Examination in Compilers, EDAN65

Department of Computer Science, Lund University

2024–10–29, 14.00-19.00

Note! **Your exam will be marked only if you have completed all six programming lab assignments in advance.**

Start each solution (1, 2, 3, 4) on a separate sheet of paper. Write only on one side of each sheet. Write your *anonymous code* and *personal identifier*[1] on every sheet of paper. Write clearly and legibly. Try to find clear, readable solutions with meaningful names. Unnecessary complexity will result in point reduction.

The following documents may be used during the exam:

- *Reference manual for JastAdd2*
- *x86 Cheat Sheet*

Max points: 60
For grade 3: Min 30
For grade 4: Min 40
For grade 5: Min 50

Good luck!

[1]The *personal identifier* is a short phrase, a code or a brief sentence of your choice. It can be anything, but not something that can reveal your identity. The purpose of this identifier is to make it possible for you to identify your exam in case something goes wrong with the anonymous code on the exam cover (such as if it is confused with another code due to sloppy writing).

1 Lexical analysis

Alice and Bob are defining a language with decimal, octal, and hexadecimal literals. A decimal literal should consist of one or more decimal digits as defined by the regular expression `[0-9]+`. An octal literal should always start with a `0` followed by one or more octal digits (`0[0-7]+`). A hexadecimal literal should always start with `0x` followed by one or more hexadecimal digits (`0x[0-9a-f]+`). They write down some test cases to define what tokens different example literals should result in:

Example	Token
`0`	`DECIMAL`
`910`	`DECIMAL`
`750`	`DECIMAL`
`019`	`DECIMAL`
`00`	`OCTAL`
`007`	`OCTAL`
`0x19`	`HEXADECIMAL`
`0x1a7e`	`HEXADECIMAL`

a) Alice and Bob realize there is some ambiguity in their regular expressions, but that they can rely on rule priority to resolve it. Which of the above examples will be disambiguated by rule priority? In what order should Alice and Bob write down the rules for `DECIMAL`, `OCTAL`, and `HEXADECIMAL` in their scanner specification to get the desired result? (3p)

b) Translate each of the regular expressions to a separate DFA. The final states should be labelled with the appropriate tokens. (3p)

c) Construct an NFA by joining the DFAs into a single automaton with a joint start state. Label each state with a unique number. Then construct an equivalent DFA where each state is labelled with the set of corresponding states in the original NFA. The DFA should have as few states as possible. Each final state should be labelled with the appropriate token. (6p)

d) Suppose there is an additional regular expression for whitespace (`[\ ]`). Consider the string `"0x19 007 910"`. If the longest match rule is used, the scanner will return `HEXADECIMAL("0x19")` as the first token. Suppose the scanner does not use the longest match rule for disambiguation. What other token(s) could it then return as the first token?

(3p)

2 Context-Free Grammars

Consider the following context-free grammar for lambdas and multiplication expressions. (`ID` is a predefined token for identifiers and `EOF` is a token representing end of file.)

p_0: `Start -> Exp EOF`
p_1: `Exp -> "(" IdList ")" Exp`
p_2: `IdList -> ID MoreIds`
p_3: `MoreIds ->` ϵ
p_4: `MoreIds -> "," ID MoreIds`
p_5: `Exp -> Exp "*" ID`
p_6: `Exp -> ID`

a) The grammar is ambiguous. Prove this by finding a sentence that can be matched with two different parse trees, and draw the two trees. (5p)

b) Construct an equivalent unambiguous grammar. The grammar should be on canonical form.

(Depending on how you do the disambiguation, the parse tree for the example sentence you used in 2a may be more similar to one or the other of the two trees you drew. It does not matter which one.) (5p)

c) Construct a grammar on EBNF form that is equivalent to the original ambiguous grammar. The EBNF grammar should have as few nonterminals as possible. (Recall that EBNF allows the use of alternatives, repetition, optionals, and parentheses.) (5p)

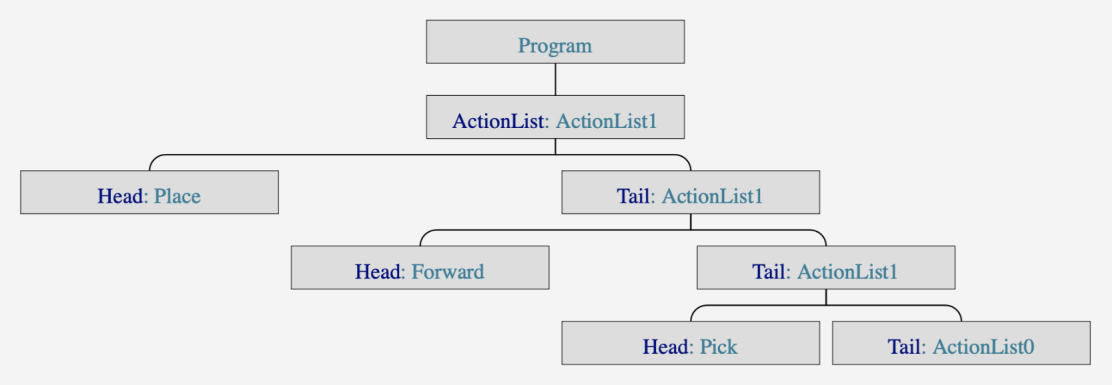
d) Construct the LL(1) parser table for the original ambiguous grammar. (5p)

3 Program analysis

The following abstract grammar defines a robot language with actions `Forward`, `Pick`, and `Place`. At the Forward action, the robot takes a step forward. At the Pick action, the robot picks up an item from the floor. However, the robot can hold at most three items, so if it already has three items, the Pick action has no effect. At the Place action, the robot places one of its items on the floor. If the robot does not hold on to any item, the Place action has no effect. The actions are arranged in a sequence, using `ActionList` nodes: `ActionList0` is an empty list, and `ActionList1` is a list with at least one action.

```
Program ::= ActionList;
abstract ActionList;
ActionList0 : ActionList;
ActionList1 : ActionList ::= Head:Action Tail:ActionList;
abstract Action;
Forward : Action;
Pick : Action;
Place : Action;
```

The figure below shows the AST for an example robot program where the robot first does a Place, then a Forward, and finally a Pick action. Initially, the robot has no items, so in this case, the Place action will have no effect, and after executing the last action, the robot will be holding one item.



Solve the problems below using attribute grammars, and without using `instanceof` or `getParent()`. Make sure to clearly indicate the kind of each attribute you declare, i.e., if it is a synthesized (`syn`), an inherited (`inh`), or a collection (`coll`) attribute.

a) Define the attributes `Action.numItemsBefore()` and `Action.numItemsAfter()`. The values should be the number of items the robot holds before and after executing the action. In the example above, the value of `numItemsAfter()` should be 0 for Place and Forward, and 1 for Pick. The values for `numItemsBefore()` should be 0 for all three actions.

You may assume the existence of the following two methods:

```
int min(int v1, int v2) { ... } // returns the minimum of v1 and v2
int max(int v1, int v2) { ... } // returns the maximum of v1 and v2
```

(5p)

b) Add an integer attribute `Program.failedPicks()` that uses a collection attribute to count the number of failed Pick actions in the program, i.e., Pick actions for which the robot already holds three items. You may use the attributes you defined in problem 3a). For the collection attribute, you may use the following `Counter` class:

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

(5p)

4 Code generation and run-time systems

Consider the following program in a Java-like language:

```
void main() {
    Stack s = new Stack();
    s.push(3);
    s.push(7);
    s.pop();
    s.push(9);
    // *** PC1 ***
    print("Sum of all stacked elements is " + s.sum()); // Should print 12
}

class Stack {
    Entry top = null;
    void push(int x) {
        Entry e = new Entry(x);
        if (top == null)
            top = e;
        else
            e.next = top;
            top = e;
    }
    void pop() {
        if (top != null)
            top = top.next;
    }
    int sum() {
        if (top == null)
            return 0;
        else
            return top.sum();
    }
}

class Entry {
    int value = 0;
    Entry next = null;
    Entry(int v) {
        this.value = v;
    }
    int sum() {
        int v = value;
        if (next == null)
            // *** PC2 ***
            return v;
        else
            return next.sum() + v;
    }
}
```

In the following problems, you should draw the situation on the stack and heap. You should assume that the code is not optimized, and that no garbage collection is performed. Your drawing should include:

- all current frames on the stack, including dynamic link and any local variables and arguments.
- the values of all local variables and arguments, including static links (`this` pointers)
- a label on each frame with the method name, e.g., `main` or `Stack.sum`.
- all objects that have been created on the heap, including their instance variables
- the values of all instance variables
- a label on each object with the class name, e.g., `Stack` or `Entry`

a) Execution starts in the method `main`. Draw the situation on the stack and heap when the program counter is at the position marked with `*** PC1 ***`. (5p)

b) Consider now that the execution has continued until the program counter is at the position marked with `*** PC2 ***`. Extend your drawing to show the additional frames created. (5p)

c) In your drawing from 4a) and 4b), mark all root pointers with an `R`, all live objects with an `L`, and all dead objects with a `D`. (5p)