University of Glasgow

7 Contextual analysis

- Aspects of contextual analysis
- Scope checking
- Type checking
- Case study: Fun contextual analyser
- Representing types
- Representing scopes

University of Glasgow

Aspects of contextual analysis

- **Contextual analysis** checks whether the source program (represented by an AST/syntax tree) satisfies the source language's **scope rules** and **type rules**.
- Contextual analysis can be broken down into:
  - **scope checking**
    (ensuring that every identifier used in the source program is declared)
  - **type checking**
    (ensuring that every operation has operands with the expected types).

University of Glasgow

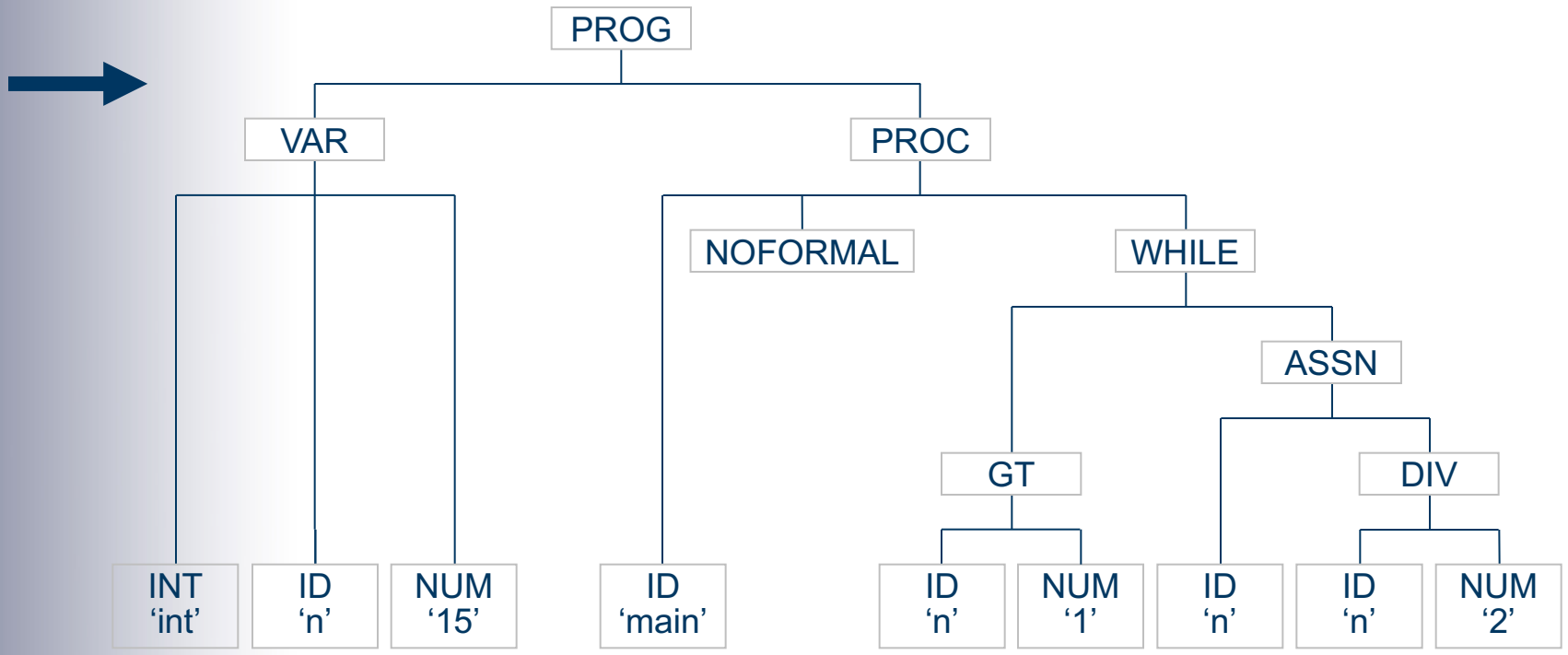
Example: Fun compilation *(1)*

- Source program:

```
int n = 15
# pointless program
proc main ():
  while n > 1:
    n = n/2 .
.
```

Example: Fun compilation *(2)*

- AST after syntactic analysis (slightly simplified):

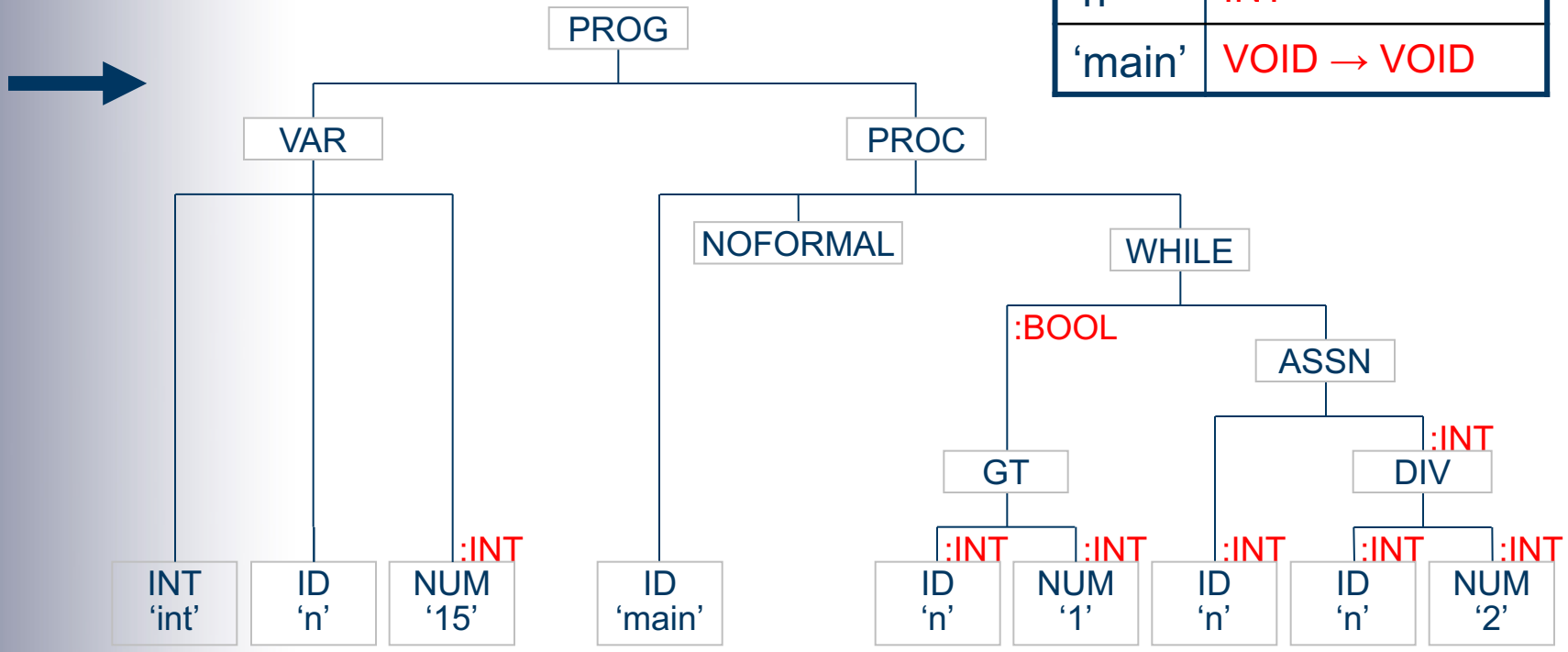


Example: Fun compilation *(3)*

- AST after contextual analysis:

Type table (simplified)

‘n’	INT
‘main’	VOID → VOID



University of Glasgow

Scope checking *(1)*

- **Scope checking** is the *collection* and *dissemination* of information about declared identifiers.
- The contextual analyser employs a **type table**. This contains the type of each declared identifier. E.g.:

‘n’	BOOL
‘fac’	INT → INT
‘main’	INT → VOID

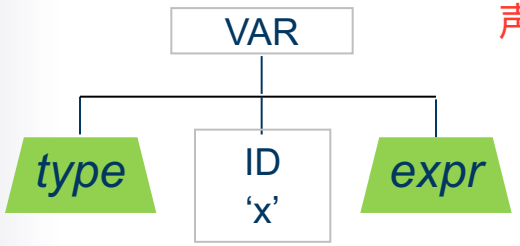
University of Glasgow

Scope checking *(2)*

- Wherever an identifier is ***declared***, put the identifier and its type into the type table.
  - If the identifier is already in the type table (in the same scope), report a scope error.
- Wherever an identifier is ***used*** (e.g., in a command or expression), check that it is in the type table, and retrieve its type.
  - If the identifier is not in the type table, report a scope error.

Example: Fun scope checking

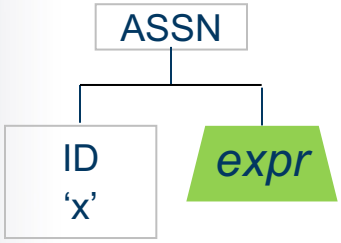
- ***Declaration*** of a variable identifier:



声明

put the identifier 'x' into the type table, along with the *type*.

- ***Use*** of a variable identifier:



使用

lookup the identifier 'x' in the type table, and retrieve its type.

University of Glasgow

Type checking *(1)*

- **Type checking** is the process of checking that every command and expression is well-typed, i.e., free of type errors.
- *Note:* The compiler performs type checking only if the source language is statically-typed.

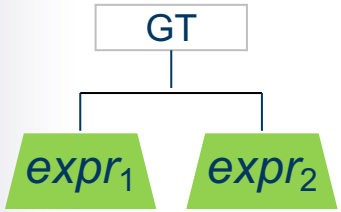
University of Glasgow

Type checking *(2)*

- At each *expression*, **check** the type of any sub-expression. **Infer** the type of the expression as a whole.
  - If a sub-expression has unexpected type, report a type error.
- At each *command*, **check** the type of any constituent expression.
  - If an expression has unexpected type, report a type error.

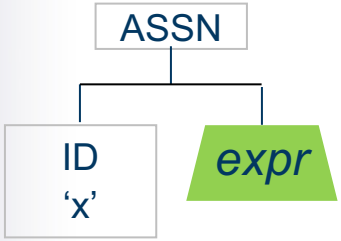
Example: Fun type checking

- Expression with binary operator:



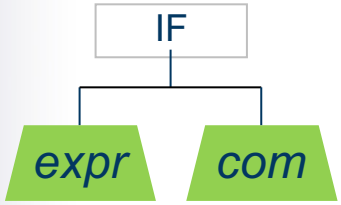
walk $expr_1$, and check that its type is INT;
walk $expr_2$, and check that its type is INT;
infer that the type of the whole expression is BOOL

- Assignment-command:



lookup 'x' and retrieve its type;
walk *expr* and note its type;
check that the two types are *equivalent*

- If-command:



walk *expr*, and check that its type is BOOL;
walk *com*

University of Glasgow

Contextual analysis with ANTLR

- Contextual analysis is done by walking over the syntax tree.
- We implement a visitor:

```
class FunCheckerVisitor extends
AbstractParseTreeVisitor<Type>
implements FunVisitor<Type> {

  SymbolTable<Type> typeTable; }
```

`SymbolTable<A>` is a table that records identifiers with attributes of type `A`.

- `Type` is the representation of Fun types. Visiting an expression returns the type of the expression.

University of Glasgow

FunChecker visitor *(1)*

```
Type visitNum(FunParser.NumContext ctx) {
   return Type.INT;
}

Type visitId(FunParser.IdContext ctx) {
   return retrieve(ctx.ID().getText(), ctx);
} // retrieve looks in typeTable.
  // ctx is used to get a file position for the
  // error message, if necessary
```

FunChecker visitor *(2)*

```
expr   :      e1=sec_expr
                ( op=(EQ | LT | GT) e2=sec_expr )?
       ;
Type visitExpr(FunParser.ExprContext ctx) {
   Type t1 = visit(ctx.e1);
   if (ctx.e2 != null) {
      Type t2 = visit(ctx.e2);
         return checkBinary(COMPTYPE, t1, t2, ctx);
         // COMPTYPE is INT x INT -> BOOL
         // checkBinary checks that t1 and t2 are INT
         // and returns BOOL.
         // If necessary it produces an error message.
   }
   else {
      return t1;
   }
}
```

FunChecker visitor *(3)*

```
com
        :       ID ASSN expr      # assn


Type visitAssn(FunParser.AssnContext ctx) {
   Type tvar = retrieve(ctx.ID().getText(), ctx);
   // retrieve looks in the type table.
   Type t = visit(ctx.expr());
   checkType(tvar, t, ctx);
   // checkType checks that tvar and t are the same type.
   // If not, it produces an error message, using ctx
   // to get the file position.
   return null; // Because it's a command not an
expression
}
```

FunChecker visitor *(4)*

```
com
        :       IF expr COLON c1=seq_com
                  ( DOT
                  | ELSE COLON c2=seq_com DOT
                  )                              # if

Type visitIf(FunParser.IfContext ctx) {
   Type t = visit(ctx.expr());
   visit(ctx.c1);
   if (ctx.c2 != null)
      visit(ctx.c2);
   checkType(Type.BOOL, t, ctx);
       // Condition must be BOOL
   return null;
}
```

FunChecker visitor *(5)*

```
seq_com
        :       com*    # seq
        ;


Type visitSeq(FunParser.SeqContext ctx) {
   visitChildren(ctx);
   return null;
}
// This is the same as the definition in FunBaseVisitor.
// If we declare FunCheckerVisitor extends FunBaseVisitor
// then we get these default definitions for free.
//
// For clarity, FunCheckerVisitor includes all definitions
// explicitly.
```

FunChecker visitor *(6)*

```
var_decl
        :       type ID ASSN expr          # var
        ;


Type visitVar(FunParser.VarContext ctx) {
   Type t1 = visit(ctx.type());
   // t1 will be INT or BOOL, representing the
   // declared type.
   Type t2 = visit(ctx.expr());
   define(ctx.ID().getText(), t1, ctx);
   // define adds an entry to the type table
   checkType(t1, t2, ctx);
   return null;
}
```

FunChecker visitor *(7)*

```
program
        :       var_decl* proc_decl+ EOF   # prog
        ;


Type visitProg(FunParser.ProgContext ctx) {
   predefine(); // Put read and write into the type table
   visitChildren(ctx);
   Type tmain = retrieve("main", ctx);
   checkType(MAINTYPE, tmain, ctx);
   // MAINTYPE is VOID -> VOID
   return null;
}
```

University of Glasgow

Driver program for the Fun syntactic and contextual analysers

- Running the Fun syntactic and contextual analysers:

```
public class FunCheck {

   public static void main (String[] args) {

      // Syntactic analysis:
      …
      ParseTree tree = parser.program();

      // Contextual analysis:
      FunCheckerVisitor checker =
         new FunCheckerVisitor(tokens);
      checker.visit(tree);
   }

}
```

University of Glasgow

Representing types

- To implement type checking, we need a way to represent the source language's types.
- We can use the concepts of §2:
  - primitive types
  - cartesian product types ($T_1 \times T_2$)
  - disjoint union types ($T_1 + T_2$)
  - mapping types ($T_1 \rightarrow T_2$)

University of Glasgow

Case study: Fun types *(1)*

- Represent Fun primitive data types by BOOL and INT.
- Represent the type of each Fun function by a mapping type:

`func` *T′* `f (`*T* `x): … .`	*T* → *T′*
`func` *T′* `f (): … .`	VOID → *T′*

- Similarly, represent the type of each Fun proper procedure by a mapping type:

`proc p (`*T* `x): … .`	*T* → VOID
`proc p (): … .`	VOID → VOID

University of Glasgow

Case study: Fun types *(2)*

- Represent the type of each Fun operator by a combination of product and mapping types:

`+  -  *  /`	(INT × INT) → INT
`==  <  >`	(INT × INT) → BOOL
`not`	BOOL → BOOL

University of Glasgow

Case study: implementation of Fun types *(1)*

- Outline of class `Type` :

```
public abstract class Type {
   public abstract boolean equiv (Type t);
   public class Primitive extends Type {
      …
   }
   public class Pair extends Type {
      …
   }
   public class Mapping extends Type {
      …
   }
}
```

University of Glasgow

Case study: implementation of Fun types *(2)*

- Subclass `Type.Primitive` has a field that distinguishes different primitive types.
- Class `Type` exports:

```
public static final Type
   VOID = new Type.Primitive(0),
   BOOL = new Type.Primitive(1),
   INT  = new Type.Primitive(2);
```

Case study: implementation of Fun types *(3)*

- Subclass `Type.Pair` has two `Type` fields, which are the types of the pair components. E.g.:

```
Type prod =
  new Type.Pair(Type.BOOL, Type.INT);
```

represents BOOL × INT

Case study: implementation of Fun types *(4)*

- Subclass `Type.Mapping` has two `Type` fields. These are the domain type and range type of the mapping type. E.g.:

```
Type proctype =
   new Type.Mapping(Type.INT, Type.VOID);

Type optype =
   new Type.Mapping(
      new Type.Pair(Type.INT, Type.INT),
   Type.BOOL);
```

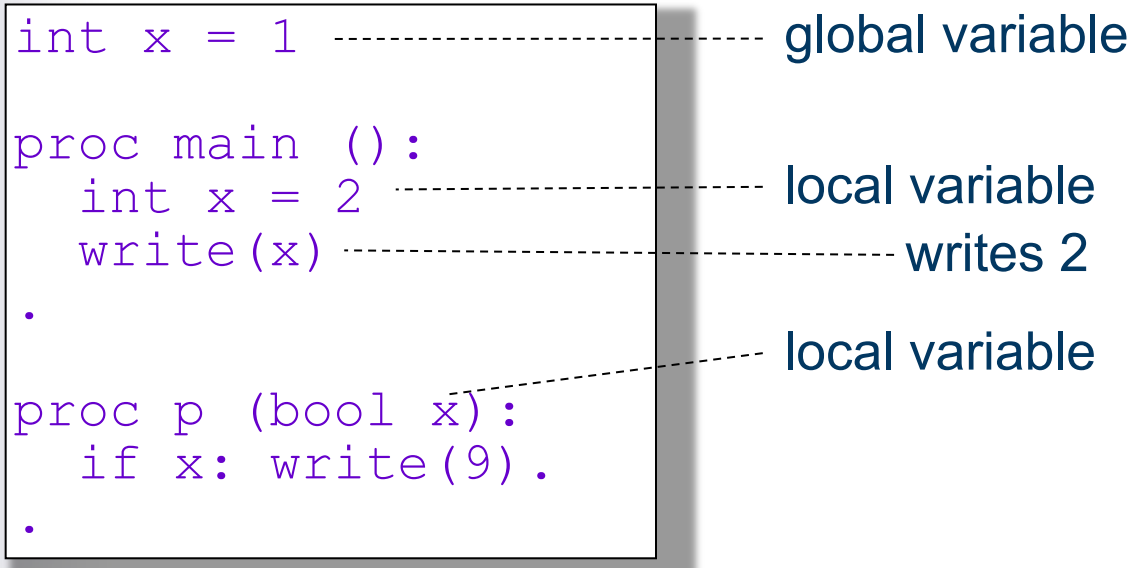
represents INT → VOID

represents (INT × INT) → BOOL

University of Glasgow

Representing scopes *(1)*

- Consider a PL in which all declarations are either *global* or *local*. Such a PL is said to have *flat block structure* (see §10).
- The same identifier can be declared both globally and locally. E.g., in Fun:



University of Glasgow

Representing scopes *(2)*

- The type table must distinguish between global and local entries.
- Global entries are *always* present.
- Local entries are present *only when analysing an inner scope*.
- At any given point during analysis of the source program, the same identifier may occur in:
  - at most one global entry, and
  - at most one local entry.
- Most PLs have nested scopes, not flat global/local scopes

Case study: Fun scopes *(1)*

- Type table during contextual analysis of a Fun program:

```
int x = 1

proc main ():
  int x = 2
  write(x)
.

proc p (bool x):
  if x: write(9).
.
```

global	‘x’	INT

global	‘x’	INT
global	‘main’	VOID → VOID
local	‘x’	INT

global	‘x’	INT
global	‘main’	VOID → VOID
global	‘p’	BOOL → VOID
local	‘x’	BOOL

University of Glasgow

Case study: Fun scopes *(2)*

- Such a table can be implemented by a pair of hash-tables, one for globals and one for locals:

```
public class SymbolTable<A> {

   // A SymbolTable<A> object represents a scoped
   // table in which each entry consists of an identifier
   // and an attribute of type A.

   private HashMap<String,A>
      globals, locals;

   public SymbolTable () {
      globals = new HashMap<String,A>();
      locals = null;   // Initially there are no locals.
   }
```

University of Glasgow

Case study: Fun scopes *(2)*

- Implementation in Java *(continued)*:

```
public void enterLocalScope () {
   locals = new HashMap<String,A>();
}

public void exitLocalScope () {
   locals = null;
}
```

University of Glasgow

Case study: Fun scopes *(3)*

- Implementation in Java *(continued)*:

```
public void put (String id, A attr) { … }
// Add an entry (id, attr) to the locals (if not null),
// otherwise add the entry to the globals.

public A get (String id) { … }
// Retrieve the attribute corresponding to id in
// the locals (if any), otherwise retrieve it from
// the globals.

}
```

- Now the type table can be declared thus:

```
SymbolTable<Type> typeTable;
```

Case study: Fun scopes *(4)*

```
proc_decl : PROC ID LPAR formal_decl RPAR COLON
          var_decl* seq_com DOT    # proc
```

```
Type visitProc(FunParser.ProcContext ctx {
   typeTable.enterLocalScope();
   Type t;
   FunParser.Formal_declContext fd = ctx.formal_decl();
   if (fd != null)
      t = visit(fd);
   else
      t = Type.VOID;
   Type proctype = new Type.Mapping(t, Type.VOID);
   define(ctx.ID().getText(), proctype, ctx); // To support recursion
   List<FunParser.Var_declContext> var_decl = ctx.var_decl();
   for (FunParser.Var_declContext vd : var_decl)
      visit(vd); // The local variables
   visit(ctx.seq_com()); // The body of the procedure
   typeTable.exitLocalScope();
   define(ctx.ID().getText(), proctype, ctx); // Define it globally
   return null; }
```