

Oblig II – INF 3110/4110

The objective of this exercise is to get a feeling for what it takes to represent programs with a syntax tree in terms of an object structure and what kind of decoration is needed in order to perform type check and do a simple interpreter.

There is no only-one right answer. In addition to the code it is therefore important that you provide a small report on your design.

Syntax tree/meta model

Design and implement in Java a syntax tree/meta model (in terms of classes) for the following language.

```
<program> ::= <block>
<block> ::= '}' {<declaration> ';' }* {<statement> ';' }+ '{'
<declaration> ::= <variable-decl> | <function-decl>
<variable-decl> ::= <type> <identifier>
<function-decl> ::= <type> <identifier> '(' {<formal parameters> }? ')' <block>
<formal parameters> ::= <type> <identifier> { ',' <type> <identifier> }*
<statement> ::= <assignment> | <call> | <block> | <return>
<assignment> ::= <target> '=' <source>
<call> ::= <function> '(' <actual parameters> ')'
<actual parameters> ::= <exp> { ',' <exp> }*
<type> ::= 'int' | 'double'
<target> ::= <identifier>
<source> ::= <exp>
<return> ::= 'return' <exp>
<exp> ::= <term> { <operator> <exp> }*
<operator> ::= '+' | '-'
<term> ::= <number> | <variable> | <call>
<variable> ::= <identifier>
<function> ::= <identifier>
```

Terminal symbols are thus enclosed by ‘’. <number> and <identifier> are the lexical units and are defined as the corresponding elements in Java. The types are the same as the corresponding types in Java. For double and int the widening of type shall applied, i.e. an expression involving both int and double shall become double, with int being converted to double. Int can be assigned to a double, but a double can not be assigned to an int; the same rules apply to parameter transfer. <return> is only allowed in function blocks. The type of the return expression must be compatible with the type of the function as return is an assignment of the expression to a function value variable.

Predefined functions are inInt and inDouble, e.g. with the definition as given in the easyIO package, and print/println with the effect of Java System.out.print/println.

Making programs

Programs shall be made by derivation. This means that each of the classes representing productions in the grammar shall have a method 'derive' for constructing a node in the syntax tree according to this production. As an example, the class Block for <block> will have a derive method that generates a number of objects of the class Declaration (representing <declaration>s) and a number of objects of the class Statement (representing <statement>s). The derive methods shall make prompts according to the production rule.

As an example, the first prompt for derive in class Block will be e.g.

```
-- <declaration>? type d
-- <statement>? type s
>
```

Typing 'd'

```
> d
```

will enter a <declaration>, and the next prompt may be

```
-- <declaration>? type d
-- <statement>? type s
>
```

Typing 's'

```
> s
```

will enter a <statement>.

After having entered at least one statement, the prompt will be:

```
-- <statement>? type s
-- end of block? type e
```

Typing 's' for <statement> will result in the generation of a Statement object and call of its derive method, which will then in turn prompts for whether the statement shall be an <assignment>, a <call> a <block> or a <return>.

The user interface may either be fancy (e.g. with widgets) or simply text-based prompts as indicated above.

At any given point in time it shall be possible to print the program as it is, with an indication of where the derivation has come. Give e.g. each of the classes a method 'print' that prints the terminal symbols of the corresponding production and calls the print

method on the objects representing the non-terminals of the production. Do not use too much time on fancy pretty-printing; it is allowed to make a line out of each production – the examples will anyhow be small test examples testing name bindings, the type checking and the interpreter.

The important thing is the design of the syntax tree classes, not the input of programs, so if the above is too challenging, you may decide to enter programs simply by a sequence of statements generating the objects of the syntax tree and linking them together to form the syntax tree. In good time before the deadline, one or two example programs will be published, and you will be asked to enter these (in whatever form you prefer), type check them and interpret them.

Representation of application-declaration bindings

One may keep the identifiers of variables and functions as they are used in expressions and look them up each time the types of the variables/functions are needed for e.g. type checking. Alternatively, the above class model may be extended with an association between classes representing applications and the class representing the declaration. Make this change to the syntax tree and implement the method for setting the link corresponding to this association.

Type checking

Implement the required methods for type checking assignment, parameter transfer and expression.

Interpreter

Make an interpreter by means of a method ‘interpret’ for each of the classes of the syntax tree model. The interpretation of a declaration will e.g. allocate an object representing the variable. The interpret for an expression will evaluate the expression by using primitives in the language of the interpreter. Do not think about efficiency, it is e.g. ok to allocate blocks and functions by means of an object that contains lists of objects representing the various parameters and local variables. It is also ok to maintain the activation stack by means of references, much the way it is indicated by the figures from Mitchell.

Parameterizing with mechanisms

Suppose that this tool shall be used in the next round of this course, in order to illustrate the implications of different language mechanisms. If you get the time, make it therefore possible to adjust the binding mechanism and the interpreter on these points:

- Static and dynamic scoping
- By value or by reference parameter transfer, that is either all parameters passed by value or all parameters passed by reference.

This means that it shall be possible to treat the same program with these different mechanisms.

Test examples

Test 1: static/dynamic scope

```
{
  int x;
  x = 1;
  {
    p(){print(x);};
    int x;
    x=2;
    {
      int x;
      x = 3;
      p();
    };
  };
}
```

Test 2: by reference or by value

```
{
  int x;
  p(int i)
  {
    i=i+1;
    x=x+1;
  };
  x=1;
  p(x);
  print(x);
}
```