Lunds Tekniska Högskola
Computer Science
Lennart Andersson

Examination
EDA180/DATE17
2010–08–19, 8.00-13.00

Examination in Compiler Construction

The total number of points is 40, 10 for each problem. In addition you can get a maximum of another 4 points from the seminar exercises 2010. At most 20 points will be required to pass the exam. Simple solutions result in more points than complex solutions. Grammars and implementations with simple and descriptive names result in more points than those with non-descriptive or misleading names.

Write clearly and legibly. Hand writing that is difficult to decipher results in point reduction. Messy layout that obscures the solution results in point reduction. Rewrite your solution on a new sheet of paper if necessary. Start each of the solutions (1, 2, 3, 4) on a separate sheet of paper.

The following documents from the course web page may be used during the exam

- *Abstract grammar and Aspects* for JastAdd
- *Icode specification*

You may also use a dictionary from English to your native language.

Prerequisites for writing the exam: You must have completed and passed the programming assignments.

1 Lexical analysis

A language has the following token definitions:

```
THROW = "throw"
THROWS = "throws"
ID = [a-z]+
```

The usual disambiguation rules of rule priority and longest match apply to these definitions.

a. Draw three small DFAs, one for each of the token definitions. Mark the final state of each DFA with the token in question.

b. Construct a DFA that combines the three small DFA's by joining their start states and eliminating nondeterminism. Mark each final state by the appropriate token.

c. Describe how the longest match rule can be implemented.

2 Syntax analysis

In most programming languages the arguments in a function call must appear in the same order as in the declaration of the function. In this problem we will consider an approach where the arguments may alternatively be specified in any order as assignments to the formal parameters. In a function call the two specification modes must not be mixed. An example:

```
float volume(float radius, float height) {
  return Math.PI * radius * radius * height;
}

...
  volume(height = 2.0, radius = diameter/2.0)
...
  volume(diameter/2.0, 2.0)
```

There may be zero och more parameters. You may assume that there is a grammar for expressions with `expr` as the start symbol. An identifier may is denoted by `ID`.

a. Construct an EBNF grammar for function calls that does not check for mixed modes.

b. Construct an EBNF grammar for function calls that does not accept mixed modes and may require more than one token look-ahead.

c. Construct an EBNF grammar for function calls that does not accept mixed modes and uses at most one token look-ahead.

3 Abstract grammar and Semantic analysis

This problem continues to the previous problem.

a. Define an abstract grammar for function declarations and function calls with parameter assignments using JastAdd notation. You may assume that abstract grammars have been defined for other statements, expressions and types extending the classes `Statement`, `Expr`, and `Type` respectively.

b. Define a JastAdd aspect to check that a function call with parameter assignments assigns every parameter exactly once.

c. Define a visitor to check that all parameter names in a function declaration are unique. You may assume that every class has an `accept` method.

```
Object accept(Visitor visitor, Object object) {
  return visitor.visit(this, object);
}
```

and that there is a traversing visitor with methods like

```
Object visit(Node node, Object object) {
// The omitted code will call the accept method for each child of
// node and return the same object as the last accept call
// or object if there are no children.
}
```

Show all methods that must be overridden. You may use classes from `java.util` even if you have forgotten the correct names of classes and methods.

4 Code generation and Runtime systems

a. This problem continues the previous one.

The following method generates code for standard positional arguments for some abstract grammar

```
public Address CallExpr.codeGen(Code p, TempFactory f, int level) {
  int args = getExprList().getNumExpr();
  p.addInstruction(new Alloc(args));
  for (int i=0; i<args; i++) {
     Operand opr = getExprList().getExpr(i).codeGen(p, f, level);
     p.addInstruction(new Move(opr, new Argument(i)));
  }
  p.addInstruction(new icode.Call(//-- arguments omitted --//));
  p.addInstruction(new Dealloc(args));
  return new Result();
}
```

Modify it to generate code for assignment arguments using your abstract grammar. You may assume that the use of the function name is linked to the declaration of the function via the attribute `decl`. Omit the arguments of the `Call` instruction.

b. Use a simple language similar to the one used in the later assignments or in your project to construct an example where the static and dynamic links in an activation record differ. Show the state of the stack when this occurs.