Examination in Compilers, EDAN65

Department of Computer Science, Lund University

2019–10–30, 08.00-13.00

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

Start each solution (1, 2, 3) on a separate sheet of paper. Write only on one side of each sheet. Write your *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*

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

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

[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

A language has the following token definitions:

```
VAR = "var"
VAL = "val"
ID = [a-z]+
```

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

a) Draw three small finite automata, one for each of the token definitions. Mark the final state of each automaton with the token in question. (3p)

b) Combine the three automata to an NFA by joining their start states, and mark each state with a unique number. Mark also the final states with the token in question. (1p)

c) Construct a DFA that is equivalent to the NFA. Mark each DFA state with the state numbers from the corresponding states in the NFA. Mark each final state by the appropriate token. (5p)

d) Consider the following strings: `"var"`, `'"vara"`, and `"val"`.

First, which tokens do these strings match? Second, for which of the strings is *longest match* needed, and what would happen if we didn't have that rule? Third, for which of the strings is *rule priority* needed, and what would happen if we didn't have that rule?

(6p)

2 Grammars

Alice and Bob are implementing a language for boolean function definitions and expressions. The functions may have optional parameters with default values. Function call arguments for optional parameters are named. Here is an example program P in the language:

```
func f1(x, y, z = true) = f2(x) and (f2(y, b = true) or z);
func f2(a, b = false) = a and not b or not (f3() and not not b);
func f3(c=true) = not c;
func main() = f1(true, false) and f2(true) and f3(c=false);
```

Alice and Bob start by constructing an abstract grammar for the language:

```
Program ::= FuncDef*;
FuncDef ::= IdDecl ParamDecl* Exp;
ParamDecl ::= IdDecl [DefaultValue:Exp];
abstract Exp;
OrExp : Exp ::= Left:Exp Right:Exp;
AndExp: Exp ::= Left:Exp Right:Exp;
NotExp : Exp ::= Exp;
TrueExp : Exp;
FalseExp : Exp;
ParExp : Exp ::= Exp;
CallExp : Exp ::= IdUse Arg*;
Arg ::= [IdUse] Exp;
IdDecl ::= <ID:String>;
IdUse : Exp ::= <ID:String>;
```

The next step is to construct a parsing grammar. Alice and Bob start by writing down a context-free grammar E for some of the boolean expressions.

```
exp → exp "or" exp
exp → exp "and" exp
exp → "not" exp
exp → "(" exp ")"
exp → ID
```

a) Alice's and Bob's first idea is to try to write a recursive-descent parser and they would therefore like to construct an LL(1) parse table for the grammar E. Help them by constructing this table.

(4p)

b) Alice and Bob decide that recursive-descent might be too difficult for them, so they instead try the E grammar on an LALR parser generator. But the generator gives several error messages, including the following (tokens within square brackets are lookahead tokens in the LR items):

```
ERROR: SHIFT/REDUCE conflict:
  exp = exp . OR exp [AND]
  exp = exp AND exp . [OR]
```

Help them to understand this problem by writing down an example expression that would be parsed as two different trees depending on if the *shift* or *reduce* action is taken in the above state. For each of these parse trees, draw the tree, and write down if shift or reduce was used for the above state.

(6p)

c) Alice and Bob now understand that they have to think about associativity and priorities for the E grammar. Normal precedence rules apply, so that `not` has higher priority than `and`, and `and` has higher priority than `or`. Both `and` and `or` should be left-associative. Help Alice and Bob by constructing an equivalent unambiguous grammar that supports these precedence rules. The grammar should be on canonical or BNF form (not EBNF).

(6p)

d) Now, help Alice and Bob by writing the remaining parts of the context-free grammar so that the example program P can be parsed using LALR parsing. The grammar should be on canonical or BNF form (not EBNF). Use the same names as in the abstract grammar when this is relevant. Note that parameter and argument lists are comma-separated, but that extra commas are not allowed. For example, the expression `p(a, b, )` should not be allowed by the grammar. Don't forget to add the expressions that were not covered by grammar E.

(8p)

3 Program analysis and code generation

a) Recall that the language from problem 2 supports optional parameters with default values. When declaring a function definition, all ordinary parameters (without default values) must be declared first, and then all the optional parameters (those with default values) should follow. It is therefore a static-semantic error if an ordinary parameter follows an optional parameter like in the following example:

```
func f(x, y, z = true, u, v, w = false, s, t) = ...
```

For this case, Alice and Bob would like to print two error messages:

```
Error at line 1, column 24: ordinary parameter u follows optional parameter
Error at line 1, column 41: ordinary parameter s follows optional parameter
```

Use reference attribute grammars to implement a boolean attribute `ordinaryFollowsOptional()` for `ParamDecl`, that captures this behavior. I.e., for the above example, the attribute should be *true* for `u` and `s`, but *false* for the other parameter declarations.

Note! You may neither use `instanceof` nor the `getParent()` method.

(6p)

b) Alice and Bob have implemented an interface `Visitor` with a method

```
void visit(C node);
```

for each class `C` in the abstract grammar. They have furthermore defined a method `accept` in `ASTNode`, and overridden it for each concrete subclass as follows:

```
void accept(Visitor v) {
  v.visit(this);
}
```

Then they have implemented a class `TraversingVisitor` that implements the `Visitor` interface and that provides a default implementation of each `visit` method. Each such default implementation calls the `accept` method on each of the children of the visited node.

Implement a visitor `ErrorMessageVisitor` that prints the error messages discussed in problem 3a on a `PrintStream` such as `System.out`. The main program should be able to call the visitor using a static method `print` as follows:

```
Program root = ... // Parse in program
ErrorMessageVisitor.print(root, System.out);
```

You may assume that `ASTNode` has two `int` methods `getLine()` and `getCol()` to get the line and column number for the node. You may also use the attribute `ordinaryFollowsOptional` from problem 3a.

(6p)

c) Alice and Bob would also like to generate code for the language.

At runtime, a called function expects *all* its arguments to be pushed on the stack by the caller, even if the function call in the source code leaves out some optional arguments. The calling function thus needs to push the default values for left out arguments. The called function furthermore expects the arguments to be pushed in the same (but reversed) order as in the function definition, even if the function call in the source code lists optional arguments in another order.

For example, consider the following function definition:

```
func f(x, y, z=true, u=false) = ... ;
```

The table below shows some example source code calls, and in what order the arguments should be pushed

```
// Source code call:     // Order in which arguments should be pushed:
f(a, b)                  // false, true, b, a
f(a, b, z=c)             // false, c, b, a
f(a, b, u=d)             // d, true, b, a
f(a, b, z=c, u=d)        // d, c, b, a
f(a, b, u=c, z=d)        // c, d, b, a
```

In Alice's and Bob's code generator, each class in the abstract grammar has a `void` method `genCode(PrintStream s)` for generating x86 64-bit code. They have already implemented large parts of the code generator. In particular:

- The `genCode` method has been implemented for all classes, except `CallExp`.
- For each expression, `genCode` leaves the result in `%rax`.
- A called function leaves its result in `%rax`.
- There is an attribute `syn FuncDef CallExp.decl()` that can be used to find the function definition for a call.
- The code generator is called only if there are no static-semantic errors.
- All static-semantic checks are already implemented, including the following:
  - The argument list of a call must start with all the ordinary (non-optional) arguments.
  - An ordinary argument is not allowed to be named in the call.
  - An optional argument must always be named in the call.

Help Alice and Bob by implementing the `genCode` method for `CallExp`. Introduce more attributes if you need them.

(9p)