

## Simple stack

- Opening a new scope simply adds on top of the existing stack.

Global scope

Global scope

```

└ multiply_and_add (function)
└─ main (func. scope)
    └ a (int)
    └ b (int)
    └─ sum (int)
        └ multiply_and_add (func. scope)
            └ x (int, parameter)
            └ y (int, parameter)
            └─ result (int)
  
```

# Representing the symbol table as a stack

Technically, two ways to represent the symbol table below.

- **Simple stack**
- **Spaghetti stack**

Global scope

```

└ multiply_and_add (function)
└─ main (func. scope)
    └ a (int)
    └ b (int)
    └─ sum (int)
        └ multiply_and_add (func. scope)
            └ x (int, parameter)
            └ y (int, parameter)
            └─ result (int)
  
```

## Simple stack

- Opening a new scope simply adds on top of the existing stack.



# Representing the symbol table as a stack

Technically, two ways to represent the symbol table below.

- **Simple stack**
- **Spaghetti stack**

Global scope

```

└ multiply_and_add (function)
└ main (func. scope)
  └ a (int)
  └ b (int)
  └ sum (int)
    └ multiply_and_add (func. scope)
      └ x (int, parameter)
      └ y (int, parameter)
      └ result (int)

```

## Simple stack

- Opening a new scope simply adds on top of the existing stack.

Global scope
Function id multiply_and_add
Main scope
Integer id a
Integer id b
Integer id sum



# Representing the symbol table as a stack

Technically, two ways to represent the symbol table below.

- **Simple stack**
- **Spaghetti stack**

```

Global scope
├ multiply_and_add (function)
└ main (func. scope)
    ├── a (int)
    ├── b (int)
    └ sum (int)
        ├── multiply_and_add (func. scope)
        │   ├── x (int, parameter)
        │   ├── y (int, parameter)
        │   └ result (int)
    
```

## Simple stack

- Opening a new scope simply adds on top of the existing stack.

Global scope
Function id multiply_and_add
Main scope
Integer id a
Integer id b
Integer id sum
Multiply_and_add func. scope
Integer id x
Integer id y
Integer id result

# Representing the symbol table as a stack

Technically, two ways to represent the symbol table below.

- **Simple stack**
- **Spaghetti stack**

Global scope

```

└ multiply_and_add (function)
└ main (func. scope)
  └ a (int)
  └ b (int)
  └ sum (int)
    └ multiply_and_add (func. scope)
      └ x (int, parameter)
      └ y (int, parameter)
      └ result (int)

```

## Simple stack

- Opening a new scope simply adds on top of the existing stack.

Global scope
Function id multiply_and_add
Main scope
Integer id a
Integer id b
Integer id sum

# Representing the symbol table as a stack

Technically, two ways to represent the symbol table below.

- **Simple stack**
- **Spaghetti stack**

Global scope

```

└ multiply_and_add (function)
└ main (func. scope)
  └ a (int)
  └ b (int)
  └ sum (int)
    └ multiply_and_add (func. scope)
      └ x (int, parameter)
      └ y (int, parameter)
      └ result (int)

```

## Simple stack

- Opening a new scope simply adds on top of the existing stack.



# Representing the symbol table as a stack

Technically, two ways to represent the symbol table below.

- **Simple stack**
- **Spaghetti stack**

## Simple stack

- Opening a new scope simply adds on top of the existing stack.

Global scope

Global scope

└ multiply\_and\_add (function)

└ main (func. scope)

└ a (int)

└ b (int)

└ sum (int)

└ multiply\_and\_add (func. scope)

└ x (int, parameter)


└ y (int, parameter)

└ result (int)

# Representing the symbol table as a stack

## Spaghetti stack

- Treat the symbol table as a linked structure of scopes.
- Each scope stores a pointer to its parents, but not vice-versa.



Global scope

# Representing the symbol table as a stack

## Spaghetti stack

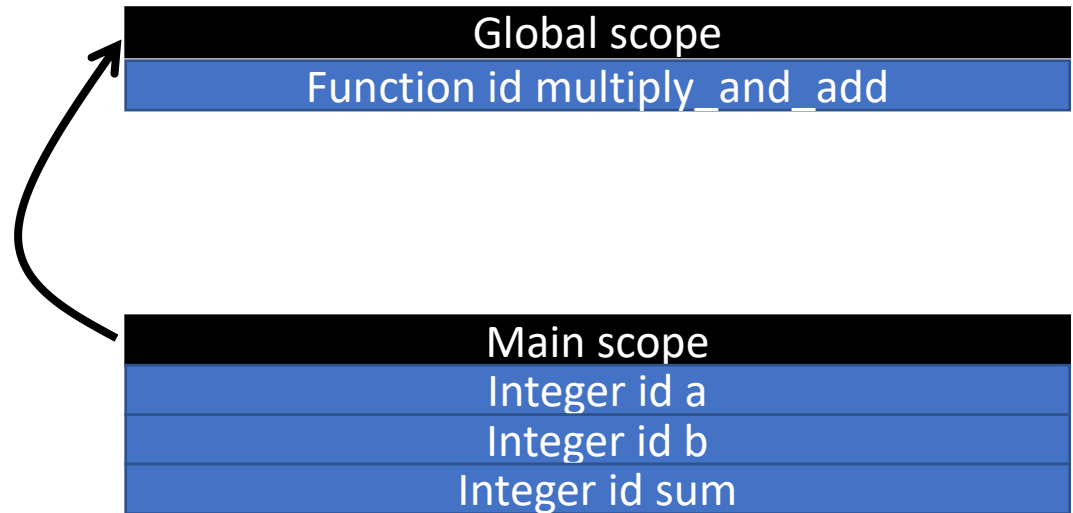
- Treat the symbol table as a linked structure of scopes.
- Each scope stores a pointer to its parents, but not vice-versa.



# Representing the symbol table as a stack

## Spaghetti stack

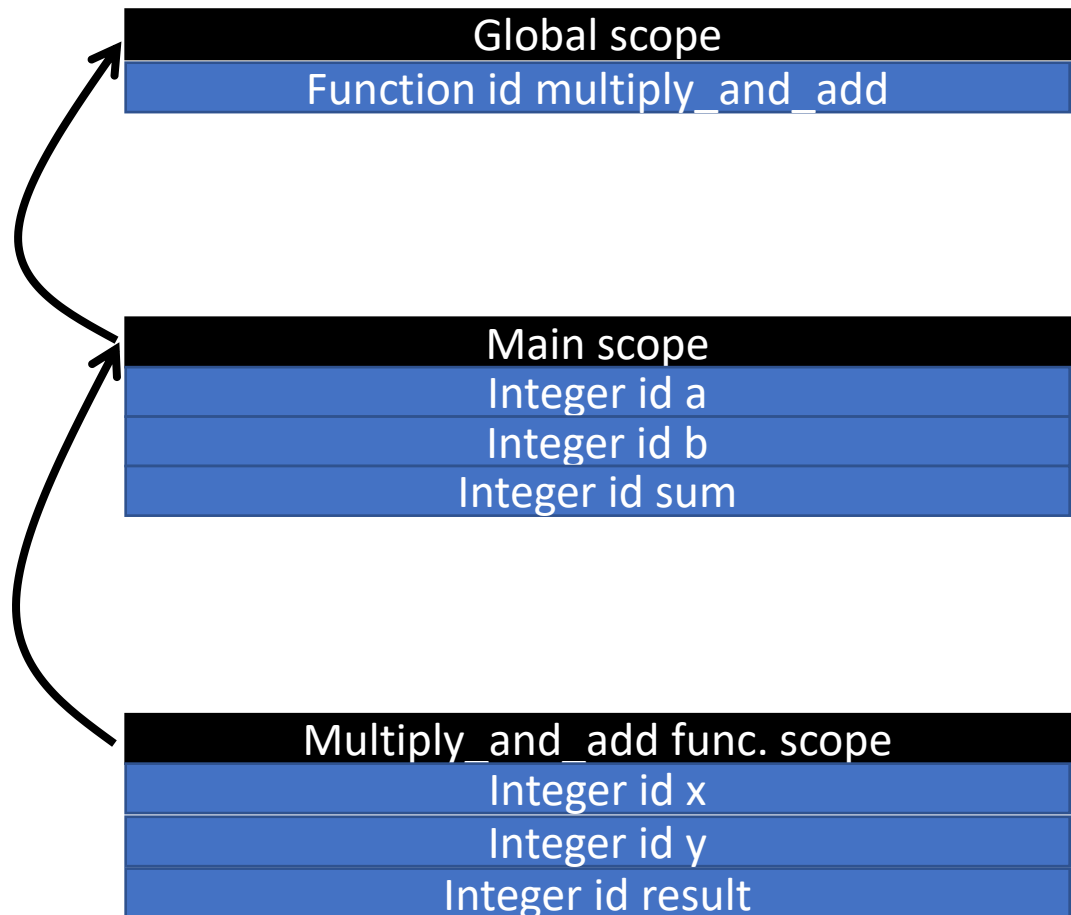
- Treat the symbol table as a linked structure of scopes.
- Each scope stores a pointer to its parents, but not vice-versa.



# Representing the symbol table as a stack

## Spaghetti stack

- Treat the symbol table as a linked structure of scopes.
- Each scope stores a pointer to its parents, but not vice-versa.

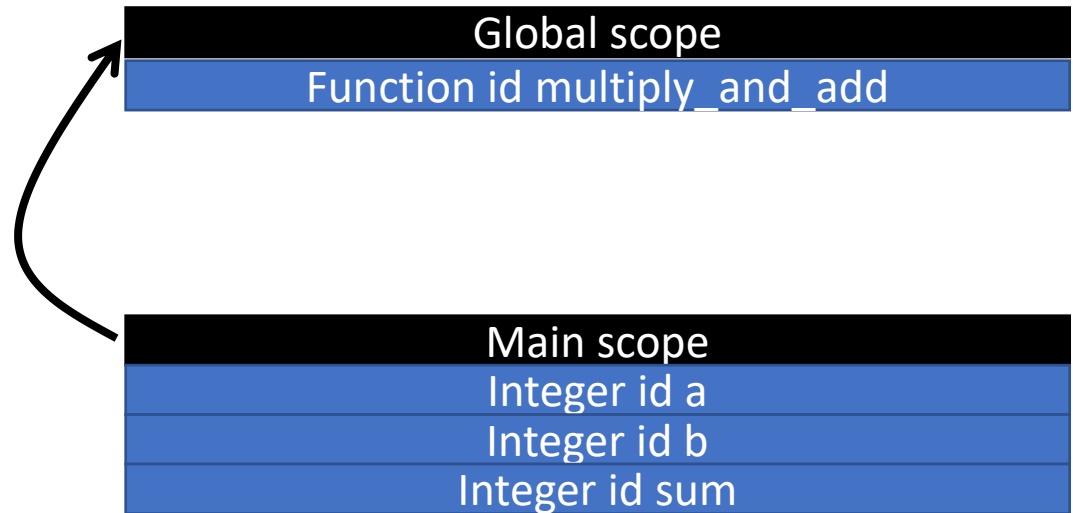




# Representing the symbol table as a stack

## Spaghetti stack

- Treat the symbol table as a linked structure of scopes.
- Each scope stores a pointer to its parents, but not vice-versa.



# Representing the symbol table as a stack

## Spaghetti stack


- Treat the symbol table as a linked structure of scopes.
- Each scope stores a pointer to its parents, but not vice-versa.



# Representing the symbol table as a stack

## Spaghetti stack

- Treat the symbol table as a linked structure of scopes.
- Each scope stores a pointer to its parents, but not vice-versa.

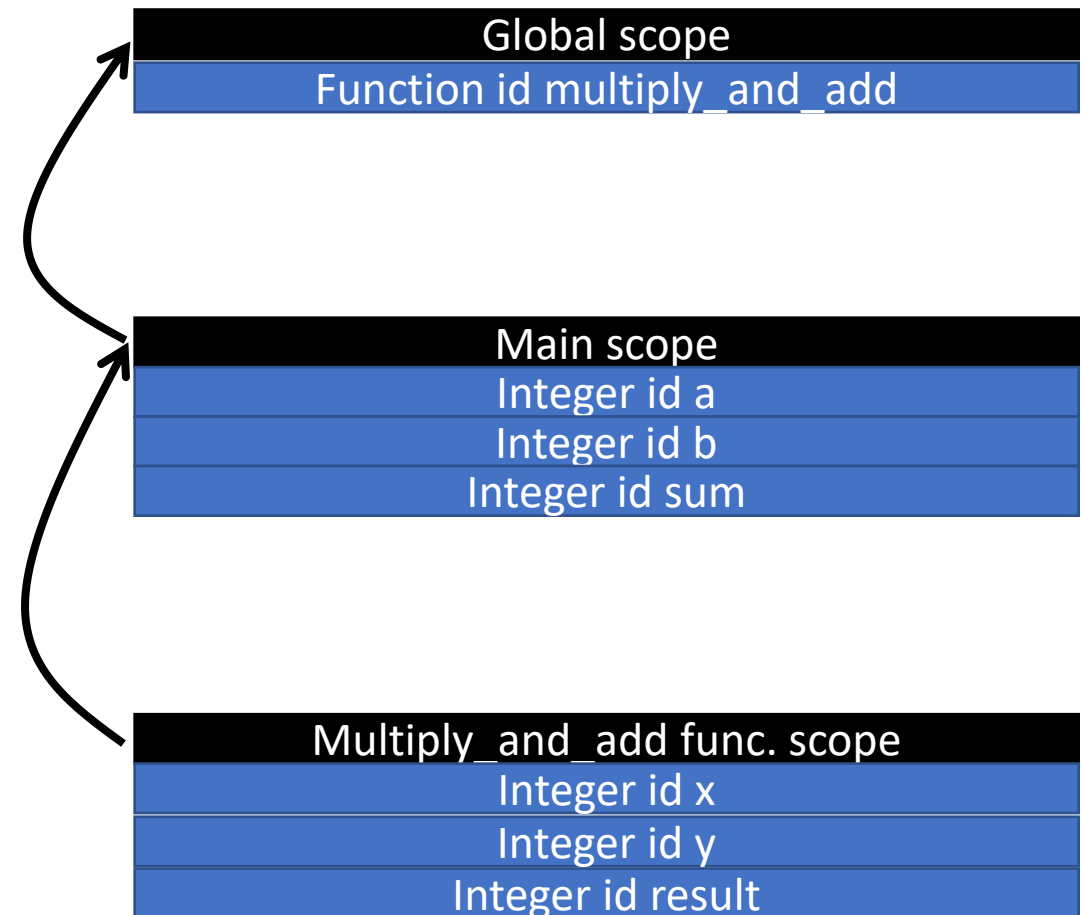


Global scope

# Representing the symbol table as a stack

## Spaghetti stack

- Treat the symbol table as a linked structure of scopes.
- Each scope stores a pointer to its parents, but not vice-versa.
- **Observation:** From any point in the program, symbol table appears to be a simple stack containing elements of the scope and the scope only.
- **Better?**



# Catching scope errors

## Scope problems

- A variable is defined after it is called for the first time.
- A function is defined after it is called for the first time and no function prototypes were mentioned.
- A variable defined in a function is called outside of the function.
- A variable name is defined twice.
- Etc.

Most scope problems will be caught when

- An operation retrieving something from the symbol table fails (variable/function undefined in scope, etc.).
- An operation tries adding a variable/function with an identifier name that already exists in table (identifier defined twice in same scope).

# Saving the day?

Consider the example on the right.

- Function `f()` requires two parameters `x` and `t` to compute the returned value.
- But `t` is not in the scope of the function, it is in the global scope instead.
- **Question:** How does Python save the day, in your opinion?

```
1 def f(x):  
2     return x + t + 2
```

```
1 t = 4  
2 x = 3  
3 y = f(x)  
4 print(y)
```

9

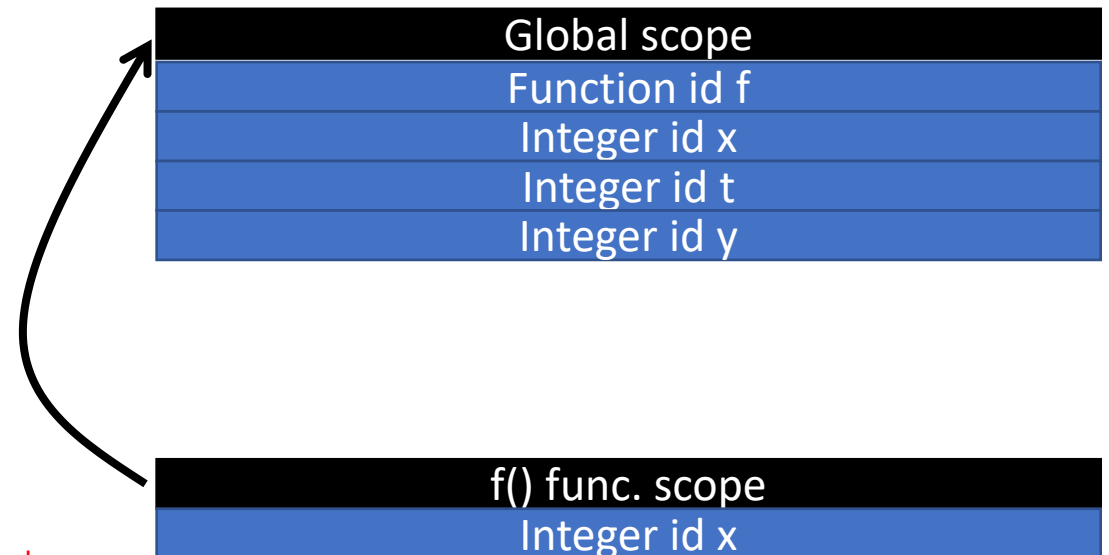
# Saving the day?

- **Reason:** In Python, when you reference a variable inside a function, it first looks for that variable within the local scope of the function.
- If it does not find it there, it tries searching for it in the next enclosing scope, which in this case is the global scope.

```
1 def f(x):  
2     return x + t + 2
```

```
1 t = 4  
2 x = 3  
3 y = f(x)  
4 print(y)
```

9



# The LEGB rule

**In fact: The scoping rule used in Python is called the “LEGB” rule, which stands for Local, Enclosing, Global, and Built-in.**

**It is a spaghetti stack with different levels of scope.**

When a variable is referenced, Python searches in the following order.

- **Local scope:** variables defined within the function.
- **Enclosing scope:** variables defined in any enclosing functions, from innermost to outermost.
- **Global scope:** variables defined at the top level.
- **Built-in scope:** predefined built-in names in Python, like `print()`, `sum()`, etc.



# How about C then?

Consider the example on the right.

- Function `f()` requires two parameters `x` and `t` to compute the returned value.
- But `t` is not in the scope of the function, it is in the global scope instead.
- **Question #2:** Also works in the equivalent implementation in C?

```
2  #include <stdio.h>
3
4  int f(int x) {
5      return x + t + 2;
6  }
7
8  int main() {
9      int x = 4;
10     int t = 3;
11     int y = f(x);
12     printf("%d", y);
13     return 0;
14 }
```

# How about C then?

**Answer:** In C, variables have a **block scope**, which means that a variable is only accessible within the block of code it is defined in.

## Better or worse than LEGB?

Usually more stable and simpler to encoder in a compiler than “LEGB”, very stable/strict and less friendly on “mistakes”.

Not an easy question...

```
2  #include <stdio.h>
3
4  int f(int x) {
5      return x + t + 2;
6  }
7
8  int main() {
9      int x = 4;
10     int t = 3;
11     int y = f(x);
12     printf("%d", y);
13     return 0;
14 }
```

# In short, the block scope (C)

## Advantages:

- Encourages local reasoning: You can understand the behavior of a variable just by looking at the block of code it is defined in.
- Reduces the chance of accidentally using the wrong variable, as variables with the same name in different scopes do not conflict.

## Disadvantages:

- Requires the use of global variables or explicit parameter passing to share data between functions, which can make the code more complex and error-prone.

# In short, the LEGB scope (Python)

## Advantages:

- Provides more flexibility in accessing variables from different scopes, which can make code shorter and more convenient to write.
- Allows for closures, which are functions that remember the environment they were created in, enabling more functional programming techniques.

## Disadvantages:

- Can lead to unintended side effects if a variable from an outer scope is accidentally used or modified.
- Can make it harder to reason about the code since a variable might be accessed or modified in multiple scopes.

# In short, the LEGB scope

## Advantages:

- Provides more flexibility in accessing variables, which can make code shorter and more readable.
- Allows for closures, which are functions that remember the environment they were created in, enabling various programming techniques.

## Disadvantages:

- Can lead to unintended side effects if a variable from an outer scope is accidentally used or modified.
- Can make it harder to reason about the code since a variable might be accessed or modified in multiple scopes.

```
1 def g(l, x):  
2     l.append(x)  
3     return len(l)
```

```
4  
5 l = [1, 2, 3]  
6 x = 7  
7 y = g(l, x)  
8 print(y)  
9 print(l)
```

```
4  
[1, 2, 3, 7]
```

# Single-pass vs. multi-pass compilers

## Definition (**single-pass vs. multi-pass compilers**):

Some compilers can combine scanning, parsing, semantic analysis, and code generation into the same pass. These are called **single-pass compilers**.

Other compilers rescan the input multiple times. These are called **multi-pass compilers**.

# What happens in multi-pass compilers

For instance, on each pass, the compiler could check one specific aspect of the source code. For instance,

- Do lexical analysis, produce tokens stream and completely parse the input file into an abstract syntax tree (first pass).
- Walk the AST, gathering information about classes definitions (second pass).
- Walk the AST gathering information about functions definitions (third pass).
- Etc.

**Could combine some of these, though they are logically distinct.**

# Single-pass vs. multi-pass compilers

**Which is better: single-pass or multi-pass compilers?**

In general, **single-pass compilers** are usually faster than **multi-pass compilers**, but the logic complexity they can check is reduced.

**Another debate I do not want to be part of!**





# Quiz time!

**What is the primary purpose of semantic analysis in a compiler?**

- A. To generate intermediate code
- B. To check the syntax of the input code
- C. To optimize the generated code
- D. To ensure the input code is semantically correct and meaningful

# Quiz time!

**What is the primary purpose of semantic analysis in a compiler?**

- A. To generate intermediate code
- B. To check the syntax of the input code
- C. To optimize the generated code
- D. To ensure the input code is semantically correct and meaningful**

# Quiz time!

**What is a symbol table used for in a compiler?**

- A. To store the intermediate code representation
- B. To store information about variables, functions, and other identifiers
- C. To store the parse tree of the input code
- D. To store the machine code generated by the compiler

# Quiz time!

**What is a symbol table used for in a compiler?**

- A. To store the intermediate code representation
- B. To store information about variables, functions, and other identifiers**
- C. To store the parse tree of the input code
- D. To store the machine code generated by the compiler

# Quiz time!

**Which of the following best describes a static scoping rule?**

- A. The visibility of a variable only depends on its position in the source code
- B. The visibility of a variable depends on the call history of functions
- C. The visibility of a variable depends on the order of function calls at runtime
- D. The visibility of a variable depends on the values assigned to it

# Quiz time!

**Which of the following best describes a static scoping rule?**

- A. The visibility of a variable only depends on its position in the source code**
- B. The visibility of a variable depends on the call history of functions
- C. The visibility of a variable depends on the order of function calls at runtime
- D. The visibility of a variable depends on the values assigned to it

# Quiz time!

**In a compiler that uses syntax-directed translation, what is the purpose of the SDT instructions?**

- A. To perform lexical analysis
- B. To create a parse tree
- C. To replace each production rule used in a derivation with intermediate code instructions
- D. To let us know how to update the symbol table
- E. To convert intermediate code to machine code

# Quiz time!

**In a compiler that uses syntax-directed translation, what is the purpose of the SDT instructions?**

- A. To perform lexical analysis
- B. To create a parse tree
- C. To replace each production rule used in a derivation with intermediate code instructions**
- D. To let us know how to update the symbol table**
- E. To convert intermediate code to machine code



# Quiz time!

**In a compiler, what is the purpose of a "spaghetti stack"?**

- A. To optimize memory usage during parsing
- B. To store the parse tree during syntax analysis
- C. To maintain the symbol table as a linked structure of scopes
- D. To generate machine code for the target architecture

# Quiz time!

**In a compiler, what is the purpose of a "spaghetti stack"?**

- A. To optimize memory usage during parsing
- B. To store the parse tree during syntax analysis
- C. To maintain the symbol table as a linked structure of scopes**
- D. To generate machine code for the target architecture

# Conclusion

- **Semantic analysis** verifies that a syntactically valid program is correctly-formed and computes additional information about the **meaning of the program**.
- **Scope checking** determines what variables, functions or classes are referred to by each name in the program.
- Scope checking is usually done with a **symbol table** implemented either as a **stack** or **spaghetti stack**.
- Simple semantic analysers will operate in a **single pass**, while some semantic analysers operate in **multiple passes** in order to gain more information about the program.