EDAN65: Compilers, Exercise set E-13

Problems and Solutions

Görel Hedin

Revised: 2024-10-15

Problem E13-1

A language L is described by following context-free grammar:

```
p1:   E -> E "+" E
p2:   E -> E "*" E
p3:   E -> ID
```

where E is the start symbol, and ID is a terminal symbol representing an identifier. Prove by writing down a left-most derivation that

```
ID "+" ID "*" ID
```

belongs to L. For each derivation step, show which production was used.

Solution

```
E => E "+" E                  (p1)
=>   ID "+" E                 (p3)
=>   ID "+" E "*" E           (p2)
=>   ID "+" ID "*" E          (p3)
=>   ID "+" ID "*" ID         (p3)
```

Note that this is a left-most derivation since the leftmost nonterminal symbol is replaced in each step.

Problem E13-2

Consider the following context-free grammar for a textual representation of a graph with labelled nodes and edges. The start symbol is Graph:

```
Graph -> ElementList
ElementList -> Element ElementList
ElementList -> ε
Element -> Node
Element -> Edge
Node -> ID
Edge -> ID "(" ID "->" ID ")"
```

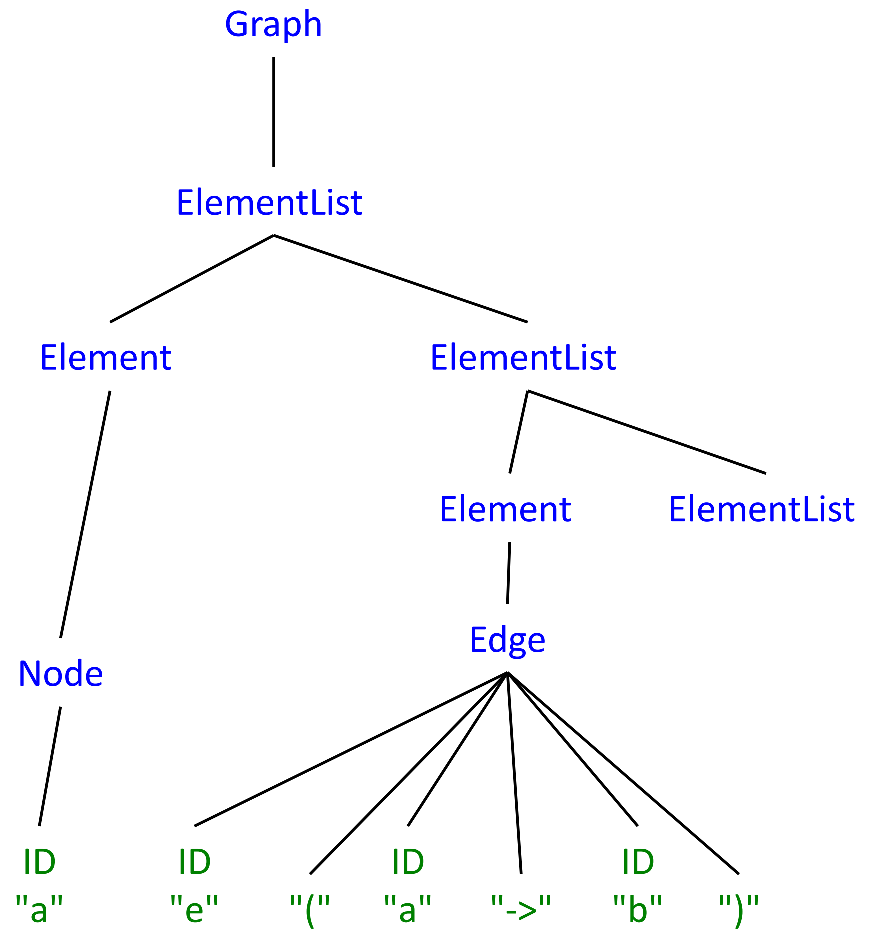
The terminal ID has the following regular expression definition:

```
ID = [a-z]+
```

Draw the parse tree for the following graph:

```
a e(a->b)
```

Solution



Note that:

- the root is labeled by the start symbol
- the terminal symbols are leaves
- each nonterminal has children corresponding to the right-hand side of one of its productions (none in case of the empty production)

Problem E13-3

Consider the following context-free grammar for a textual representation of a graph with labelled nodes and edges. The start symbol is Graph:

```
p1: Graph -> ElementList
p2: ElementList -> Element ElementList
p3: ElementList -> ε
p4: Element -> Node
p5: Element -> Edge
p6: Node -> ID
p7: Edge -> ID "(" ID "->" ID ")"
```

This grammar is not LL(1). Explain why.

Solution

The grammar is not LL(1) since the nonterminal Element has two productions (p4 and p5) with an indirect common prefix (ID).

Problem E13-4

The following grammar contains a common prefix. Transform the grammar to an equivalent grammar where the common prefix is eliminated.

```
Graph -> ElementList
ElementList -> Element ElementList
ElementList -> ε
Element -> Node
Element -> Edge
Node -> ID
Edge -> ID "(" ID "->" ID ")"
```

Solution

Step 1: Substitute the definitions of Node and Edge into the Element productions:

```
Graph -> ElementList
ElementList -> Element ElementList
ElementList -> ε
Element -> ID
Element -> ID "(" ID "->" ID ")"
```

Step 2: Factor out the common prefix by introducing a new nonterminal ElementRest:

```
Graph -> ElementList
ElementList -> Element ElementList
ElementList -> ε
Element -> ID ElementRest
ElementRest -> ε
ElementRest -> "(" ID "->" ID ")"
```

The common prefix is now eliminated.

Problem E13-5

The following grammar is left-recursive and therefore not LL(1). Transform the grammar to an equivalent grammar that is LL(1). Argue for that your resulting grammar is LL(1).

```
T -> T "*" F
T -> F
F -> ID
F -> "(" T ")"
```

Solution

Step 1: Replace left recursion with right recursion.

```
T -> F "*" T
T -> F
F -> ID
F -> "(" T ")"
```

Step 2: Eliminate the common prefix

```
T     -> F TRest
TRest -> ε
TRest -> "*" T
F     -> ID
F     -> "(" T ")"
```

A grammar is LL(1) if the LL(1) parse table is without conflicts. The T row cannot have any conflicts since T has only one production. The F row clearly has no conflicts since the two productions start with different tokens.
For TRest, we need to compare FOLLOW of its first production (which is {EOF, ")"}) with FIRST of its second production (which is {"*"}). Since these sets do not overlap, there is no conflict here either. The grammar above is therefore LL(1).

Problem E13-6

Consider the following context-free grammar for a textual representation of a graph with labelled nodes and edges. The start symbol is G:

```
p1: G -> ElemList
p2: ElemList -> Elem ElemList
p3: ElemList -> ε
p4: Elem -> Node
p5: Elem -> Edge
p6: Node -> ID
p7: Edge -> ID "(" ID "->" ID ")"
```

The terminal ID has the following regular expression definition:

```
   ID = [a-z]+
```

Show how an LR parser would parsing the following program:

```
   a e(a->b)
```

Show the stack contents, the remaining input, and the parsing action taken in each step.

Solution

To the left, the stack and remaining input is shown, separated by an asterisk.
To the right, the next action is shown.
We consider the tokenized input:

```
ID ID ( ID -> ID )
```

The LR parse is then:

```
* ID ID ( ID -> ID )        shift ID
ID * ID ( ID -> ID )        reduce p6
Node * ID ( ID -> ID )      reduce p4
Elem * ID ( ID -> ID )      shift ID
Elem ID * ( ID -> ID )      shift (
Elem ID ( * ID -> ID )      shift ID
Elem ID ( ID * -> ID )      shift ->
Elem ID ( ID -> * ID )      shift ID
Elem ID ( ID -> ID * )      shift )
Elem ID ( ID -> ID ) *      reduce p7
Elem Edge *                 reduce p5
Elem Elem *                 reduce p3
Elem Elem ElemList *        reduce p2
Elem ElemList *             reduce p2
ElemList *                  reduce p1
G *                         accept
```

Problem E13-7

Consider the following abstract grammar for a graph of nodes and edges.

```
G ::= Element*;
abstract Element;
Node:Element ::= <ID>;
Edge:Element ::= Src:NodeUse Dst:NodeUse;
NodeUse ::= <ID>;
```

Suppose there is an attribute

```
   Node NodeUse.maybeNode()
```

that refers to the node of the same name as the NodeUse, or to null if there is no such node.

Define a boolean synthesized attribute wellFormed() for Edge nodes, that is true iff both its source and destination nodes exist.

Solution

Implement Edge.wellFormed(). Discover that a helper attribute NodeUse.wellFormed() would be convenient:

```
syn boolean Edge.wellFormed() =
  getSrc().wellFormed() &
  getDst().wellFormed();
```

Implement NodeUse.wellFormed():

```
syn boolean NodeUse.wellFormed() =
  maybeNode()!=null;
```

Problem E13-8

Consider the following abstract grammar for a graph of nodes and edges.

```
G ::= Element*;
abstract Element;
Node:Element ::= <ID>;
Edge:Element ::= Src:NodeUse Dst:NodeUse;
NodeUse ::= <ID>;
```

Suppose there is an attribute

```
   Node NodeUse.maybeNode()
```

that refers to the node of the same name as the NodeUse, or to null if there is no such node.

To represent missing nodes, introduce a new AST class UnknownNode, and create an object of this class as an NTA of the root.

Define a new attribute

```
   Node NodeUse.node()
```

that refers to the UnknownNode object instead of to null.

Solution

The new class is

```
  UnknownNode:Node;
```

The NTA:

```
  syn nta UnknownNode G.unknown() =
    new UnknownNode("Unknown");
```

Propagation of the UnknownNode object downwards in the AST:

```
  inh UnknownNode NodeUse.theUnknown();
  eq G.getElement().theUnknown() =
    unknown();
```

Definition of node():

```
  syn Node NodeUse.node() {
    if (maybeNode()==null)
      return theUnknown();
    else
      return maybeNode();
  }
```

Problem E13-9

Consider the following abstract grammar for a graph of nodes and edges.

```
G ::= Element*;
abstract Element;
Node:Element ::= <ID>;
Edge:Element ::= Src:NodeUse Dst:NodeUse;
NodeUse ::= <ID>;
```

Implement an attribute

```
   Node NodeUse.maybeNode()
```

that refers to the node of the same name as the NodeUse, or to null if there is no such node.

Solution

Implement the attribute. Discover that it would be convenient with a helper attribute lookup:

```
syn Node NodeUse.maybeNode() =
  lookup(getID());
```

Implement the lookup attribute. Discover that another helper attribute localLookup would be convenient.

```
inh Node NodeUse.lookup(String s);
eq G.getElement().lookup(String s) {
  for (Element e : getElementList()) {
    Node n = e.localLookup(s);
    if (n != null) return n;
  }
  return null;
}
```

Implement localLookup as well.

```
syn Node Element.localLookup(String s) =
  null;
eq Node.localLookup(String s) {
  if (s.equals(getID())) return this;
  return null;
}
```

Problem E13-10

Consider the following abstract grammar for a graph of nodes and edges.

```
G ::= Element*;
abstract Element;
Node:Element ::= <ID>;
Edge:Element ::= src:NodeUse dst:NodeUse;
NodeUse ::= <ID>;
```

Define an attribute

```
  int G.nbrOfEdges()
```

that counts the number of edges in the graph. Use a collection attribute to compute the attribute. You can use a class Counter with the following implementation:

```
public class Counter {
  private int count = 0;
  public void add(int n) {
    count = count + n;
  }
  public int count() {
    return count;
  }
}
```

Solution

Declare the collection:

```
  coll Counter G.edgeCount()
       [new Counter()] with add;
```

Let each Edge contribute 1 to the counter:

```
  Edge contributes 1
  to G.edgeCount()
  for theGraph();
```

Propagation of a reference to the graph to all edges:

```
  inh G Edge.theGraph();
  eq G.getElement().theGraph() = this;
```

Define G.nbrOfEdges:

```
  syn int G.nbrOfEdges() =
    edgeCount().count();
```

Problem E13-11

Consider the following abstract grammar for a graph of nodes and edges.

```
G ::= Element*;
abstract Element;
Node:Element ::= <ID>;
Edge:Element ::= src:NodeUse dst:NodeUse;
NodeUse ::= <ID>;
```

Suppose there is an attribute

```
   Node NodeUse.maybeNode()
```

that refers to the node of the same name as the NodeUse, or to null if there is no such node.

If there is an edge a->b, we say that the node b is a target of a. Implement a collection attribute Node.targets() containing all the target nodes for a given node.

For sets, you may use the Java type HashSet.

Solution

Declare the collection attribute

```
  coll HashSet<Node> Node.targets()
       [new HashSet<Node>()] with add;
```

Each Edge contributes its target node to the source node's target set (if the dst and src nodes exist).

```
  Edge contributes getDst().maybeNode()
  when getDst().maybeNode() != null &
       getSrc().maybeNode() != null
  to Node.targets()
  for getSrc().maybeNode();
```

Problem E13-12

Consider the following abstract grammar for a graph of nodes and edges.

```
G ::= Element*;
abstract Element;
Node:Element ::= <ID>;
Edge:Element ::= Src:NodeUse Dst:NodeUse;
NodeUse ::= <ID>;
```

If there is an edge a->b, we say that the node b is a target of a. Suppose there is a collection attribute

```
  Set<Node> Node.targets()
```

containing all the target nodes for a given node.

The reachable set of a node is the transitive set of target nodes. Implement the reachable set as a circular attribute. You can use the Java class HashSet with operations add and addAll, for adding one element or a set of elements.

Solution

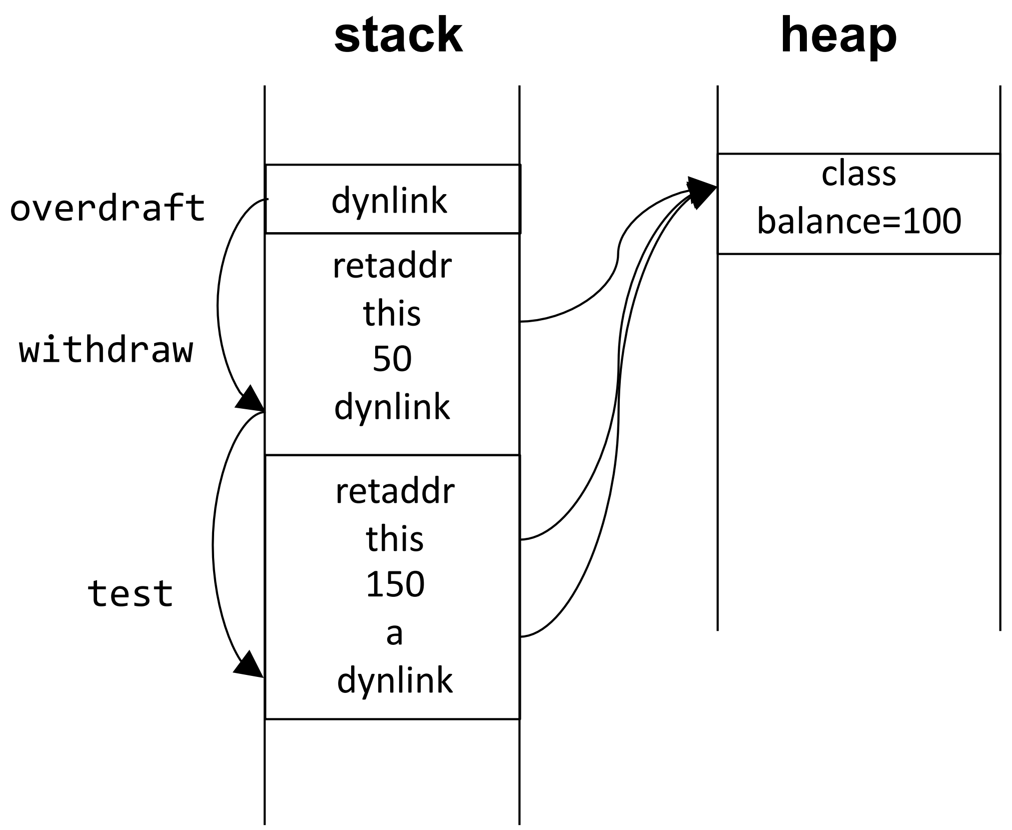
```
syn Set<Node> Node.reachable()
    circular [new HashSet<Node>()] {
    HashSet<Node> s =
      new HashSet<Node>();
    for (Node t : targets()) {
      s.add(t);
      s.addAll(t.reachable());
    }
    return s;
}
```

Problem E13-13

```
class Account {
  int balance = 0;
  void deposit(int amount) {
    balance = balance + amount;
  }
  void withdraw(int amount) {
    if (amount > balance)
      overdraft(amount – balance);
    else
      balance = balance – amount;
  }
  void overdraft(int am) {
    /* PC */
    System.out.println
      ("Overdraft with amount "+am);
  }
}
void test() {
  Account a = new Account();
  a.deposit(100);
  a.withdraw(150);
}
```

Suppose that test() is called. Draw the situation on the stack and heap at /* PC */. Your sketch should include dynamic link, fields, local variables, "this" pointer, and arguments including their values. Arguments should be passed on the stack. Explain the contents of the withdraw activation.

Solution

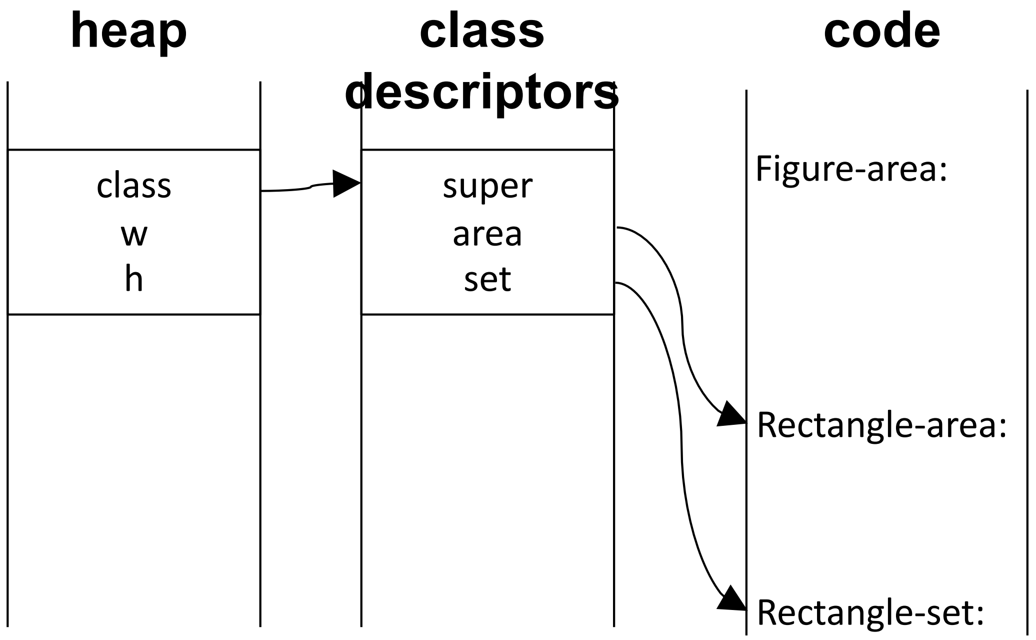


The withdraw activation contains:

- the dynamic link (pointer to previous activation)
- the argument to overdraft (50 in this case)
- the static link ("this") for the overdraft method, i.e., the Account object (viewed as argument 0).
- the return address, i.e., the point in the withdraw code to which the overdraft method should return.

Problem E13-14 Solution

```
class Figure {
  int area() { return 0; }
}
class Rectangle extends Figure {
  int w;
  int h;
  void set(int w, int h) {
    this.w = w;
    this.h = h;
  }
  int area() {
    return w * h;
  }
  ...
}
```



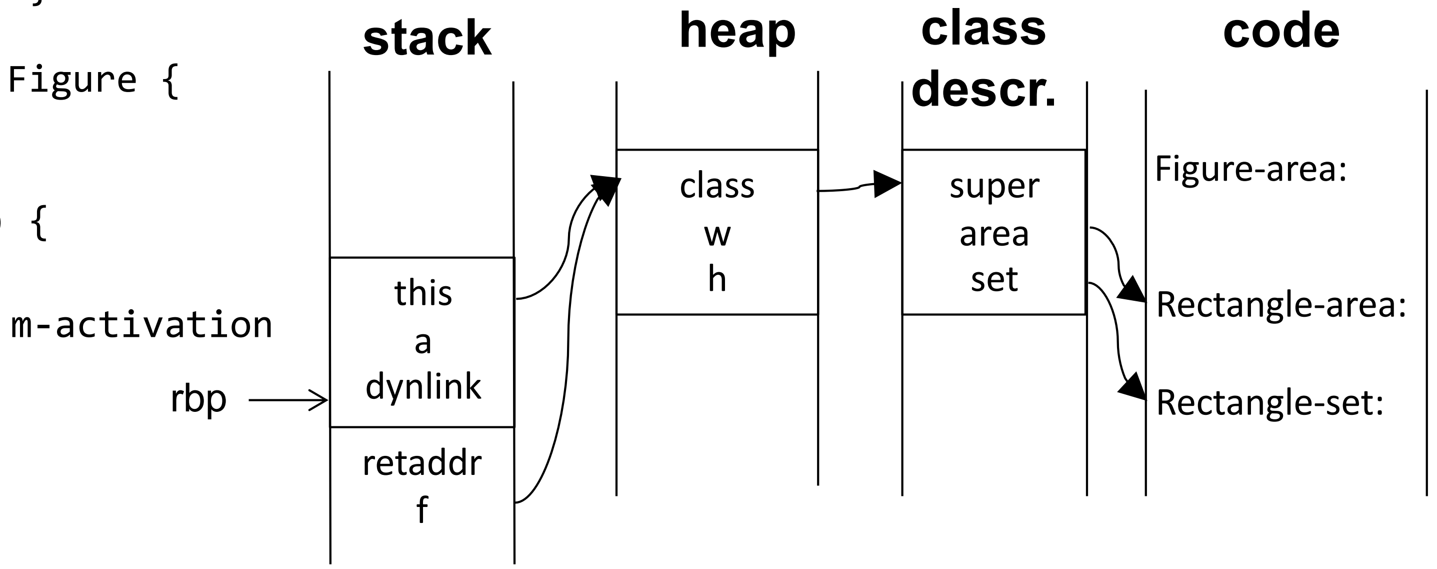
Suppose this language is implemented using virtual tables. Draw a sketch over the memory showing a Rectangle object, its class descriptor, and its code. Your sketch should include fields, class link, virtual table, and methods.

Problem E13-15

```
class Figure {
  int area() { return 0; }
}
class Rectangle extends Figure {
  int w;
  int h;
  void set(int w, int h) {
    this.w = w;
    this.h = h;
  }
  int area() {
    return w * h;
  }
  ...
}
void m(Figure f) {
  int a;
  a = f.area(); // S
}
```

This language is implemented using virtual tables. Draw the situation on stack and heap at statement S, right before the call to f.area() is made. Assume f is a Rectangle object and include the class descriptor in your sketch. Sketch the code for the statement S. Use x86 instructions according to the assignment 6 cheatsheet. Add comments to the code, explaining what it does.

Solution



```
pushq 16(%rbp)          # push "this" arg (f)
movq 16(%rbp), %rax     # f -> rax
movq (%rax), %rax       # class -> rax
callq 8(%rax)           # call area
popq                    # pop "this" arg
movq %rax -8(%rbp)      # return val -> a
```