CMPSC 160
Translation of Programming Languages

Lecture 9: Context-sensitive Analysis and Type System

Recap on Attribute Grammars

- **Context-sensitive analysis**
  - non-local information
  - value beyond syntax
  - computation

- **Attribute grammar:** An attribute grammar consists of a context-free grammar augmented by **a set of rules** that specify **computations**.

- **Category of attribute grammar**
  - S-Attributed Grammars
  - L-Attributed Grammars

Rules must be “local” for Attribute Grammar

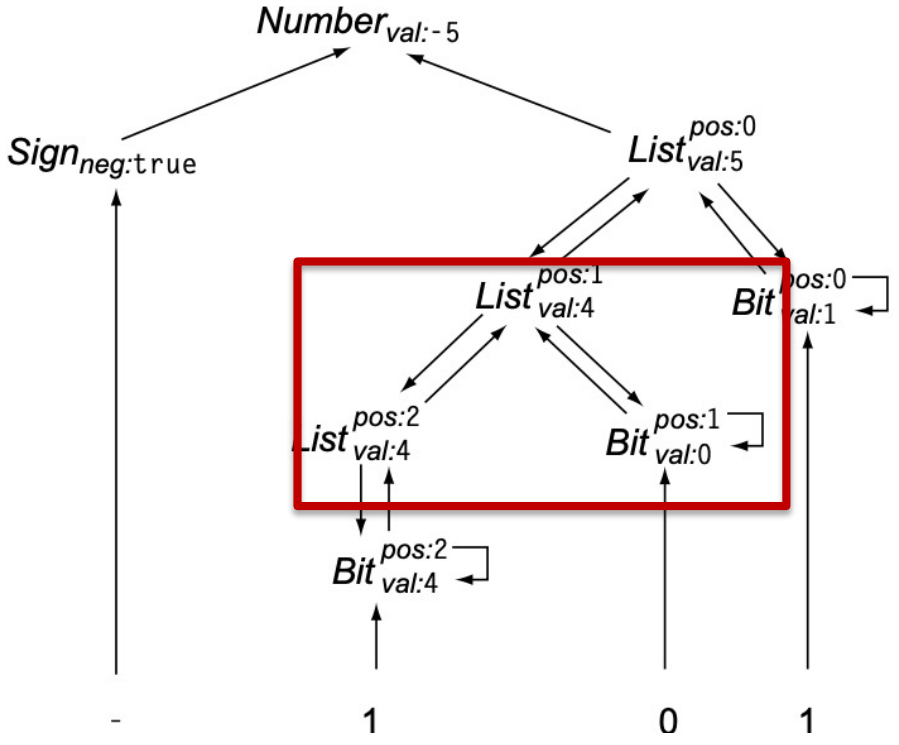
	Production	Attribution Rules
1	*Number* → *Sign List*	*List*.position ← 0 if *Sign*.negative then *Number*.value ← -*List*.value else *Number*.value ← *List*.value
2	*Sign* → +	*Sign*.negative ← false
3	*Sign* → -	*Sign*.negative ← true
4	*List* → *Bit*	*Bit*.position ← *List*.position *List*.value ← *Bit*.value
5	$List_0 \rightarrow List_1\ Bit$	$List_1$.position ← $List_0$.position + 1 *Bit*.position ← $List_0$.position $List_0$.value ← $List_1$.value + *Bit*.value
6	*Bit* → 0	*Bit*.value ← 0
7	*Bit* → 1	*Bit*.value ← $2^{Bit.position}$

- The scope of each rule is constrained by production
- Each rule only touch attributes of NT/T of the local production.

Recap on Attribute Dependence Graph

- Information flow within each subgraph.
- Far-away information can **only be transferred along the tree.**

	Production	Attribution Rules
1	*Number* → *Sign List*	*List*.position ← 0 if *Sign*.negative then *Number*.value ← -*List*.value else *Number*.value ← *List*.value
2	*Sign* → +	*Sign*.negative ← false
3	*Sign* → -	*Sign*.negative ← true
4	*List* → *Bit*	*Bit*.position ← *List*.position *List*.value ← *Bit*.value
5	$List_0 \rightarrow List_1\ Bit$	$List_1$.position ← $List_0$.position + 1 *Bit*.position ← $List_0$.position $List_0$.value ← $List_1$.value + *Bit*.value
6	*Bit* → 0	*Bit*.value ← 0
7	*Bit* → 1	*Bit*.value ← $2^{Bit.position}$



Based on the dependence graph, we can define an evaluation order to calculate all attributes.

An Attribute-Grammar Example

Another example to illustrate the **weaknesses** of attribute grammars

$Block_0$	→	$Block_1$ Assign
	|	Assign
Assign	→	Ident = Expr ;
$Expr_0$	→	$Expr_1$ + Term
	|	$Expr_1$ – Term
	|	Term
$Term_0$	→	$Term_1$ * Factor
	|	$Term_1$ / Factor
	|	Factor
Factor	→	(Expr)
	|	Number
	|	Identifier

Estimate execution time

- **Each operation has a COST**
  - **load/store**
  - **compute**
- **Assume a load per value**
- **Assume no reuse**

Grammar for a block of assignments

An Attribute-Grammar Example (continued)

$Block_0$	$\rightarrow$	$Block_1$ *Assign*	$Block_0.cost \leftarrow Block_1.cost + Assign.cost$
	|	*Assign*	$Block_0.cost \leftarrow Assign.cost$
Assign	$\rightarrow$	*Ident = Expr ;*	$Assign.cost \leftarrow COST(store) + Expr.cost$
$Expr_0$	$\rightarrow$	$Expr_1$ + *Term*	$Expr_0.cost \leftarrow Expr_1.cost + COST(add) + Term.cost$
	|	$Expr_1$ – *Term*	$Expr_0.cost \leftarrow Expr_1.cost + COST(add) + Term.cost$
	|	*Term*	$Expr_0.cost \leftarrow Term.cost$
$Term_0$	$\rightarrow$	$Term_1$ * *Factor*	$Term_0.cost \leftarrow Term_1.cost + COST(mult) + Factor.cost$
	|	$Term_1$ / *Factor*	$Term_0.cost \leftarrow Term_1.cost + COST(div) + Factor.cost$
	|	*Factor*	$Term_0.cost \leftarrow Factor.cost$
Factor	$\rightarrow$	(*Expr*)	$Factor.cost \leftarrow Expr.cost$
	|	*Number*	$Factor.cost \leftarrow COST(loadi)$
	|	*Identifier*	$Factor.cost \leftarrow COST(load)$

All the attributes are synthesized !
Good fit to bottom-up, shift/reduce parser

What about an improvement?

Values are **loaded only once** per block (**not at each use**)

- Need to track which values have been already loaded

Adding load tracking

- Need new attributes: Two sets: *Before* and *After* for each production

A Better Execution Model

Factor → (Expr)	Factor.cost ← Expr.cost ; Expr.Before ← Factor.Before ; Factor.After ← Expr.After
| Number	Factor.cost ← COST(loadi) ; Factor.After ← Factor.Before

| Identifier	If (Identifier.name ∉ **Factor.Before**)
	then
	Factor.cost ← COST(load);
	Factor.After ← Factor.Before ∪ Identifier.name
	else
	Factor.cost ← 0
	Factor.After ← Factor.Before

Factor → Identifier is the only rule that is associated with loading.

A Better Execution Model

$Block_0 \rightarrow Block_1\ Assign$	$Block_0.cost \leftarrow Block_1.cost + Assign.cost$ $Block_1.before \leftarrow Block_0.before$ $Assign.before \leftarrow Block_1.after$ $Block_0.after \leftarrow Assign.after$
$\mid\ Assign$	$Block_0.cost \leftarrow Assign.cost$ $Assign.before \leftarrow Block_0.before$ $Block_0.after \leftarrow Assign.after$
$Assign \rightarrow Ident = Expr\ ;$	$Assign.cost \leftarrow \text{COST}(store) + Expr.cost$ $Expr.before \leftarrow Assgin.before$ $Assign.after \leftarrow Expr.after$
.....	.....

- Sets *Before* and *After* for each production
  - Must be **updated**, and **passed** around the tree

A Better Execution Model

$Expr_0 \rightarrow Expr_1 + Term$ ….	$Expr_0.cost \leftarrow Expr_1.cost + COST(add) + Term.cost$; $Expr_1.Before \leftarrow Expr_0.Before$; $Term.Before \leftarrow Expr_1.After$; $Expr_0.After \leftarrow Term.After$ …..
$Term0 \rightarrow Term1 * Factor$ ……	$Term0.cost \leftarrow Term1.cost + COST(mult) + Factor.cost$ $Term_1.Before \leftarrow Term_0.Before$; $Factor.Before \leftarrow Term_1.After$; $Term_0.After \leftarrow Factor.After$ ……

- Sets *Before* and *After* for each production
  - Must be **updated**, and **passed** around the tree

Huge Overhead

- These copy rules incurs large overhead
- Each creates an instance of the set, but many of them are very similar to each other.
- Lots of work, lots of space, lots of rules to write

Problems with the Attribute-Grammar Approach

- **Non-local computation**
  - An attribution rule can only connect attribute values associated with a grammar symbol that appears in the same production rule.
  - Thus, it only allows nearby, or local, information communications.
  - Copy rules are needed to move those “non-local” values to the points where they are used.
  - But copy rules increase computation complexity and space requirements
- Non-local computation are the key to many context-sensitive analysis
  - e.g., to pass declaration information around the parse tree
  - This problem reflects a **fundamental clash** between the functional nature of the attribute-grammar paradigm and the imperative use to which it might be put in the compiler.

Addressing the Problem

Use rules with **side effects**, store the results in global variables

- This is called ***ad-hoc syntax directed translation***
  - This is the approach we use when we write semantic actions in Yacc (Bison is the GNU implementation/extension of Yacc).
- Avoids **all the copy rules**, allocation and storage headaches
- All **inter-assignment attribute flow** is through table
  - Clean, efficient implementation
  - Good techniques for implementing the (symbol) table
  - When its done, information is in the table!
  - Cures most of the problems
- This design violates the functional paradigm

Reworking the Example (with load tracking) Using Ad-Hoc Syntax Directed Translation

$Block_0$	→	$Block_1$ Assign	
	|	Assign	cost ← 0;
Assign	→	Ident = Expr ;	cost ← cost + COST(store);
$Expr_0$	→	$Expr_1$ + Term	cost ← cost + COST(add);
	|	$Expr_1$ – Term	cost ← cost + COST(sub);
	|	Term	
$Term_0$	→	$Term_1$ * Factor	cost ← cost + COST(mult);
	|	$Term_1$ / Factor	cost ← cost + COST(div);
	|	Factor	
Factor	→	(Expr)	
	|	Number	cost ← cost + COST(loadI);
	|	Identifier	i ← hash(Identifier);
			if (Table[i].loaded = false)
			then
			cost ← cost + COST(load);
			Table[I].loaded ← true;

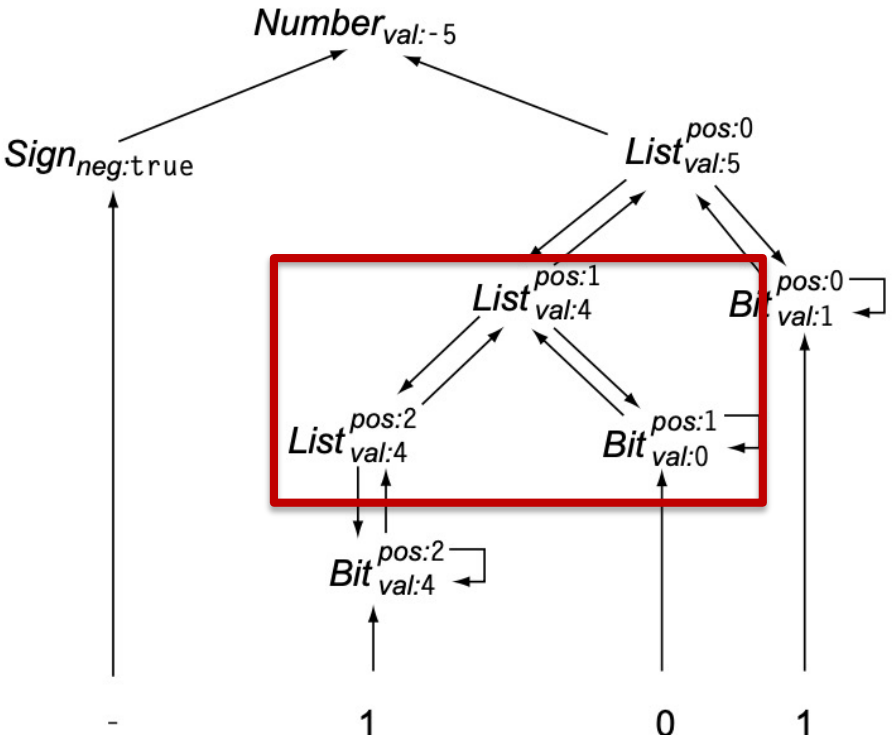
- For the previous example, use a **central repository for attributes**: symbol table to indicate information about loaded/not loaded state
- The compiler can accumulate **cost** in a single variable, rather than creating a cost attribute at each node in the parse tree.

Actions for each production in ad-hoc method are applied as a unit rule. The entire evaluation order has to be specified by the programmer.

Actions in ad-hoc method are applied as a unit; not true for attribute grammar rules

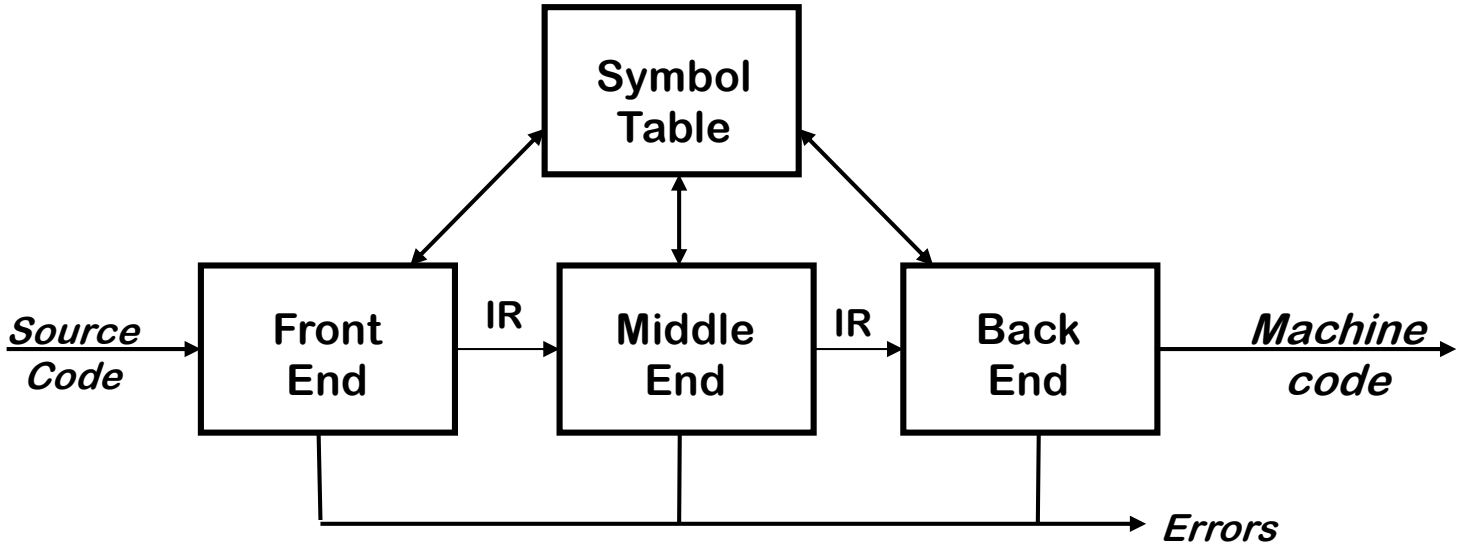
- Based on the dependence graph, we can define an evaluation order to calculate different attributes.
- For attribute rule 5, the position part is **evaluated much earlier** than the value part.

	Production	Attribution Rules
1	*Number* → *Sign List*	*List*.position ← 0 if *Sign*.negative then *Number*.value ← -*List*.value else *Number*.value ← *List*.value
2	*Sign* → +	*Sign*.negative ← false
3	*Sign* → -	*Sign*.negative ← true
4	*List* → *Bit*	*Bit*.position ← *List*.position *List*.value ← *Bit*.value
5	$List_0 \rightarrow List_1\ Bit$	$List_1$.position ← $List_0$.position + 1 *Bit*.position ← $List_0$.position $List_0$.value ← $List_1$.value + *Bit*.value
6	*Bit* → 0	*Bit*.value ← 0
7	*Bit* → 1	*Bit*.value ← $2^{Bit.position}$



Symbol Tables in a Compiler

- **Symbol Table** is an important data structure created and maintained by the compiler in order to keep track of **semantics** of variables.



- It is initially built during the front-end processing (often integrated into lexical and syntax analysis).
- Its information is used/refined during later passes: semantics analysis in the front end, compiler optimizations in the middle end, and code synthesis in the back end.
- We can rediscover this information every time we need it by traversing the Irs (e.g., AST), but using a symbol table is more efficient.

Symbol Tables: Contents and Operations

- **Key information stored in Symbol Tables**
  - textual name
  - data type
  - for arrays: dimension, upper and lower bounds for each dimension
  - for records or structures: list of fields and their information
  - for functions and procedures: number and type of arguments
  - storage information (memory location: base address and offset)
  - …
- **Key operators**
  - allocate(): to allocate a new empty symbol table
  - free(): to remove all entries and free the storage of the symbol table
  - insert(): frequently used to add information about unique names occurring in the source code.
  - lookup(): frequently used to search a name if the symbol is initialized + if the symbol declared multiple times.

Symbol Tables: Implementations

- **Array (Sorted) :**
  - **Insertion Time** O(n). When inserting an element, traversing must be done in order to shift elements to right.
  - **Lookup Time** O(log n) time. A binary search can be used to find an element.
- **Linked List (unsorted):**
  - **Insertion Time** O(1). If we have to check that an entry was not entered before, then insert will be O(*N*).
  - **Lookup Time –** O(n). While fetching a data item the linked list must be traversed completely (linear search).
- **Hash**
  - **Insertion Time** O(1).
  - **Lookup Time** O(1)
  - potential issue: the disadvantage of this implementation is when there are too many collisions the time complexity increases to O(n).
  - *Open hashing* and *Open addressing*

Problems with the Ad-hoc ***syntax directed translation*** Approach

- An attribute rule can record information directly into a global table, where other rules can read the information.
  - the implicit dependence between them is removed from the attribute dependence graph.
- The missing dependence should constrain the evaluator to ensure that the two rules are processed in the correct order.
- Compiler writer have to deal with this evaluation order manually.
  - The common routine is to integrate arbitrary snippets of code into the parser and lets the parser sequence the actions and pass values between them.

Ad-hoc methods vs. Attributed Grammar

Most parsers are based on this *ad-hoc* style of context-sensitive analysis using ad-hoc syntax directed translation

Advantages

- Addresses the shortcomings of the attribute grammars
- Efficient, flexible

Disadvantages

- Must write the code with little assistance
- Programmer deals directly with the details

Real Case Study for Semantics Analysis

To generate code, a compiler needs to answer many questions:

- **Type analysis**
  - is x a scalar, an array or a function?
  - is the expression x*y+z type-consistent?
  - in an array reference a[i , j , k], does a have three dimensions?
  - how many arguments does a function take?

    . . .
- **Name analysis**
  - is x declared? Are there names declared but not used?
  - which declaration of x does each use reference?

Type Analysis

Type Analysis

- Type: Definition and Benefits.
- Two notions of typing for programming languages are to be distinguished:
  - static vs. dynamic
  - strong vs. weak
- **Static** (compilation time) Type Checking
  - Type equivalence
  - Type Inference for expressions
  - **Ad-hoc Syntax-directed Translation on a simple language**

Type

- **Type**: for each data value in the program, there are **a collection of properties** associated it, known as the value's *type.*

- With each **type t** we associate **values** and **operators** that we can apply to values of type t.
  - type of int also gives ranges of values $-2^{31} \le i < 2^{31}$
  - To values of type string, we can apply operations such as *println*, but we cannot multiply two string.
- Conversely, with each **operator** with associate types that describe the nature of the operator's arguments and result.

+	integer	real	double	complex
integer	integer	real	double	complex
real	real	real	double	complex
double	double	double	double	*illegal*
complex	complex	complex	*illegal*	complex

Result Types for Addition in FORTRAN77

Benefits: types as two-version programming

- In languages such as Java, programs are annotated with types.
- This can be seen as a weak form of two-version programming: the programmer specifies twice what the program should do, once by the actual code, and a second time through the types.
- By saying something twice, but in somewhat different languages (Java computation vs types) the probability that we make the same mistake in both expressions is lower than if we state our intention only once.
- The key idea behind semantic analysis is to look for **contradictions between the two specifications** and reject programs with such contradictions.

Benefits: type for ***expressiveness***

- An operator that has different meanings based on the types of its arguments is "overloaded."
- Still, for the previous example. Fortran has a single addition operator, +, and uses type information to determine how it should be implemented.

Type of			Code
a	b	a+b	
integer	integer	integer	iADD r_a, r_b ⇒ r_{a+b}
integer	real	real	i2f f_a ⇒ r_{a_f} fADD r_{a_f}, r_b ⇒ r_{a_f+b}
integer	double	double	i2d r_a ⇒ r_{a_d} dADD r_{a_d}, r_b ⇒ r_{a_d+b}
real	real	real	fADD r_a, r_b ⇒ r_{a+b}
real	double	double	r2d r_a ⇒ r_{a_d} dADD r_{a_d}, r_b ⇒ r_{a_d+b}
double	double	double	dADD r_a, r_b ⇒ r_{a+b}

Type is not everything

- Note that types can only prevent stupid mistakes like
  "hello" * "world".

- They cannot (usually) prevent more complicated problems, like out-of-bounds indexing of arrays.

```
int [] a = new int [ 10 ]
a [ 20 ] = 3
```

Type-checking

- An important distinction is that between type-checking (old-fashioned) and type-inference (modern).
- In type-checking (e.g., Java), we verify that the programmer-written type-annotations are consistent with the program code.
  - E.g.,

```
def f ( x : String ) : Int = {
    if ( x = "Moon" ) true
   else false
 }
```

  is easy to see as inconsistent.

Type Inference

- Types are determined from the context of the reference, rather than just by assignment statement
- The compiler can trace how values flow through variables and function arguments
- Any remaining ambiguity is treated as an error the programmer must fix by adding explicit declarations

Type inference

```
def f ( y : ??? ) : ??? = {
    if ( x = y ) y
    else x+1
}
```

What types could you give to x, y and the return value of f?

Clearly x has type integer, y has type integer.

Another Example

That was easy. What about this program

```
def f ( x : ??? ) : ??? = {
    while ( x.g ( y ) ) { y = y+1 };
     if ( y > z ) z = z+1
     else println ( “hello” ) ;
}
```

What types could you give to x, y, z, g and f?

y and z are integers, x must be a class A such that A has a method g which takes and integer and returns a boolean.

Finally, f returns nothing, so should be of type void

Contrasts in Type Systems

Type systems are often described by their design decisions along several dimensions.

- Static vs. dynamic types
  - time of the type binding
- Strong vs. Weak typing
  - Explicit vs. implicit type conversion

Type Binding

Type Binding is an association between a name and a type attribute

Type Binding time is the time at which a binding takes place.

- Language design time, e.g., bind operator symbols to operations
- Language implementation time, e.g., bind floating point type to a representation
- Compile time, e.g., bind a variable to a type in C or Java
- Link time
- Load time, e.g., references to dlls in C/C++
- Runtime, e.g., dynamic type bindings

Static Type Binding

- In a static type system, types are fixed before the program is run (e.g., compile time)
- Compatibility checking can be done by a compiler and errors flagged
- Some claim that most program errors are type errors

- The advantage is that the resulting code need not check for type mismatches at run time, which speeds up execution

- It typically requires adding type declarations (a pain) but these can also be seen as a kind of documentation (a benefit), note that this is only for explicit declarations

Dynamic Type Binding

- A variable's type can change as the program runs
- might be re-bound on every assignment.
- Used in scripting languages (Javascript, PHP, Python) and some older languages (Lisp, Basic, Prolog, APL)

- Here's a javascript example

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

Dynamic Type Binding

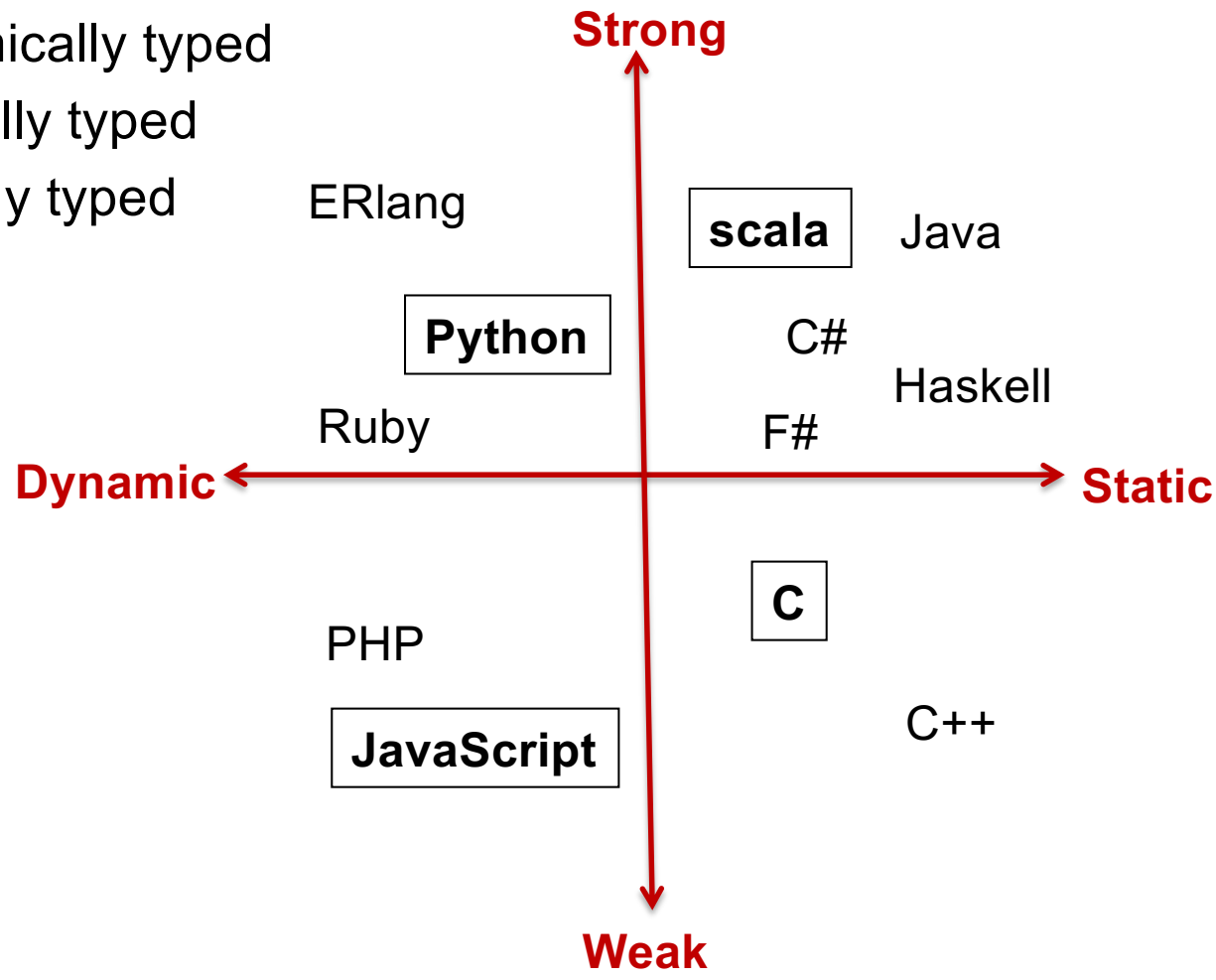
- Flexibility for the programmer
- Obviates the need for “polymorphic” types
- Development of generic functions (e.g., sort)
- But there are disadvantages as well
  - Types have to be constantly checked at run time
  - A compiler can’t detect errors via type mis-matches
  - Mostly used by scripting languages today

Strong/Weak Typing

- Definition: Strong/weak typing
  - Strong/weak typing is about how strictly types are distinguished (e.g., implicit conversion).
  - If a language specification requires its typing rules strongly (i.e., more or less allowing only those automatic type conversions that do not lose information), one can refer to the process as S*trongly* typed, if not, as We*akly* typed.
  - **Weakly-typed** languages make conversions between unrelated types *implicitly.*
  - ***Strongly-typed*** languages don't allow implicit conversions between unrelated types.

Languages

1. Weakly and dynamically typed
2. Strongly and dynamically typed
3. Strongly and statically typed
4. Weakly and statically typed



Weakly and Dynamically Typed Language

A small **JavaScript** program:

```
function add(a, b) {
    result = a + b;
    return result;
}

x = "10"
y = 10

result = add(x, y);

console.log(result);
```

- Any Error?
  - No Error
- What is the result?
  - 1010
- What happened？
  - 10 is implicitly converted to a string “10”, and then concatenated with the other string together.

Strongly and Dynamically Typed Language

A small **Python** program:

```python
def add(a, b):
    result = a + b
    return result

x = "10"
y = 10

result = add(x, y)
print(result)
```

- Any Error?
- What is the result?
- What happened ?

```
Traceback (most recent call last):
  File "add.py", line 8, in <module>
    result = add(x, y)
  File "add.py", line 2, in add
    result = a + b
TypeError: can only concatenate str (not "int") to str
```

Note: all these discussion are based on common features of these languages. For example, we could also use type annotations in Python and enable checking statically the types.

Strongly and Statically Typed Language

A small **Scala** program:

```
object Add {
  def main(args: Array[String]): Unit = {
    val x = "5"
    val y = 5

    val result = add(x, y)
    println(result)
  }

  def add(a: Int, b: Int) :Int = {
    val result = a + b
    result
  }
}
```

This makes the code feel dynamically typed.
But it is not.
The key is "**type inference**".

- Any Error?
- What is the result?
- What happened ?

```
add.scala: error: type mismatch;
found    : String
required: Int
    val result = add_(x, y)
                      ^
one error found
```

Weakly and Statically Typed Language

A small **C** program:

```c
#include <stdio.h>

int add(int a, int b) {
    int result = a + b;
    return result;
}

int main() {
    char x = '10';
    int y = 10;

    int result = add(x, y);
    printf("%d\n", result);
    return 0;
}
```

Any Error?

- No Error

What is the result?

- 58

What happened ?

```
add.c:9:14: warning: multi-character
 character constant [-Wmulticharch]
    char x = '10';
             ^
add.c:9:14: warning: implicit conver
sion from 'int' to 'char' changes va
lue from 12592 to 48 [-Wconstant-con
version]
    char x = '10';
          ~    ^~~~
2 warnings generated.
```

Untyped Language?

- Assembly language for instance is said to be ***untyped*** since there is no **type checking**.
- The absence of type checking allows a lot of freedom, necessary for strong optimizations.
- In general, you don‘t write Assembly code except for very specific projects.
- We write in a higher-level language and the compiler produces Assembly code for you).