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) 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).

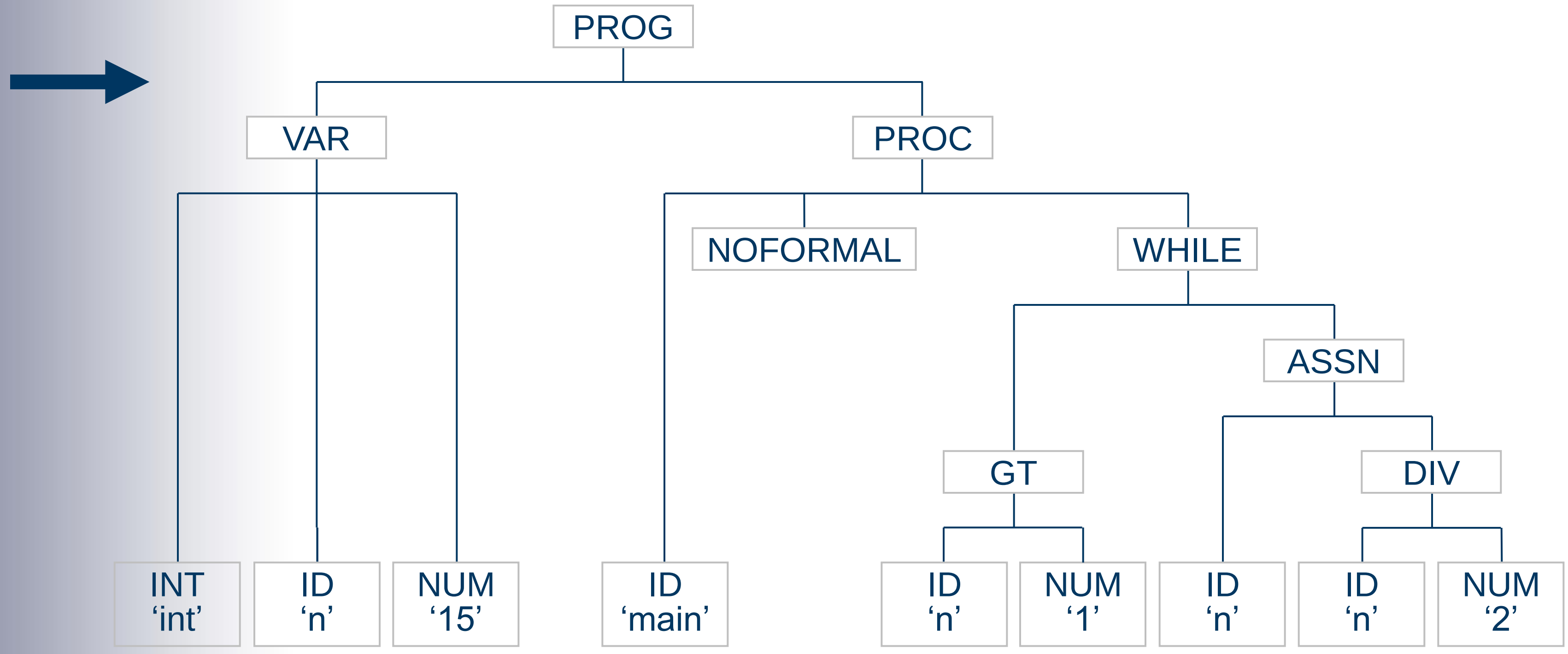
Example: Fun compilation *(1)*

- 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):

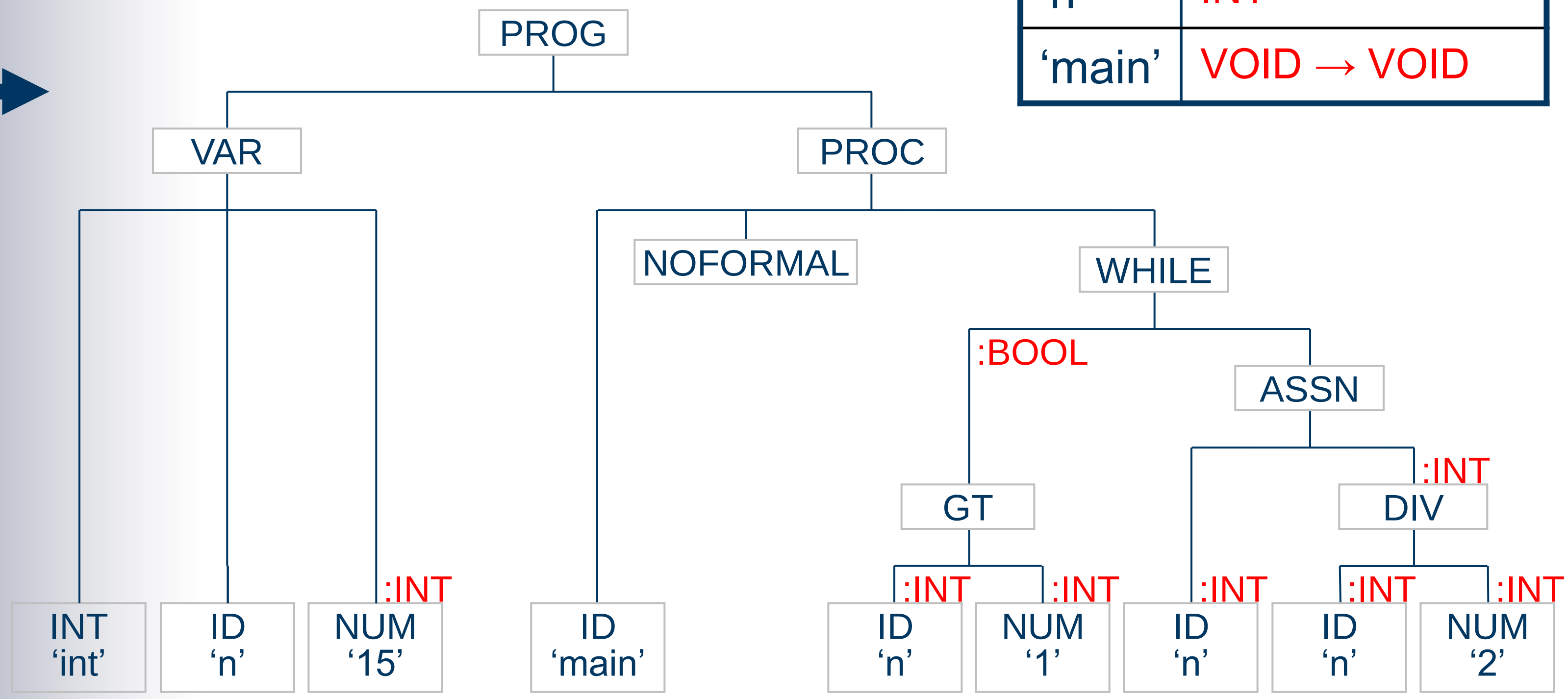


Example: Fun compilation *(3)*

- 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

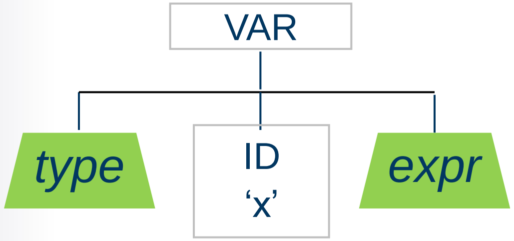
University of Glasgow

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.

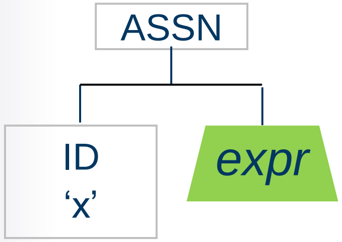
Example: Fun scope checking

- Declaration of a variable identifier:



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

- Use of a variable identifier:



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

Type checking *(1)*

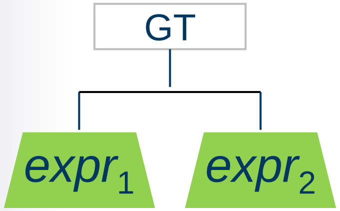
- **Type checking** is the process of checking that every command and expression is well-typed, i.e., free of type errors.
- *Note:* The compiler performs type checking only if the source language is statically-typed.

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.

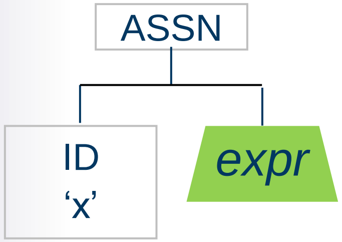
Example: Fun type checking

- Expression with binary operator:



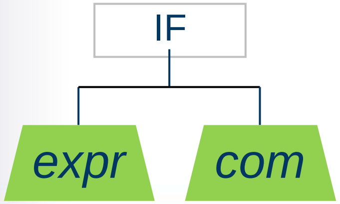
walk $expr_1$, and check that its type is INT;
walk $expr_2$, and check that its type is INT;
infer that the type of the whole expression is BOOL

- 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*

Contextual analysis with ANTLR *(1)*

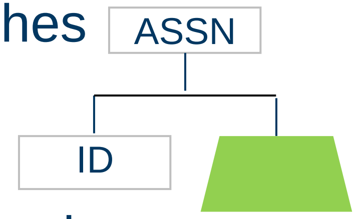
- In ANTLR we can write a “tree grammar” which describes the ASTs.
- Each rule in the tree grammar is a pattern match for part of the AST.
- From the tree grammar, ANTLR generates a depth-first left-to-right tree walker.
- We can enhance the tree grammar with actions to perform scope and type checking. ANTLR will insert these actions into the tree walker.
- *Important:* The position of an action determines when it will be performed during the tree walk.

Contextual analysis with ANTLR *(2)*

- Examples of AST pattern matches:

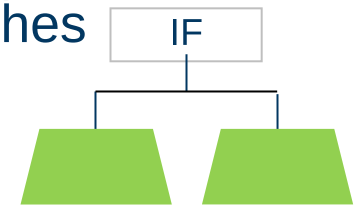
```
com
        :  ^(ASSN ID expr)

        |  ^(IF expr com)

        ;
```

This pattern matches



and makes `expr` refer to the right subtree.

This pattern matches



and makes `expr` and `com` refer to the left and right subtrees.

Case study: Fun tree grammar in ANTLR *(1)*

- Fun tree grammar *(outline)*:

```
tree grammar FunChecker;

options {
        tokenVocab = Fun;
        …;
}

prog
        :  ^(PROG var_decl* proc_decl+)
        ;
```

Case study: Fun tree grammar in ANTLR *(2)*

- Fun tree grammar *(continued)*:

```
var_decl
    :  ^(VAR type ID expr)
    ;

type
    :  BOOL
    |  INT
    ;
```

Case study: Fun tree grammar in ANTLR *(3)*

- Fun tree grammar *(continued)*:

```
com
        :  ^(ASSN ID expr)
        |  ^(IF expr com)
        |  ^(SEQ com*)
        |  …
        ;
```

Case study: Fun tree grammar in ANTLR *(4)*

- Fun tree grammar *(continued)*:

```
expr
      :  NUM
      |  ID
      |  ^(EQ expr expr)
      |  ^(PLUS expr expr)
      |  ^(NOT expr)
      |  …
      ;
```

Case study: Fun tree grammar with contextual analysis actions *(1)*

- Fun tree grammar with actions *(outline)*:

```
tree grammar FunChecker;

options {
      tokenVocab = Fun;
      …;
}

@members {
      private SymbolTable<Type> typeTable;
      …
}
```

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

Case study: Fun tree grammar with contextual analysis actions *(2)*

- Fun tree grammar with actions *(continued)*:

```
expr                    returns [Type typ]
        :  NUM
                        {  set $typ to INT;  }
        |  ID
                        {  lookup the identifier in type-
                             Table, and let its type be t;
                           set $typ to t;  }
        |  ^(EQ
              t1=expr   //check the left expr
              t2=expr   //check the right expr
             )          {  check that t1 and t2 are INT;
                           set $typ to BOOL;}
        |  …
```

Case study: Fun tree grammar with contextual analysis actions *(3)*

- Fun tree grammar with actions *(continued)*:

```
|  ^(PLUS
      t1=expr       //check the left expr
      t2=expr       //check the right expr
     )              { check that t1 and t2 are INT;
                      set $typ to INT;  }
|  ^(NOT
      t=expr        //check the expr
     )              { check that t is BOOL;
                      set $typ to BOOL;  }
|  …
;
```

Case study: Fun tree grammar with contextual analysis actions *(4)*

- Fun tree grammar with actions *(continued)*:

```
com
        :  ^(ASSN
              ID
              t=expr       //check the expr
             )             { lookup the identifier in type-
                              Table, and let its type be ti;
                             check that ti is t; }
        |  …
```

Case study: Fun tree grammar with contextual analysis actions *(5)*

- Fun tree grammar with actions *(continued)*:

```
|  ^(IF
      t=expr          //check the expr
      com             //check the com
    )                 { check that t is BOOL; }
|  ^(SEQ
      com*            //check the com*
    )
|  …
;
```

Case study: Fun tree grammar with contextual analysis actions *(6)*

- Fun tree grammar with actions *(continued)*:

```
var_decl
      :  ^(VAR
           t1=type
           ID
           t2=expr    //check the expr
          )           { put the identifier into
                        typeTable along with t1;
                        check that t1 is t2; }
      ;
```

Case study: Fun tree grammar with contextual analysis actions *(7)*

- Fun tree grammar with actions *(continued)*:

```
type                  returns [Type typ]
      :  BOOL         {  set $typ to BOOL;  }
      |  INT          {  set $typ to INT;  }
      ;
```

Case study: Fun tree grammar with contextual analysis actions *(8)*

- Fun tree grammar with actions *(continued)*:

```
prog
        :   ^(PROG
                        { put 'read' and 'write' with their
                          types into typeTable; }
              var_decl* //check the var_decl*
              proc_decl+//check the proc_decl+
             )          { check that 'main' is in
                          typeTable and has type
                          VOID → VOID; }
        ;
```

Case study: Fun syntactic and contextual analysers *(1)*

- Put the above tree grammar in a file named `FunChecker.g`.
- Feed this as input to ANTLR:

```
…$ java org.antlr.Tool FunChecker.g
```

- ANTLR generates a class `FunChecker` containing methods that walk the AST and perform the contextual analysis actions.

Case study: Fun syntactic and contextual analysers *(2)*

- Program to run the Fun syntactic and contextual analysers:

```
public class FunRun {
   public static void main (String[] args) {
      // Syntactic analysis:
      …
      CommonTree ast = (CommonTree)
         parser.prog().getTree();
      // Contextual analysis:
      FunChecker checker =
         new FunChecker(
            new CommonTreeNodeStream(ast));
      checker.prog();
   }
}
```

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 ($T_1 \to T_2$)

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): … .`	$T \rightarrow T'$
`func` *T'* `f (): … .`	VOID $\rightarrow T'$

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

`proc p (`*T* `x): … .`	$T \rightarrow$ VOID
`proc p (): … .`	VOID $\rightarrow$ VOID

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

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 {
      …
   }
}
```

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 *(2)*

- 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

University of Glasgow

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

- 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);
```

represents INT → VOID

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

represents (INT × INT) → BOOL

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:

```
int x = 1

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

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

- `int x = 1` — global variable
- `int x = 2` — local variable
- `write(x)` — writes 2
- `bool x` — local variable

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.

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

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;
}
```

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;
```

University of Glasgow

Case study: Fun scopes *(4)*

- In the Fun tree grammar (simplified):

```
proc_decl
      :   ^(PROC
            ID
                        {enter local scope in
                          typeTable; }
            t=formal
            var_decl*   //check the var_decl*
            com         //check the com
           )            {exit local scope in typeTable;
                         put the identifier into
                          typeTable with t → VOID; }
      |   …
      ;
```