



Compiling expressions

Javier Esparza

Computer Science
School of Informatics
The University of Edinburgh



The goal

Construct a compiler for *arithmetic expressions*

```
[criollo]jav: java Compiler "2 + ( 3 * 4 )"
iconst_2
iconst_3
iconst_4
imul
iadd
```

```
[criollo]jav: java Compiler "2 + ("
expected expression
```

```
[criollo]jav: java Compiler "2 + ( 3 * 4"
expected )
```

Simplifying assumptions

- Adjacent *tokens* separated by at least one space

```
[criollo]jav: java Compiler "2+3"  
expected integer
```

- Never two consecutive constants

```
[criollo]jav: java Compiler "2 + 3 4"  
iconst_2  
iconst_4  
iadd
```

- Expression is fully parenthesised (except for the outermost parenthesis)

```
[criollo]jav: java Compiler "2 * 3 + 4"  
iconst_2  
iconst_3  
iconst_4  
iadd  
imul
```

```
[criollo]jav: java Compiler "( 2 * 3 ) + 4"  
iconst_2  
iconst_3  
imul  
iconst_4  
iadd
```

Stages of the process

- **Lexical analysis** \longrightarrow array of tokens

"2 + (3 * 4)" \longrightarrow {"2", "+", "(", "3", "*", "4", ")"}

- **Syntactic analysis** \longrightarrow syntax tree

{"2", "+", "(", "3", "*", "4", ")"} \longrightarrow
$$\begin{array}{c} + \\ / \quad \backslash \\ 2 \quad * \\ \quad / \quad \backslash \\ \quad 3 \quad 4 \end{array}$$

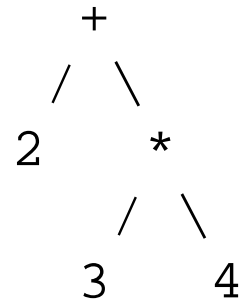
- **Semantic analysis** (empty stage for arithmetic expressions)

- **Translation** \longrightarrow Java byte code

$$\begin{array}{c} + \\ / \quad \backslash \\ 2 \quad * \\ \quad / \quad \backslash \\ \quad 3 \quad 4 \end{array} \longrightarrow \begin{array}{l} \text{iconst_2} \\ \text{iconst_3} \\ \text{iconst_4} \\ \text{imul} \\ \text{iadd} \end{array}$$

Why do we need the syntax tree ?

Reading the syntax tree



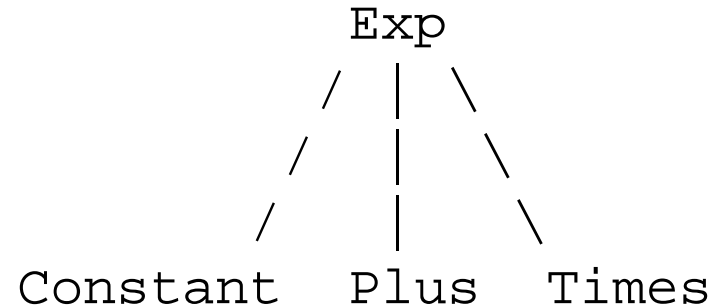
in postorder (left child then right child then parent) yields

2, 3, 4, *, +

which immediately yields the Java byte code

```
iconst_2  
iconst_3  
iconst_4  
imul  
iadd
```

A class for syntax trees of expressions



- A constant is represented as an instance of

```
class Constant extends Exp {  
    private int value;  
    public Constant (int value) {  
        this.value = value;  
    }  
}
```

- An expression with + as root is represented as an instance of

```
class Plus extends Exp {  
    private Exp left, right;  
    public Plus (Exp left, Exp right) {  
        this.left = left;  
        this.right = right;  
    }  
}
```

- An expression with * as root is represented as an instance of

```
class Times extends Exp {  
    private Exp left, right;  
    public Times (Exp left, Exp right) {  
        this.left = left;  
        this.right = right;  
    }  
}
```

Translation

Translate a syntax tree (an instance of `Exp`) into byte code

- A constant c is compiled into `iconst_c`

```
class Constant extends Exp {  
    private int value;  
    public Constant (int value) {  
        this.value = value;  
    }  
    public void compile() {  
        System.out.println("iconst_" + value);  
    }  
}
```

- An expression $e1 + e2$ is compiled into

```
    result of compiling e1
    result of compiling e2
    iadd
```

```
class Plus extends Exp {
    private Exp left, right;
    public Plus (Exp left, Exp right) {
        this.left = left;
        this.right = right;
    }
    public void compile() {
        left.compile();
        right.compile();
        System.out.println("iadd");
    }
}
```

Lexical analysis

Convert the source into an array of tokens

```
import java.util.*;
class Tokens {
    String[] tokens;

    Tokens (String source) {
        StringTokenizer st = new StringTokenizer(source);
        int tokenCount = st.countTokens();
        tokens = new String[tokenCount];
        for (int i = 0 ; i < tokenCount ; i++) {
            tokens[i] = st.nextToken();
        }
    }
}
```

Syntactic analysis

Parse the tokens into a syntax tree

- If input is `2 + 3` output should be the result of executing

```
Exp e = new Plus(new Constant(2), new Constant(3));
```

- A parser builds the syntax tree and simultaneously checks for errors.
 - We first assume that no error checking is needed, and then modify the implementation to include it.
-

Recursive descent

- Expressions are defined by the following rules

Constant Any integer constant is an expression

Plus If *left* and *right* are expressions then so is *left + right*

Times If *left* and *right* are expressions then so is *left * right*

Parentheses If *subexp* is an expression then so is *(subexp)*

Closure Nothing else is an expression

- We assume that the parser receives as input
 - The position of the first token that has not been processed yet
 - A parsed subexpression (possibly `null`)
 - The parser
 - reads the next token;
 - deducts the syntax rule that applies from the list above;
 - acts accordingly, calling itself recursively if necessary, and returns the parse tree of the result of applying the rule
-

-
- An example
 - Parser is called with parameters `n` and `exp`
 - Next token is `+`
 - Parser selects rule **Plus**; identifies `exp` with *left*;
 - Parser calls itself recursively with parameters `n+1` and `null`; it stores the result of the call in `right`
 - Parser returns `new Plus(exp, right)`
 - Reason for the name ‘recursive descent’: the parser recursively ‘descends’ through the syntax rules
-

A first implementation

- We add a current position as new field to Tokens, as well as some new methods.

```
class Tokens {  
    int pos;  
    String[] tokens;  
  
    Tokens (String source) { ... }  
    boolean atEnd() { ... }  
    boolean nextIs(String target) { ... }  
    String next() { ... }  
}
```

- The parser contains several cases according to the next token

```
class Exp {
    Tokens tokens;
    public Exp parse(Tokens tokens) {
        this.tokens = tokens;
        return parseExp(null);
    }
    public Exp parseExp(Exp exp) {
        if (tokens.atEnd()) {
            return exp;
        }
        else if (tokens.nextIs("+")) {
            tokens.eat("+");
            Exp right = parseExp(null);
            return parseExp(new Plus(exp, right));
        }
    }
}
```

```
    else if (tokens.nextIs("(")) {
        tokens.eat("(");
        Exp subexp = parseExp(null);
        tokens.eat(")");
        return parseExp(subexp);
    }
    else if (tokens.nextIs(")")) {
        return exp;
    }
    else {
        String s = tokens.next();
        int i = Integer.parseInt(s);
        return parseExp(new Constant(i));
    }
}
public void compile() {}
}
```

```
class Compiler {  
    public static void main (String[] args) throws SyntaxError  
        Tokens tokens = new Tokens(args[0]);  
        Exp e = new Exp();  
        e.parse(tokens).compile();  
    }  
}
```

Adding error checking

- We introduce a new class `SyntaxError`

```
class SyntaxError extends Exception {  
    String message;  
    SyntaxError (String symbol) {  
        this.message = "expected " + symbol;  
    }  
}
```

- We throw syntax errors where necessary
-

Changes in Exp

```
if (tokens.atEnd()) {  
    return exp;  
}
```

```
if (tokens.atEnd()) {  
    if (exp == null)  
        throw new SyntaxError("expression");  
    else return exp;  
}
```

```
else {
    String s = tokens.next();
    int i = Integer.parseInt(s);
    return parseExp(new Constant(i));
}

else {
    try {
        String s = tokens.next();
        int i = Integer.parseInt(s);
        return parseExp(new Constant(i));
    }
    catch (NumberFormatException e) {
        throw new SyntaxError ("integer");
    }
}
```

Changes in Tokens

```
void eat(String s) { pos++; }
```

```
void eat(String s) throws SyntaxError {  
    if (pos != tokens.length && tokens[pos].equals(s))  
        pos++;  
    else  
        throw new SyntaxError(s);  
}
```

Changes in Compiler

```
class Compiler {
    public static void main (String[] args) {
        Tokens tokens = new Tokens(args[0]);
        Exp e = new Exp();
        e.parse(tokens).compile();
    }
}
```

```
class Compiler {
    public static void main (String[] args) {
        try {
            Tokens tokens = new Tokens(args[0]);
            Exp e = new Exp();
            e.parse(tokens).compile();
        }
        catch (SyntaxError e) {
            System.out.println(e.message);
        }
    }
}
```