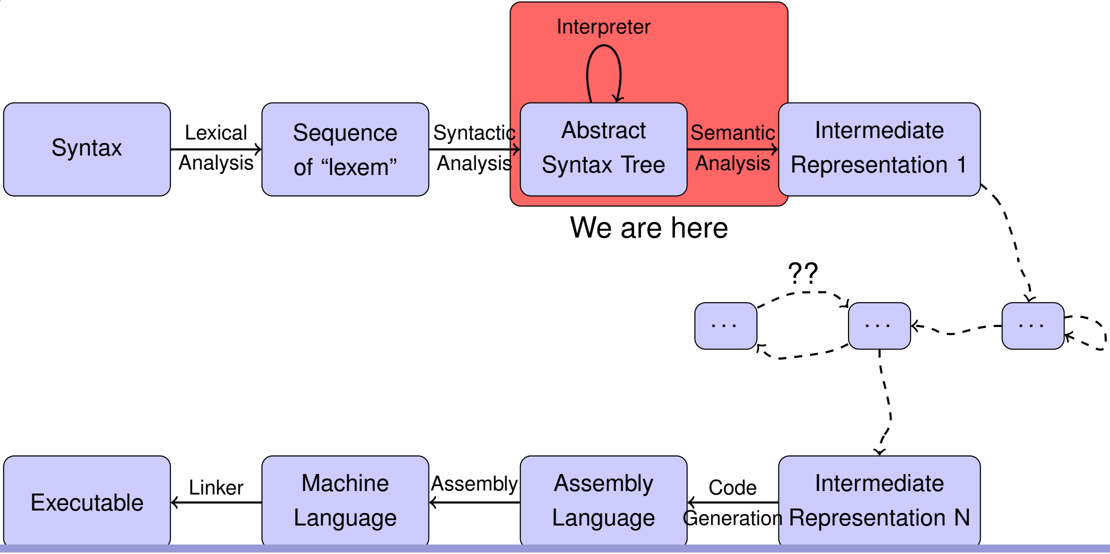
Compilation and Program Analysis (#3): Semantics, Interpreters from theory to practice.

Laure Gonnord & Matthieu Moy & Gabriel Radanne & other
https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2024-2025

Big Picture



Interpreter
Syntax
Lexical Analysis
Sequence of “lexem”
Syntactic Analysis
Abstract Syntax Tree
Semantic Analysis
Intermediate Representation 1
We are here
??
...
...
...
Executable
Linker
Machine Language
Assembly
Assembly Language
Code Generation
Intermediate Representation N

Objective of this course

How to **Implement** semantics with interpreters, analysis, and compilers

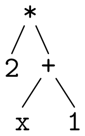
A bit about syntax

The texts:

- `2*(x+1)`
- `(2 * ((x) + 1))`
- `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 ...):

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

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

Semantics

On the abstract syntax we define a semantics (its meaning):

- The example of numerical expressions
- And programs!

1. The Abstract Syntax Tree

2. Interpreter
   - Interpreter with semantic actions
   - Interpreter with explicit AST construction
   - Interpreter with implicit AST

Definition

From Wikipedia:

> *In computer science, an interpreter is a computer program that* ***directly executes instructions*** *written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program.*

- An **interpreter** executes the input program according to the programming language **semantics**.

Implementation strategies

From Wikipedia:

An interpreter generally uses one of the following strategies for program execution:

1. *Parse the source code and perform its behavior directly;* ▶ *Semantic actions !*
2. *Translate source code into some* ***efficient intermediate representation*** *and immediately execute this;* ▶ *Explicit or implicit* ***Abstract Syntax Tree****.*
3. *(Explicitly execute stored precompiled code made by a compiler which is part of the interpreter system.)*

2 Interpreter

- Interpreter with semantic actions
- Interpreter with explicit AST construction
- Interpreter with implicit AST

How

Use semantic attributes to “evaluate” your input program, by induction on the syntax.

$$(string)"37+5" \rightarrow \ldots \rightarrow (int)42$$

Recall the example

The evaluation of arithmetic expressions is defined by induction:

ArithExprParser.g4 - Warning this is java

```
parser grammar ArithExprParser;
options {tokenVocab=ArithExprLexer;}

prog : expr EOF { System.out.println("Result: "+$expr.val); } ;

expr returns [ int val ] // expr has an integer synthesized attribute
: LPAR e=expr RPAR { $val=$e.val; } // e=expr just names 'expr' as 'e'
| INT { $val=$INT.int; } // implicit attribute for INT (given by lexer)
| e1=expr PLUS e2=expr { $val=$e1.val+$e2.val; /* synthesize attribute val based on e1 and e2 */}
| e1=expr MINUS e2=expr { $val=$e1.val-$e2.val; }
;
```

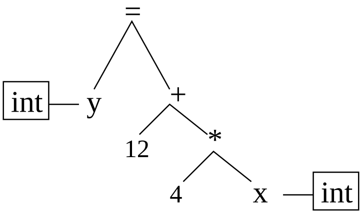
Separation of concerns

- The meaning/semantics of the program could be defined in the semantic actions (of the grammar). Usually though:
  - Syntax analyzer only produces the Abstract Syntax Tree.
  - The rest of the compiler directly **works with this AST**.
- Why ?
  - Manipulating a tree (AST) is easy (recursive style);
  - Separate language syntax from language semantics;
  - During later compiler phases, we can assume that the AST is **syntactically correct** $\Rightarrow$ simplifies the rest of the compilation.

2 Interpreter

- Interpreter with semantic actions
- Interpreter with explicit AST construction
- Interpreter with implicit AST

Abstract Syntax Tree



- AST: memory representation of a program;
- Node: a language construct;
- Sub-nodes: parameters of the construct;
- Leaves: usually constants or variables.

Running example : semantics for numerical expressions

$$
\begin{array}{rclr}
e & ::= & c & \textit{constant} \\
  & | & x & \textit{variable} \\
  & | & e + e & \textit{add} \\
  & | & e \times e & \textit{mult} \\
  & | & \ldots &
\end{array}
$$

Explicit construction of the AST

- Declare a type for the abstract syntax.
- Construct instances of these types during parsing (trees).
- Evaluate with tree traversal.

Example in OCaml 1/3

Types for the abstract syntax:

```
type binop = Add | Mul | ...

type expr =
  | Constant of int
  | Var of string
  | Bin of binop * expr * expr
  | ...
```

Example in OCaml 2/3

Pattern matching in parsing rules:

```
%type<Mysyntax.expr> expr

expr:
| INT { Constant (int_of_string $1) }
| LPAREN expr RPAREN { $2 }
| expr PLUS expr { Bin (Add, $1, $3) }
| var { Var $1 }
```

Example in OCaml 3/3

Tree traversal with pattern matching (for expression eval):

```
let rec eval sigma = function
  | Constant i -> i
  | Bin (bop, e1, e2) ->
   let num1 = eval sigma e1 in
   let num2 = eval sigma e2 in
   ....
  | Var s -> Hashtbl.find sigma s
```

- we need σ, the environnement (map variables to values).

See the interpreter order, we made a choice !

Example in Java 1/3

AST definition in Java: one class per language construct.

AExpr.java

```
public class APlus extends AExpr {
    AExpr left, right;

    public APlus (AExpr left, AExpr right) { this.left=left; this.right=right; }
}
public class AMinus extends AExpr { AExpr left, right; ... }
public class AUnaryMinus extends AExpr { AExpr expr; ... }
public class AConst extends AExpr { int value; ... }
```

Example in Java 2/3

The parser builds an AST instance using AST classes defined previously.

ArithExprASTParser.g4

```
parser grammar ArithExprASTParser ;
options {tokenVocab=ArithExprASTLexer;}

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); }
;
```

Example in Java 3/3

Evaluation is an eval function per class:

AExpr.java

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

APlus.java

```java
public class APlus extends AExpr {
    AExpr left, right;
    public APlus (AExpr left, AExpr e2) { this.left=left; this.right=e2; }
    @Override
    int eval() { return left.eval() + right.eval(); }
}
```

2 Interpreter

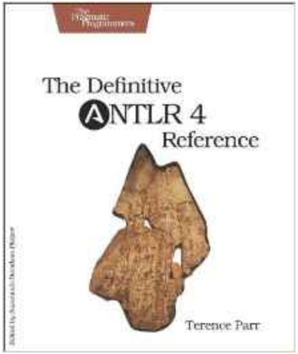
- Interpreter with semantic actions
- Interpreter with explicit AST construction
- Interpreter with implicit AST

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:
```

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)
```

Wrap Up

1. The Abstract Syntax Tree
2. Interpreter
   - Interpreter with semantic actions
   - Interpreter with explicit AST construction
   - Interpreter with implicit AST