CM4106 - LANGUAGES AND COMPILERS

SEMANTIC ANALYSIS

THIS WEEK

- Semantic Analysis
- Semantic Rules (Context Constraints)
- Symbol Table
- Type Checking

SO FAR

- Up until now we have, mainly, been concerned with the structure of the source program
- Our parser check the source program matches the language definition
- We saw last week how the parser creates the abstract syntax tree which defines what the program is going to do

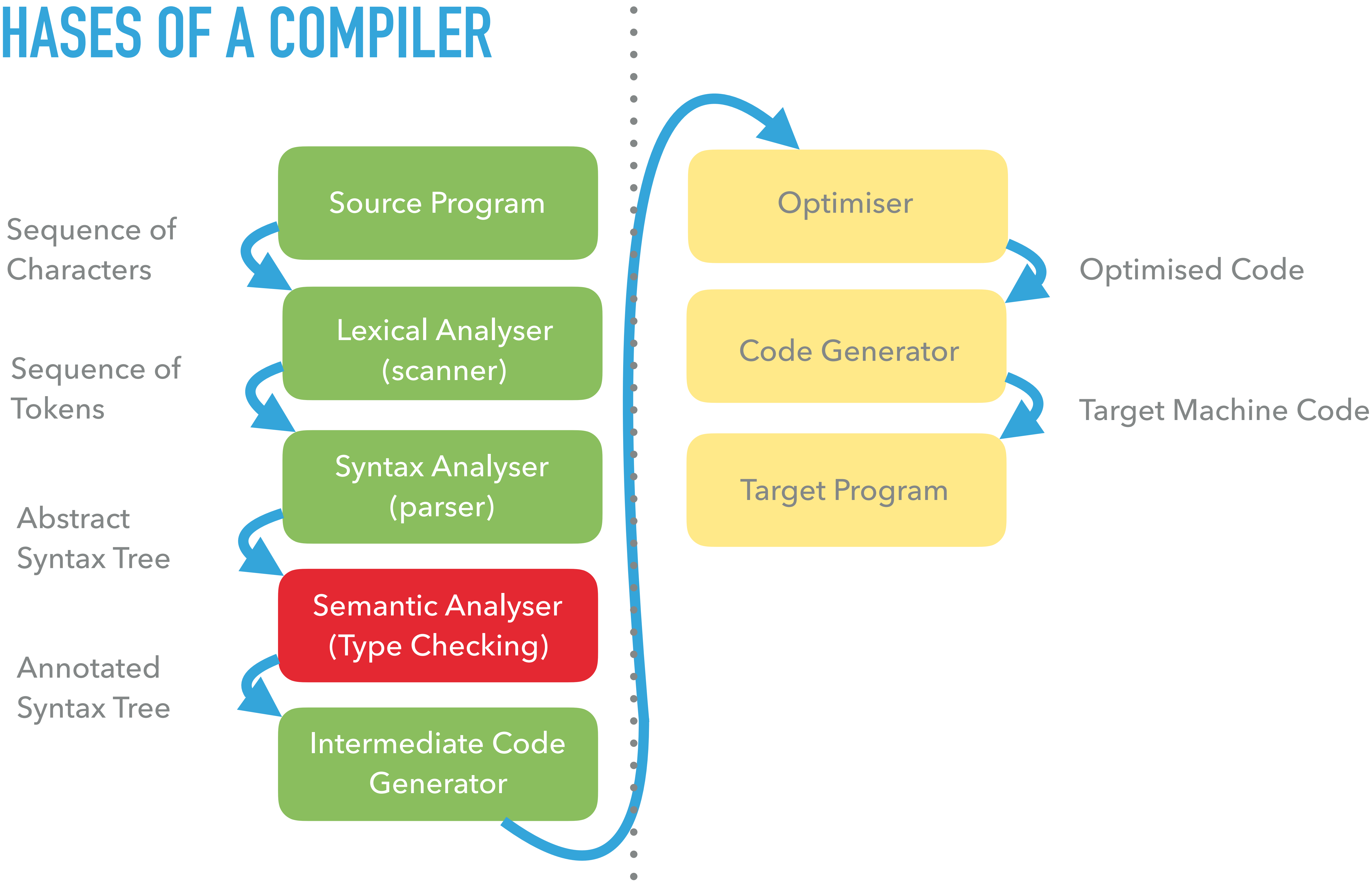
SEMANTIC RULES

- Every language will have semantic rules, or constraints which govern what the language can do.
- You already know these for the languages you use
  - if (3) then 4 else 8
  - int v = “wobble”
- You know inherently these are wrong, but they are actually defined somewhere

SEMANTIC RULES

- Semantic rules fall into two categories
- **Scope Rules**
  - Govern declarations and occurrences of identifiers
- **Type Rules**
  - Govern the types used and determines if expressions have valid types

PHASES OF A COMPILER



THE SEMANTIC ANALYSER

- these two sets of rules form the two sub-processes of the Semantic Analysis phase
- **Identification**
  - we apply the source language's scope rules to relate each identifier to its declaration (if it has one)
- **Type Checking**
  - we apply the source language's type rules to determine if the an expressions type matches the expected type

IDENTIFICATION PHASE

- The following code is syntactically correct, it fits our language

```
let
 var x : Integer
in
begin
 x:=1;
 putint(x);
 putint(y);
end
```

Semantic error

- But our semantic analyser should detect that the identifier **x** has been declared but that **y** has not

IDENTIFICATION PHASE

- We have two areas of scope here.
- Although syntactically correct, y has still not been defined in the scope it is used.

This is ok

This is't

```
let
 var x : Integer
in
begin
 let
  var y : Integer
 in
 begin
  y:=2;
 end;
 x:=1;
 putint(x);
 putint(y)
end
```

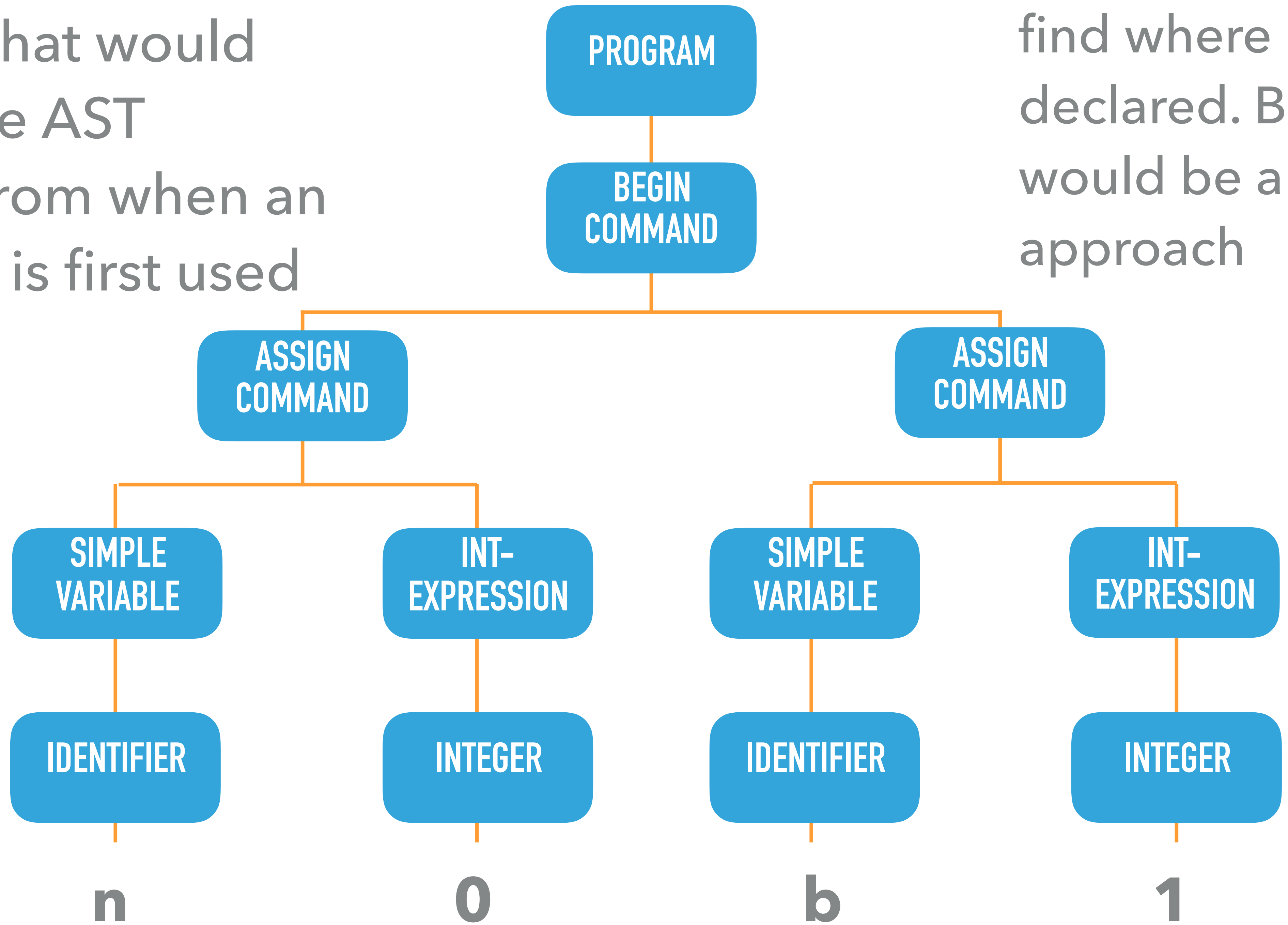
IDENTIFICATION ISSUES

- the identification phase can be slow.
- Longer, more complicated programs will have more identifiers and this is the main reason why long programs take so long to compile
- How do we implement the identification?

OUR SYNTAX TREE FROM LAST WEEK

begin n:=0; b:=1 end

We could create a method that would search the AST starting from when an identifier is first used



And work back to find where it was declared. But this would be a difficult approach

SYMBOL TABLE

- A better method is to use an Symbol Table
- a simple table that associates an Identifier with some attributes representing that Identifier (type, kind, visibility etc)
    - start with an empty table
    - when an identifier is found add it to the list (**enter)**
    - attempt to get the attributes when you come across another identifier (**retrieve**)

TABLE ORGANISATION

- How does our identification table deal with scope?
- It depends on the languages **scope rules** and **block structure**
  - **Monolithic Block Structure**
  - **Flat Block Structure**
  - **Nested Block Structure**

FLAT BLOCK STRUCTURE

```
program
 (1)integer b = 10
 (2)integer n
 (3)char c

 begin
  …
   n=n*b
  …
  write c
  …
 end
```

Ident	Attr
b	1
n	2
c	3

- **Characteristics**
  Only **one block**: entire program
  All declarations are **global**
- **Scope rules**
  identifier declared only **once**
  identifier **cannot** be used if **not declared**
- **Symbol table**
  For every **identifier** there is a **single entry** in the symbol table.
  Retrieval should be **fast**
  (e.g. binary search tree or hash table).
- **BASIC and COBOL use this approach**

NESTED BLOCK STRUCTURE

```
let
  (1)integer c
  (2)boolean b

  (3)procedure Q
    (4)real r
    (5)real pi

  (6)procedure R
     begin
        call Q
     end
```

Level	Ident	Attr
global	c	1
global	b	2
global	Q	3
local	r	4
local	pi	5
global	R	6

- **Characteristics**
  program has **several code blocks**
  two scope levels: global and local
- **Scope rules**
  globally declared identifier **cannot be redeclared** globally.
  locally declared identifier **cannot be redeclared** in the same block.
  identifier **must be declared** to use
- **Symbol table**
  Symbol table contains entries for **global** and **local** declarations.
  After analysis of a block has completed, its local declarations can be discarded.
- **FORTRAN and C use this approach**

NESTED BLOCK STRUCTURE

```
let !level 1
 var a: Integer;
 var b: Boolean
 in
  begin
  …;
  let !level 2
   var b: Integer;
   var c: Boolean
  in
   begin
    let !level 3
     const x ~ 3
    in …;
   end
   …;
   let !level 2
   var d: Boolean;
   var e: Integer
  in …;
end
```

- **Characteristics**
  blocks can be **nested** within each other
  many scope levels
- **Scope rules**
  Identifier **cannot be redeclared** in the same block
  Identifier cannot be used unless declared
- **Symbol table**
  Several entries for each identifier
  One entry for each (scope, identifier, combination)
  Highest level returned first
- **PASCAL, Java, C# etc**

SYMBOL TABLE CREATION

```
let
 var a: Integer;
 var b: Boolean
 in
  begin
  …;
  let
   var b: Integer;
   var c: Boolean
  in
   begin
    let
     const x ~ 3
    in …;
   end
   …;
   let
   var d: Boolean;
   var e: Integer
  in …;
end
```

Level	Ident	Attr
1	a	**1**
1	b	**2**

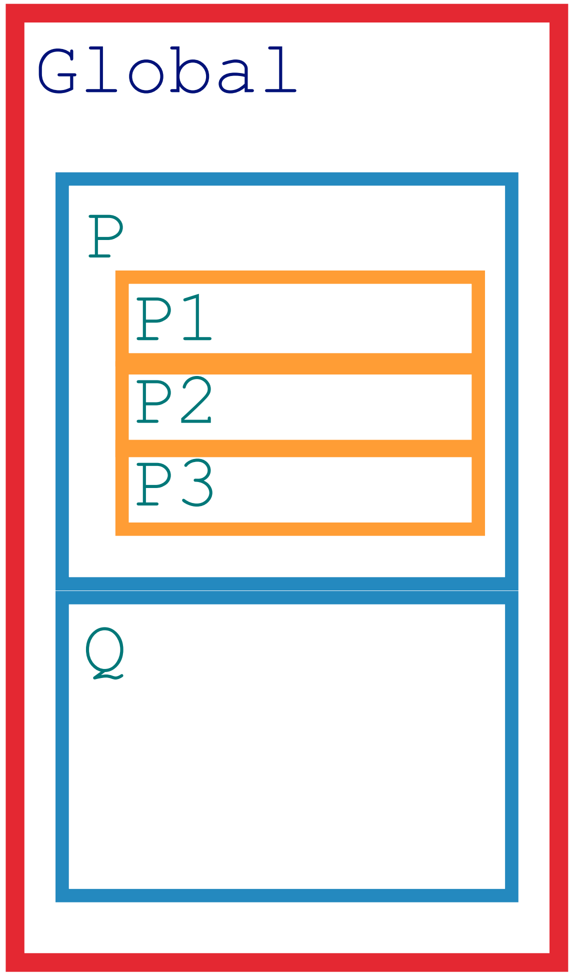
Level	Ident	Attr
1	a	**1**
1	b	**2**
2	b	**3**
2	c	**4**

Level	Ident	Attr
1	a	**1**
1	b	**2**
2	b	**3**
2	c	**4**
3	x	**5**

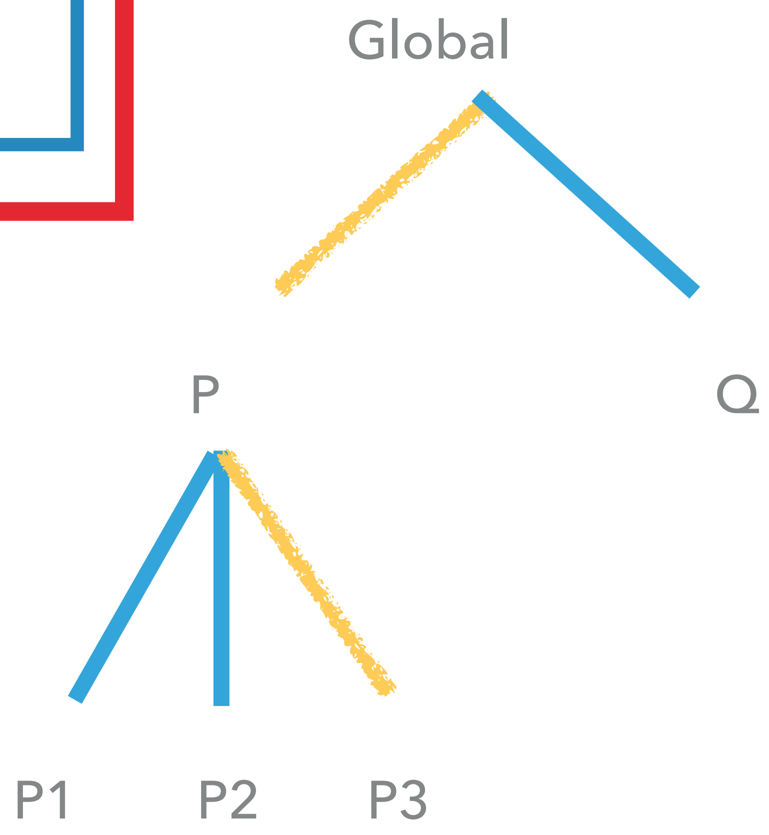
Level	Ident	Attr
1	a	**1**
1	b	**2**
2	d	**6**
2	e	**7**

SCOPE STRUCTURE

- Triangle uses a **nested block structure** and is **statically scoped**



- This means that the structure of the scoping can be shown as a tree



- When analysing the program only one path is visible

SYMBOL TABLE SKELETON IMPLEMENTATION

```
public class SymbolTable {

 public void openScope()

 public void closeScope()


 public void enter(String id, Attribute attr);

 public Attribute retrieve(String id)

 public int currentLevel()
}
```

Opens a new scope

Closes current (highest) scope

Creates a new entry with an id & attributes.
Attributes will be different for different languages.

Retrieves an attribute for an ID (or null)

gets current level

```
let
 var a: Integer;
 var b: Boolean
 in
  begin
  let
   var b: Integer;
   var c: Boolean
  in
   begin
    let
     const x ~ 3
     a = x + a
    in
   end

   let
   var d: Boolean;
   var e: Integer
  in
end
```

openScope()
entry(a, level1)
entry(b, level1)

openScope()
entry(b, level2)
entry(c, level2)

openScope()
entry(x, level3)
retrieve(x)
retrieve(a)
closeScope()

closeScope()
openScope()
entry(e, level2)
entry(d, level2)
closeScope()
closeScope()

the let token tells us to start a new scope

ATTRIBUTES

- Attributes should contain information for
  - Checking the scope rules
    - if a **retrieve** command is successful it means an **identifier has been declared** at that point.
  - Checking the type rules
    - the **type** of an **identifier** must be stored
  - Code Generation (later)

STORING ATTRIBUTES

- We could store Attributes as Objects

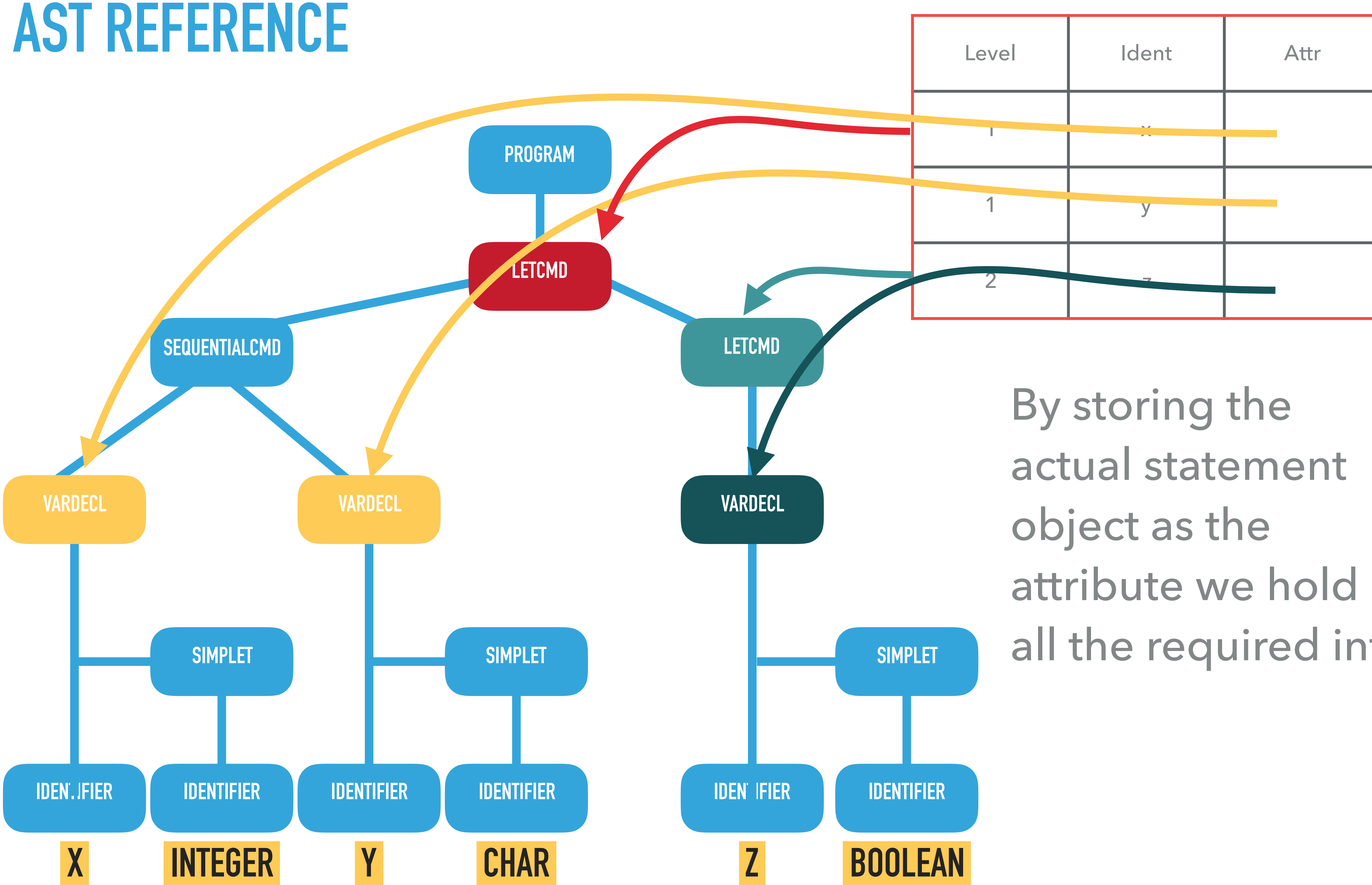
```
public class Attribute {
 public Kind kind;
 public Type type;
}
```

- You would need to create a class of every possible type of expression and statement. eg bool, char, int, var, const
- And then record the tree for each scope
- This works but could become pretty tedious and complex for realistic languages

ATTRIBUTES AND TREE REFERENCES

- We already have a scoped tree with classes for each part that contains the info we need
- Our AST that comes from the parser already has the info we need.
- the Let commands are identified to determine the scope
- and any declaration and use of a variable will already have a class to represent it

AST REFERENCE



By storing the actual statement object as the attribute we hold all the required info

TYPES

- **What is a type**?
  - ‘A restriction on the possible interpretations of a segment of memory or a program construct’
  - Or, more simply, a **set of values** defined by the language semantics

WHY USE TYPES

- **Error avoidance**
  - Prevent programmer from making type errors
  - eg char X = 4 or int V = “hello”
- **Runtime optimisation**
  - earlier binding to a type will make the compiler more efficient

DO WE NEED TYPES

- **It depends on the language!**
  - Java is strictly typed.
  - C# (core) isn't, kind of (var type)
  - Javascript isn't
  - Languages can work without strict types, but generally defining types leads to less runtime mistakes

TYPE CHECKING

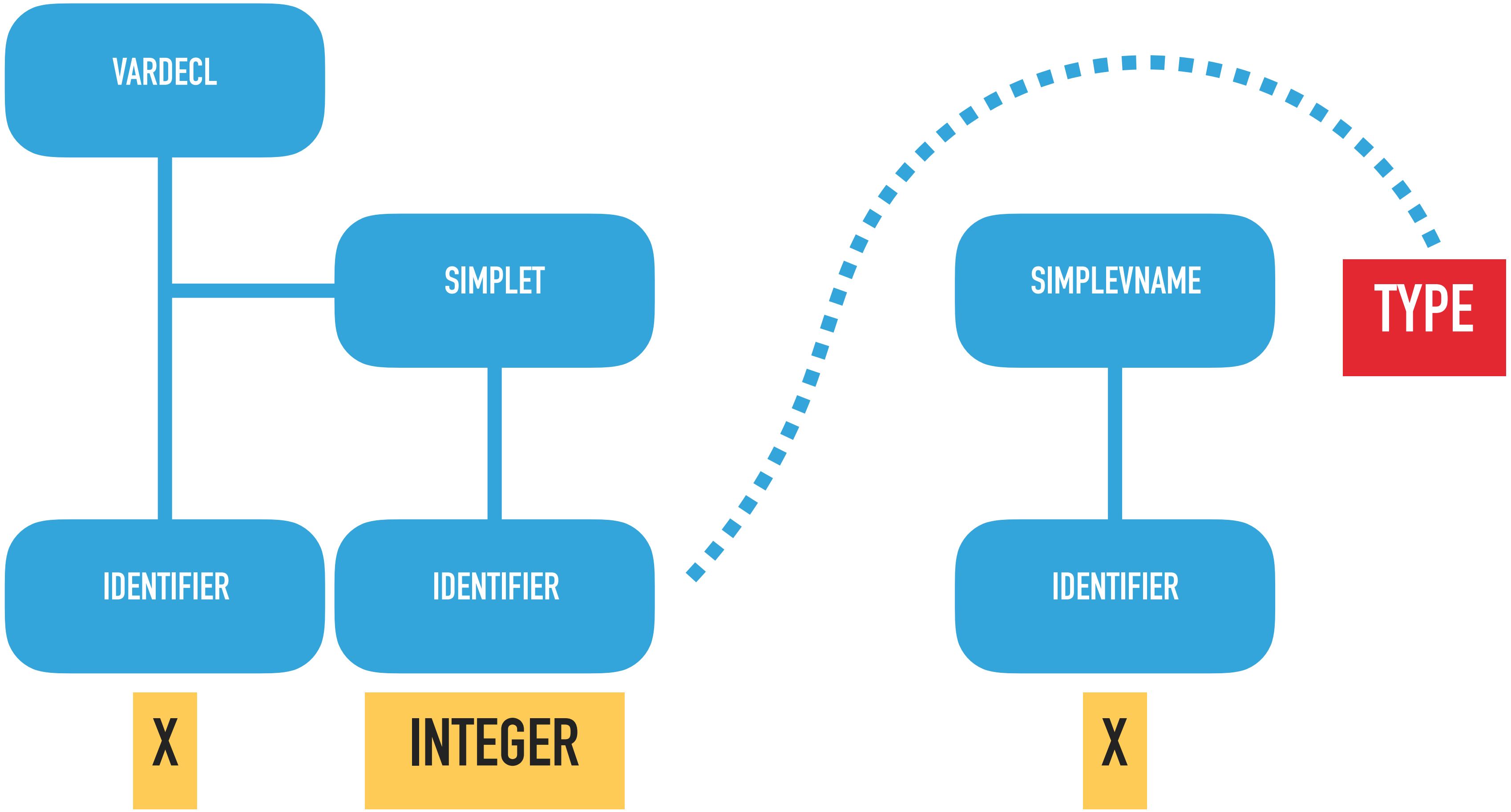
- In a Statically Typed language the compiler can determine type errors without actually running the program.
- For every expression in the language the compiler can determine if the expression is
  - (i) ill-typed (error)
  - (ii) or has a type that can be determined without looking at the value and that will remain same through out runtime

TYPE CHECKING

- For most statically typed languages Type Checking is straight forward
- The Type-Checker infers the type of each expression
  - Literal - type is immediately known (char, int etc)
  - Identifier - type is obtained from the first declaration of that identifier (from symbol table)
  - Unary Operator - the resulting type is the same as the initial type eg (x>5) is a boolean as is /(x>5), other wise there is a type error
  - Binary Operator - the types are equivalent eg x * 6 not x * 'h'

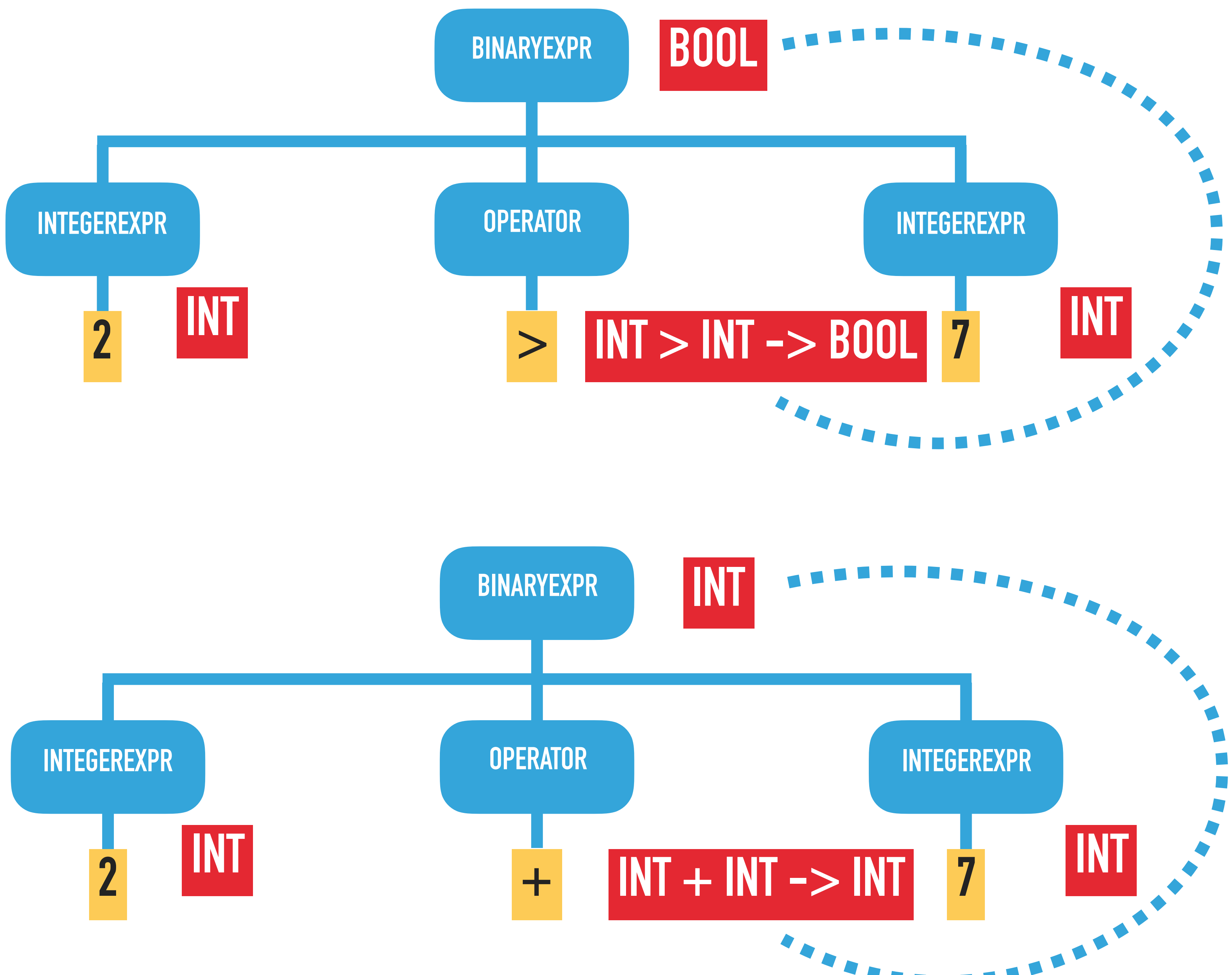
TYPE CHECKING

Vname from declaration



TYPE CHECKING

BinaryExpression - inferred from operator



REDUCED TRIANGLE SEMANTIC RULES

- the Skip command ‘ ’ has no effect when executed
- the assignment command **Variable := Expression** is executed as follows. The expression is evaluated to a value then the variable is updated to this value (V and E must be equivalent)
- the procedure calling command **Identifier(actual-parameter-sequence)** is executed as follows. The parameter list is evaluated to a list of parameters the procedure identified is then called with that list.

REDUCED TRIANGLE SEMANTIC RULES

- For the sequential command: **single-command1; single-command2.** single-command is evaluated first
- for the bracketed command **begin Command end** is executed by simply executing command
- the block command **let declaration in command** is executed as follows; the declarations are handled first then the command executed. Let defined scope

REDUCED TRIANGLE SEMANTIC RULES

- The If command **if E then C1 else C2** is executed as follows; the expression E is evaluated, if it is true C1 is executed, otherwise C2 is executed. **E must be a boolean.**

- The while command **while E do C** is executed as follows; the expression E is evaluated, if it is true C is executed and the while command is checked again. If the value is false the execution of the while command is completed

IMPLEMENTATION

- We will implement this Symbol table and Type Checking using an algorithm called the Visitor Pattern
- A visitor pattern let us check the a program using the AST Objects without changing the AST it's self
- Very help full as it lets us add more operations (semantics) without changing the Language Syntax

SUMMARY

- Semantics define the meaning of the language
- Our semantic analysis phase looks at ensuring that the source program fits the semantic rules of the language
- the Symbol table plays an extensive role in this process