

50.051 Programming Language Concepts

W11-S3 Semantic Analysis(Part 2) and intermediate code representation

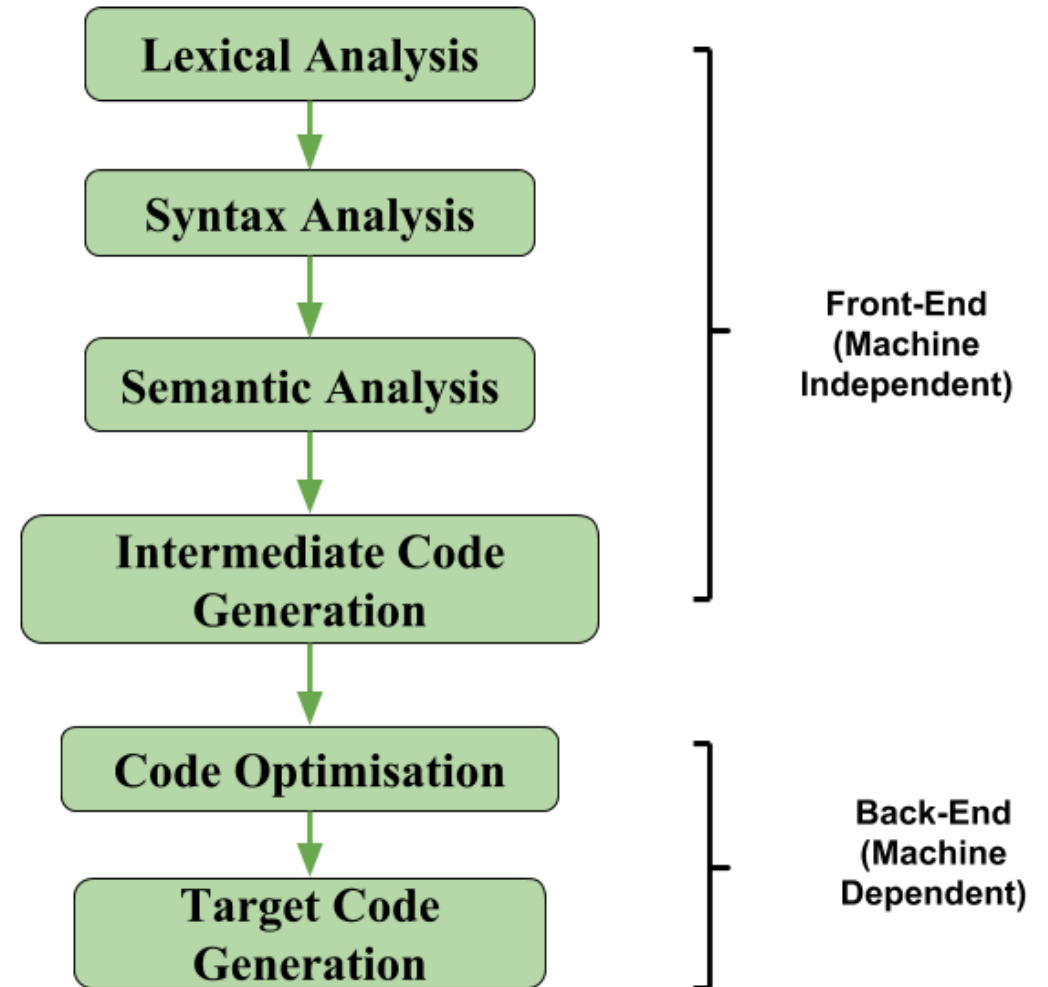
Matthieu De Mari



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

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

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 (duh!) 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.

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.

What are Types?

Definition (**type**):

This is a difficult definition, by here is my attempt.

A **type** is a **classification** that defines a **set of values and the operations that can be performed on variables of said type** (e.g. types methods and operators behaviours) .

It also defines a **way to encode said information** about the variable in the memory (e.g. char, short, long, int, etc.)

In short, the purpose of a **types system** is to provide a way to describe and reason about data and their behaviour within a program.

```
1 # Show all methods and special methods for int
2 print(dir(int))
```

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattr__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'as_integer_ratio', 'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

```
1 x = 10
2 print(x.real)
3 print(x.imag)
```

10

0

Strong vs. weak type systems

Definition (**Strong type system**):

A **strong type system** strictly enforces rules about which types can be used in specific operations and contexts.

In languages with a strong type system, we typically see more explicit type declarations, and the compiler or runtime will enforce type constraints, often causing errors when there is a type mismatch.

This helps catch potential errors early, as well as promotes code readability and allows for better optimization.

Strong vs. weak type systems

Definition (**Weak type system**):

A **weak type system** is **more lenient with type enforcement**.

In languages with a weak type system, you may see implicit type conversions (coercions) and less strict rules about which types can be used in certain operations.

This can lead to more flexibility and shorter code, but also introduces the risk of subtle bugs and less predictable behaviour.

For instance, Python allows things like “`x = True and 1`”. It resolves the type mismatch (bool and int) by performing an implicit conversion of the int 1 into a Boolean. Might not be allowed in other languages.

Implicit vs. Explicit type conversion

Definition (**Implicit type conversion**):

Implicit type conversion, also known as **type coercion** or **automatic type conversion**, occurs when the compiler or runtime automatically converts a value of one type to another type without the programmer explicitly requesting the conversion.

This typically happens when performing operations with operands of different types, and the language's type system automatically promotes or converts one of the values to match the other.

Implicit type conversions can make code more concise, but they can also introduce unexpected behaviour if the conversion rules are not well-defined or understood.

Implicit vs. Explicit type conversion

Definition (**Explicit type conversion**):

Explicit type conversion, also known as **type casting** or **manual type conversion**, occurs when the programmer explicitly requests the conversion of a value from one type to another using language-specific syntax or functions.

This is often necessary in languages with strong type systems that do not allow implicit type conversions or when the programmer wants to override the default behaviour of implicit conversions.

Explicit type conversions provide more control and predictability, but they can also make the code more verbose and may introduce errors if not used carefully.

Strong vs. weak type systems

Strong vs. Weak defines a spectrum (not a binary classification).

- **C:** will allow to sum together a char and an int, same thing with a float and an int. An **implicit conversion** will happen.
- **Python:** will allow to sum together a float and an int using an **implicit conversion**. The char and int addition is not straightforward and requires an **explicit Unicode conversion** of the char using `ord()`.
- **Strict languages (e.g. Haskell):** will simply refuse to operate if the types do not exactly match.

Strong vs. weak type systems

Strong vs. Weak defines a spectrum (not a binary classification).

- **C:** will allow to sum together a char and an int, same thing with a float and an int. An **implicit conversion** will happen.
- **Python:** will allow to sum together a float and an int using an **implicit conversion**. The char and int addition is not straightforward and requires an **explicit Unicode conversion** of the char using `ord()`.
- **Strict languages (e.g. Haskell):** will simply refuse to operate if the types do not exactly match. Explicit conversions only.

Which is better then? Another open debate, for you to decide!

Types errors

Type errors will typically occur in the following two scenarios:

- **An explicit method or operation is used on a type that does not support it** (e.g. in Python the explicit conversion of a string “123.4” into int, using + operation between int “7” and string “hello”, calling append() method on string, etc.).
- **There is no implicit conversion that might resolve the issue above, if any** (e.g. summing a char “c” and an int “7” in C, will simply require an implicit conversion, but this will not do in Python).

This can be implemented with an “if-else” again and a set of rules on the basic types in our language.

Static vs Dynamic type checking

Definition (**Static vs. Dynamic type checking**):

In **Static Type Checking**, types are checked **at compile-time**, errors are caught **before program execution** (e.g., Java, C++).

In **Dynamic Type Checking**, types are checked **at run-time**, errors are caught **during program execution** (e.g., Python, JavaScript).

In **No (or None?) Type Checking**, types are not checked and everything is processed in binary! (Ok, maybe not the best idea).

Which one is best? Another open debate.

Endless debate

There is an **endless debate** about the best type system to use.

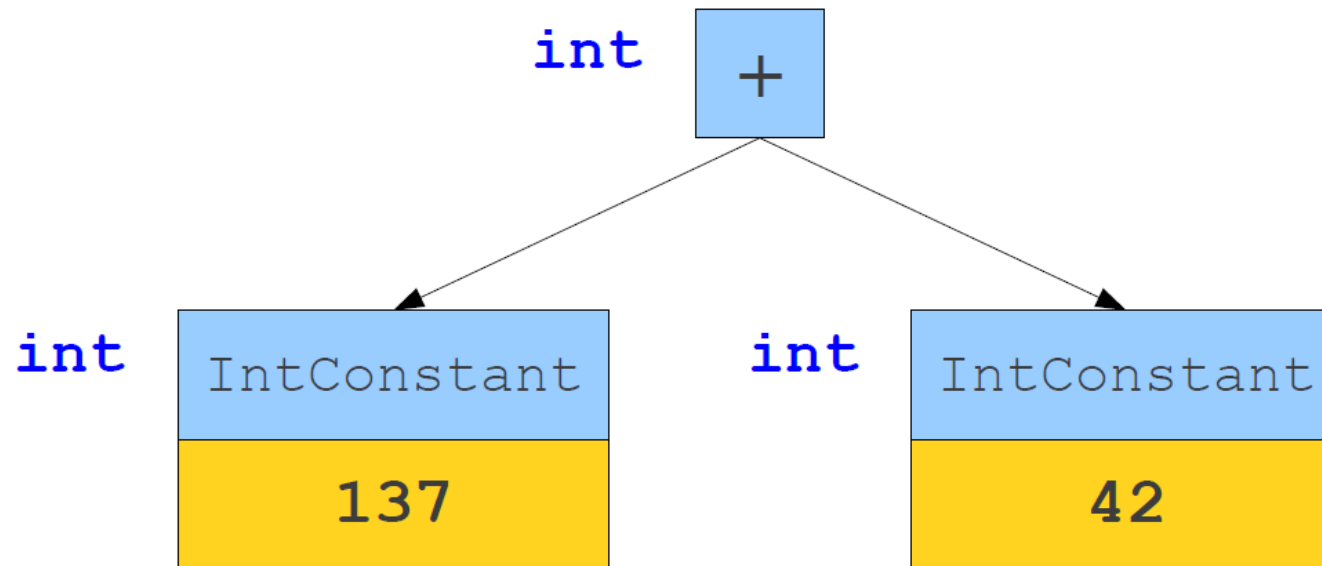
- Dynamic type checking systems make it easier to prototype.
- Static type checking systems have fewer bugs.
- None type checking is quite frankly a terrible idea.
- Strongly-typed languages are often more robust.
- Weakly-typed systems are often faster.

I am staying out of this!

Types inference

Definition (**types inference**):

The bread-and-butter concept behind a types system is **types inference**, that is a logical set of pre-established rules that will define the behaviour to use for any operation on any given type.



Types inference for operators (Python)

The case of Python

- In the case of operators (+, *, -, /, etc.), the behaviour of a given operation “x operator y” is often decided by looking at the left-hand variable type first and calling its special method for said operator, here `__add__()`.

```

1 class MyInt():
2     def __init__(self, n):
3         self.value = n
4     def __add__(self, other):
5         print("Using the MyInt behavior")
6         return self.value + other.value
7
8 class MyOtherInt():
9     def __init__(self, n):
10        self.value = n
11    def __add__(self, other):
12        print("Using the MyOtherInt behavior")
13        return self.value * other.value

```

```

1 x = MyInt(10)
2 y = MyOtherInt(4)

```

```

1 z = x + y
2 print(z)

```

Using the MyInt behavior
14

```

1 z = y + x
2 print(z)

```

Using the MyOtherInt behavior
40

Types inference for operators (Python)

The case of Python

- In the case of operators (+, *, -, /, etc.), the behaviour of a given operation “x operator y” is often decided by looking at the left-hand variable type first and calling its special method for said operator, here `__add__()`.
- If `__add__()` is not defined for left-hand type, try using the `__radd__()` for right-hand type.

```

1 class MyInt():
2     def __init__(self, n):
3         self.value = n
4     def __add__(self, other):
5         print("Using the MyInt behavior")
6         return self.value + other.value
7     def __radd__(self, other):
8         print("Using the MyInt Radd behavior")
9         return self.value - other.value
10
11 class MyOtherInt():
12     def __init__(self, n):
13         self.value = n

```

```

1 x = MyInt(10)
2 y = MyOtherInt(4)

```

```

1 z = x + y
2 print(z)

```

Using the MyInt behavior
14

```

1 z = y + x
2 print(z)

```

Using the MyInt Radd behavior
6

Types inference for operators (Python)

The case of Python

- In the case of operators (+, *, -, /, etc.), the behaviour of a given operation “x operator y” is often decided by looking at the left-hand variable type first and calling its special method for said operator, here `__add__()`.
- If `__add__()` is not defined for left-hand type, try using the `__radd__()` for right-hand type.
- This behaviour is Python has to do with the way they designed their object-oriented structure for the programming language.
- It defines clear types inferences rules for **operator overloading**.
- It is, however, not universal and not the only framework.

Types inference for operators (C)

The case of C

- In C, operator behaviour is different.
- C follows specific rules for type promotion and conversion when performing operations with different types.
- For example, when adding an int and a float, C will implicitly convert the int to a float before performing the addition.
- Behaviour of operators in C is determined by the language's type system and promotion rules, rather than the left-hand operand's type alone.

Types inference for operators (C)

The case of C

- In C, operator behaviour is different
- C follows specific rules for type promotion when performing operations with different types
- For example, when adding an `int` to a `float` before performing the addition, the `int` is promoted to a `float`
- Behaviour of operators in C is defined by the C standard and promotion rules, rather than by the compiler

```

1
2  #include <stdio.h>
3
4  int main() {
5      int a = 5;
6      float b = 3.5;
7
8      // When adding 'a' (int) and 'b' (float),
9      // C will implicitly convert 'a' to a float
10     // before performing the addition
11     // The 'int' value 5 is promoted to a 'float'
12     // value (5.0) before the addition
13     float result = a + b;
14
15     // Output: Result: 8.500000
16     printf("Result: %f\n", result);
17
18     return 0;
19 }
20

```

Types inference for operators (C++)

The case of C

- In C, operator behaviour is different.
- C follows specific rules for type promotion and conversion when performing operations with different types.
- For example, when adding an int and a float, C will implicitly convert the int to a float before performing the addition.
- Behaviour of operators in C is determined by the language's type system and promotion rules, rather than the left-hand operand's type alone.
- To some extent, operator overloading can also be made in C++ for custom objects (to be seen later).

Types inference for operators (C++)

The case of C

- In C, operator behaviour is different.
- C follows specific rules for type promotion performing operations with different types.
- For example, when adding an int and a float, the int is promoted to a float before performing the addition.
- Behaviour of operators in C is determined by precedence and promotion rules, rather than the left-to-right rule.
- To some extent, operator overloading can be used for custom objects (to be seen later).

```

1  #include <iostream>
2
3
4  class MyInt {
5  public:
6      MyInt(int n) : value(n) {}
7      int operator+(const MyInt& other) {
8          std::cout << "Using the MyInt behavior" << std::endl;
9          return value + other.value;
10     }
11     friend int operator+(int other, const MyInt& self);
12 private:
13     int value;
14 };
15
16 int operator+(int other, const MyInt& self) {
17     std::cout << "Using the MyInt Radd behavior" << std::endl;
18     return self.value + other;
19 }
20
21 class MyOtherInt {
22 public:
23     MyOtherInt(int n) : value(n) {}
24     int value;
25 };
26
27 int main() {
28     MyInt x(10);
29     MyOtherInt y(4);
30
31     int z = x + y.value;
32     std::cout << z << std::endl;
33
34     z = y.value + x;
35     std::cout << z << std::endl;
36
37     return 0;
38 }

```


Type Checking as Proofs

- We can think of type checking in semantic analysis as proving claims about the types of expressions.
- We begin with a set of **axioms**, then apply our **inference rules** to determine the types of expressions.
- Many type systems can be thought of as proof systems.
- Typically defined as ‘If ... Then ...’ statements.

Some examples of types inference rules

Some examples of types inferences rules.

- “If x is an identifier that refers to an object of type T , the expression x has type T .”
- “If e is an integer constant, e has type int .”
- “If the operands e_1 and e_2 of $e_1 + e_2$ are known to have types int and int , then $e_1 + e_2$ has type int .”

Formalism for Type Inference

- We will represent our axioms and inference rules using the formalism below.

Preconditions

Postconditions

- It reads as “If preconditions are true, then we can infer postconditions are true.”
- We also denote “ $S \vdash e : T$ ” if the expression e in scope S has type T .

Formalism for Type Inference

- We will represent our axioms and inference rules using the formalism below.

Preconditions

Postconditions

- It reads as “If preconditions are true, then we can infer postconditions are true.”
- We also denote “ $S \vdash e : T$ ” if the expression e in scope S has type T .

Some examples of types inference rules

Some examples of types inferences rules.

- “True is of type Boolean, whatever the scope is (no preconditions).”

$$\frac{}{\vdash \text{true} : \text{bool}}$$

- “If e is an integer constant in scope S, then e has type int.”

$$\frac{\text{e is an integer constant}}{S \vdash e : \text{int}}$$

Some examples of types inference rules

Some examples of types inferences rules

- “If the operands $e1$ and $e2$ of the operation $e1 + e2$ are known to have types int and int , then $e1 + e2$ has type int .”

$$\frac{\begin{array}{c} S \vdash e1 : \text{int} \\ S \vdash e2 : \text{int} \end{array}}{S \vdash e1 + e2 : \text{int}}$$

Some examples of types inference rules

Some examples of types inferences rules.

- “If the operands $e1$ and $e2$ of the operation $e1 + e2$ are known to have types `int` and `float`, then $e1 + e2$ has type `float`, but will require $e1$ to be converted implicitly to `float`.”

$$\frac{\begin{array}{c} S \vdash e1 : \text{int} \\ S \vdash e2 : \text{float} \end{array}}{S \vdash e1 : \text{float} \\ S \vdash e1 + e2 : \text{float}}$$

- Can be used to introduce **implicit conversions of types!**

Operations tables

Many times, these rules can be assembled in result types tables, where each cell will indicate

- If the operation is allowed,
- What is the resulting type,
- What is the operation to use for said types
- If any implicit conversions must be made.

Op: +	bool	int	float	string
bool
int
float
string

Op: +, rows are left operand, columns are right operand.

	bool	int	float	string
bool	Res: bool Behavior: or	Convert int to bool Res: bool Behavior: or	Convert float to bool Res: bool Behavior: or	Convert string to bool? Res: bool Behavior: or
int	Convert int to bool Res: bool Behavior: or	Res: int Behavior: math. addition	Convert int to bool Res: float Behavior: math. addition	Illegal, should trigger a type error
float	Convert float to bool Res: bool Behavior: or	Convert int to bool Res: float Behavior: math. addition	Res: float Behavior: math. addition	Illegal, should trigger a type error
string	Convert string to bool? Res: bool Behavior: or	Illegal, should trigger a type error	Illegal, should trigger a type error	Res: string Behavior: concatenation

Some examples of types inference rules

- When an identifier (e.g. a variable with name x) is called, a lookup procedure will check the symbol table for a given scope S (or multiple ones if LEGB).

$$\begin{array}{c}
 x \text{ is in identifier called in scope } S \\
 x \text{ appears in scope } S \text{ with type int } (S \vdash x : \text{int}) \\
 S \vdash 7 : \text{int} \\
 \hline
 S \vdash x + 7 : \text{int}
 \end{array}$$

- Can also be used to resolve **resulting types of operations mixing constants and identifiers!**

Some examples of types inference rules

- **What if the identifier refers to a function?**

$$\frac{\begin{array}{l} f \text{ is in identifier called in scope } S \\ f \text{ appears in scope } S \text{ as a non-member function} \\ f \text{ expects types } (T_1, \dots, T_n) \text{ and returns type } U \\ S \vdash e_i : T_i \text{ for } 1 \leq i \leq n \end{array}}{S \vdash f(e_1, \dots, e_n) : U}$$

- Can also be used to resolve **resulting types of function calls!**

Some examples of types inference rules

- **Also works on Compound types (e.g. arrays) and their operations, for instance indexing.**

$$\frac{S \vdash e1 : T[] \quad S \vdash e2 : \text{int}}{S \vdash e1[e2] : T}$$

- **Question about array bounds and index checking:** Semantic analysis should also include checking that array indices are within bounds to avoid out-of-bounds errors at runtime. How would you modify statement above to illustrate that?

Some examples of types inference rules

- How about assignments?

$$\frac{S \vdash e1 : T \quad S \vdash e2 : T}{S \vdash e1 = e2 : T}$$

Question: Is this rule good enough to approve the assignment procedures in the code shown on the right ?

```

1
2  #include <stdio.h>
3
4  int main() {
5      int a = 5;
6      int b;
7      b = a;
8
9      return 0;
10 }
```

Some examples of types inference rules

- How about assignments?

$$\frac{S \vdash e1 : T \quad S \vdash e2 : T}{S \vdash e1 = e2 : T}$$

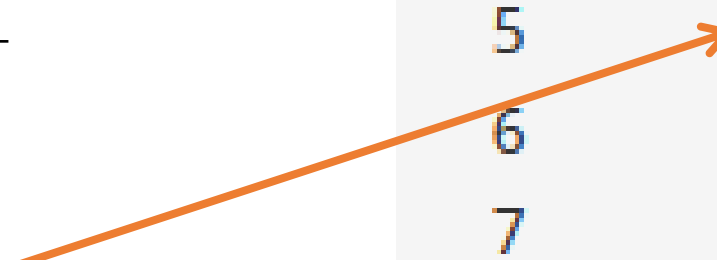
Question 2: How about this?

Is “5 = a;” an acceptable syntax?
And, therefore, something that
would have escaped the syntax
analysis phase?

```

1
2  #include <stdio.h>
3
4  int main() {
5      int a = 5;
6      int b;
7      5 = a;
8
9      return 0;
10 }
11

```



Defining custom classes

How do we factor in, any new classes the user might have defined into our inference rules?

- Let us start with the simplest case, a custom class that does not inherit from anybody.

```

1
2  #include <iostream>
3
4  class MyInt {
5  public:
6      MyInt(int n) : value(n) {}
7      int operator+(const MyInt& other) {
8          return value + other.value;
9      }
10     friend int operator+(int other, const MyInt& self);
11 private:
12     int value;
13 };

```

```

1  class MyInt():
2      def __init__(self, n):
3          self.value = n
4      def __add__(self, other):
5          print("Using the MyInt behavior")
6          return self.value + other.value
7      def __radd__(self, other):
8          print("Using the MyInt Radd behavior")
9          return self.value - other.value
10

```

Defining custom classes

Let us start with the simplest case, a custom class that does not inherit from anybody.

- In both cases, these class statements indicate to the compiler that there is a new type in town now, on top of the built-in ones!
- Operator methods basically tell the compiler to amend operations table and account for new possible + behaviour!

```

1
2  #include <iostream>
3
4  class MyInt {
5  public:
6      MyInt(int n) : value(n) {}
7      int operator+(const MyInt& other) {
8          return value + other.value;
9      }
10     friend int operator+(int other, const MyInt& self);
11 private:
12     int value;
13 };

```

```

1  class MyInt():
2      def __init__(self, n):
3          self.value = n
4      def __add__(self, other):
5          print("Using the MyInt behavior")
6          return self.value + other.value
7      def __radd__(self, other):
8          print("Using the MyInt Radd behavior")
9          return self.value - other.value
10

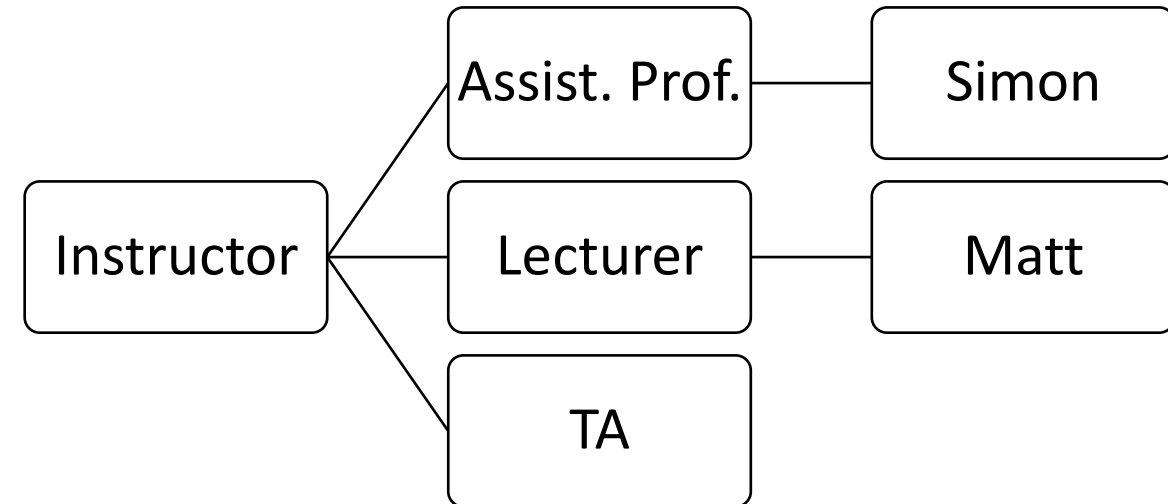
```


Op: +	bool	int	float	string	MyInt
bool	Res: bool Behavior: or	Convert int to bool Res: bool Behavior: or	Convert float to bool Res: bool Behavior: or	Convert string to bool? Res: bool Behavior: or	Illegal, trigger a type error
int	Convert int to bool Res: bool Behavior: or	Res: int Behavior: math. addition	Convert int to bool Res: float Behavior: math. addition	Illegal, should trigger a type error	Illegal, trigger a type error
float	Convert float to bool Res: bool Behavior: or	Convert int to bool Res: float Behavior: math. addition	Res: float Behavior: math. addition	Illegal, should trigger a type error	Illegal, trigger a type error
string	Convert string to bool? Res: bool Behavior: or	Illegal, trigger a type error	Illegal, trigger a type error	Res: string Behavior: concatenation	Illegal, trigger a type error
MyInt	Illegal, should trigger a type error	Illegal, trigger a type error	Illegal, trigger a type error	Illegal, trigger a type error	Res: int Behavior: in add method of MyInt class

Defining custom classes

How do we factor in, any new classes the user might have defined into our inference rules?

- How about classes that inherit from a single other class or type?
- Will have to factor in the possible inheritance conversions for types and methods.



Properties of inheritance structures

In the definitions below, “convertible” means that type A can be converted to type B, 100% of the time (no shenanigans).

- **Reflexivity:** Any type is convertible to itself.
- **Transitivity:** If A is convertible to B and B is convertible to C, then A is convertible to C.
- **Antisymmetry:** if A is convertible to B and B is convertible to A, then A and B are the same.

Properties of inheritance structures

In the definitions below, “convertible” means that type A can be converted to type B, 100% of the time (no shenanigans).

- **Reflexivity:** Any type is convertible to itself.
- **Transitivity:** If A is convertible to B and B is convertible to C, then A is convertible to C.
- **Antisymmetry:** if A is convertible to B and B is convertible to A, then A and B are the same.

In mathematics, having these three properties allows to define a **partial order operation (\leq) over elements**, in our case, types.

Properties of inheritance structures

In the definitions below, “convertible” means that type A can be converted to type B, 100% of the time (no shenanigans).

- **Reflexivity:** Any type is convertible to itself.
- **Transitivity:** If A is convertible to B and B is convertible to C, then A is convertible to C.
- **Antisymmetry:** if A is convertible to B and B is convertible to A, then A and B are the same.

In mathematics, having these three properties allows to define a **partial order operation (\leq) over elements**, in our case, types.

Using this notation, we say that $A \leq B$ if A is convertible to B.

Properties of inheritance structures

This idea and the notation \leq can be used to provide **order among types** and **define possible implicit conversion rules in operations**.

$$\frac{\begin{array}{l} S \vdash e1 : \text{int} \\ S \vdash e2 : \text{float} \\ \text{int} \leq \text{float} \end{array}}{S \vdash e1 : \text{float} \quad S \vdash e1 + e2 : \text{float}}$$

Typically added to operations tables!

Careful about operations overloading

```

1 class MyInt():
2     def __init__(self, n):
3         self.value = n
4     def __add__(self, other):
5         print("Using the MyInt behavior")
6         return self.value + other.value
7
8 class MyOtherInt(MyInt):
9     def __init__(self, n):
10        self.value = n
11
12 class MyLastInt(MyInt):
13     def __init__(self, n):
14         self.value = n
15     def __add__(self, other):
16         print("Using the MyLastInt behavior")
17         return self.value * other.value

```

```

1 x = MyInt(4)
2 y = MyInt(3)
3 w = MyOtherInt(2)
4 z = MyLastInt(5)

```

```
1 print(x + y)
```

Using the MyInt behavior
7

```
1 print(x + w)
```

Using the MyInt behavior
6

```
1 print(x + z)
```

Using the MyInt behavior
9

```
1 print(z + x)
```

Using the MyLastInt behavior
20

Some examples of types inference rules

- **Updated functions calls with implicit types conversions**

$$\begin{array}{c}
 \text{f is in identifier called in scope S} \\
 \text{f appears in scope S as a non-member function} \\
 \text{f expects types (T1, ... , Tn) and returns type U} \\
 S \vdash e_i : R_i \text{ for } 1 \leq i \leq n \\
 R_i \leq T_i \text{ for } 1 \leq i \leq n \\
 \hline
 S \vdash f(e_1, \dots, e_n) : U
 \end{array}$$

Some examples of types inference rules

- **Updated functions calls with implicit types conversions and member functions/methods of a class**

f is in identifier called in scope S

$S \vdash e_0 : M$

f appears in scope S as a member function in class M

f expects types (T_1, \dots, T_n) and returns type U

$S \vdash e_i : R_i$ for $1 \leq i \leq n$

$R_i \leq T_i$ for $1 \leq i \leq n$

$S \vdash e_0.f(e_1, \dots, e_n) : U$

Some examples of types inference rules

- **Open Question: How would you modify this to ensure that f is a method of class $e0$ or any parent class than $e0$ inherits from?**

f is in identifier called in scope S

$S \vdash e0 : M$

f appears in scope S as a member function in class M

f expects types $(T1, \dots, Tn)$ and returns type U

$S \vdash ei : Ri$ for $1 \leq i \leq n$

$Ri \leq Ti$ for $1 \leq i \leq n$

$S \vdash e0.f(e1, \dots, en) : U$

Defining custom classes

How do we factor in, any new classes the user might have defined into our inference rules?

- **Open question #1:** How about classes that inherit from a multiple other classes or types?
- Usually starts to become a mess at this point! (Which is why I am not really a big fan of multiple inheritance...)

- **Open question #2:** How about function overloading/polymorphism?

```
void display(int var1, double var2) {  
    // code  
}  
  
void display(double var) {  
    // code  
}  
  
void display(int var) {  
    // code  
}  
  
int main() {  
    int a = 5;  
    double b = 5.5;  
  
    display(a);  
    display(b);  
    display(a, b);  
  
    ...  
}
```

The diagram illustrates function calls from the `main()` function to three overloaded `display` functions. Three blue lines originate from the right side of the `main()` block and point to the opening curly brace of each `display` function: `display(a)` calls the `display(int)` version, `display(b)` calls the `display(double)` version, and `display(a, b)` calls the `display(int, double)` version.

Error reporting and recovery

Type errors will typically occur in these two scenarios:

- **An explicit method or operation is used on a type that does not support it** (e.g. in Python the explicit conversion of a string “123.4” into int, using + operation between int “7” and string “hello”, calling append() method on string, etc.).
- **There is no implicit conversion that might resolve the issue above, if any** (e.g. summing a char “c” and an int “7” in C, will simply require an implicit conversion, but this will not do in Python).

This can be implemented with an “if-else” again and a set of rules we have defined in our **types inference framework and **tables**.**

Compiler Error Recovery

Definition (**Compiler Error Recovery**):

In modern compilers, **compiler error recovery** refers to the process of attempting to continue the semantic analysis despite type errors.

This allows the compiler to potentially detect and report multiple errors in a single compilation pass, saving the programmer time by reducing the need for multiple recompilations.

Limited in terms of possibilities, however, as semantic errors will often carry to other expressions...

- **For instance:** E.g. if y uses x and x undefined, then y undefined, and if z uses y , then z undefined, etc.
- **How to bypass error, pretend code worked and continue execution?**

Quiz time!

What is the main purpose of type checking during semantic analysis?

- A. To optimize the generated machine code
- B. To ensure that variables and expressions are used correctly according to their types
- C. To find syntax errors in the code
- D. To allocate memory for variables, matching the types requirements

Quiz time!

What is the main purpose of type checking during semantic analysis?

- A. To optimize the generated machine code
- B. To ensure that variables and expressions are used correctly according to their types**
- C. To find syntax errors in the code
- D. To allocate memory for variables, matching the types requirements

Quiz time!

In the context of type systems, what does polymorphism refer to?

- A. The ability of a single function or method to work with multiple types of arguments
- B. The automatic conversion of one type to another when needed
- C. The process of inferring the types of variables and expressions without explicit type annotations
- D. The relationship between types and their subtypes

Quiz time!

In the context of type systems, what does polymorphism refer to?

- A. The ability of a single function or method to work with multiple types of arguments**
- B. The automatic conversion of one type to another when needed
- C. The process of inferring the types of variables and expressions without explicit type annotations
- D. The relationship between types and their subtypes

Quiz time!

What is the term used to describe the process by which a programming language algorithm derives the types of expressions and variables based on the context in which they are used?

- A. Type inference
- B. Type coercion
- C. Type instantiation
- D. Type resolution

Quiz time!

What is the term used to describe the process by which a programming language algorithm derives the types of expressions and variables based on the context in which they are used?

- A. Type inference**
- B. Type coercion
- C. Type instantiation
- D. Type resolution

Other tasks in semantic analysis

Beyond the scope of this class, there are many other tasks in Semantic Analysis, worth discussing:

1. **Control-Flow:** Checks that the control structures, such as loops and conditional statements, are used correctly, e.g. ensuring that every branch of an 'if-else' is reachable, and that loops have proper termination conditions, etc.
2. **Access control and visibility:** Checks that some attributes/methods of an object can be called at a given time, given their public, private, protected status.
3. **Self/This:** At any given time, who is the “self/this” keyword referring to? How do you resolve this and know which object in the scope it corresponds to?

Other tasks in semantic analysis

Beyond the scope of this class, there are many other tasks in Semantic Analysis, worth discussing:

4. **Exception handling:** Ensures that exceptions are properly caught and handled, and that appropriate error messages or fallback behaviour are in place.
5. **Garbage collection and memory management:** Ensures that you are freeing variables, that can be freed, and that you are freeing them the correct way. Somewhat related to scope?

And many other concepts!

Code is legal!

Definition (**Legal Code**):

A **legal code** is a program that adheres to all the rules of a programming language. It passed through various stages, including:

- **Lexical Analysis:** The code is tokenized into meaningful symbols, keywords, identifiers, and literals according to the language's rules. No weird symbols have been used.

Syntax Analysis: The code follows the grammar rules and structural guidelines of the language, ensuring that keywords, punctuation, and language constructs are used correctly.

Semantic Analysis: The code is meaningful and adheres to the language's conventions, including well-formedness, type checking, identifier scoping, and control flow analysis.

Code is legal!

Definition (**Legal Code**):

A **legal code** is a program that adheres to all the rules of a programming language. It passed through various stages, including:

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis

Our code is **legal** if no errors were reported during each of these three stages.

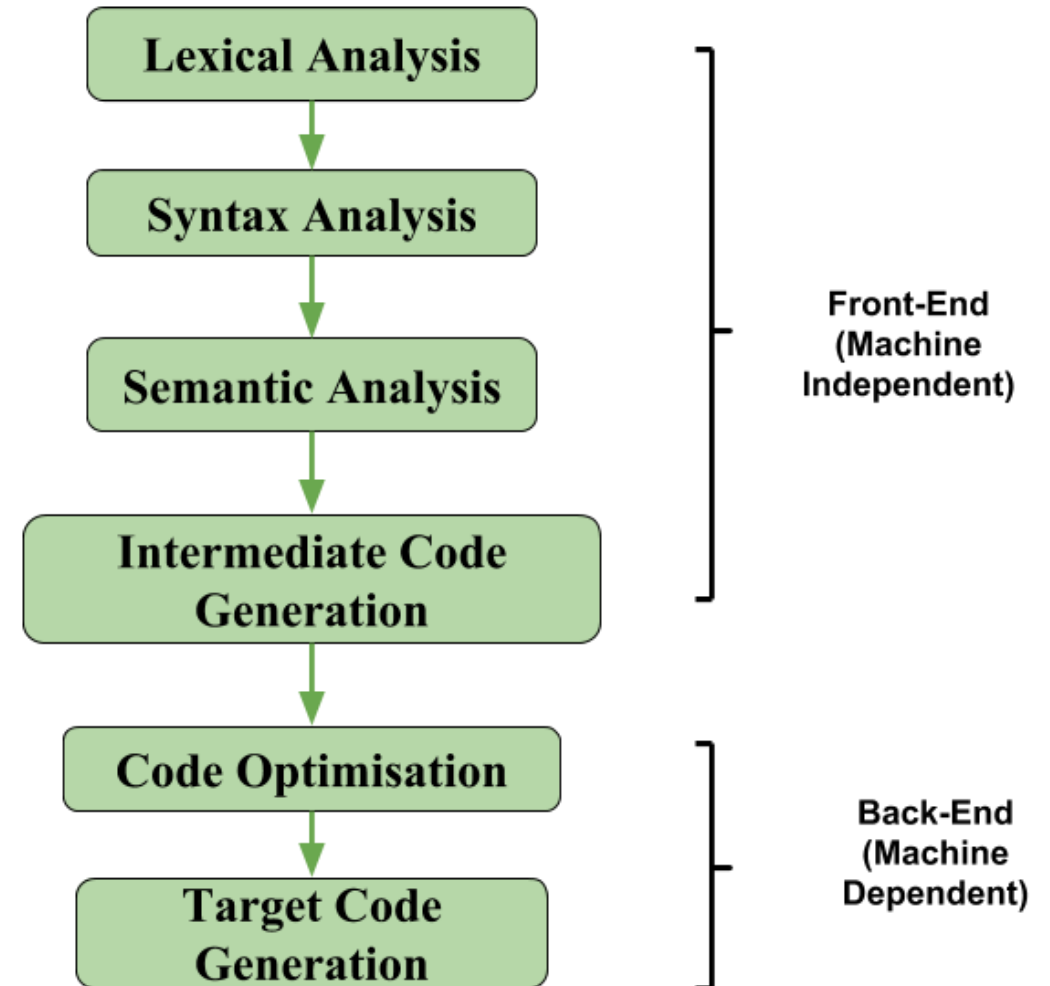
Property: A code is considered legal if it satisfies the requirements of lexical analysis, syntax analysis, semantic analysis.

A code that is deemed **legal should be able to execute (or run) correctly, after it has been transformed into machine code.**

The end of the front-end?

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.

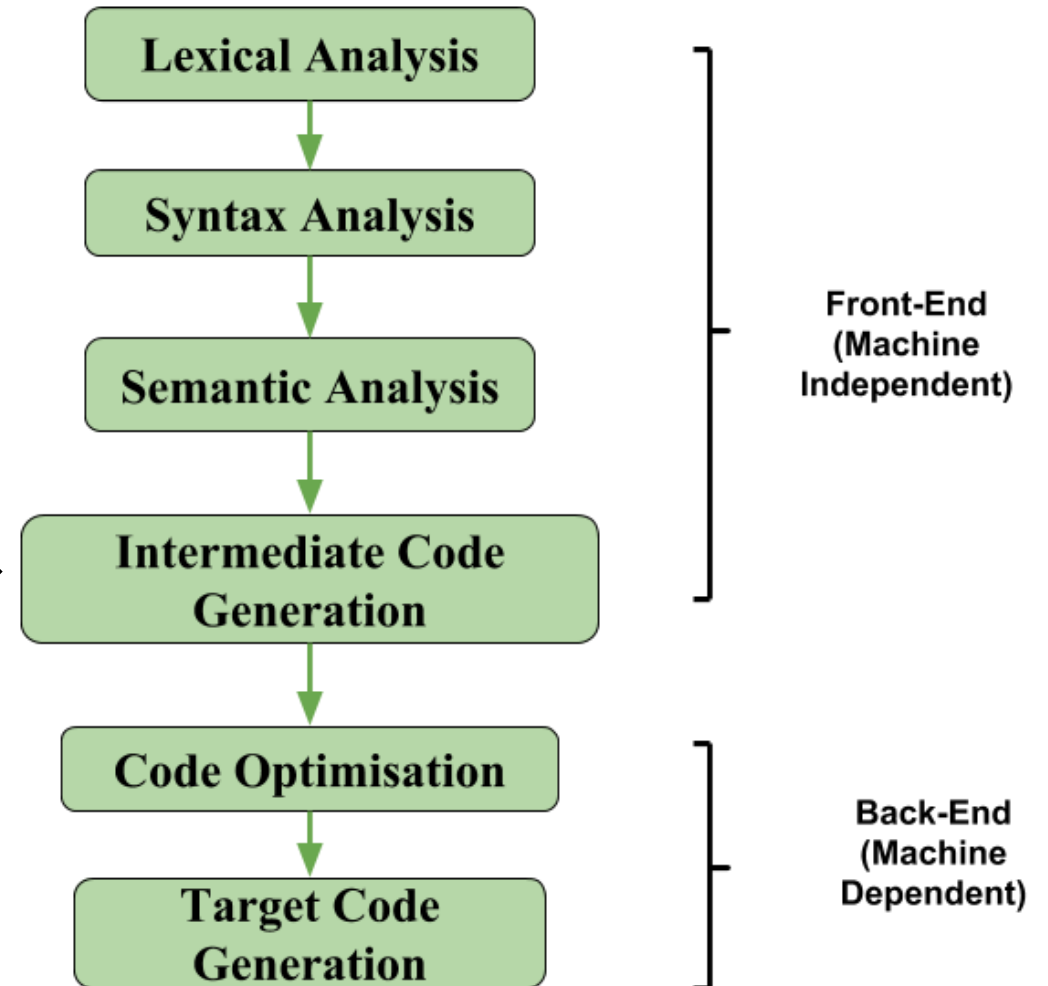
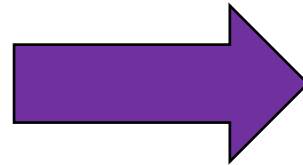


The end of the front-end?

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.

In an upcoming (or bonus) lecture, Intermediate Code Generation?

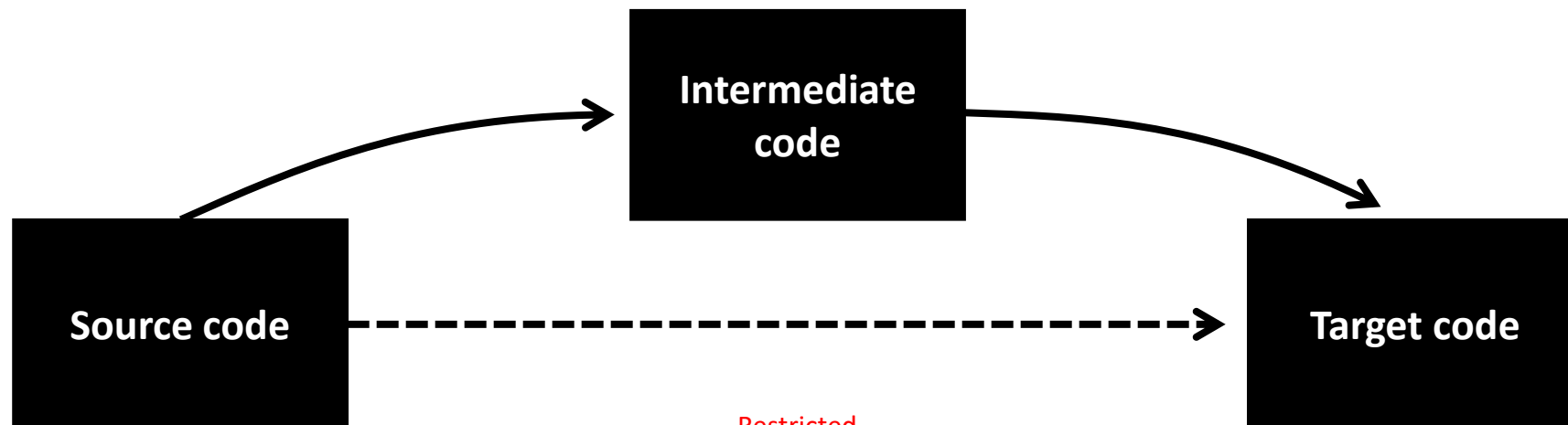


Intermediate code generation

Definition (Intermediate code generation):

Intermediate code generation is the process of transforming the source code into a code that is more abstract and closer to machine language.

Making a direct jump from source code to target code might prove difficult.



Intermediate code generation

Definition (Intermediate code generation):

Intermediate code generation is the process of transforming the source code into a code that is more abstract and closer to machine language.

Making a direct jump from source code to target code might prove difficult.

For this reason, an intermediate code representation is often easier to manipulate and allows for optimization.

The intermediate code is not specific to any particular hardware or operating system and can be easily transformed into the final machine code (**Intermediate code is roughly a high-level assembly code?**).

Three-address code representation

Definition (Three-address code representation):

Three-address code (or TAC) is a low-level intermediate code representation used by compilers to facilitate optimization and code generation.

It is called “**three-address**” because each instruction in the code **can have at most three operands**.

A typical three-address code instruction has the following format:

$$\textit{operand1} = \textit{operand2} \textit{operator} \textit{operand3}$$

Three-address code representation

For instance, the C code below can be transformed...

...into its equivalent three-address code representation.

```
#include <stdio.h>

int main() {
    int x = 10;
    int y = x + 5;
    printf("The value of y is %d\n", y);
    return 0;
}
```

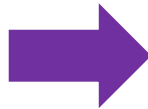
```
t1 = 10
t2 = t1 + 5
y = t2
printf("The value of y is %d\n", y)
```

From CFG to SDT to IR?!

A blast from the past (W9S2).

We managed to find a derivation for the string “26*5+7” and built an **Abstract Syntax Tree** for this derivation...

Having an **AST**, following from using SDT Rules on our Parse Tree/CFG, **brings us very close to being able to produce an Intermediate Code Representation!**

- | | | |
|----------------------------|---|----------------|
| • I1.val = 26 | | • i1 = 26 |
| • I2.val = 5 | | • i2 = 5 |
| • T1.val = I2.val | | • t1 = i2 |
| • T2.val = T1.val * I1.val | | • t2 = t1 * i1 |
| • I3.val = 7 |  | • i3 = 7 |
| • T3.val = I3.val | | • t3 = i3 |
| • E1.val = T3.val | | • e1 = t3 |
| • E.val = T2.val + E1.val | | • e = t2 + e1 |

TAC generation for SDT expressions

Our SDT expressions are quite close to the TAC expressions already.

All we need is:

- A simple mapping to translate our SDT elementary operations into their TAC equivalents (SDT: $e.val = a.val + b.val \rightarrow$ TAC: $e := a + b$).
- A function that will correctly link variables using the edges of our abstract syntax tree. And a function that will open registers $_{tn}$ to store variable values and keep a counter on n .
- We do not even need to verify that the SDT code is legal! (already guaranteed by previous steps that brought us to the SDT code!)

Could be easily implemented (but would require an implementation of an abstract syntax tree object during parsing, which we did not do!).

A quick note before we start

An important note before we start

- When generating IR at this level, you do not need to worry about optimizing it, as long as it produces the correct outcome.
- It is okay to generate IR that has lots of unnecessary assignments, redundant computations, uses many temporary variables, etc.
- In the next lecture, we will discuss how to optimize IR code (aka. The middle-end part of the compiler).
- Optimization is tricky, but interesting!

CFG for TAC

- Technically, the three-address code has a syntax and CFG of its own.

$P \rightarrow SP \mid \epsilon$

$S \rightarrow id := id \ op \ id$

$S \rightarrow id := op \ id$

$S \rightarrow id := id$

$S \rightarrow \text{push } id$

$S \rightarrow \text{if } id \text{ goto } L$

$S \rightarrow L:$

$S \rightarrow \text{jump } L$

$id \rightarrow$ (any identifier name)

$id \rightarrow$ (any constant literal)

$op \rightarrow$ (any basic operator
e.g. +, -, *, /, ==, <, ||, &&, etc.)

$L \rightarrow L0 \mid L1 \mid \dots$ (block name)

Normally, many more
production rules for S...

Basic block in TAC

Definition (**basic block** in TAC):

A **basic block** consists of a sequence of instructions in TAC, with:

- No labels (except at the first instruction),
- No jumps (except at the last instruction of the block).

Core idea behind basic blocks:

- Cannot jump in the middle of a basic block, only the beginning,
- Cannot jump out of a block, except at the end of it,
- Basic block is then a single-entry and single-exit code segment.

Basic block in TAC

A basic block always writes as:

- Start with a **block name**, by convention, in the form of L_n , with n an integer.
- Has **elementary operations** in TAC format.
- Could have **conditional structures**, using if.
- **Goto/jump** could reference current block or another block.

L0:

...

L1:

$w := x - 1$

$z := w > 0$

if z goto L1

jump L2

L2:

...

Basic block in TAC

Question: Which C code could have potentially generated this TAC?

L0:

...

L1:

w := x - 1

z := w > 0

if z goto L1

jump L2

L2:

...

Basic block in TAC

Question: Which C code could have potentially generated this TAC?

int x = 4;

...

do {

 x = x - 1

} while (x > 0)

...

(Some code L0)

(Some more code in L2)

L0:

...

L1:

w := x - 1

z := w > 0

if z goto L1

jump L2

L2:

...

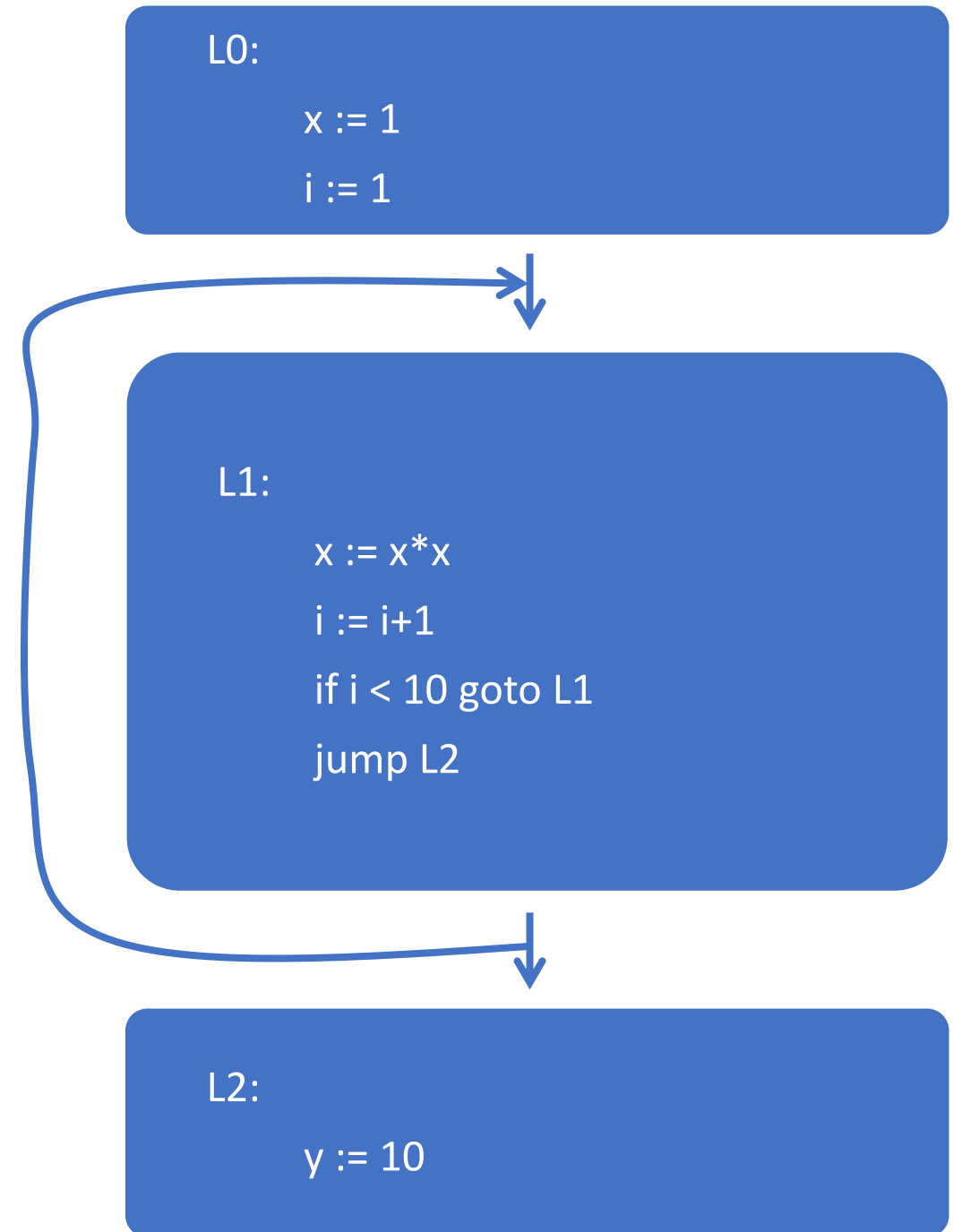
Control flow graph

Definition (**control-flow graph**):

A **control-flow graph** is a directed graph with basic blocks as nodes, and an edge from block A to block B if the execution can pass from the last instruction in A to the first instruction in B.

- E.g., the last instruction in A is jump LB and execution can fall-through from block A to block B.

Used for if/for/while/functions!



Conclusion

Finished with the front-end part of the compiler.

- Can now safely check if a code is legal or not.

The SDT/AST gives us a solid candidate for a first translation, using an intermediate language of some sort.

- Easily done with a mapping of some sort.
- Curious about how the generation of the TAC code is done using a mapping on the SDT instructions?
 - Check bonus slides attached to today's lecture (tough!).

Quiz time!

Which of the following best describes the structure of a three-address code?

- A. Three operands and one operator
- B. Two operands and two operators
- C. One operand and three operators
- D. Three operands and no operator

Quiz time!

Which of the following best describes the structure of a three-address code?

- A. Three operands and one operator**
- B. Two operands and two operators
- C. One operand and three operators
- D. Three operands and no operator

Quiz time!

In the context of three-address code, what is the purpose of the "basic block" concept?

- A. To define a sequence of code with a single entry and exit point
- B. To divide code into smaller, manageable units for optimization
- C. To handle loops and conditional statements correctly
- D. All of the above
- E. None of the above

Quiz time!

In the context of three-address code, what is the purpose of the "basic block" concept?

- A. To define a sequence of code with a single entry and exit point
- B. To divide code into smaller, manageable units for optimization
- C. To handle loops and conditional statements correctly
- D. All of the above (more on B in the next lecture!)**
- E. None of the above

Quiz time!

Which of the following is NOT a common operation that can be implemented using three-address code?

- A. Arithmetic operations
- B. Conditional jumps
- C. Loops
- D. Dynamic memory allocation

Quiz time!

Which of the following is NOT a common operation that can be implemented using three-address code?

- A. Arithmetic operations
- B. Conditional jumps
- C. Loops
- D. Dynamic memory allocation (that is something the backend will have to resolve later on during register allocation!)**