UNIVERSITY OF TWENTE.

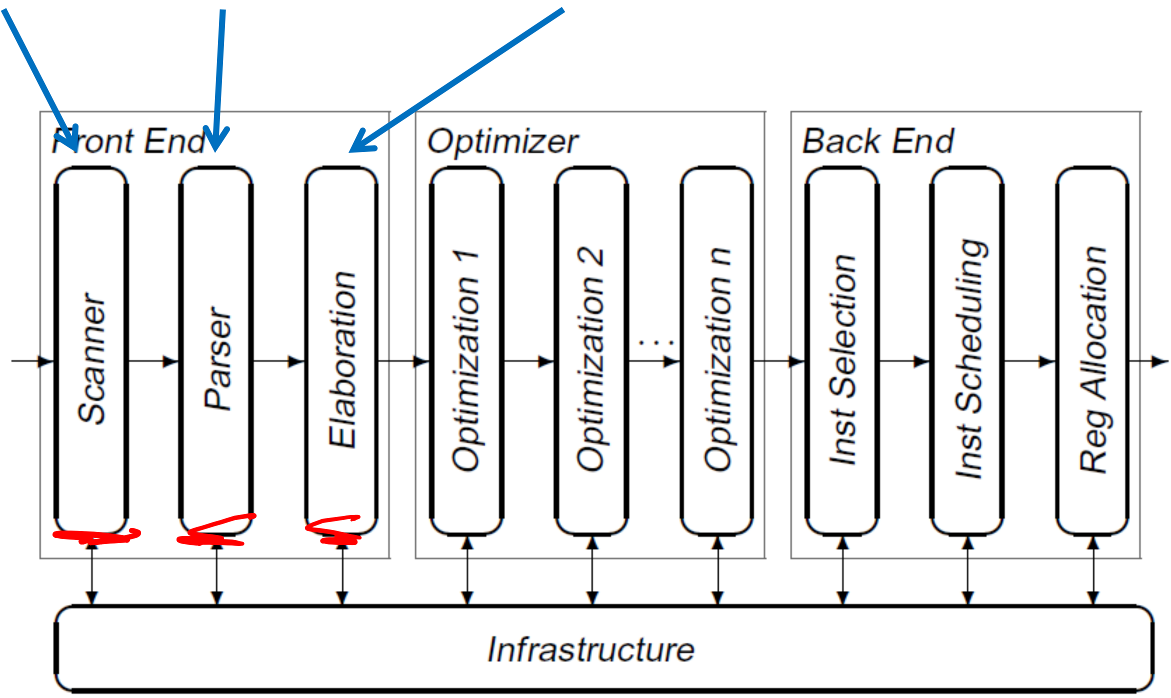
COMPILER CONSTRUCTION:

CC 3-1: TYPE SYSTEMS

MODULE 8: PROGRAMMING PARADIGMS

6 MAY 2019

WHERE ARE WE?



TYPING IN PROGRAMMING LANGUAGES

- What programming languages have you worked with?

- as part of your studies -

statically typed?	typed?	
(✓)	✓	Java
✗	✓	Python
✗	✓	JavaScript
✓	✓	C
✓	✓	Haskell
n/a	✗	SPARC Assembler

- ~~otherwise~~ -

```
B b = getSomething()
A a = (A) b;
```

ok if B derives from A

may be okay or not at runtime if A derives from B

TYPING IN PROGRAMMING LANGUAGES

- Typed versus untyped languages
  - Actually, there's many degrees of typedness
  - Strength of the type system: can you subvert it?
  - What examples do you know?
- For typed languages: Compile-time versus run-time checking
  - A.k.a. dynamic versus static ~~typing~~ *checking*
  - Again, there's degrees
  - Advantages/disadvantages?
  - Examples of run-time type checking in Java?
- For compile-time checking: explicit versus implicit typing
  - Implicit typing known as "type inference"
  - What examples do you know?

See also: http://en.wikipedia.org/wiki/Programming_language

a + b   a & b   a * b   *int

int long   int long   *long   *double

PURPOSES OF TYPING

- Ensuring run-time safety
  - Many errors prevented
- Improving expressiveness
  - Operator overloading
  - What operators are overloaded in Java?

PURPOSES OF TYPING

- Ensuring run-time safety
  - Many errors prevented
- Improving expressiveness
  - Operator overloading
  - What operators are overloaded in Java?
- Generating better code
  - Appropriate instructions known at compile time (with static typing)

From FORTRAN:

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

vs.

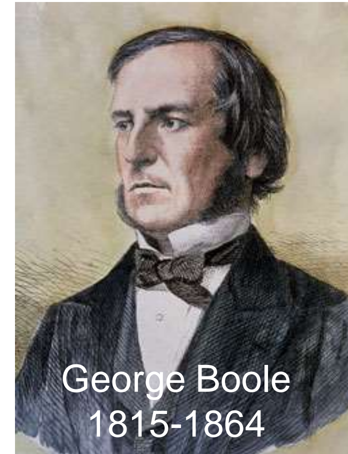
```
// partial code for "a+b ⇒ c"
if (tag(a) = integer) then
    if (tag(b) = integer) then
        value(c) = value(a) + value(b);
        tag(c) = integer;
    else if (tag(b) = real) then
        temp = ConvertToReal(a);
        value(c) = temp + value(b);
        tag(c) = real;
    else if (tag(b) = ...) then
        // handle all other types ...
    else
        signal runtime type fault
else if (tag(a) = real) then
    if (tag(b) = integer) then
        temp = ConvertToReal(b);
        value(c) = value(a) + temp;
        tag(c) = real;
    else if (tag(b) = real) then
        value(c) = value(a) + value(b);
        tag(c) = real;
    else if (tag(b) = ...) then
        // handle all other types ...
    else
        signal runtime type fault
else if (tag(a) = ...) then
    // handle all other types ...
else
    signal illegal tag value;
```

TYPES OF TYPES

- Base ("primitive") types
  - Numbers (integers, "reals", …)
  - Booleans
  - Characters
- Compound/constructed types
  - Array types
  - Strings (sometimes treated as primitive)
  - (Enumerated types)
  - Structures and variants (unions)
  - Pointers
  - Function types
- Polymorphic types
  - Parameterised: Only a "proper" type when instantiated

struct B { int a; double* b };

ArrayList<___>
type



George Boole
1815-1864

RELATIONS BETWEEN TYPES (1)

int int(0..10)

S+7 S+7

- Type equivalence
  - By name
  - By structure
- Subtyping
  - Criterion for subtyping?
    - Values of subtype ⊆ values of supertype
    - Subtype behaves as supertype *in every context*
  - Can be inferred (structure-based) or declared (name-based)
  - Inheritance ≠ subtyping!
    - Subclass may behave (quite) differently from superclass
    - Consider: Circle subtype of Ellipse, or vice versa, or neither?

certainly not!

depending on methods: setHeight behaves differently

- Type casting
  - Actual (dynamic) type may be subtype of known (static) type
  - Static type “changed” by casting
  - Up-casting (implicit) or down-casting (explicit)

```
struct Tree {
  struct Tree *left;
  struct Tree *right;
  int value
}
```

```
struct STree {
  struct STree *left;
  struct STree *right;
  int value
}
```

Equivalent?
By name: no, by structure: yes

```
String s = "hi";
Object o = s;
String t = (String) o;
```

RELATIONS BETWEEN TYPES (2)

- Type coercion

```
int a = 1;
double b = a;
int c = (int) b;
```

  - Forcibly changes (values of) one type to (values of) another
    - Widening (often implicit) or narrowing (typically explicit)
  - Mostly for base types
    - However, see Java boxing/unboxing

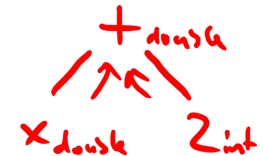
```
int a = 1;
Integer b = a;
int c = b;
```

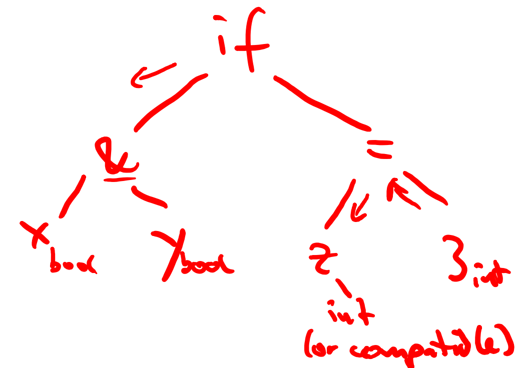
- Coerced types are not equivalent or subtypes
  - Coercion ≠ casting!
  - Even though notation in Java coincides

TYPE CHECKING AND INFERENCE

- In parse tree, every node is (an instance of) a grammar symbol
  - Symbol is terminal (= token type) or nonterminal
- Some symbols are typed
  - Namely, those corresponding to expressions
- How can the type of the parse tree node be established?
  1. *Synthesized* from symbol & children in parse tree
     - E.g. type of x+2 in Java?

       int if x is **int**, String if x is String, invalid if x is Object
  2. *Inherited* from siblings & parent in parse tree
     - E.g., type of x, y, z in **if** (x & y) { z = 3; }?





TYPE CHECKING AND INFERENCE

- In parse tree, every node is (an instance of) a grammar symbol
  - Symbol is terminal (= token type) or nonterminal
- Some symbols are typed
  - Namely, those corresponding to expressions
- How can the type of the parse tree node be established?
  1. *Synthesized* from symbol & children in parse tree
     - E.g. type of x+2 in Java?

       int if x is **int**, String if x is String, invalid if x is Object
  2. *Inherited* from siblings & parent in parse tree
     - E.g., type of x, y, z in **if** (x & y) { z = 3; }?

       x, y are **boolean** because x & y is **boolean** (which is what **if** expects),
       z is **int** because z = 3 is **int** (which is because 3 is **int**, synthesized from 3)
     - Especially important in implicitly typed languages
- Type errors:
  - Type synthesis: error if subnodes of symbol have “wrong” type
  - Type inheritance: error if inherited type differs from actual (synthesized) type

TYPING RULES

- *Not* part of the actual grammar
  - Context-free grammars not powerful enough
  - Could be done using context-sensitive grammars, but not practical
- Rules added on top of grammar
  - Attribute rules → “attributed grammar”
  - Kinds of rules: Synthesized versus inherited
  - Not just used for typing
- In Antlr, *tree listeners* (or *visitors*) are used to implement such rules