Compilation (#5): AST and Typing.

C. Deleuze & L. Gonnord

Grenoble INP/Esisar

2022-2023

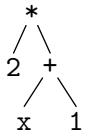
Objective

- Construction of our first intermediate representation for programs.
- Typing.

A bit about syntax

The texts: `2*(x+1)` and `(2 * ((x) + 1))` and `2 * /* comment */ ( x + 1 )`

have the same semantics ▶ they should have the **same internal representation**.



Example: syntax of expressions

The (abstract) grammar of arithmetic expressions is (avoiding parenthesis, syntactic sugar . . .):

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
 & | & x & \textit{variable} \\
 & | & e + e & \textit{addition} \\
 & | & e \times e & \textit{multiplication} \\
 & | & \ldots &
\end{array}
$$

Remark : to properly define the semantics of the expression, it is sufficient to define $\mathcal{A}(e)$.

AST Definition (Wikipedia is your friend!)

In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of text (often source code) written in a formal language. Each node of the tree denotes a construct occurring in the text.

The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. For instance, grouping parentheses are implicit in the tree structure, so these do not have to be represented as separate nodes.

Warning

An AST is not! a derivation tree.

Give an example

Content

- Interpret with semantic actions : in the previous course.
- 2 other techniques here.

From grammar to an AST

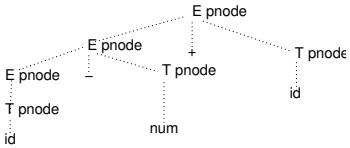
$$E \rightarrow E + T \mid E - T$$
$$E \rightarrow T$$
$$T \rightarrow (E)$$
$$T \rightarrow \textbf{id} \mid \textbf{nb}$$

```
node *create_node(top op, node *left, node *right)
node *create_leaf_id(int num)
node *create_leaf_nb(int value)
```

- **Attribution ?**

AST construction with attributes

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	E.pnode := create_node('+', E_1.pnode, T.pnode)
$E \rightarrow E_1 - T$	E.pnode := create_node('–', E_1.pnode, T.pnode)
$E \rightarrow T$	E.pnode := T.pnode
$T \rightarrow (E)$	T.pnode := E.pnode
$T \rightarrow$ **id**	T.pnode := create_leaf_id(**id**.num)
$T \rightarrow$ **nb**	T.pnode := create_leaf_nb(**nb**.val)



Example : expr to AST (in ANTLR w/Java backend)

AST Construction

```
prog returns [ AExpr e ] : expr EOF { $e=$expr.e; } ;

// We create an AExpr instead of computing a value
expr returns [ AExpr e ] :
| INT { $e=new AInt($INT.int); }
| LPAR x=expr RPAR { $e=$x.e; } // Parenthesis not represented in AST
| e1=expr PLUS e2=expr { $e=new APlus($e1.e,$e2.e); }
| e1=expr MINUS e2=expr { $e=new AMinus($e1.e,$e2.e); }
;
```

AST construction (main)

```
AExpr expr = parser.prog().e; // get the AST in var expr
// everything is ready to eval interpret (by tree traversal)
```

Example in Java - eval function

Evaluation is an eval function per class:

AExpr.java

```
public abstract class AExpr {
    abstract int eval(); // need to provide semantics
}
```

APlus.java

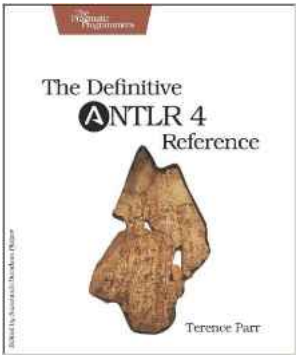
```
public class APlus extends AExpr {
    AExpr e1,e2;
    public APlus (AExpr e1,AExpr e2) { this.e1=e1; this.e2=e2; }
    // semantics below
    int eval() { return (e1.eval()+e2.eval()); }
}
```

Principle - OO programming

The visitor design pattern is a way of separating an algorithm from an object structure on which it operates.[...] In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function.

https://en.wikipedia.org/wiki/Visitor_pattern

Application



Designing interpreters / tree traversal in ANTLR-Python

- The ANTLR compiler generates a Visitor class.
- We override this class to traverse the parsed instance.

Arit Example with ANTLR/Python 1/3

AritParser.g4

```
expr: expr mdop=(MULT | DIV) expr #multiplicationExpr
    | expr pmop=(PLUS | MINUS) expr #additiveExpr
    | atom #atomExpr
    ;

atom: INT #int
    | ID #id
    | '(' expr ')' #parens
    ;
```

- compilation with `-Dlanguage=Python3 -visitor`

Arit Example with ANTLR/Python 2/3 -generated file

AritVisitor.py (generated)

```
class AritVisitor(ParseTreeVisitor):
...
    # Visit a parse tree produced by AritParser#multiplicationExpr.
    def visitMultiplicationExpr(self, ctx):
        return self.visitChildren(ctx)


    # Visit a parse tree produced by AritParser#atomExpr.
    def visitAtomExpr(self, ctx):
        return self.visitChildren(ctx)
..
```

Arit Example with ANTLR/Python 3/3

Visitor class overriding to write the interpreter:

MyAritVisitor.py

```
class MyAritVisitor(AritVisitor):

    def visitInt(self, ctx):
        return int(ctx.getText())

    def visitMultiplicationExpr(self, ctx):
        leftval = self.visit(ctx.expr(0))
        rightval = self.visit(ctx.expr(1))
        if ctx.mdop.type == AritParser.MULT:
            return leftval * rightval
        else:
            return leftval / rightval
```

Arit Example with ANTLR/Python - Main

And now we have a full interpret for arithmetic expressions!

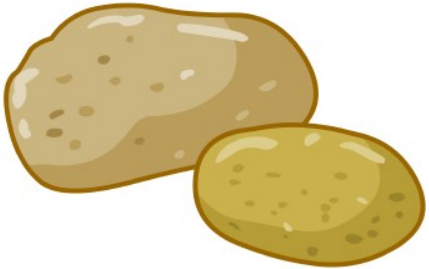
arit.py (Main)

```
lexer = AritLexer(InputStream(sys.stdin.read()))
stream = CommonTokenStream(lexer)
parser = AritParser(stream)
tree = parser.prog()
print("I'm here : nothing has been done")

visitor = MyAritVisitor()
visitor.visit(tree)
```

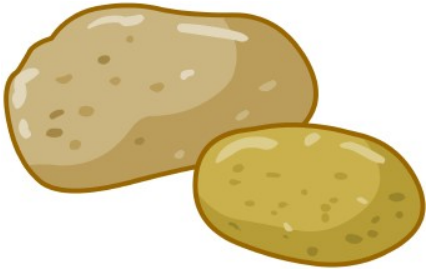
Typing

+



Typing

/



Typing

If you write: `"5432" + 1`

what do you want to obtain

- a compilation error? (OCaml)
- an exec error? (Python)
- the int 5433? (Visual Basic, PHP)
- the string `"54321"`? (Java)
- (too insane to put on a slide) (C, C++)
- anything else?

and what about `5432 / "1"` ?

Typing

When is

```
e1 + e2
```

legal, and what is its semantics?

- Typing: an analysis that gives a type to each subexpression, and reject incoherent programs.

When

- Dynamic typing (during execution): Lisp, PHP, Python
- Static typing (at compile time, after lexing+parsing): C, Java, OCaml

▶ Here: the second one.

Slogan

well typed programs do not go wrong

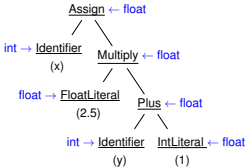
Typing objectives

- Should be **decidable**.
- It should reject programs like `(1 2)` in OCaml, or `1+"toto"` in C before an actual error in the evaluation of the expression: this is **safety**.

  The type system is related to the kind of error to be detected: **operations on basic types** / method invocation (message not understood) / correct synchronisation (e.g. session types) in concurrent programs / ...
- The type system should be expressive enough and not reject too many programs. (**expressivity**)

Principle

All sub-expressions of the program must be given a type



What does the programmer write?

- The type of all sub-expressions (like above) easy to verify, but tedious for the programmer
- Annotate only variable declarations (Pascal, C, Java, . . .)

```
{int x, y; x = 2.5 * (y + 1);}
```

- Only annotate function parameters (Scala)

```
def toto(y : Int) { var x = 2.5 * (y + 1) }
```

- Annotate nothing: complete inference : Ocaml, Haskell, . . .

```
# let foo y = 2 * (y + 1);;
val foo : int -> int = <fun>
```

Properties

- correction: “yes” implies the program is well typed.
- completeness: the converse.

(optional)

- principality : The most general type is computed.

What is a good output for a type-checker?

- We do not want:

  `failwith "typing error"`

  the origin of the problem should be clearly stated.
- We keep the types for next phases.

In practice

- Input: Trees are decorated by source code lines (and columns).
- Output: Trees are decorated by types in addition.

Mini-While (Abstract) Syntax

Expressions:

e	$::=$	c	*constant*
	$\mid$	x	*variable*
	$\mid$	$e + e$	*addition*
	$\mid$	$e \times e$	*multiplication*
	$\mid$	...	

Mini-while:

$S(Smt)$	$::=$	$x := expr$	assign
	$\mid$	$skip$	do nothing
	$\mid$	$S_1; S_2$	sequence
	$\mid$	`if` b `then` S_1 `else` S_2	test
	$\mid$	`while` b `do` S `done`	loop

Some vocabulary

We will define how to compute **typing judgements** denoted by:

$$\Gamma \vdash e : \tau$$

and means "in environment Γ, expression e has type τ" (here, $\tau \in \{int, bool\}$).

- Γ associates a type $\Gamma(x)$ to all declared variables x (that may or may not appear in e).

```
{
    int x = 42, y;
    float z;
}
```

$\rightsquigarrow$

$$\{ \quad x \rightarrow int, \quad y \rightarrow int, \quad z \rightarrow float \quad \}$$

Typing rules format

We will use typing rules to prove these typing judgments (the fact that it is correct to do so is admitted). Typing rules are of the form $\frac{P_1 \dots P_n}{B}$ and mean $P_1 \wedge \dots P_n \Rightarrow B$.

Typing rules for expr

Here types are basic types: {`int`, `bool`}

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \texttt{int}} \quad (\text{or } \mathrm{tt}\colon \mathrm{bool}, \ldots)$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

and plenty of similar rules.

Typing: an example

Considering $\Gamma = \{x_1 \mapsto int\}$, prove $\Gamma \vdash x1 + 9 : int$.

Typing rules for statements: $\Gamma \vdash S : \texttt{void}$

A statement S is well-typed in environment Γ, written: $\Gamma \vdash S : \texttt{void}$

on board!

- Example with expressions
- Guess/construct typing rules for assignment

Typing Mini-While: recap

$$\frac{c \in \mathbb{Z}}{\Gamma \vdash c : \texttt{int}} \qquad \frac{\Gamma(x) = t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x : t}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash S_1 : \texttt{void} \quad \Gamma \vdash S_2 : \texttt{void}}{\Gamma \vdash S_1; S_2 : \texttt{void}} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash x : t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x = e : \texttt{void}}$$

$$\frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S_1 : \texttt{void} \quad \Gamma \vdash S_2 : \texttt{void}}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \texttt{void}}$$

Typing: an example

Considering $\Gamma = \{x_1 \mapsto int\}$, prove that the given sequence of instructions is well typed (with a correct proof tree using proof rules).

```
x1 = 3 ;
x1 = x1+9 ;
```

on board!

You can have a look at my YT video : https://youtu.be/2A-hQy_6YIE?t=808 at 13min30

Hybrid expressions

What if we have `1.2 + 42` ?

- reject?
- compute a float?

Hybrid expressions

What if we have `1.2 + 42` ?

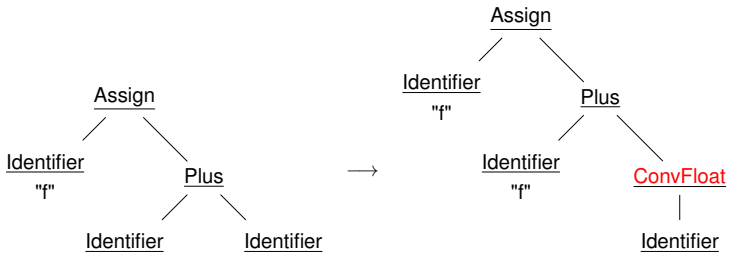
- reject? OCaml
- compute a float? Most mainstsream languages, e.g. C, Python, Java, etc.

Hybrid expressions

What if we have `1.2 + 42` ?

- reject? OCaml
- compute a float? Most mainstsream languages, e.g. C, Python, Java, etc.

▶ This is **type coercion**. Clean way to implement it: modify the AST to make the conversion explicit:



More complex expressions

What if we have types `pointer of bool`, or `array of int`? We might want to check equivalence (for addition ...).

- This is called **structural equivalence** (see Dragon Book, "type equivalence").

This is solved by a basic graph traversal checking that each element are equivalent/compatible.

Principle

- Γ is constructed going through the declaration part of the AST (may raise typing errors in case of double declarations for example)
- Γ is used to typecheck the program itself (e.g complain about type mismatch in an assignment or an expression, ...)
- One really wants typechecking on the AST, not the concrete program

Type Checking with a Visitor

MiniCTypingVisitor.py, rule for atoms

```
# In MiniC.g4 :
# expr : atom #atomExpr
#
# atom: OPAR expr CPAR #parExpr
# | INT #intAtom
# ...
def visitAtomExpr(self, ctx):
    return self.visit(ctx.atom())
```

Type Checking with a Visitor

MiniCTypingVisitor.py, rule for "or"

```
# In MiniC.g4 :
# expr: expr OR expr #orExpr
def visitOrExpr(self, ctx):
    ltype = self.visit(ctx.expr(0))
    rtype = self.visit(ctx.expr(1))
    if BaseType.Boolean == ltype and BaseType.Boolean == rtype:
        return BaseType.Boolean
    else:
        self._raise(ctx, 'boolean operands', ltype, rtype)
```

In practice for MiniC (lab sessions)

- No annotation is added to the AST (MiniC is simple enough, they are not needed)
- Typing is given (may serve as an example visitor ...)

Summary

1. Abstract syntax, Abstract syntax tree
   - The notion of AST
2. Another view on interpreters
   - AST construction with grammar attributions
   - Interpreter with implicit AST
3. Typing
   - Generalities about typing
   - Simple Type Checking for Mini-While
   - A bit of implementation (for expr)