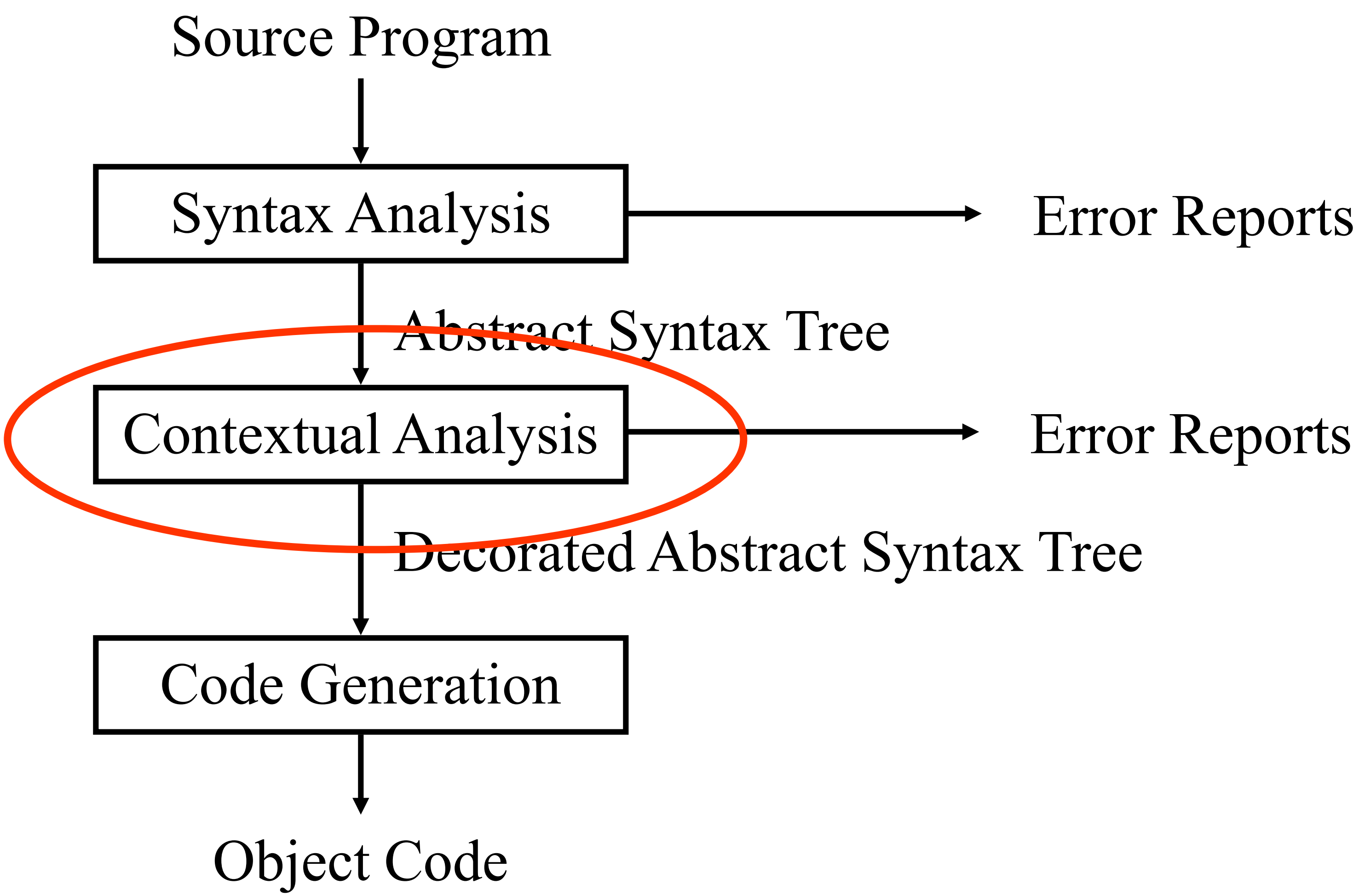
Languages and Compilers (SProg og Oversættere)

Semantic Analysis

Semantic Analysis

- Describe the purpose of the Semantic analysis phase
- Discuss Identification and type checking
- Discuss scopes/block structure and implication for implementation of identification tables/symbol tables
- Discuss Implementation of semantic analysis

The “Phases” of a Compiler



Contextual Constraints

Syntax rules alone are not enough to specify the format of well-formed programs.

Example 1:

```
let const m~2;
in  m + x
```

Undefined! → Scope Rules

Example 2:

```
let const m~2 ;
    var   n:Boolean
in begin
   n := m<4;
   n := n+1
end
```

Type error! → Type Rules

Scope Rules

Scope rules regulate visibility of identifiers. They relate every **applied occurrence** of an identifier to a **binding occurrence**

Example 1

Binding occurrence

```
let const m~2;
    var   r:Integer
in
    r := 10*m
```

Applied occurrence

Example 2:

?

```
let const m~2
in  m + x
```

Terminology:

Static binding vs. *dynamic binding*

Static scope/block structured scope vs. dynamic scope

Implicit vs. explicit binding

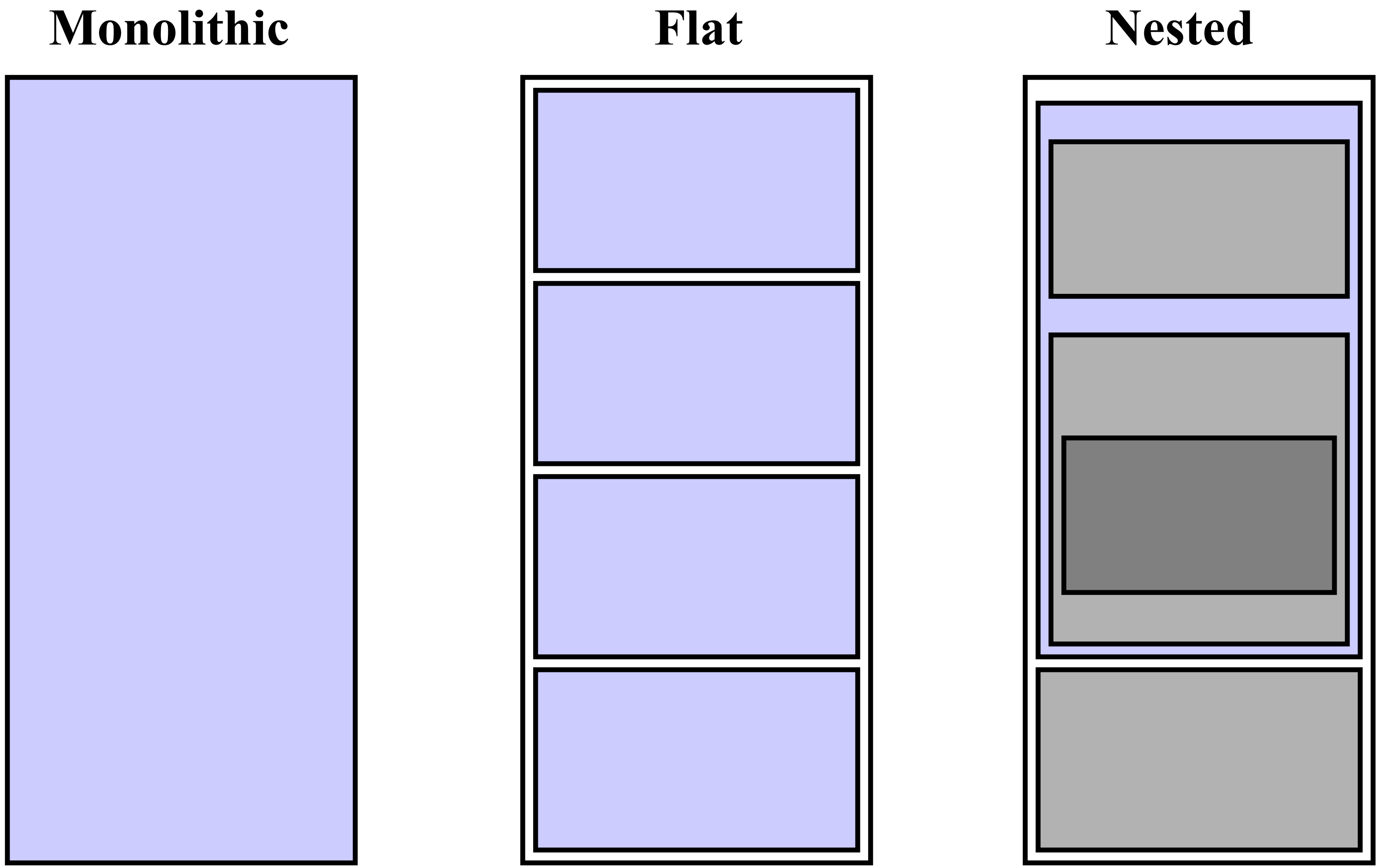
Example (from p. 88 in Transitions and Trees)

```
begin
        var x:= 0;
        var y:= 42

        proc p is x:= x+3;
        proc q is call p;

        begin
                var x:=9;
                proc p is x := x+1;
                call q;
                y := x
        end
end
```

Assuming static scope for procedures and variables,
What is the value assigned to y ?

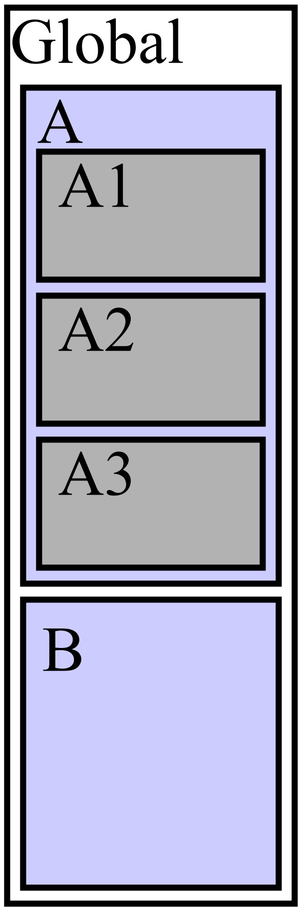
Assuming dynamic scope for procedures and variables,
What is the value assigned to y ?

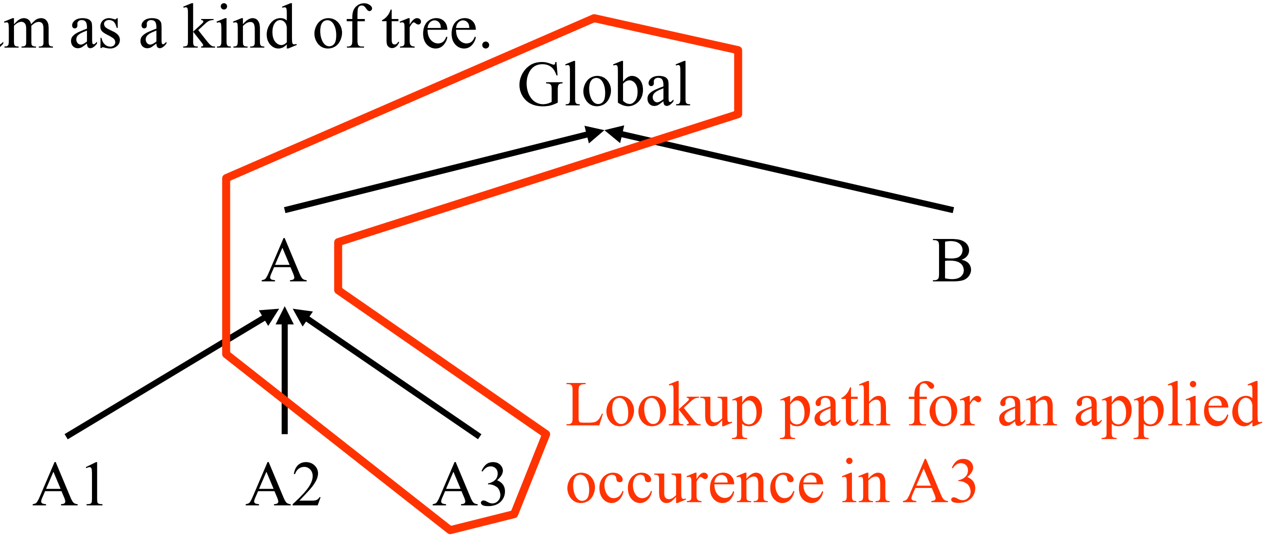
Different kinds of Block Structure... a picture



Identification Table

For a typical programming language, i.e. statically scoped language and with nested block structure we can visualize the structure of all scopes within a program as a kind of tree.





→ = "direction" of identifier lookup

At any one time (in analyzing the program) only a single path on the tree is accessible.
=> We don't necessarily need to keep the whole "scope" tree in memory all the time.

Identification Table: Example

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

Level	Ident	Attr
1	a	(1)
1	b	

Level	Ident	Attr
1	a	(1)
1	b	(2)
2	b	(3)
2	c	(4)

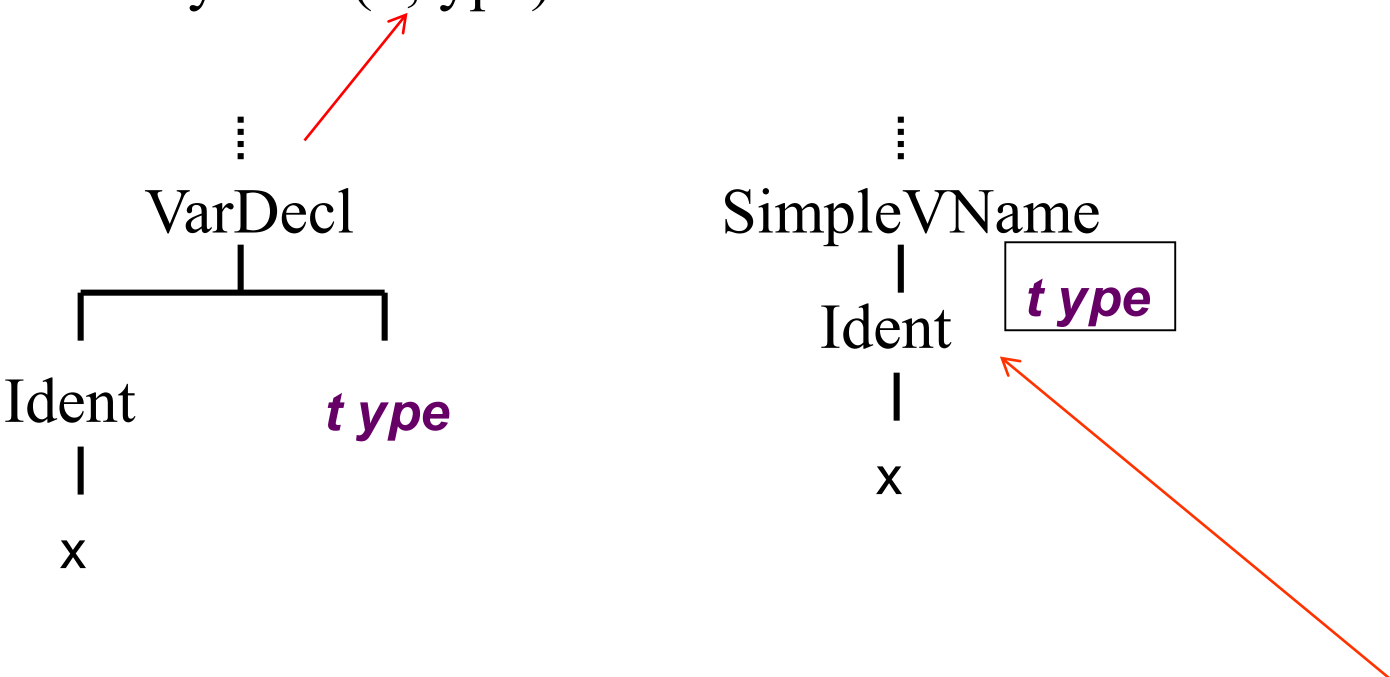
Level	Ident	Attr
1	a	(1)
1	b	
2	d	
2	e	

Level	Ident	Attr
1	a	(1)
1	b	(2)
2	d	(5)
2	e	(6)
3	x	(7)

Type Checking: How Does It Work

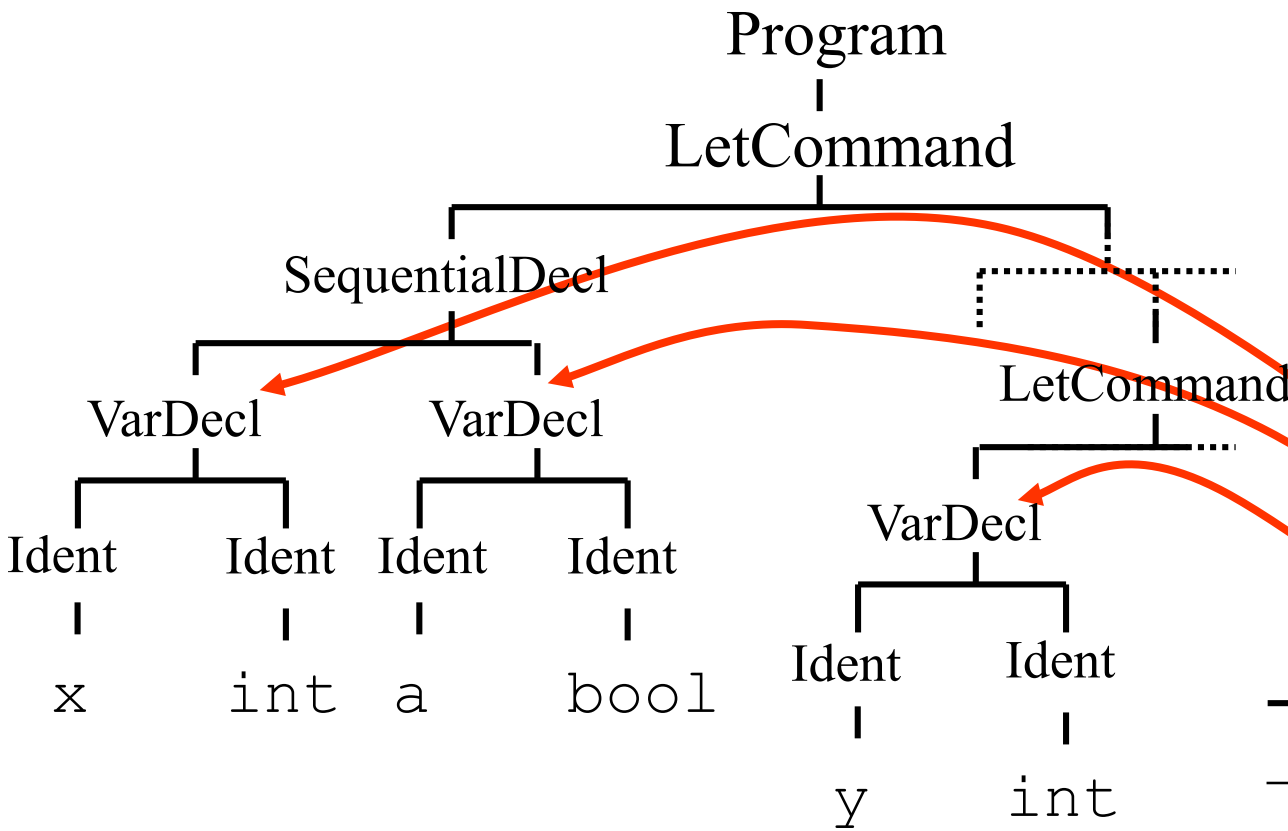
Example: Type of a variable (applied occurrence)

During Identification/SymbolTableFilling:
EnterSymbol(x,type)



During typeChecking:
RetreiveSymbol(x) -> type

Attributes as pointers to Declaration AST's

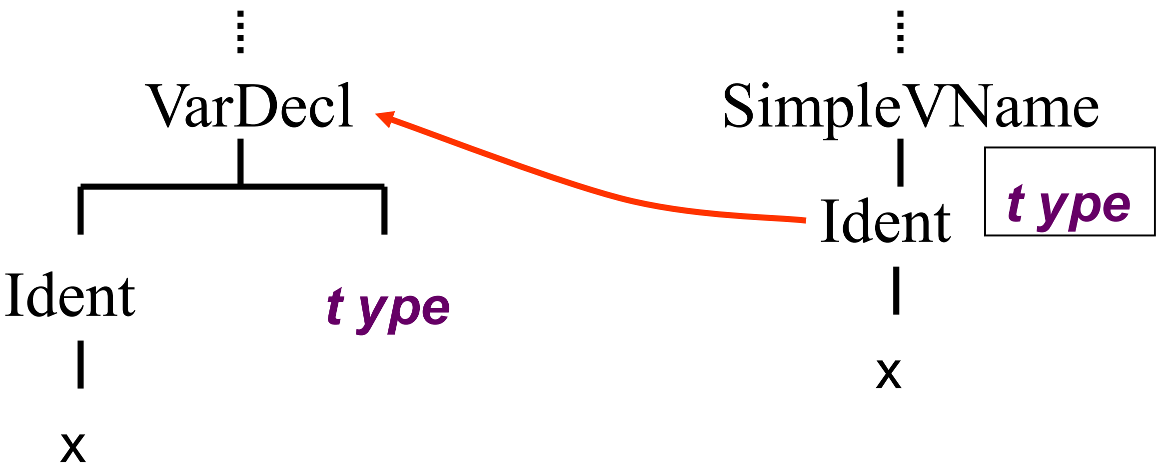


Id table

Level	Ident	Attr
1	x	•
1	a	•
2	y	•

Type Checking: How Does It Work

Example: Type of a variable (applied occurrence)



Type Checking

For most statically typed programming languages, a bottom up algorithm over the AST:

- Types of expression AST leaves are known immediately:
  - literals => obvious
  - variables => from the ID table
  - named constants => from the ID table
- Types of internal nodes are inferred from the type of the children and the type rule for that kind of expression

Type Rules

Type rules regulate the expected types of arguments and types of returned values for the operations of a language.

Examples

Type rule of `<` :

E1 `<` *E2* is type correct and of type `Boolean`
if *E1* and *E2* are type correct and of type `Integer`

Type rule of `while`:

`while` *E* `do` *C* is type correct
if *E* of type `Boolean` and *C* type correct

Terminology:

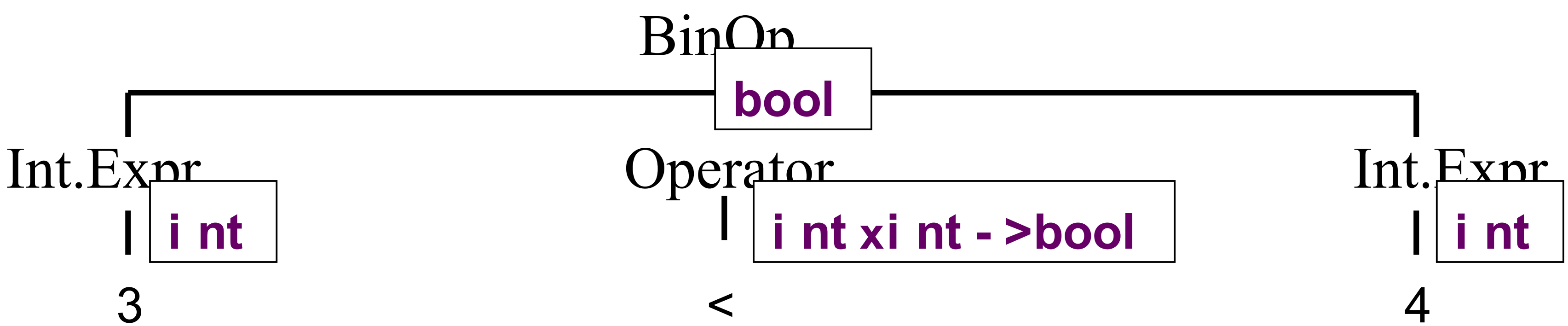
Static typing vs. *dynamic typing*

Type Checking: How Does It Work

Example: the type of a binary operation expressions

Type rule:

If ***`op`*** is an operation of type ***`T1xT2->R`*** then ***`E1 op E2`*** is type correct and of type ***`R`*** if ***`E1`*** and ***`E1`*** are type correct and have type compatible with ***`T1`*** and ***`T2`*** respectively



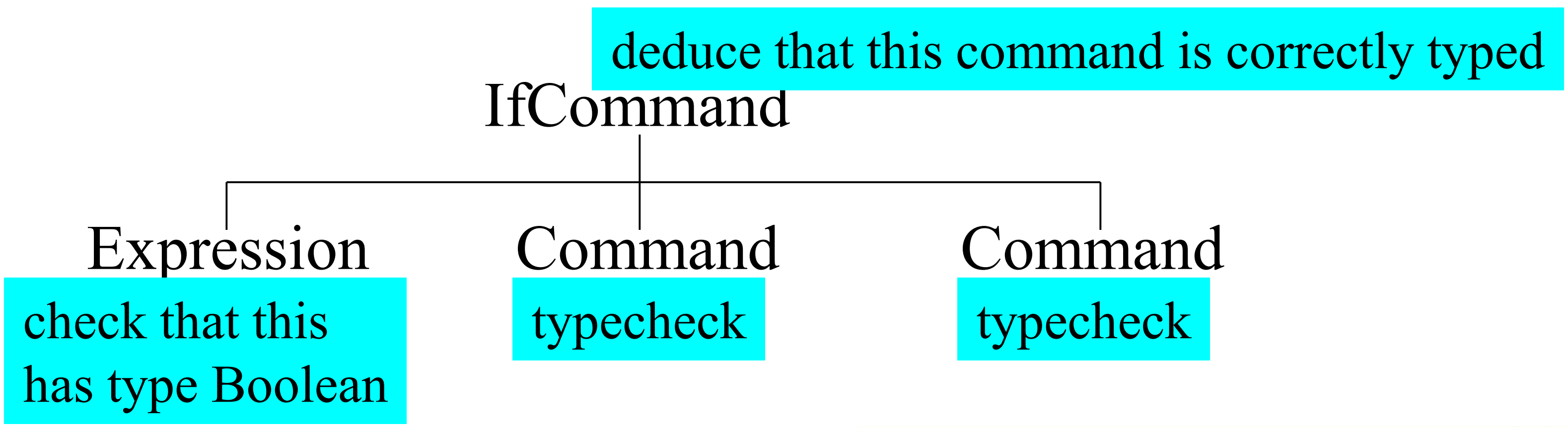
Type checking

Commands which contain expressions:

Type rule of **IfCommand**:

if *E* **do** *C1* **else** *C2* is type correct

if *E* of type **Boolean** and *C1* and *C2* are type correct

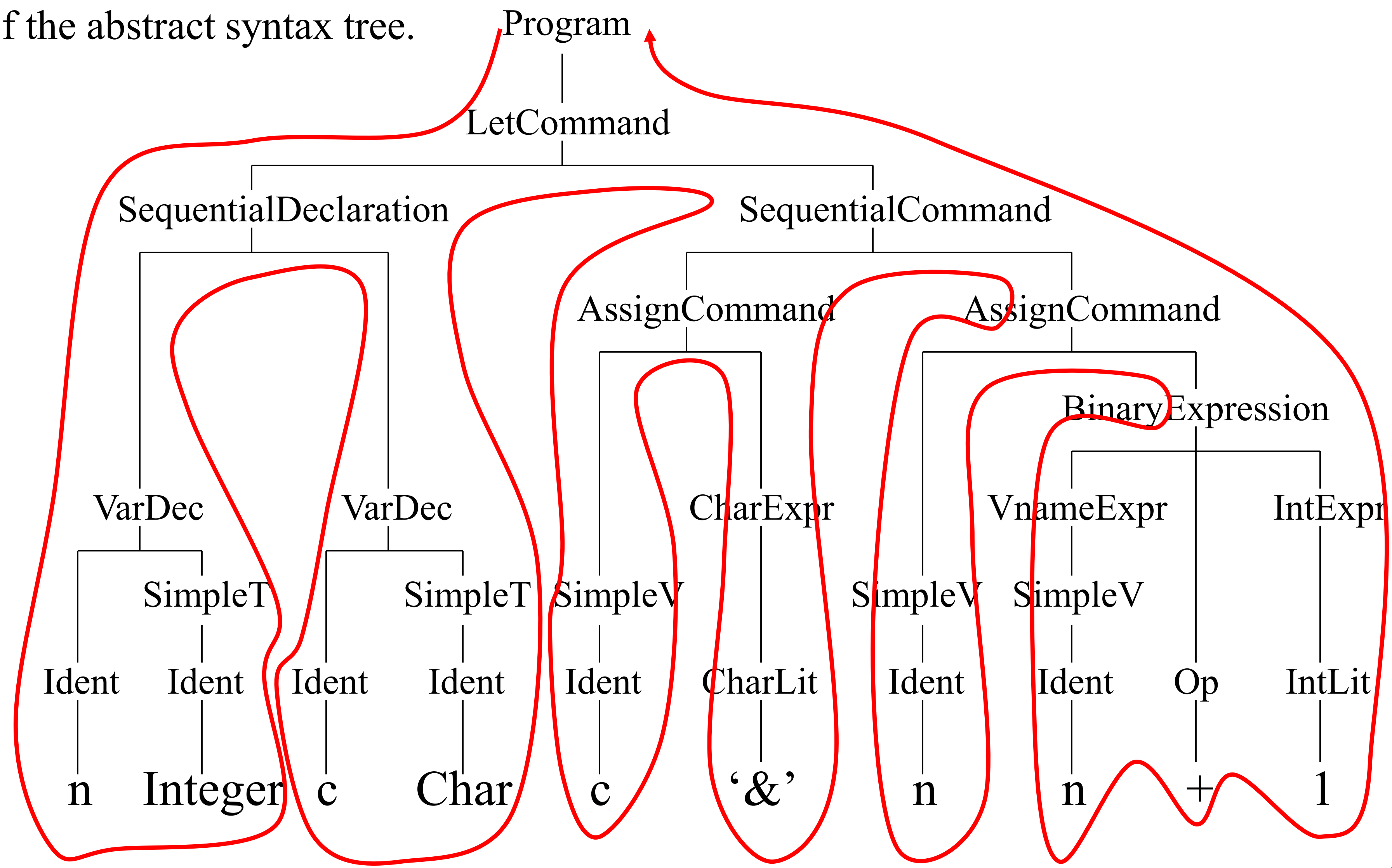


WhileCommand is similar.

$$[\text{IFSTM}] \quad \frac{E \vdash e : \mathsf{Bool} \quad E \vdash S_1 : \mathsf{ok} \quad E \vdash S_2 : \mathsf{ok}}{E \vdash \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else};\ S_2 : \mathsf{ok}}$$

Contextual Analysis

Identification and type checking are combined into a depth-first traversal of the abstract syntax tree.



Implementing Tree Traversal

- “Traditional” OO approach
- Visitor approach
  - GOF
  - Using static overloading
  - Reflective
  - (dynamic)
  - (SableCC style)
- “Functional” approach
- Active patterns in Scala (or F#)
- (Aspect oriented approach)

Implementing type checking from type rules

(conditional)

$$\frac{\Gamma \vdash E: T_E,\ T_E=\text{bool},\ \Gamma \vdash S_1: T_1,\ \Gamma \vdash S_2: T_2,\ T_1=T_2}{\Gamma \vdash \text{if } E \text{ then } S_1 \text{ else } S_2: T_1}$$

```
  public Object visitIfExpression (IfExpression
com, Object arg)
  {
     Type eType = (Type)com.E.visit(this, null);
     if (! eType.equals(Type.boolT) )
        report error: expression in if not boolean
     Type c1Type = (Type)com.C1.visit(this, null);
     Type c2Type = (Type)com.C2.visit(this, null);
     if (! c1Type.equals(c2Type) )
        report error: type mismatch in expression
branches
     return c1Type;
  }
```

Why contextual analysis can be hard

- Questions and answers involve non-local information
- Answers mostly depend on values, not syntax
- Answers may involve computations

Solution alternatives:

- Abstract syntax tree
  - specify non-local computations by walking the tree
- Identification tables (sometimes called symbol tables)
  - central store for facts + checking code
- Language design
  - simplify language