Compiling expressions

We have seen that a compiler translates a high-level source program into a low-level executable version. In this lecture we look in detail at a small part of this process, compiling integer expressions from their familiar mathematical syntax — for example, an expression such as \((2 + 3) * 4\) — into a simpler syntax which does not support parenthesized sub-expressions. This will require us to order the evaluation of the sub-expressions in such a way that their values are computed from the most basic sub-expressions upwards. One syntax for arithmetic expressions called reverse Polish notation requires expressions to be written in this order — instead of the familiar notation which uses parentheses and infix operators \((2 + 3) * 4\) in reverse Polish one would write \(2 3 + 4 *\) which uses no parentheses and has postfix operators. (An infix operator appears between its two operands whereas a postfix operator appears after its two operands.)

Reverse Polish notation is not the only language where the operators appear after their operands. In a stack-based machine model such as the Java Virtual Machine sub-expressions are also evaluated before their results are combined by an operator. Accordingly, we will look here at the compilation of simple arithmetic expressions into sequences of Java byte code instructions.

8.1 Detecting syntax errors

Even with a very simple language such as the language of arithmetic expressions it is possible to form expressions which contain syntax errors — for example, \(2 +\) is not well-formed because the plus operator should have both a left and a right operand. Similarly the input could contain lexical errors — for example, the token \(2abc\) is not a number and accordingly could not be used in an arithmetic expression. It is possible for parentheses not to enclose a well-formed sub-expression — for example, \(2 + ()\) is not well-formed.

Upon encountering any of these errors we will not be able to compile the expression which we were given. In this case we should report the error as a exception to the normal routine of processing and give back a diagnostic error message which seeks to help the user of the compiler to detect the error in the expression with the aim of repairing it before the next compilation attempt. This process of error detection cannot guarantee to find the true cause of the error in the expression. Perhaps an earlier error
causes a later expression to be faulted instead of the earlier one. Even though this is
the case, we hope that the (limited) diagnostic error messages which we produce will
be of some use in tracking down the real source of the error.

The class SyntaxError shown below is used for reporting errors. It carries a mes-
sage string which records the token or grammatical construct which was expected but
not found.

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

### 8.2 Representing arithmetic expressions

We will define a class Exp, to come later, which represents any arithmetic expression.
This class will have three subclasses, Constant, Plus and Times, which represent
numbers, addition expressions and multiplication expressions respectively. (It would
be easy to add other subclasses for subtractions, divisions and other operations but
we omit these here in order to be brief. Integer variable identifiers are also omitted for
the sake of brevity.)

The Constant class stores an integer value, which is the value which it represents.
Once this value is set by the constructor of the class it can neither be changed nor
inspected. This is achieved by marking the integer value private at the point of its
declaration.

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

The method which we are considering here is the compile() method which will
print the compiled representation of our expression on the standard output stream.
This method is defined at the level of the Exp class but each subclass will override
this method with a specific version which is appropriate for its data. For a constant
we compile into a single Java byte code instruction, the iconst (integer constant)
instruction. This instruction pushes an integer constant onto the top of the evaluation
stack. That is, iconst_1 pushes the integer constant 1 onto the top of the evaluation
stack, iconst_2 pushes the integer constant 2 onto the top of the evaluation stack,
and so on.
A slightly more complex example is that of arithmetic expressions. Here we have two sub-expressions to consider. When an addition operation is constructed we must supply both the left sub-expression and the right sub-expression. These might only be constants but they might be other compound expressions which use addition or multiplication and themselves have sub-expressions. Because we do not know the class of the two sub-expressions they are stored as expressions of class \texttt{Exp} (instead of the more specific subclasses \texttt{Constant}, \texttt{Plus} or \texttt{Times}).

```java
1 class Plus extends Exp {
2     private Exp left, right;
3     public Plus (Exp left, Exp right) {
4         this.left = left;
5         this.right = right;
6     }
7     public void compile() {
8         left.compile();
9         right.compile();
10        System.out.println("iadd");
11    }
12 }
```

When we compile an addition expression we first compile the left operand (the method invocation \texttt{left.compile()} on line 8) and then the right operand (the method invocation \texttt{right.compile()} on line 9) and then finally generate an \texttt{iadd} (integer add) instruction on line 10. The effect of the \texttt{iadd} instruction in the Java Virtual Machine is to pop the top two integers from the evaluation stack, add them, and then push the resulting integer back onto the top of the stack.

The \texttt{Times} class is virtually identical to the \texttt{Plus} class except that it generates an \texttt{imul} (integer multiply) instruction at the end instead of an \texttt{iadd} instruction. The effect of the \texttt{iadd} instruction in the Java Virtual Machine is to pop the top two integers from the evaluation stack, multiply them, and then push the resulting integer back onto the top of the stack. Again, we omit the code of the \texttt{Times} class for brevity.

### 8.3 Dynamic method dispatch

We now have three different versions of the \texttt{compile()} method, defined in the three subclasses \texttt{Constant}, \texttt{Plus} and \texttt{Times}. In the case of the method invocations on lines 8 and 9 of the method \texttt{compile()} in the \texttt{Times} class shown above how do we know which of these \texttt{compile()} methods to invoke on the objects \texttt{left} and \texttt{right}? The answer is that this question is resolved dynamically at run-time by inspecting the run-time class information associated with the objects \texttt{left} and \texttt{right}. This is known as \textit{dynamic method dispatch} or, in some object-oriented languages, \textit{late binding}. At the point where we come to carry out these method invocations we know that the object identifiers \texttt{left} and \texttt{right} will refer to some allocated object of class \texttt{Exp} or one of its subclasses, \texttt{Constant}, \texttt{Plus} or \texttt{Times}. At that point the correct version of the method will be chosen, based on this information about the class of the object.
8.4 Generating a stream of tokens

So far we have concentrated on the final stage of the compilation process, after we have accepted the concrete syntax of the expression and built from this an abstract syntax tree of the expression, using the constructors of the classes which we have just seen. Before this stage we define a class which produces a token stream which can be parsed in order to build the abstract syntax tree.

```java
import java.util.*;
class Tokens {
    int pos;
    String[] tokens;
    /* Convert the source into an array of strings. */
    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();
        }
    }
    boolean atEnd() {
        return pos == tokens.length;
    }
    boolean nextIs(String target) {
        return tokens[pos].equals(target);
    }
    String next() {
        return tokens[pos++];
    }
    void eat(String s) throws SyntaxError {
        if (pos != tokens.length && tokens[pos].equals(s))
            pos++;
        else
            throw new SyntaxError(s);
    }
}
```

The constructor of this class (lines 7–14) is given a single string which is the source of the expression which is to be compiled. For simplicity we assume that adjacent tokens are separated by at least one space character (so that we would have 2 + 3, but never 2+3). In this case we can use a StringTokenizer to split the string into tokens, which are accessed by the nextToken() method. These tokens are entered into an array to allow us to look ahead in the input and inspect tokens without consuming them. At various points when parsing expressions we will expect to see certain tokens and to consume them. The eat() method checks for the presence of a particular token.
and throws an exception if it is not found.

## 8.5 Recursive descent parsing

The most complex part of our small compiler is the parser. This processes the token stream which is generated from the `Token` class and tries to make sense of it in order to build the abstract syntax tree of the expression. For example, if our input string was `2 + 3` we would like the parser to produce the result of executing the Java assignment

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

Part of the difficulty of achieving this lies in detecting at the same time very similar expressions which are erroneous. For example, `2 +` and `2 + ()`. The method which we use is *recursive descent parsing*. That is, we expect to find an expression which is defined by the following rules.

- **Constant** Any integer constant is an expression.
- **Plus** If `left` is an expression and `right` is an expression then so is `left + right`
- **Times** If `left` is an expression and `right` is an expression then so is `left * right`
- **Parentheses** If `subexp` is an expression then so is `( subexp )`
- **Closure** Nothing else is an expression.

In recursive descent parsing, we recursively descent through these rules parsing sub-expressions using recursive invocations of the method. When we reach single tokens, which have no recursive structure, we consume them using the `eat()` method. When we reach the point where an integer constant is expected we consume it with the use of another method (`Integer.parseInt()`) because it has no recursive sub-structure. When we reach the end of the token stream we know that what we have should be a well-formed expression.

When errors occur during the parsing process we report them as syntax errors which are flagged by throwing a new `SyntaxError` exception. This carries with it a diagnostic message string, giving feedback to the user. One place where this occurs is if our attempt to parse an integer constant fails. In this case the `Integer.parseInt()` method will throw the exception `NumberFormatException`. This is caught by a `try .. catch` construct and the `SyntaxError` exception is thrown instead.

Another source of `SyntaxError` exception is the `eat()` method. For example, if we have an unclosed parenthesis the method invocation of `tokens.eat("")` at line 26 will fail and correctly report a syntax error.

Another source of syntax errors arises where an infix binary operator has a left operand but no right operand (as in `2 +`). In this case the statement

```java
Exp right = parseExp(null);
```

will fail when the exception `SyntaxError` is thrown at line 10.

As noted in Section 8.3, each of the classes `Exp, Constant, Plus` and `Times` defines a `compile()` method. The empty method definition on line 42

```java
public void compile() { }
```
might seem pointless but it is needed because we need to have a definition of this method for the Exp method as the default method to use in the compile() methods in the subclasses of Exp.

```java
class Exp {
    Tokens tokens;

    public Exp parse(Tokens tokens) throws SyntaxError {
        this.tokens = tokens;
        return parseExp(null);
    }

    private Exp parseExp(Exp exp) throws SyntaxError {
        if (tokens.atEnd()) {
            if (exp == null)
                throw new SyntaxError("expression");
            else 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 right = parseExp(null);
            return parseExp new Times(exp, right);
        }
        else if (tokens.nextIs("(" ) {
            tokens.eat("(");
            Exp subexp = parseExp(null);
            tokens.eat(")" );
            return parseExp(subexp);
        }
        else if (tokens.nextIs(")") ) {
            return exp;
        }
        else {
            try {
                String s = tokens.next();
                int  i = Integer.parseInt(s);
                return parseExp new Constant(i);
            }
            catch (NumberFormatException e) {
                throw new SyntaxError("integer");
            }
        }
    }

    public void compile() {}
}
```
8.6 Putting it all together

The compiler is built from an object of the Token class and then constructs an expression to parse the token stream (with `parse(tokens)` and compile the resulting abstract syntax tree (with `compile`). If syntax errors are detected then these are caught and reported to the user by printing their message field.

```java
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);
        }
    }
}
```

8.7 Running the compiler

We run the compiler by passing in the expression to be compiled as a command line argument.

```
\textit{[blarff]\textit{stg}:} java Compiler "2 + 3"
\textit{iconst\_2}
\textit{iconst\_3}
\textit{iadd}
```

More complex examples allow us to see that structured expressions with sub-expressions are compiled down into sequences of Java byte code instructions which are preformed in first-to-last order to achieve the same effects.

```
\textit{[blarff]\textit{stg}:} java Compiler "( 2 + 3 ) * ( 4 + 5 )"
\textit{iconst\_2}
\textit{iconst\_3}
\textit{iadd}
\textit{iconst\_4}
\textit{iconst\_5}
\textit{iadd}
\textit{imul}
```

Stephen Gilmore.