Handout Progress

Interpretation and Compilation
4-OUT-2020

Luis Caires

Handout Phase 0.1

Implement a complete interpreter and compiler for a tiny arithmetic expression language

Use the approach we are developing in the course

- LL(1) parser using JAVACC
- AST Model
- Interpreter
- Compiler

Fully understanding the handout statement is part of the handout as well.

Contact me anytime if you need help.

Handout Phase 0.1

Learning Outcomes

- you learn how to develop a simple parser using JavaCC
  - understand how to specify tokens using regular expressions
  - you understand how to specify a simple non ambiguous LL(1) context free grammar
- you understand the basics of abstract syntax trees (AST)
- you learn how to define the semantics evaluation function over the AST (this provides an interpreter for the language)
- you learn how to define the semantics compilation function over the AST (this provides a compiler for the language, and allows you to meet the Java Virtual Machine (JVM) internals

Handout Phase 0.1

Abstract Syntax (Abstract Constructors)

ADD: Exp x Exp -> Exp
SUB: Exp x Exp -> Exp
MUL: Exp x Exp -> Exp
DIV: Exp x Exp -> Exp
UMINUS: Exp -> Exp
NUM: int -> Exp

Handout Phase 0.1

Concrete Syntax (Examples)

2*3+4

2*(3+4)

4-2/5*2

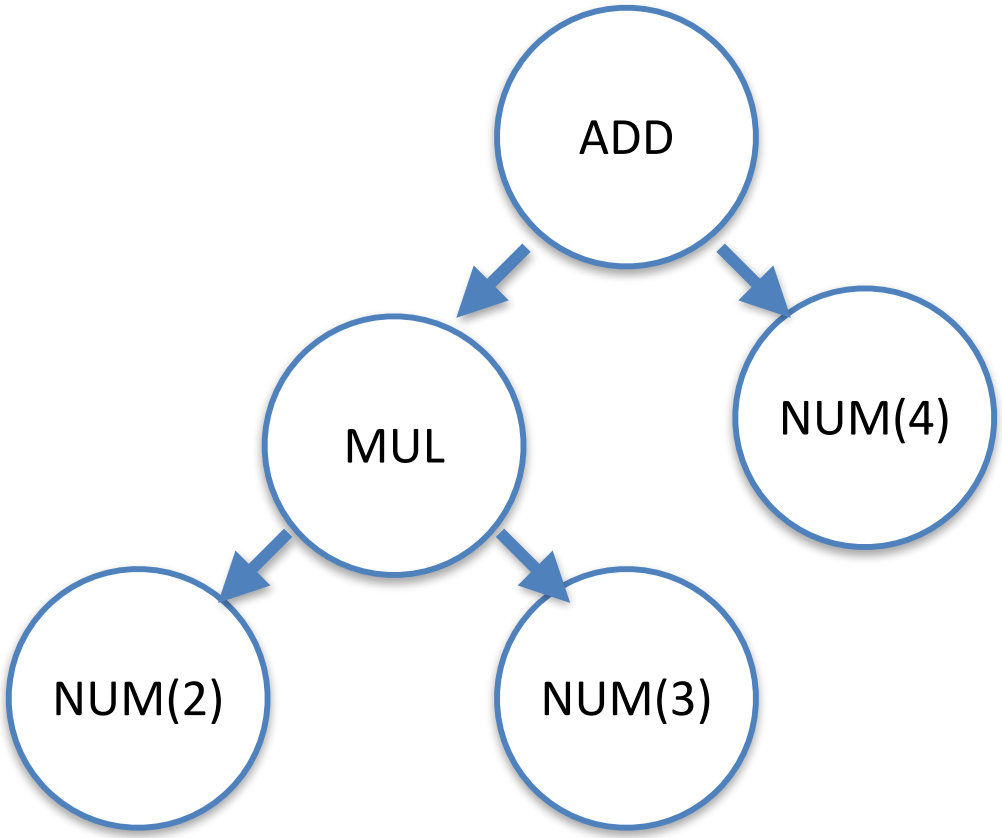
-(2+2-4)

-2

Handout Phase 0.1

Abstract Syntax (Abstract Constructors) **2*3+4**

ADD(MUL(NUM(2),NUM(3)), NUM(4))

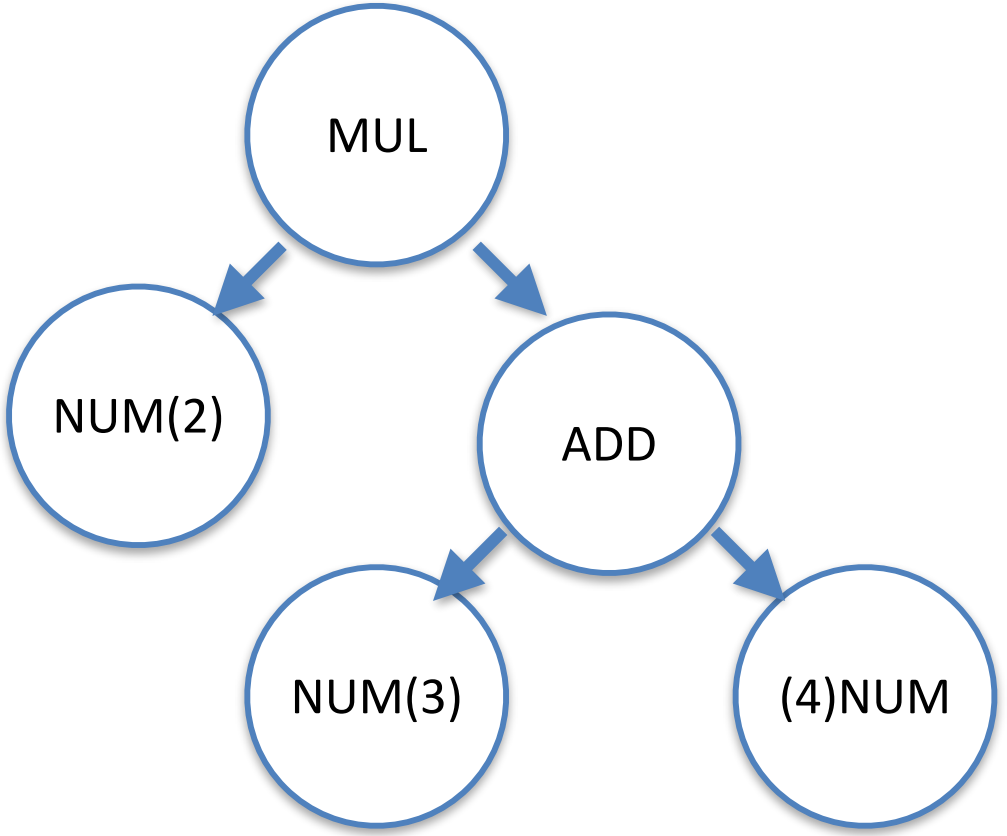


Handout Phase 0.1

Abstract Syntax (Abstract Constructors)

2*(3+4)

MUL(NUM(2), ADD(NUM(3),NUM(4))



Handout Phase 0.1

Grammar

Alphabet = { **num**, **+**, **-**, *, **/**, **(**, **)** }

E -> **num**
E -> E **+** E
E -> E **-** E
E -> E ***** E
E -> E **/** E
E -> **-** E
E -> **(** E **)**

Handout Phase 0.1

Grammar (ambiguous)

E ->

| **num**

| E **+** E | E **-** E

| E ***** E | E **/** E | **-**E | **(** E **)**

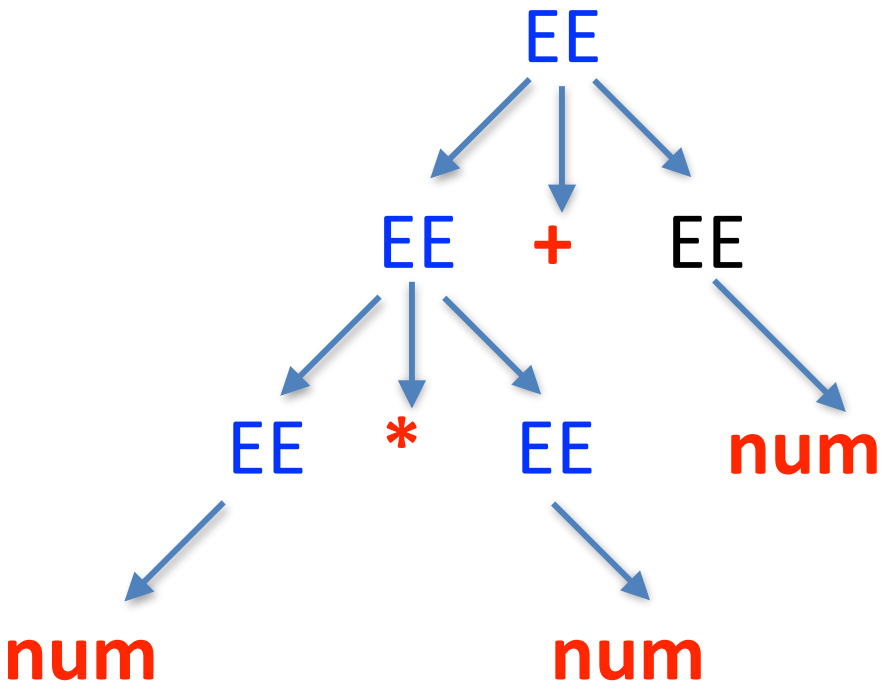
num * num + num has two derivations

EE -> EE **+** EE -> EE ***** EE **+** EE -> **num * num + num**

EE -> EE ***** EE -> EE ***** EE **+** EE -> **num * num + num**

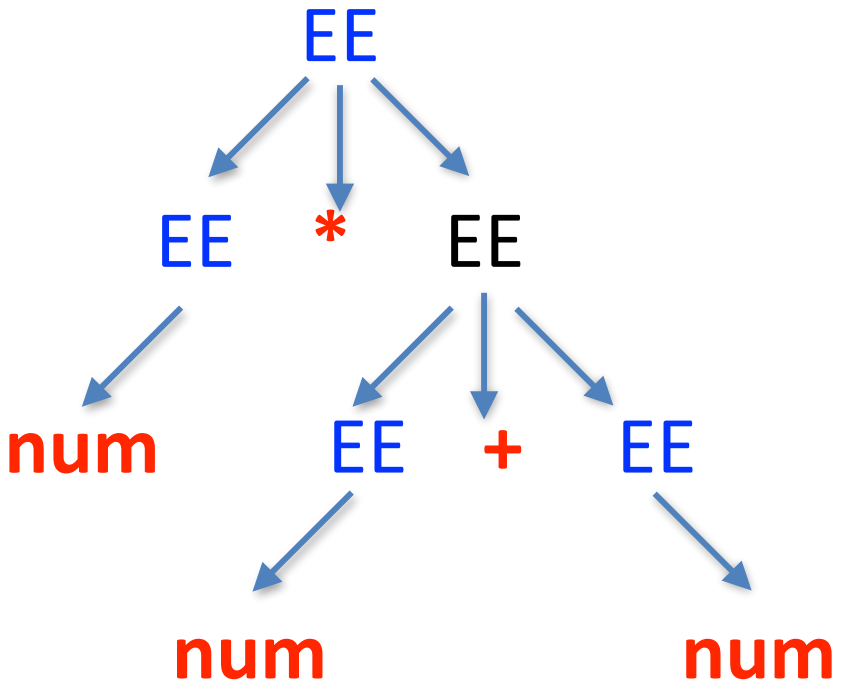
Handout Phase 0.1

EE -> EE + EE -> EE * EE + EE -> num * num + num



Handout Phase 0.1

EE -> EE * EE -> EE * EE + EE -> num * num + num



Handout Phase 0.1

Grammar (non-ambiguous LL(1))

E -> T

E -> T + E

T -> F

T -> F * T

F -> num

F -> (E)

F -> - F

Handout Phase 0.1

Grammar (non-ambiguous)

E -> T

E -> T + E

T -> F

T -> F * T

F -> num

F -> (E)

F -> - F

Handout Phase 0.1

Grammar (non-ambiguous and LL(1))

E -> TE’

E’ -> ε | **+** E

T -> FT’

T’ -> ε | ***** T

F -> **num**

F -> **(** E **)**

F -> **-** F

E -> TE’ -> T+E -> T+TE’ -> T +T+E -> … -> T + T + … + T

Handout Phase 0.1

Grammar (non-ambiguous and LL(1))

E -> TE’

E’ -> ε | **+** E

T -> FT’

T’ -> ε | ***** T

F -> **num**

F -> **(** E **)**

F -> **-** F

num * num + num

E -> TE’ -> FT’E’ -> **num** T’E’ ->

num * T E’ -> **num *** FT’ E’ ->

num * num T’ E’ ->

num * num + E ->

num * num + E ->

num * num + E ->

num * num + TE’ -> num * num + num

E -> TE’ -> T+E -> T+TE’ -> T +T+E -> … -> T + T + … + T

Handout Phase 0.1

Grammar (non-ambiguous and LL(1))

EBNF (Extended BNF)

E -> T [(**+** | **-**) T] *
T -> F [(***** | **/**) F] *
F -> **num** | **(** E **)** | **-** **F**

AST (schematic)

```
interface ASTNode {
int eval() ...
}

class AST??? implements ASTNode {

}
```

AST (schematic)

```
class ASTAdd implements ASTNode {
ASTNode lhs;
ASTNode rhs;
public ASTAdd (ASTNode l, ASTNode r) {
        lhs = l;
        rhs = r;
}
}
```

AST (schematic)

```
class ASTAdd implements ASTNode {


public eval() {
        int vl = lhs.eval();
        int rv = rhs.eval();
        return vl + rv;
}


}
```

Handout Phase 0.2

Learning Outcomes

- you learn how to develop a simple parser using JavaCC
  - understand how to specify tokens using regular expressions
  - you understand how to specify a simple non-ambiguous LL(1) context free grammar
- you understand the basics of abstract syntax trees (AST)
- you learn how to define the semantics evaluation function over the AST (this provides an interpreter for the language)
- you learn how to define the semantics compilation function over the AST (this provides a compiler for the language, and allows you to meet the Java Virtual Machine (JVM) internals

AST (schematic)

```
interface ASTNode {
int eval();
void compile(CodeBlock c);
}

class CodeBlock {
String code[];
int pc;
void emit(String opcode){
        code[pc++] = opcode;
}
void dump(PrintStream f) { … // dumps code to f }
}
```

AST (schematic)

```
class ASTAdd implements ASTNode {

public void compile(CodeBlock c) {
        lhs.compile(c);
        rhs.compile(c);
        c.emit(“iadd”);
}

}
```

JVM bytecodes

- sipush *n*
- iadd
- imul
- isub
- idiv
- …

```
.class public Main
.super java/lang/Object
;
; standard initializer
.method public <init>()V
  aload_0
  invokenonvirtual java/lang/Object/<init>()V
  return
.end method
.method public static main([Ljava/lang/String;)V
      .limit locals 10
       .limit stack 256
       ;    1 - the PrintStream object held in java.lang.System.out
       getstatic java/lang/System/out Ljava/io/PrintStream;
       ; place your bytecodes here between START and END
       ; START

       sipush 20
       sipush 20
       iadd
       sipush 2
       imul

       ; END
       ; convert to String;
       invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
       ; call println
       invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
       return
.end method
```

JVM bytecodes

sipush

Operation

Push `short`

Format

```
sipush
byte1
byte2
```

Forms

sipush = 17 (0x11)

Operand Stack

... →

..., *value*

Description

The immediate unsigned *byte1* and *byte2* values are assembled into an intermediate `short`, where the value of the `short` is (*byte1* << 8) | *byte2*. The intermediate value is then sign-extended to an `int` *value*. That *value* is pushed onto the operand stack.

JVM bytecodes

iadd

Operation

Add `int`

Format

`iadd`

Forms

iadd = 96 (0x60)

Operand Stack

..., *value1*, *value2* →

..., *result*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *iadd* instruction never throws a run-time exception.