Examination in Compilers, EDAN65

Department of Computer Science, Lund University

2019–10–30, 08.00-13.00

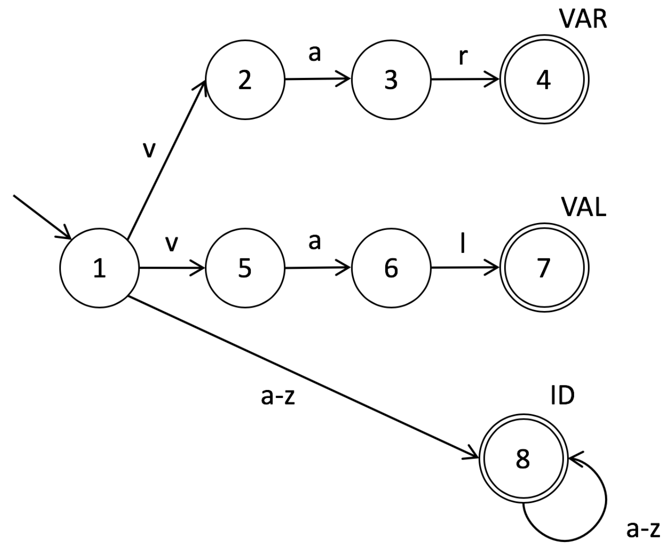
SOLUTIONS

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

1 Lexical analysis

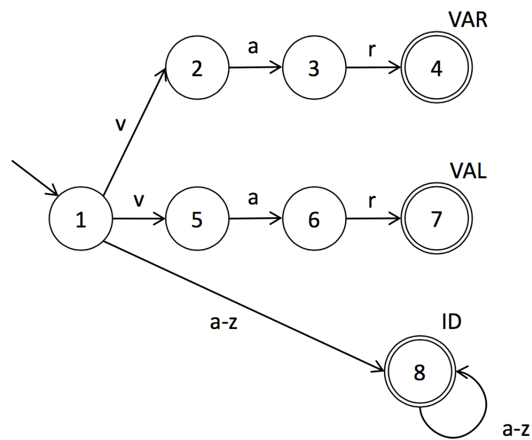
a) (3p)

Three small automata



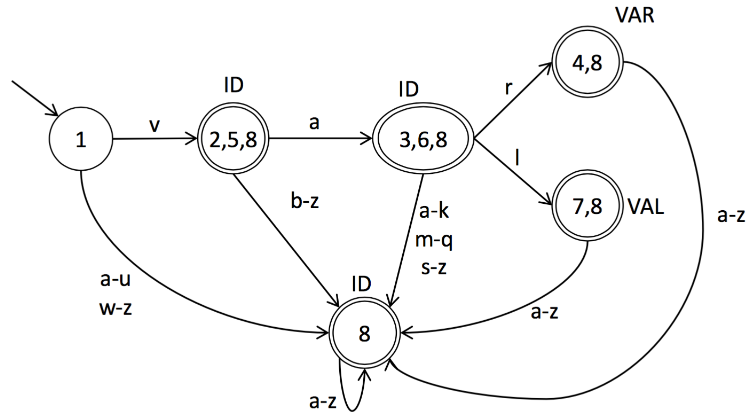
b) (1p)

The combined NFA



c) (5p)

Equivalent DFA



d) (6p)

The strings should match the tokens `VAR`, `ID("vara")`, and `VAL`.

Longest match is needed for all three strings. If we didn't have longest match, the specification would be ambiguous, and shorter tokens could be matched for each of the strings. For "vara" we could match, for example, `ID("v") ID("a") ID("r") ID("a")`, or `ID("va") ID("ra")`, or `VAR` and `ID("a)`. Similarly for "var" and "val".

Rule priority is needed fot the `"var"` and `"val"` strings. If we didn't have rule priority, the specification would be ambiguous, and these strings could match `ID("var")` and `ID("val")` instead of `VAR` and `VAL`.

2 Grammars

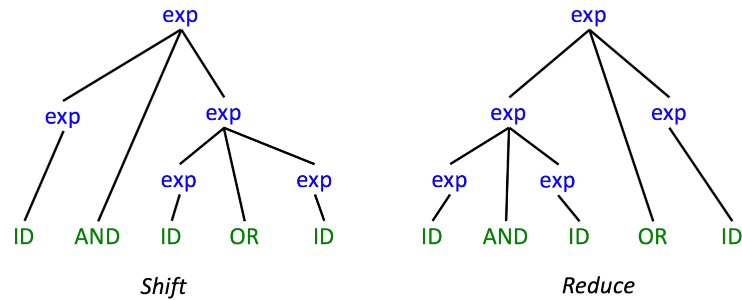
a) (4p)

Assume that the five productions in grammar E are numbered p1, .. , p5. The LL(1) table is then:

```
               "and"    "or"    "not"       "("         ")"       ID
               ------------------------------------------------------------
   exp                          p1,p2,p3    p1,p2,p4              p1,p2,p5
```

b) (6p)

Consider the example `a and b or c`. This is scanned to `ID AND ID OR ID`. When the current token is `OR`, the parser is in the described state of LR-items. The figure below shows which parse tree we get if the parser then shifts or reduces.



c) (6p)

An equivalent unambiguous grammar on BNF form:

```
exp → exp "or" term | term
term → term "and" factor | factor
factor → "not" factor | "(" exp ")" | ID
```

d) (8p)

Remaining grammar: Here is one style of writing itr:

```
program → funcdefs
funcdefs → funcdef funcdefs | ϵ
funcdef → "func" ID "(" optparamdecls ")" "=" exp ";"
optparamdecls → paramdecl moreparamdecls | ϵ
moreparamdecls → "," paramdecl moreparamdecls | ϵ
paramdecl → ID optdefaultexp
optdefaultexp → "=" exp | ϵ
factor → "true" | "false" | ID "(" optargs ")"
optargs → arg moreargs | ϵ
moreargs → "," arg moreargs | ϵ
arg → optname exp
optname → ID "=" | ϵ
```

Here is another style for optparamdecls and optargs:

```
optparamdecls → paramdecls | ϵ
paramdecls → paramdecl | paramdecl "," paramdecls

optargs → args | ϵ
args → arg | arg "," args
```

Alternative formulation, eliminating the optional productions (except for the first, which cannot be eliminated):

```
program → funcdefs
funcdefs → funcdef funcdefs | ϵ
funcdef →
    "func" ID "(" ")" "=" exp ";"
  | "func" ID "(" paramdecls ")" "=" exp ";"
paramdecls →
    paramdecl
  | paramdecl "," paramdecls
paramdecl →
    ID
  | ID "=" exp
factor →
    "true"
  | "false"
  | ID "(" ")"
  | ID "(" args ")"
args →
    arg
  | arg "," args
arg →
    exp
  | ID "=" exp
```

It is not clear from the specification if the function definition list can be empty, so if we require there to be at least one function def, we could also use this alternative rule for funcdefs:

```
funcdefs → funcdef | funcdef funcdefs
```

3 Program analysis and code generation

a) (6p)

There are two possible ways to solve the problem: when defining the value for a ParamDecl, we can either check if the immediately preceding ParamDecl is an optional, or we can check if there is an optional ParamDecl anywhere among the preceding ParamDecls.

Let's first define two helper attributes that makes it easier for us to talk about ordinary and optional parameters.

```
syn boolean ParamDecl.isOrd() = !hasDefaultValue(); // Ordinary parameter
syn boolean ParamDecl.isOpt() = hasDefaultValue(); // Optional parameter
```

The following solution checks only the *immediately* preceding ParamDecl:

```
inh boolean ParamDecl.ordinaryFollowsOptional();
eq FuncDef.getParamDecl(int index).ordinaryFollowsOptional() {
  if (index==0)
    return false;
  else
    return getParamDecl(index).isOrd() && getParamDecl(index-1).isOpt();
}
```

The following solution checks if there is an optional ParamDecl among *any* of the preceding ParamDecls:

```
inh boolean ParamDecl.optToLeft(); // true if it has an opt among its left siblings
eq FuncDef.getParamDecl(int index).optToLeft() {
  if(index==0)
    return false;
  else
    return getParamDecl(index-1).optToLeft();
}
syn boolean ParamDecl.ordinaryFollowsOptional() {
  return isOrd() && optToLeft();
}
```

An alternative to the above solution is to make oFO inherited, and let the equation scan all preceding ParamDecls to see if any is optional.

There are many alternative ways of coding up the two different solutions, but all involve at least one inherited attribute.

b) (6p)

Visitor:

```
public class ErrorPrintingVisitor extends TraversingVisitor {
  public static void print(Program node, PrintStream s) {
    node.accept(new ErrorPrintingVisitor(s));
  }

  private PrintStream s;

  public ErrorPrintingVisitor(PrintStream s) {
    this.s = s;
  }

  @Override
  public void visit(ParamDecl node) {
   if (node.ordinaryFollowsOptional())
      s.println(
        "Error at line " +
        node.getIdDecl().getLine() +
        ", column " +
        node.getIdDecl().getCol() +
        ": ordinary parameter " +
        node.getIdDecl().getID() +
        " follows optional parameter"
        );
  }
}
```

Note, it would also work to let the `print` method call `visit` instead of `accept`, since the `node` arg has the type `Program`). However, this would not work if the `node` arg is declared as `ASTNode`.

```
public static void print(Program node, PrintStream s) {
  new ErrorPrintingVisitor(s).visit(node);
}
```

Other variations of the solutions are also possible. For example, instead of storing a PrintStream field, we could store the error messages in a field, and let the `print` method print them at the end of the method.

c) (9p)

The main idea in the solution is to go through all the parameters in the definition in reverse order, and for each param generate code for either the default value, the named arg, or the ordinary arg.

```
  syn Arg CallExp.lookupArg(String id) {
    for (Arg a : getArgs())
      if (a.hasIdUse() && a.getIdUse().getID().equals(id))
        return a;
    return null;
  }

  public void CallExp.genCode(PrintStream s) {
    FuncDef fd = decl();
    int n = fd.getNumParamDecl();
    for (int i = n-1; i>=0; i--) {
      ParamDecl pd = fd.getParamDecl(i);
      if (pd.hasDefaultValue()) {
        Arg a = lookupArg(pd.getIdDecl().getID());
        if (a != null)
          a.genCode(s);
        else
          pd.getDefaultValue().genCode(s);
      }
      else
        getArg(i).genCode(s);
      s.println("  push %rax");
    }
    s.println("  call " + fd.getIdDecl().getID());
    s.println("  addq $" + n*8 + ", %rsp");
  }
}
```