Programming Languages H

Dr Ornela Dardha

University of Glasgow

5 Compilation

- Overview
- Compilation phases
  - syntactic analysis
  - contextual analysis
  - code generation
- Abstract syntax trees
- Case study: Fun language and compiler

University of Glasgow

Overview

- A ***S*** → ***T*** **compiler** translates a *source program* in *S* to *object code* in *T*, if it conforms to the source language's syntax and scope/type rules.
- This suggests a decomposition of the compiler into three phases:
  - syntactic analysis
  - contextual analysis
  - code generation.

University of Glasgow

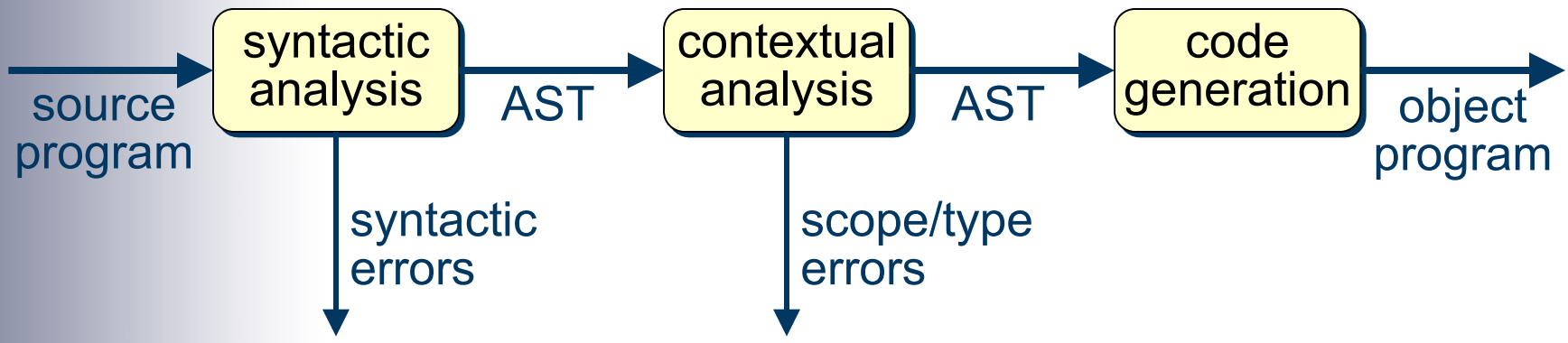
Compilation phases *(1)*

- **Syntactic analysis**: *Parse* the *source program* to check whether it is well formed, and to determine its phrase structure, in accordance with the source language's *syntax*.
- **Contextual analysis**: *Analyze* the *parsed program* to check whether it conforms to the source language's *scope rules* and *type rules*.
- **Code generation**: *Translate* the *parsed program* to object code, in accordance with the source language's *semantics*.

University of Glasgow

Compilation phases *(2)*

- Data flow between phases:



- An AST (abstract syntax tree) is a convenient way to represent a source program after syntactic analysis *(see later for details)*.

University of Glasgow

Case study: Fun language *(1)*

- **Fun** is a simple imperative language.
- A Fun program declares some global variables and some procedures/functions, always including a procedure named `main()`.
- A Fun procedure/function may have a single parameter. It may also declare local variables. A function returns a result; not a procedure.
- Fun has two data types, `bool` and `int`.
- Fun com: assignment, proc/func call, if-command, while-command, sequential command.

University of Glasgow

Case study: Fun language *(2)*

- Sample Fun program:

```
func int fact (int n): # returns n!
   int f = 1
   while n > 1:
      f = f*n
      n = n-1 .
   return f .

proc main ():
   int num = read()
   write(num)
   write(fact(num)) .
```

- Fun programs are free-format: spaces, tabs, and EOLs (ends-of-lines) are not significant.

University of Glasgow

Case study: Fun language *(3)*

- Fun syntax *(extracts)*:

prog	=	*var-decl* * *proc-decl* + *eof*
var-decl	=	*type id* ‘=’ *expr*
type	=	‘`bool`’
	|	‘`int`’
com	=	*id* ‘=’ *expr*
	|	‘`if`’ *expr* ‘`:`’ *seq-com* ‘`.`’
	|	…
seq-com	=	*com* *

Case study: Fun language *(4)*

- Fun syntax *(continued)*:

expression ---------- *expr* = *sec-expr* …

secondary-expression ----- *sec-expr* = *prim-expr*
((‘+’ | ‘–’ | ‘*’ | ‘/’) *prim-expr*) *

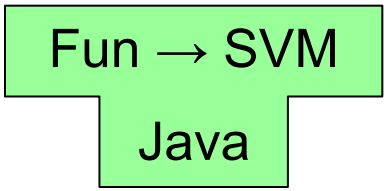
primary-expression ------- *prim-expr* = *num*
| *id*
| ‘(’ *expr* ‘)’
| …

- For a full description, see *Fun Specification* (available from the PL Moodle page).

University of Glasgow

Case study: Fun compiler *(1)*

- The Fun compiler generates SVM code. It is expressed in Java:



- This contains the following classes:
  - syntactic analyser (`FunLexer`, `FunParser`)
  - contextual analyser (`FunChecker`)
  - code generator (`FunEncoder`).

University of Glasgow

Case study: Fun compiler *(2)*

- The compiler calls each of these in turn:
  - The syntactic analyser *lexes* and *parses* the source program, printing any error messages, and *generates* an AST. Then the AST is printed.
  - The contextual analyser performs *scope/type checking*, printing any error messages.
  - The code generator emits object code into the SVM code store. Then the object code is printed.
- Compilation is terminated after syntactic or contextual analysis if any errors are detected.

University of Glasgow

Case study: Fun driver

- The driver `FunRun` does the following:
  - It compiles the source program into an SVM object program.
  - If no errors are detected, it calls the SVM interpreter to run the object program.

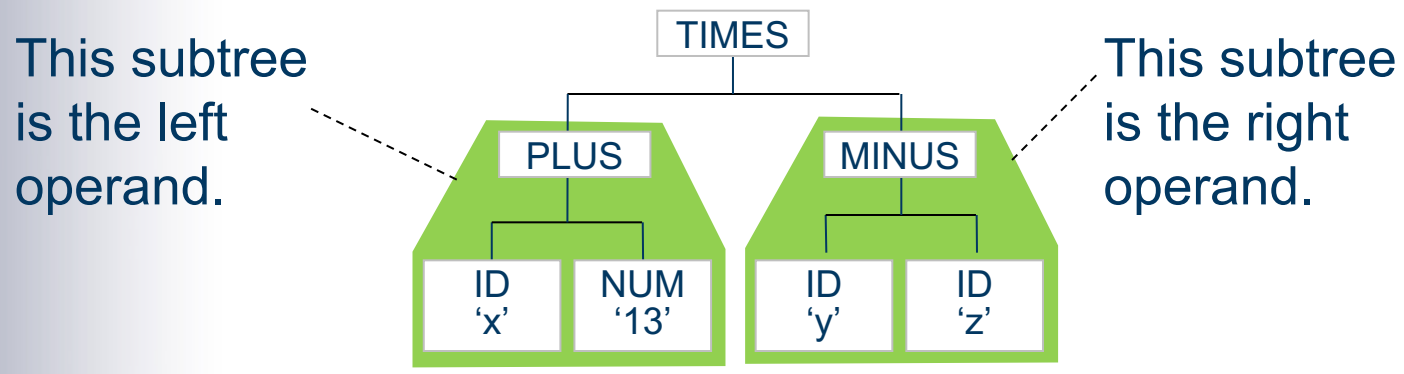
University of Glasgow

Abstract syntax trees

- An **abstract syntax tree** (**AST**) is a convenient way to represent a source program's phrase structure.
- Structure of an AST:
  - Each *leaf node* represents an identifier or literal.
  - Each *internal node* corresponds to a source language construct (e.g., a variable declaration or while-command). The internal node's subtrees represent the parts of that construct.
- ASTs are much more compact than *syntax trees* (§1).

Example: AST for Fun expression

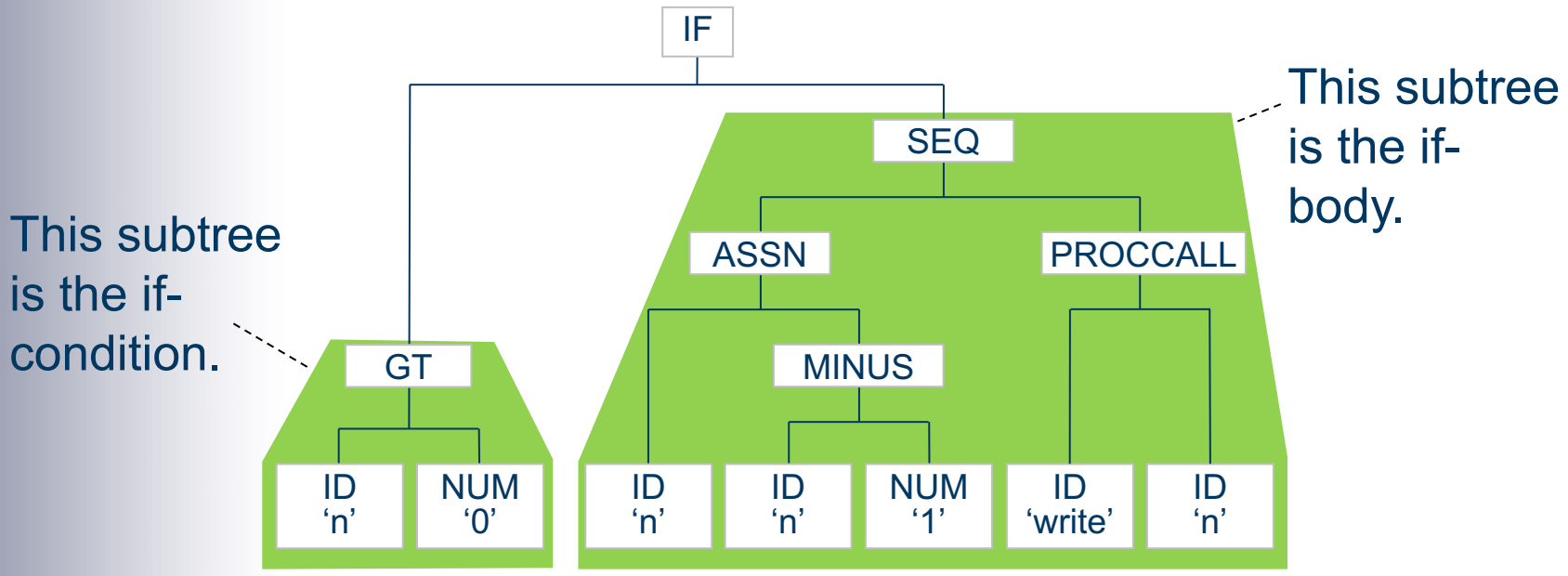
- AST for expression '`(x+13)*(y-z)`':



- *Note:* The AST makes no distinction between *expr*s, *sec-expr*s, etc.: they are all just expressions.

Example: AST for Fun command

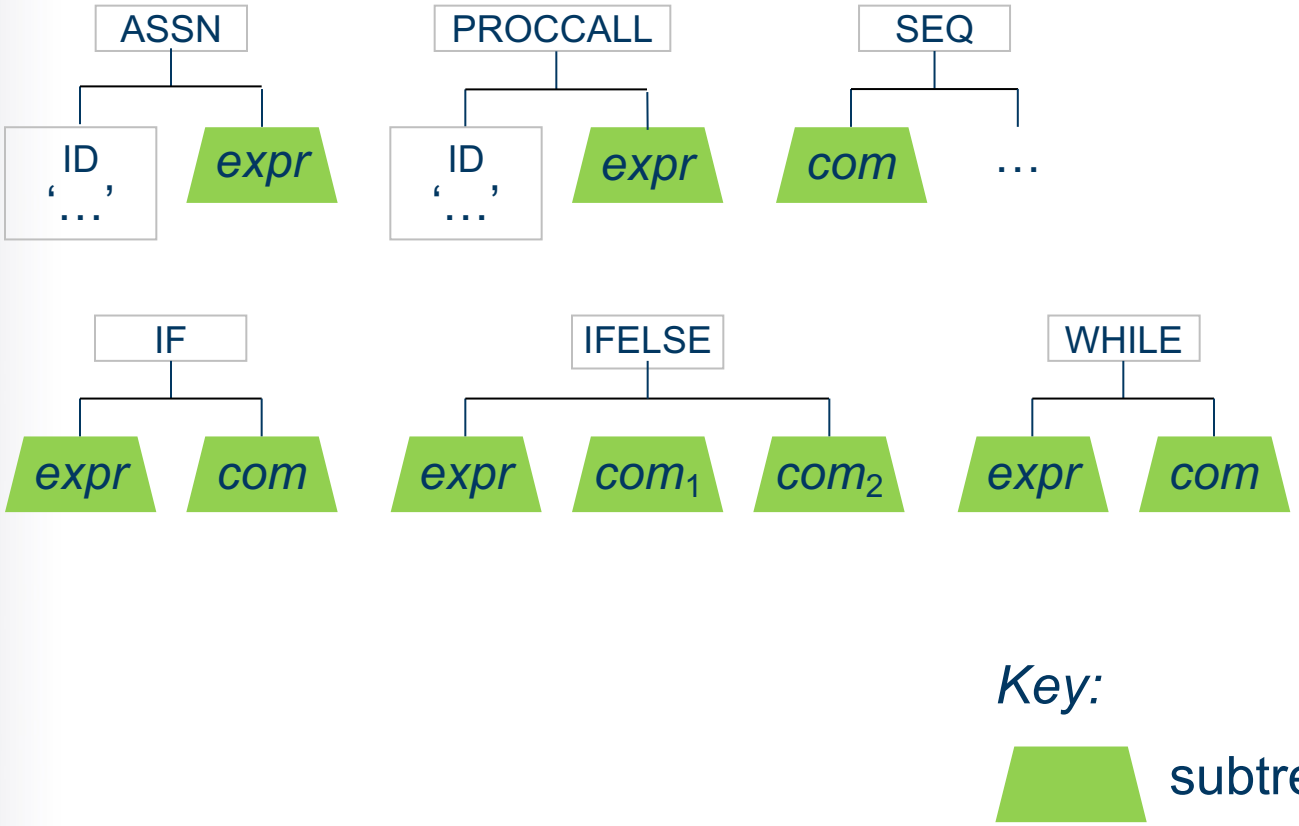
- AST for '`if n>0: n = n-1 write(n).`':



- *Note:* The AST makes no distinction between *com*s and *seq-com*s: they are all just commands.

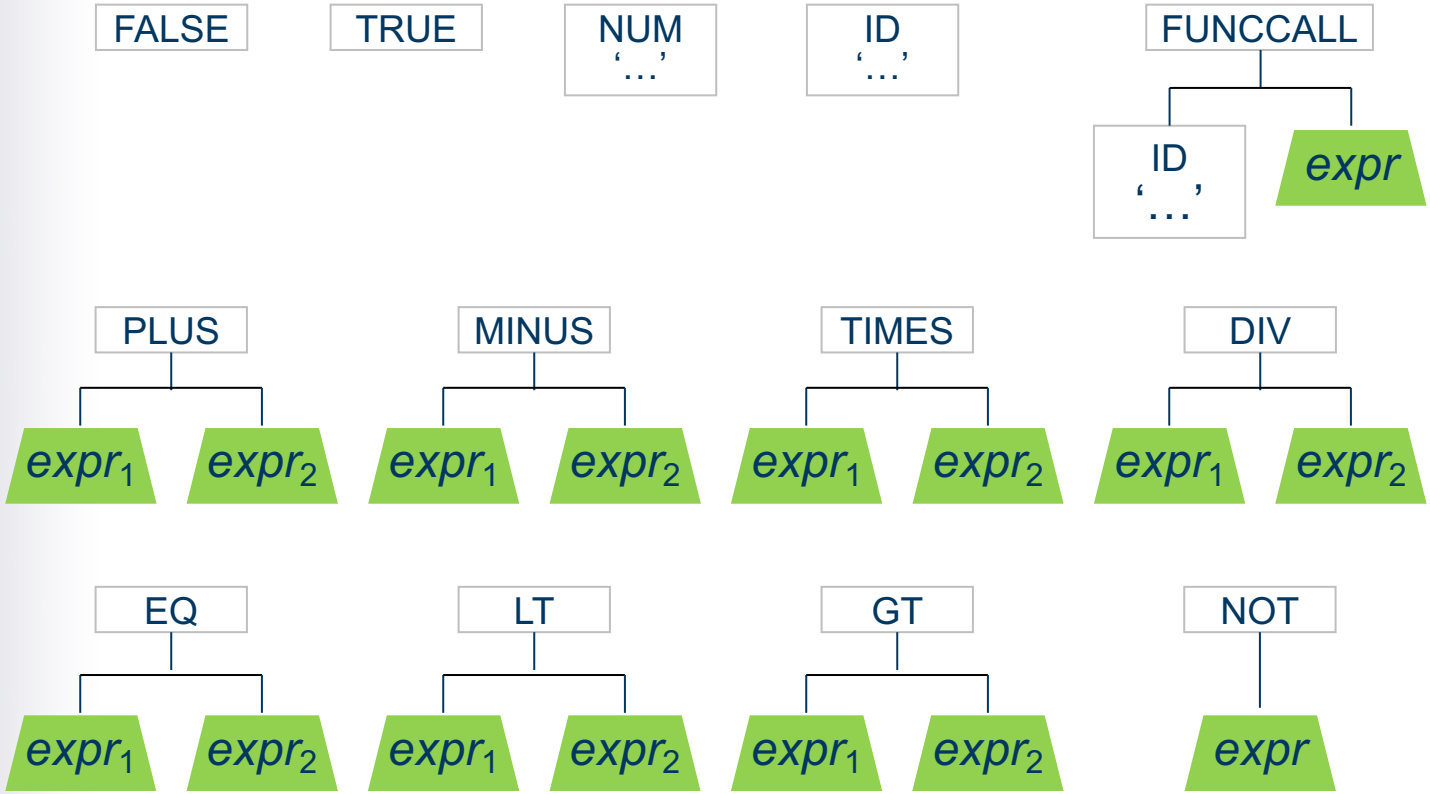
Case study: summary of Fun ASTs *(1)*

- ASTs for Fun commands (*com*):



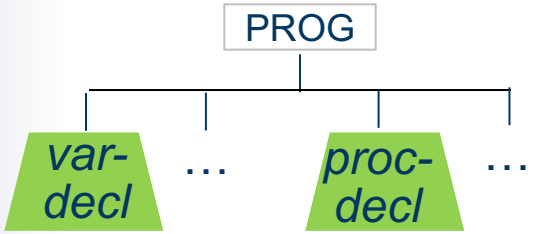
Case study: summary of Fun ASTs *(2)*

- ASTs for Fun expressions (*expr*):



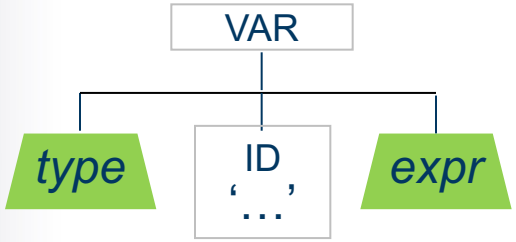
Case study: summary of Fun ASTs *(3)*

- AST for Fun programs:



Case study: summary of Fun ASTs *(4)*

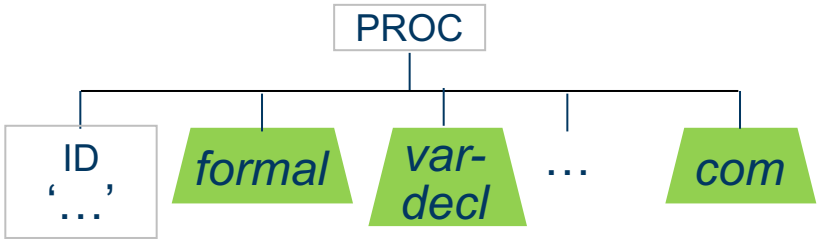
- ASTs for Fun variable declarations (*var-decl*):

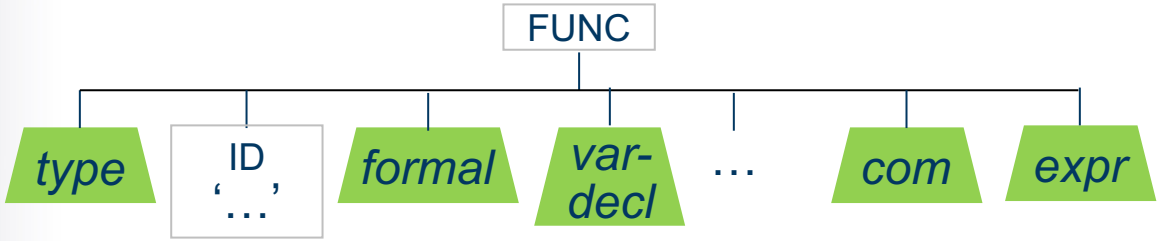


- ASTs for Fun types (*type*):

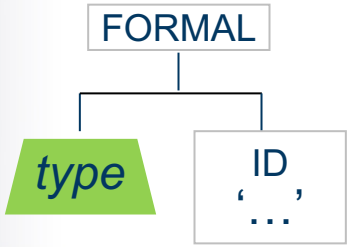
Case study: summary of Fun ASTs *(5)*

- ASTs for Fun procedure declarations (*proc-decl*):





- ASTs for Fun formal parameters (*formal*):



University of Glasgow

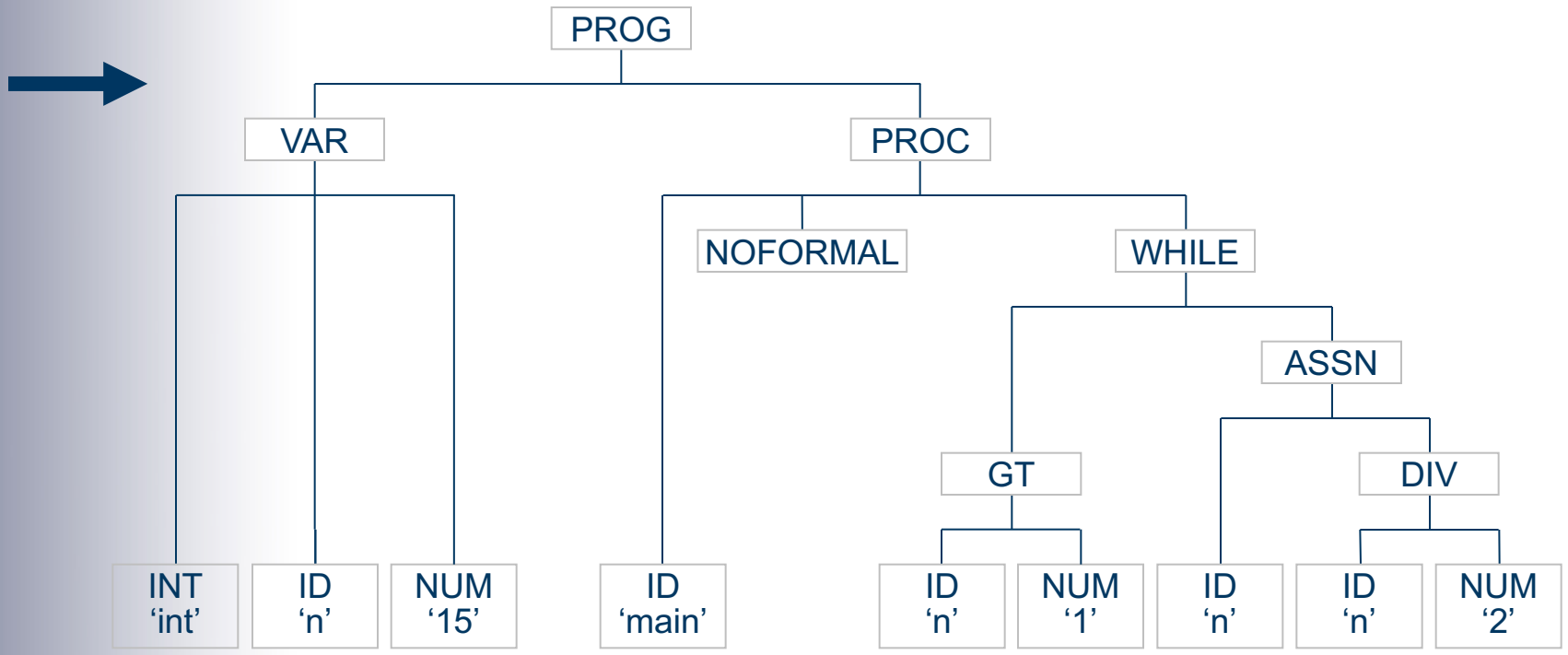
Example: Fun compilation *(1)*

- Source program:

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

Example: Fun compilation *(2)*

- AST after syntactic analysis (slightly simplified):

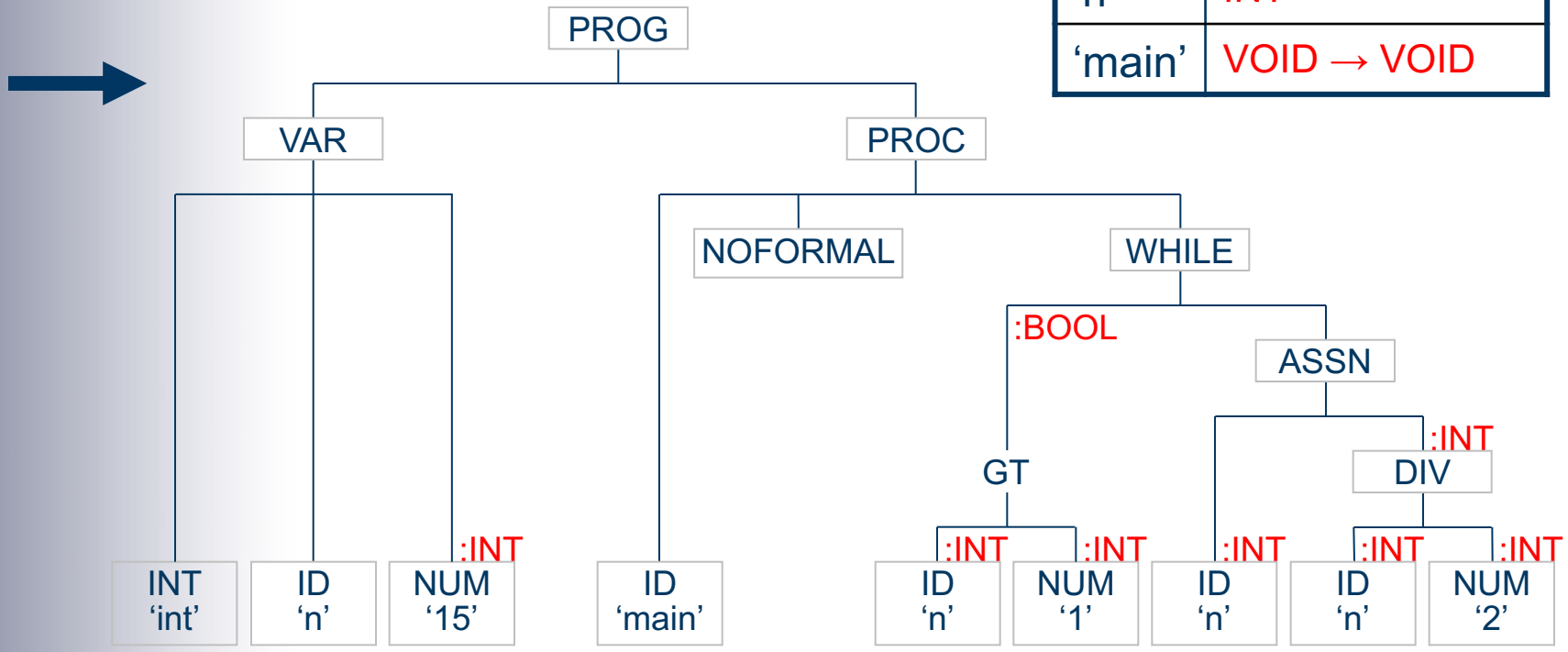


Example: Fun compilation *(3)*

- AST after contextual analysis:

Type table

‘n’	INT
‘main’	VOID → VOID



Example: Fun compilation *(3)*

- SVM object code after code generation:

```
 0: LOADC 15
 3: CALL 7
 6: HALT
 7: LOADG 0
10: LOADC 1
13: COMPGT
14: JUMPF 30
17: LOADG 0
20: LOADC 2
23: DIV
24: STOREG 0
27: JUMP 7
30: RETURN 0
```

Address table

‘n’	0 (global)
‘main’	7 (code)