Individual Exercises - Lecture 9

1. (optional - but recommended) Follow the studio associated with Crafting a Compiler Chapter 7
   http://www.cs.wustl.edu/~cytron/cacweb/Chapters/7/studio.shtml
   A virtual machine is available on Moodle with the studio/lab exercises loaded inside Eclipse.

2. (optional - by recommended) Do the exercises associated with the Tutorial on the Visitor Pattern
   Can be found here: https://www.cse.wustl.edu/~cytron/cacweb/Tutorial/Visitor/

3. *Do Fischer et al exercise 10, 12, 22 on pages 272-278 (exercise 11, 15, 22 on pages 304-310 in GE)*

   10. *Consider extending the grammar in Figure 7.14 to include binary subtraction and unary negation, so that the expression "minus y minus x times 3"*
   has the effect of negating y prior to subtraction of the product of x and 3.

   (a) Following the steps outlined in Section 7.4, modify the grammar to include these new operators. Your grammar should grant negation strongest precedence, at a level equivalent to deref. Binary subtraction can appear at the same level as binary addition.

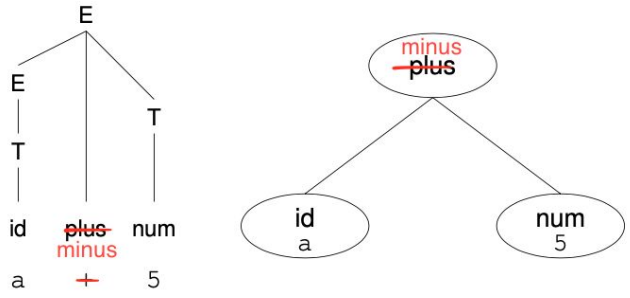
```
 1  Start  → Stmt $
 2  Stmt   → id assign E
 3         | if lparen E rparen Stmt else Stmt fi
 4         | if lparen E rparen Stmt fi
 5         | while lparen E rparen do Stmt od
 6         | begin Stmts end
 7  Stmts  → Stmts semi Stmt
 8         | Stmt
 9  E      → E plus T
10         | T
11  T      → id
12         | num
```

Figure 7.14: Grammar for a simple language.

Alter the modification:

9 E → E plus T
10 | E minus T
11 | T
12 T → id
13 | num
15 | minus num

(b) Modify the chapter's AST design to include subtraction and negation by including a minus operator node.



(a) (b)

For negation you could introduce a negativeNum node : num(5) -> negativeNum(5),

Or you could keep the current num node and allow for negative numbers: num(-5).

(c) Install the appropriate semantic actions into the parser.

```
1  Start          → Stmt_ast $
                    return (ast)                                          (13)
2  Stmt_result    → id_var assign E_expr
                    result ← MAKEFAMILY(assign, var, expr)                (14)
3                 | if lparen E_p rparen Stmt_s fi
                    result ← MAKEFAMILY(if, p, s, MAKENODE( ))            (15)
4                 | if lparen E_p rparen Stmt_s1 else Stmt_s2 fi
                    result ← MAKEFAMILY(if, p, s1, s2)                    (16)
5                 | while lparen E_p rparen do Stmt_s od
                    result ← MAKEFAMILY(while, p, s)                      (17)
6                 | begin Stmts_list end
                    result ← MAKEFAMILY(block, list)                      (18)
7  Stmts_result   → Stmts_sofar semi Stmt_next
                    result ← sofar.MAKESIBLINGS(next)                     (19)
8                 | Stmt_first
                    result ← first                                        (20)
9  E_result       → E_e1 plus T_e2
                    result ← MAKEFAMILY(plus, e1, e2)                     (21)
10                | T_e
                    result ← e                                            (22)
11 T_result       → id_var
                    result ← MAKENODE(var)                                (23)
12                | num_val
                    result ← MAKENODE(val)                                (24)
```

Figure 7.17: Semantic actions for grammar in Figure 7.14.

To E_{result} add:

| E_{e1} minus T_{e2}

Result <- MakeFamily(minus, e1, e2)

// This create a MinusNode with children e1 and e2

To T_{result} add:

| minus num_{val}

Result <- MakeNode(-val)

12. *The grammar below generates nested lists of numbers. The semantic actions are intended to count the number of elements just inside each parenthesized list. For each list found by Rule 2 (Marker 44), the number of elements found just inside the list.*

For example, the input

((1 2 3) (1 2 3 4 5 6))

Should print 3, 6 and 2

```
1  Start        → List_avg $
2  List_result  → lparen Operands_ops rparen
                    PRINT(count)                    (44)
3               | num_val
4  Operands     → Operands List
                    count ← count + 1
5               | List
                    count ← 1
```

(a) The grammar uses a global variable count to determine the number of elements in a list. What is wrong with that approach?

Relying on a single global variable for counting is a problem because it cannot handle nested lists, since it overwrites the current element count when entering a new list.

(b) Change the semantic actions so that the appropriate values are synthesized by *the rules to allow counting without a global variable.*

Instead of using a global variable, use synthesized attributes. Remember, a synthesized attribute is a value associated with some node in the tree, where the value is calculated from the children of said node. See Fischer figure 7.3 in section 7.2.1 (p. 240).

```
Start          → list_avg $

List_result    → lparen Operands_ops rparen
                   Print(ops);
               | numval

Operands_count → Operands_previous List
                   count ← previous + 1;
               | List
                   count ← 1
```

22. In addition to the ***NeedsBooleanPredicate*** type discussed in Section 7.7.3, Figure 7.24 references the following types: ***NeedCompatibleTypes*** and ***NeedsLeftChildType***.

```
class ReflectiveVisitor
    /*  Generic visit                                          */
    procedure VISIT(AbstractNode n)                              (35)
        this.DISPATCH(n)                                         (36)
    end
    procedure DISPATCH(Object o)
        /*  Find and invoke the VISIT(n) method                 */
        /*  whose declared parameter n is the closest match     */
        /*  for the actual type of o.                           */
    end
    procedure DEFAULTVISIT(AbstractNode n)                       (37)
        foreach AbstractNode c ∈ Children(n) do this.VISIT(c)
    end
end
class IfNode                                                     (38)
    extends AbstractNode
    implements {NeedsBooleanPredicate}
end
class WhileNode                                                  (39)
    extends AbstractNode
    implements {NeedsBooleanPredicate}
end
class PlusNode
    extends AbstractNode
    implements {NeedsCompatibleTypes}
end
class TypeChecking extends ReflectiveVisitor                     (40)
    procedure VISIT(NeedsBooleanPredicate nbp)                   (41)
        /*  Check the type of nbp.GETPREDICATE()                */
    end
    procedure VISIT(NeedCompatibleTypes nct)                     (42)
    end
    procedure VISIT(NeedsLeftChildType nlct)                     (43)
    end
end
```

Figure 7.24: Reflective Visitor

(a) Which node types inherit from those types?

NeedCompatibleType probably requires all children to have the same type, and NeedComptaibleType could be inherited by operation nodes such as +, -, / etc. For example addition nodes may allow addition of a float and an integer.

NeedsLeftChildType probably requires that the children has the same type as the leftmost child. This might be inherited by assignment nodes, since most languages require the right hand side to be of the same type (or at least castable to the same type), as the left hand side of the assignment.

*(b) Describe the actions that should be performed by the visitor at Markers 42 and 43 on behalf of the **NeedCompatibleTypes** and **NeedsLeftChildType** types.*

NeedCompatibleType

Find the set of types for all immediate children of the node and check the list of compatible types for that node type. For example, addition operations might allow adding doubles and floats, but not doubles and integers.

NeedsLeftChildType

Declare local variable Type = LeftMostChild.getType()
Check all the other children have the same (or convertible) types.

Group Exercises - Lecture 9

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Do Fischer et al exercise 20, 19, 21 on page 272-278 (exercise 14, 19 20 on pages 304-310 in GE) (you may use the language you are defining in your project when designing ast nodes in exercise 19)

   20. In contrast to the approach discussed in Section 7.7.2, Figure 7.25 shows the partial results of a design in which each phase contributes code to each node type.

```
class IfNode extends AbstractNode
    procedure TYPECHECK( )
        /*  Type-checking code for an if                    */
    end
    procedure CODEGEN( )
        /*  Generate code for an if                         */
    end
    ...
end

class PlusNode extends AbstractNode
    procedure TYPECHECK( )
        /*  Type-checking code for a plus                   */
    end
    procedure CODEGEN( )
        /*  Generate code for a plus                        */
    end
    ...
end
...
```

Figure 7.25: Inferior design: phase code distributed among node types.

(a) What are the advantages and disadvantages of the approach used in Figure 7.25?
This approach might be more manageable initially when the amount of phases is low. Inside each node class you can easily follow what happens to that node type during the different compiler phases.

A problem with this approach is that many modern compilers have up to a couple of hundred phases(mostly optimization steps). If the node classes themselves contained the methods for all phases, they would very quickly get very bloated. Consider software design principles such as the ‘Single responsibility principle’. If you use the approach in 7.25, each Node class will end up being responsible for many different steps of the compiler. If you change a phase, or add a new one, you now have to go through all the node classes to implement this change.

(b) How does the visitor pattern discussed in Section 7.7.2 address the disadvantages?

A separate class for each phase is created. This class then contains visitor methods for all types of nodes. This makes modifying and adding of phases much simpler. If we modify a phase, we only have to modify the visitor for that phase. If we add a phase, we simply add a new visitor and then dispatch it to the root.

19. Based on the discussion of Section 7.4 and using the pseudocode in Figure 7.13 as a guide, design a set of AST classes and methods to support AST construction in a real programming language.

```
/*   Assert: y ≠ null                                          */
function MAKESIBLINGS(y) returns Node
    /*   Find the rightmost node in this list                  */
    xsibs ← this
    while xsibs.rightSib ≠ null do xsibs ← xsibs.rightSib
    /*   Join the lists                                        */
    ysibs ← y.leftmostSib
    xsibs.rightSib ← ysibs
    /*   Set pointers for the new siblings                     */
    ysibs.leftmostSib ← xsibs.leftmostSib
    ysibs.parent ← xsibs.parent
    while ysibs.rightSib ≠ null do
        ysibs ← ysibs.rightSib
        ysibs.leftmostSib ← xsibs.leftmostSib
        ysibs.parent ← xsibs.parent
    return (ysibs)
end

/*   Assert: y ≠ null                                          */
function ADOPTCHILDREN(y) returns Node
    if this.leftmostChild ≠ null
    then this.leftmostChild.MAKESIBLINGS(y)
    else
        ysibs ← y.leftmostSib
        this.leftmostChild ← ysibs
        while ysibs ≠ null do
            ysibs.parent ← this
            ysibs ← ysibs.rightSib
end
```

Figure 7.13: Methods for building an AST.

Below is an example for an if statement very similar to the one in Java / C# / C. In the first code snippet the AST node for the if statement can be seen. In the second code snippet a visitor method for creating the AST node when visiting the non-terminal ifstmt node in the parse tree. The ListNode inherits from an abstract node class.

Other node classes could be created in a similar fashion. Usually an abstract class with methods like getChildren() and accept(visitor) is created as a foundation. This class can then be extended like in this case by a ListNode, but often BinaryNodes and LeafNodes are also included, since their behaviour when visited is slightly different. The IfStmtNode below is of type listnode, since it has more than 2 children.

Node class for if statements

```
public class IfStmtNode extends ListNode {
    public IfStmtNode(ArrayList<Node> children) {
        super(children);
    }
    public Node getLogicalExprNode(){
        return super.getChildren().get(0);
    }
    public Node getIfBlock(){
        return super.getChildren().get(1);
    }
    public Node getElseBlock(){
        // No else block present
        if(super.getChildren().size() < 3){
            return null;
        } else {
            return super.getChildren().get(2);
        }
    }
    // The generics here allows different visitors to have different
    // return types. For example the ASTBuilder visitor will probably
    // return objects of the type Node(to return a new modified tree),
    // while a pretty printer visitor might return strings.
    // Don't fret if you don't get this 100% right now
    public <T> T accept(ASTBaseVisitor<? extends T> astBaseVisitor) {
         return astBaseVisitor.visit(this);
    }
}
```

Visitor method for IfStmt inside the ASTBuilderVisitor<Node>(which extends the BaseVisitor)

```
@Override
public Node visitIfstmt(ifStmtParseTreeNode node) {
    ArrayList<Node> children = new ArrayList<>();
    Node logicalExpr = visit(node.logical_expr());
    // Block of code inside if stmt
    Node ifBlock = visit(node.blck);
    children.add(logicalExpr);
    children.add(ifBlock);
    if(node.elseblck != null){
        Node elseBlocK = visit(node.elseblck);
        children.add(elseBlocK);
    }

    return new IfStmtNode(node, children);
}
```

21. In contrast with the approaches discussed in Sections 7.7.2 and 7.7.3, consider the idea sketched in Figure 7.26.

```
foreach AbstractNode n ∈ AST do
    switch (n.GETTYPE( ))
        case IfNode
            call f.VISIT(⟨IfNode ⇓ n⟩)
        case PlusNode
            call f.VISIT(⟨PlusNode ⇓ n⟩)
        case MinusNode
            call f.VISIT(⟨MinusNode ⇓ n⟩)
```

Figure 7.26: An alternative for achieving double dispatch.

(a) What are the advantages and disadvantages of the approach used in Figure 7.26?

The advantage is that it is easy to understand/implement and can give a good overview of the node types visited by the specific visitor.

The disadvantage is that it is essentially a lot of duplicated code and it can quickly become very big with 50+ node types. Also one node type can easily be forgotten in the switch case.

(b) How does the visitor pattern discussed in Sections 7.7.2 and 7.7.3 address the disadvantages?

Instead of switching on the type, each node has an accept function that takes the visitor and makes the visitor visit “this” (i.e. the node itself and it’s children), which would invoke a visit method specific to that type. This eliminates the large switch statement, but still contains a lot of duplicated code (you have to have an accept() method in every node class).

The reflective visitor pattern eliminates the need for the accept() method through the dispatch() method. It looks through the available node classes and finds the “closest match”.

3. (optional - but recommended) Follow the Lab associated with Crafting a Compiler Chapter 7
   http://www.cs.wustl.edu/~cytron/cacweb/Chapters/7/lab.shtml
   For this a virtual machine will be provided with the necessary tools pre installed.