Compilation (#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

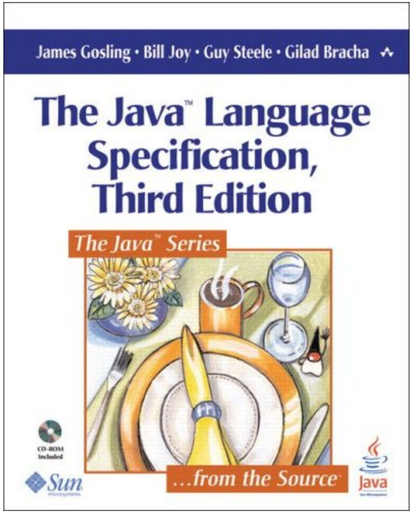
2022-2023

Meaning

How to define the meaning of programs in a given language ?

- Informal description most of the time (natural language, ISO, reference book. . .)
- Unprecise, ambiguous.

Informal Semantics



The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.

It is recommended that code not rely crucially on this specification.

Formal semantics

The formal semantics mathematically characterises the computations done by a given program:

- useful to design tools (compilers, interpreters).
- mandatory to reason about programs and properties of the language.
- (Usually a bunch of greek letters with weird symbols)

Objective of this course

Implementation of program semantics with interpreters.

So far

From our grammars, we only generated **acceptors**.

- We want to execute some action/code each time a grammar rule is matched.

Semantic actions: example (ANTLR)

Semantic actions: code executed each time a grammar rule is matched:

Printing as a semantic action in ANTLR

```
s : A s B {
   // Host language (Java, Python, etc.,
   // depending on the back-end)
   System.out.println("rule A s B just applied");
}
```

Semantic actions in theory - attributes

An attribute is a set “of information” attached to non-terminals/terminals of the grammar

They are usually of two types:

- synthetized: children $\rightarrow$ father ($\uparrow$)
- inherited: the converse ($\downarrow$)

Synthetized grammar attributes

We extend production rules $S \rightarrow S_1 S_2$ with attributes r_i, and we write:

$$S \uparrow r \rightarrow S_1 \uparrow r_1 S_2 \uparrow r_2; \{r := f(r_1, r_2)\}$$

with the meaning:

- S recognizes a chain if the beginning is recognized with S_1 and the rest by S_2.
- Recognizing a S (resp. S_1, S_2) produces a result r (resp. r_1, r_2)
- The result r is computed from the two results r_1, r_2 by the instruction $r := f(r_1, r_2)$
- All rules that produce a S should have attributes of the same type.

Example of a synthetized attribute

Value of an arithmetic expression, simple grammar: $E \to E_1 + E_2 | c$
We define : $value(E) = v$ and $value(c) = v_c$ two attributes <u>of type int</u> for the propagation. Then: $E \uparrow v \to E_1 \uparrow v_1 + E_2 \uparrow v_2 \; ; \; \{v := v_1 + v_2\}$
$E \uparrow v \to c \uparrow v_c ; \; \{v := v_c\}$

In practice the value of c is given by the lexer.

Inherited grammar attributes

(left : inherited/right : synthetised) Now

$$S \downarrow r \uparrow r' \to \{constraint : r'_1 = c(r)\}$$
$$S_1 \downarrow f(r) \uparrow r'_1 \quad S_2 \downarrow g(r, r'_1) \uparrow r'_2$$
$$; \quad \{r' := h(r, r'_1, r'_2)\}$$

with the meaning:

- S recognizes a chain if the beginning is recognized with S_1 and the rest by S_2.
- Recognizing a S_1 produces r'_1 from $f(r)$ st $r'_1 = c(r)$.
- After recognizing $S_1 S_2$, the result r' is computed with $h(r, r'_1, r'2)$.

Example

Consider the grammar: $G = \begin{cases} Start \rightarrow S \\ S \rightarrow \varepsilon | SC \\ C \rightarrow' 0' |' 1' | \ldots |' 9' \end{cases}$

To compute $eval("27") = (int)27$:
(attribution for C is left as exercice)

$$\begin{array}{rcll} Start & \rightarrow & S\downarrow 0 \quad \uparrow res & (1) \\ S\downarrow i\uparrow o & \rightarrow & \varepsilon \quad \{o := i\} & (2) \\ & \rightarrow & C\uparrow c \quad S\downarrow\{10 * i + c\}\uparrow o'\{o := o'\} & (3) \end{array}$$

An important remark

- Synthetised attributes are easy to implement, thus they exist in most parser generators (matching a rule “produces” a value).
- Inherited attributes are often implemented as global/class variables (see later)

Exo time – XML Files

We give the following grammar:

```
L   ->   E   L
      |
E   ->   A   L   B
      |  ident
A   ->   < ident >
B   ->   </ ident >
```

1. Give the derivation tree for the chain `<html><head>toto</head>titi</foo>`.
2. Attribute this grammar to verify that opening and closing tags refer to the same identifiers.

Exo time – Variable declarations

Write a grammar that accepts declarations of variables like:

```
int x=1;
float y,z;
int t;
float u,v=0;
```

and rejects:

```
int x, int y;
```

Then write an attribution that prints individual declarations (of the first case) like:

```
int x=1; float y; float z; int t; float u; float v=0;
```

Semantic action in practice - ANTLR

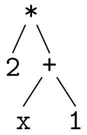
ArithExprParser.g4 - Warning this is java

```
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 // name sub-parts
   { $val=$e1.val+$e2.val; } // access attributes
| e1=expr MINUS e2=expr { $val=$e1.val-$e2.val; }
;
```

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 ...):

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

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.

Semantics

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

- The example of numerical expressions
- And programs!

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.)*

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:

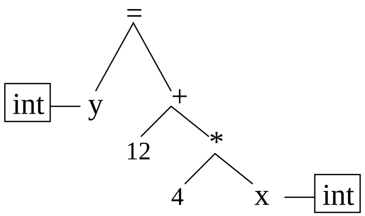
ArithExprParser.g4 - Warning this is java

```
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 // name sub-parts
   { $val=$e1.val+$e2.val; } // access attributes
| 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.

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

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

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 e1,e2;

    public APlus (AExpr e1,AExpr e2) { this.e1=e1; this.e2=e2; }
}
public class AMinus extends AExpr { ...
```

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

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

APlus.java

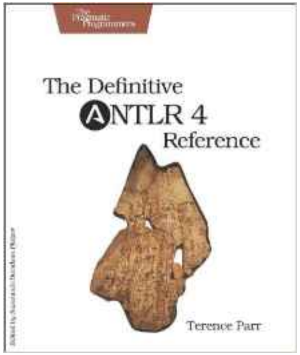
```
public class APlus extends AExpr {
    AExpr e1,e2;
    public APlus (AExpr e1,AExpr e2) { this.e1=e1; this.e2=e2; }
    // semantics below
    int eval() { return (e1.eval()+e2.eval()); }
}
```

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

    def visitAdditiveExpr(self, ctx):
```

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. Program Semantics
2. Grammars attributions and semantic actions
3. Useful notions: abstract syntax, AST
4. Interpreter
    - Interpreter with semantic actions
    - Interpreter with explicit AST construction
    - Interpreter with implicit AST