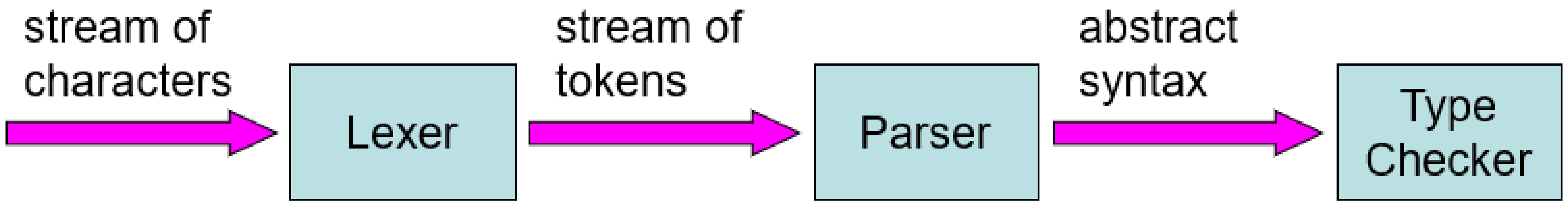
Programming Language Concepts

Lecture 6

The Front End

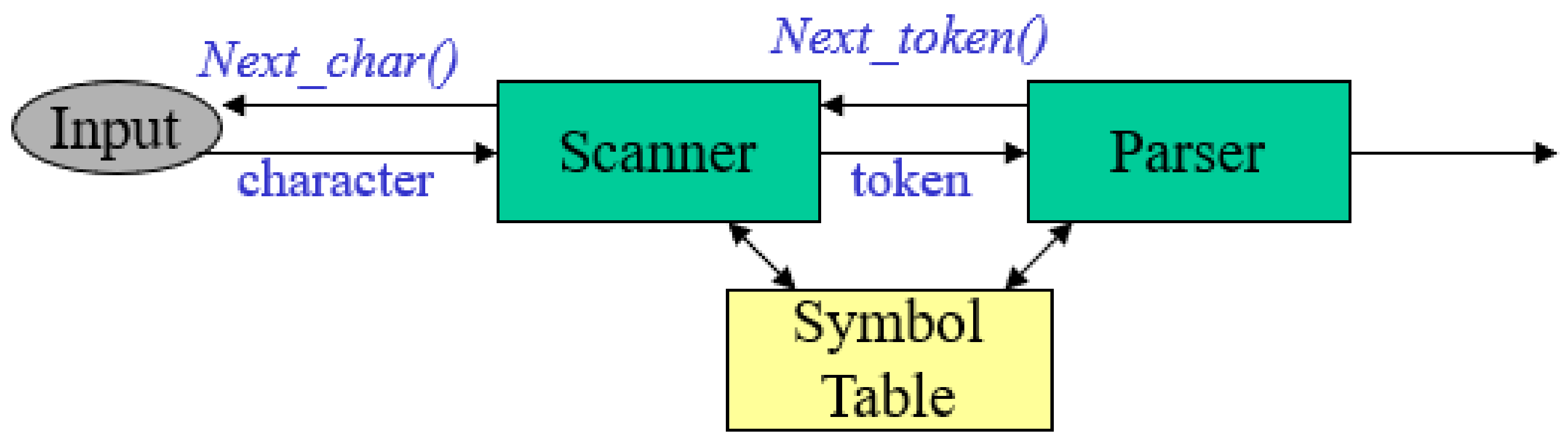


- Lexical Analysis: Create sequence of tokens from characters
- Syntax Analysis: Create abstract syntax tree from sequence of tokens
- Type Checking: Check program for well-formedness constraints

Lexical Analysis

INPUT: sequence of characters

OUTPUT: sequence of tokens



A lexical analyzer is generally a subroutine of parser:

- Simpler design
- Efficient
- Portable

Definitions

- **token** – set of strings defining an atomic element with a defined meaning
- **pattern** – a rule describing a set of string
- **lexeme** – a sequence of characters that match some pattern

Examples

Token	**Pattern**	**Sample Lexeme**
while	while	while
relation_op	= | != | < | >	<
integer	(0-9)*	42
string	Characters between “ “	“hello”

Input string: size := r * 32 + c

<token,lexeme> pairs:

- <id, size>
- <assign, :=>
- <id, r>
- <arith_symbol, *>
- <integer, 32>
- <arith_symbol, +>
- <id, c>

Lexical Analysis

- Lexical Analysis: Breaks stream of ASCII characters (source) into tokens
- Token: An atomic unit of program syntax
  - i.e., a word as opposed to a sentence
- Tokens and their types:

Characters Recognized:	**Type:**	**Token:**
foo, x, listcount	ID	ID(foo), ID(x), ...
10.45, 3.14, -2.1	REAL	REAL(10.45), REAL(3.14), ...
;	SEMI	SEMI
(	LPAREN	LPAREN
50, 100	NUM	NUM(50), NUM(100)
if	IF	IF

Lexical Analysis Example

```
x   =   (   y   +   4.0   )   ;
```

$\Big\downarrow$ Lexical Analysis

```
ID(x)  ASSIGN  LPAREN  ID(y)  PLUS  REAL(4.0)  RPAREN  SEMI
```

Exercise:

```
int main()
{
int a = 10, b = 20;
printf("sum is:%d",a+b);
return 0;
}
```

Write down the valid tokens for the program.

HOW DO WE KNOW WHETHER CODE IS WELL-TYPED?

- Most typed languages have some sort of annotation to indicate the type of data. E.g.
  - C asks variables to be declared with a (weak) type so that the compiler can allocate enough memory to store values of that type.
  - Java asks for methods to declare the type of their input parameters as well as their return value
- The way that the compiler makes use of these annotations determines whether the type system is strong/weak.
- We'll talk about strong (static) type systems for the remainder of this lecture.
- The compiler needs to check that, at any point in a program's execution, the program doesn't attempt to consume data of the wrong type.
- Certain operators of each programming language consume data. So the compiler needs to find uses of these operators.
- Clearly then, in order to check well-typedness of programs, this will amount to checks on abstract syntax trees according to the types of the data being used.

A TOY LANGUAGE

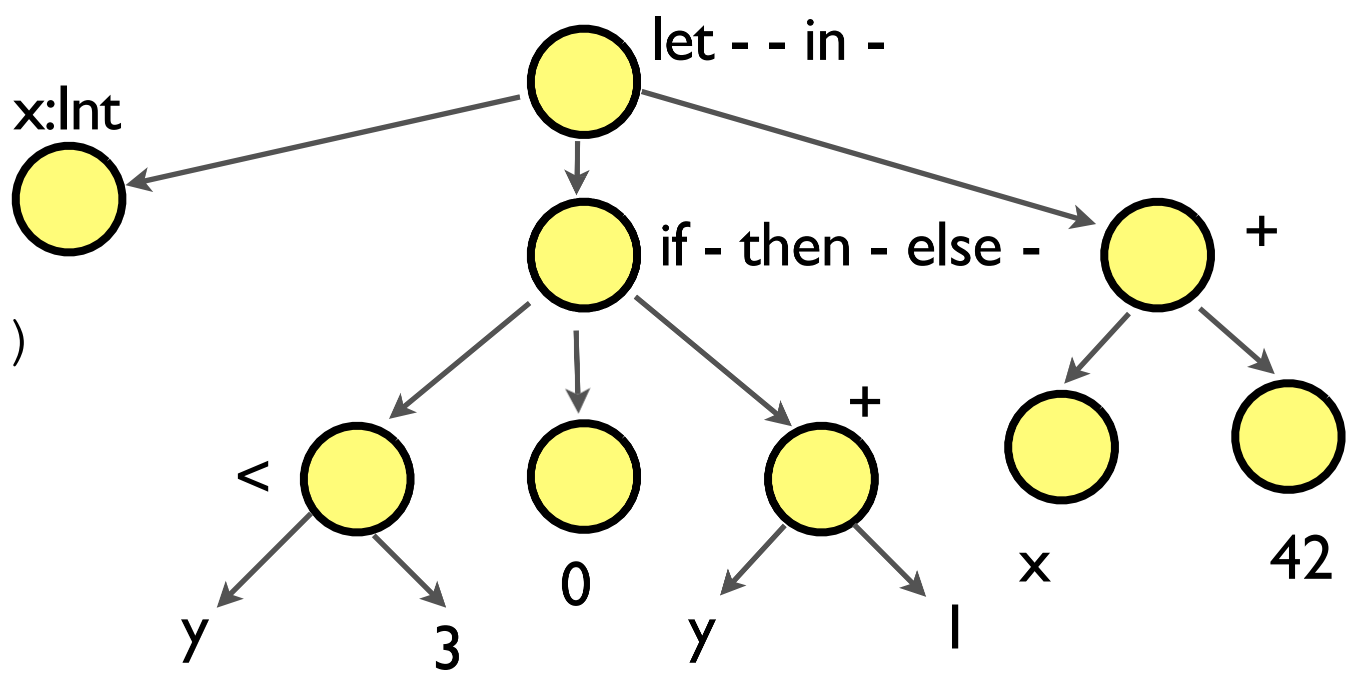
Let's play a little game with ASTs for a Toy language:

```
T , U ::=  Int | Bool
E ::=  n | true | false | E < E | E + E | x |
          | if E then E else E
          | let (x : T) = E in E
```

where n ranges over natural numbers and x ranges over some set of variable names.

Consider the following example and its AST

```
let (x : Int) =
  if (y < 3) then 0 else (y + 1)
in x + 42
```

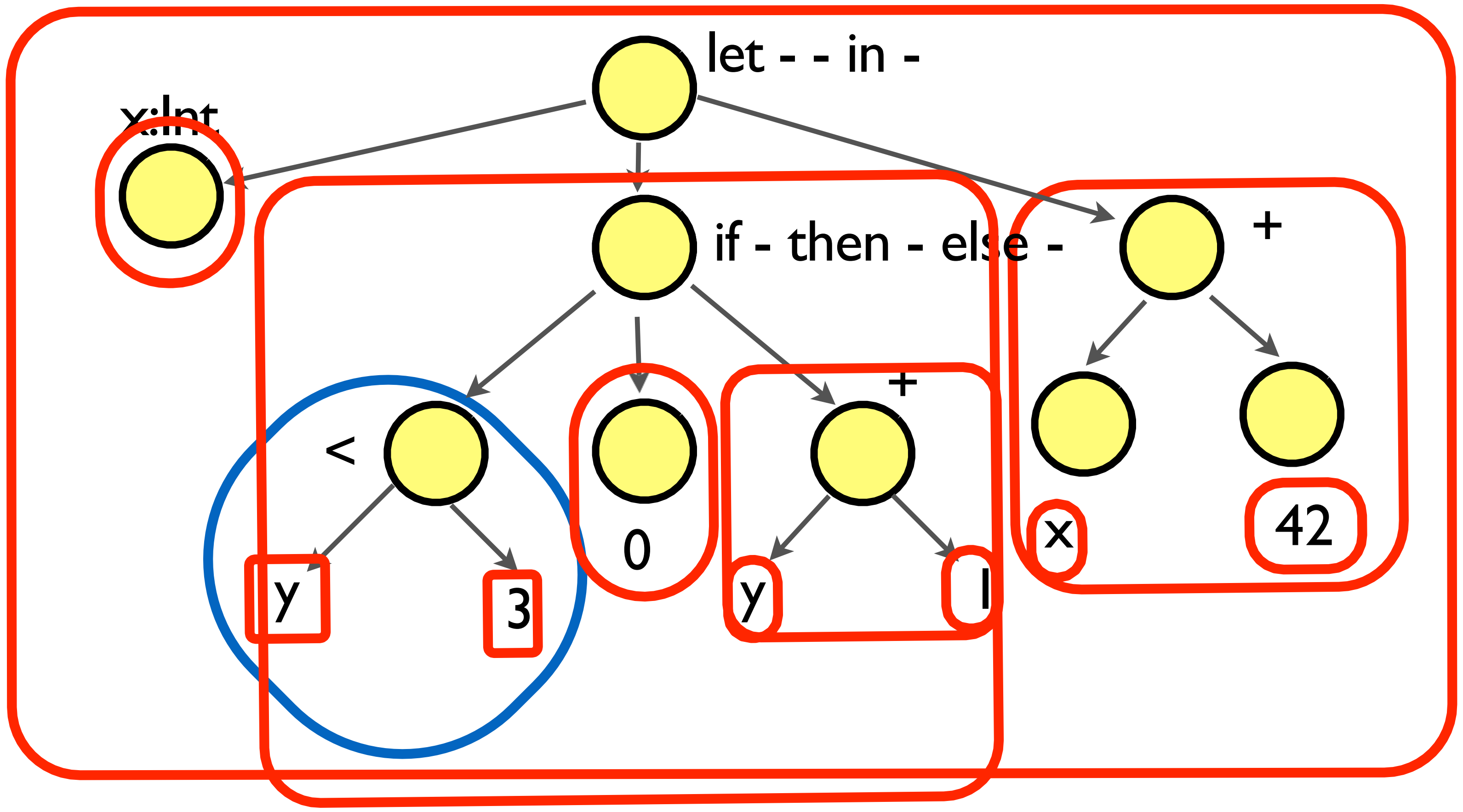


ABSTRACT ABSTRACT SYNTAX TREES

Let's rewrite the previous example with Types in place of the values and value expressions! e.g.

```
let ( Int )
   = if ( Int < Int ) then Int else Int + Int
in
 Int + Int
```

We can track the types during computation also!



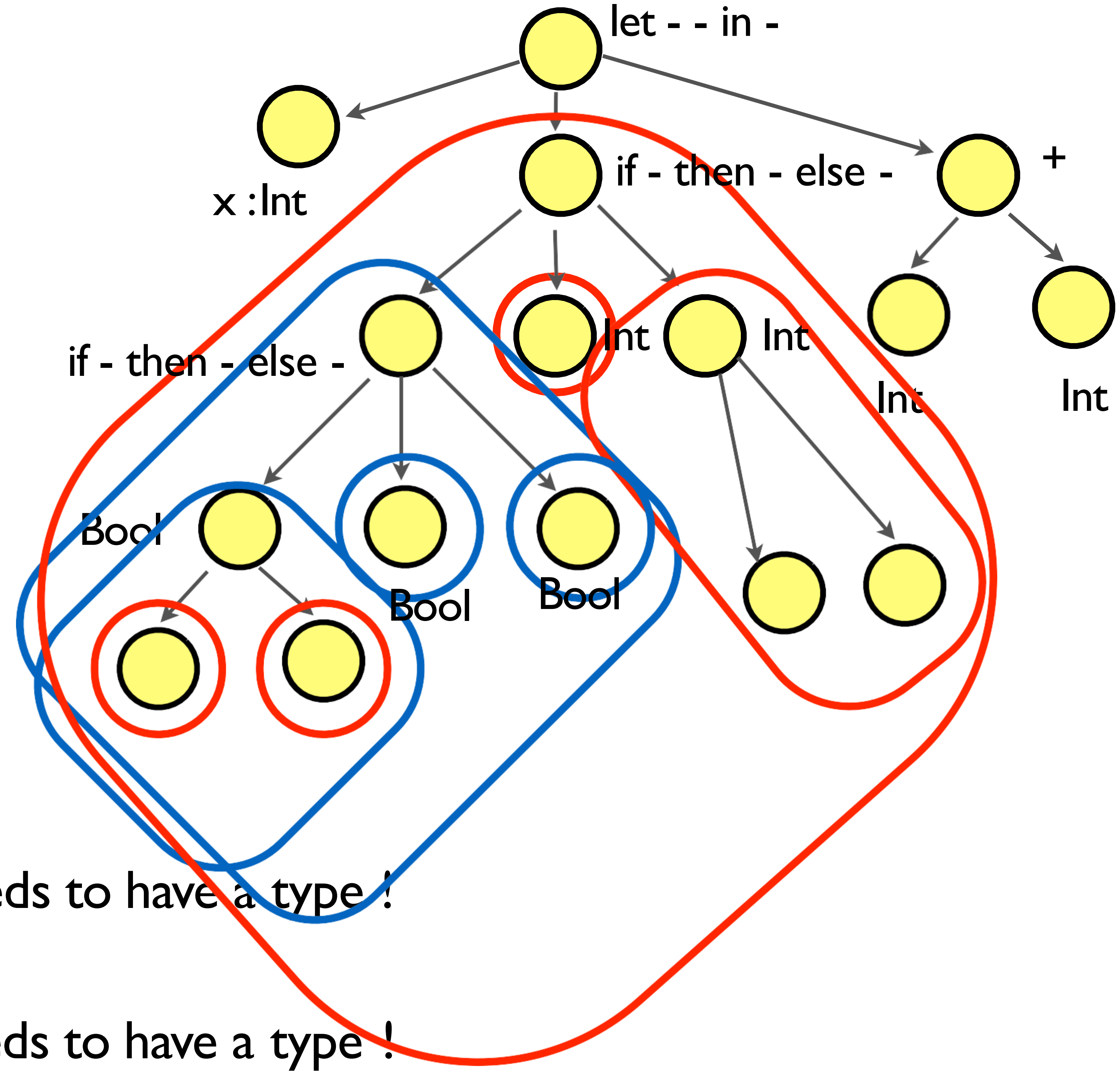
Now, one way of checking types is to traverse this type abstracted AST and see whether the operators that consume data are given data of the correct type.

The operators that consume data are : if-then-else (this reads a Bool), < consumes two Ints and + consumes two Ints

COMPLICATE THE EXAMPLE

Let's make it more fun

```
let (x : Int)
  = if ( if ( y < 0)
           then false
           else true
        )
      then 0
      else (y + 1)
in
 x + 42
```



Every subtree in the AST needs to have a type !

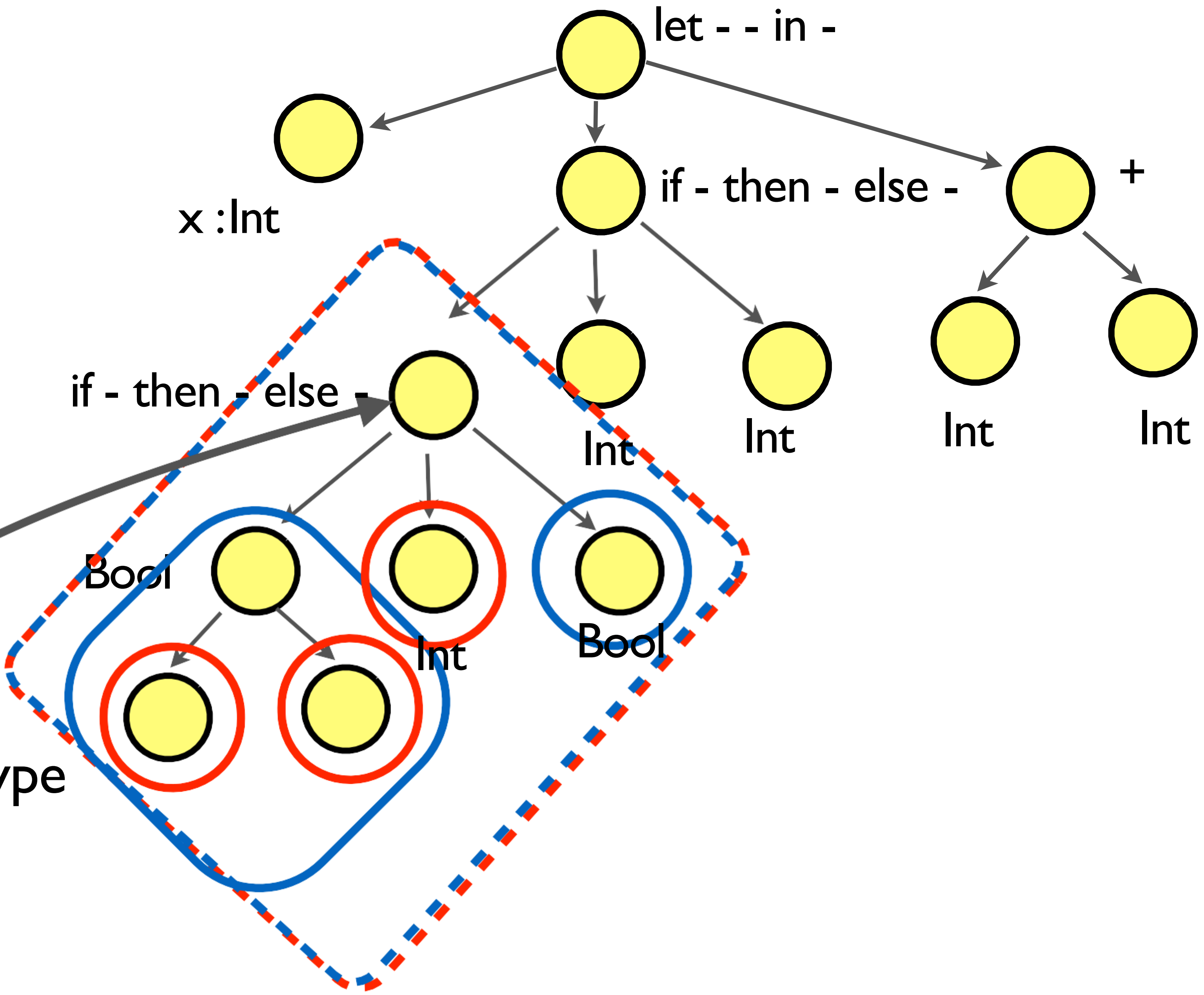
Every program fragment needs to have a type !

COMPLICATE AGAIN

So is this program well typed? In a dynamically typed language you might say so.

What about a static type?

```
let (x : Int)
  = if ( if ( 1 < 0)
          then 0
          else false
       )
     then 0
     else 1
in
 x + 42
```



So what is the type of the subtree rooted here?

We know it is either an Int or Bool - but which one in general depends on the outcome of the conditional check. This can depend on run time information - e.g. user inputs - even without this, it is undecidable in general.

WHAT HAVE WE LEARNED?

- Every program fragment E needs to be given a type T in order to build up a picture of whether the whole AST is well-typed.
  - We need to define the 'typing relation' written $\vdash$ E :T and read 'E' has type 'T')
  - We need to define this for **every** possible program E !
- We will want to do local checking on syntax trees
  - The type of a program op (E1, E2, ..., En) should depend only on the types of E1, E2,..., En
  - Only certain operators generate actual checks for correct usage of types
  - We may need to approximate the type where we can't determine it statically
- To define a relation over the set of **all** programs is an interesting challenge.
  - But we know something special about the shape of the set of all programs!
  - This is going to help us enormously.
- But what would the definition of such a relation look like?

TYPE DERIVATION RULES AND TREES

The general form of a type derivation rule is

$$\dfrac{\text{`}\ E_1 : T_1 \qquad \text{`}\ E_2 : T_2 \qquad \ldots \qquad \text{`}\ E_n : T_n}{E\ : T}$$

This can be read as " If the relation holds for the things above the line then the relation holds for things below the line also". Sometimes there are no premises above the line.

In general, a programming language will be given a set of such rules. Then, in order to show that ⊢ E : T holds, the rules must be formed in to a tree such that the leaf nodes of the tree have no premises. For example

$$\dfrac{\dfrac{\dfrac{}{\text{`}\ E_0 : T_0}}{\text{`}\ E_1 : T_1} \qquad \dfrac{\dfrac{\dfrac{}{\text{`}\ E_4 : T_4} \qquad \dfrac{}{\text{`}\ E_5 : T_5}}{\text{`}\ E_3 : T_3}}{\text{`}\ E_2 : T_2}}{E\ : T}$$

Let's consider how this leads to a relation between all programs and types.