Examination in Compilers, EDAN65

Department of Computer Science, Lund University

2021–10–29, 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)

A regular expression for INT can be written, for example

```
INT = [0-9] ( [_]* [0-9]+)*
```

or

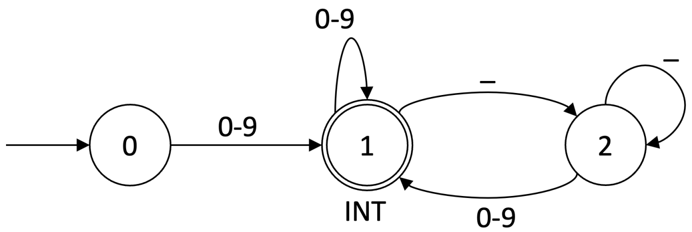
```
INT = [0-9] ( [0-9_]* [0-9])?
```

or

```
INT = [0-9]+ ( [_]+ [0-9]+)*
```

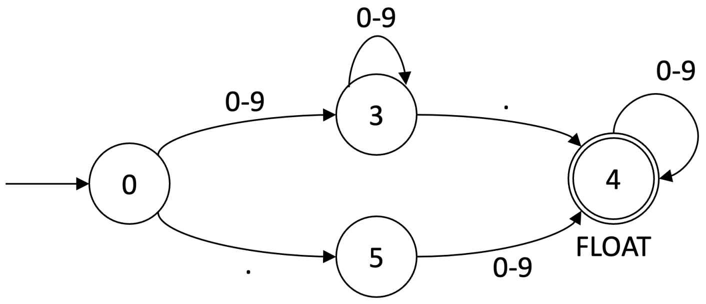
b) (3p)

A DFA for INT



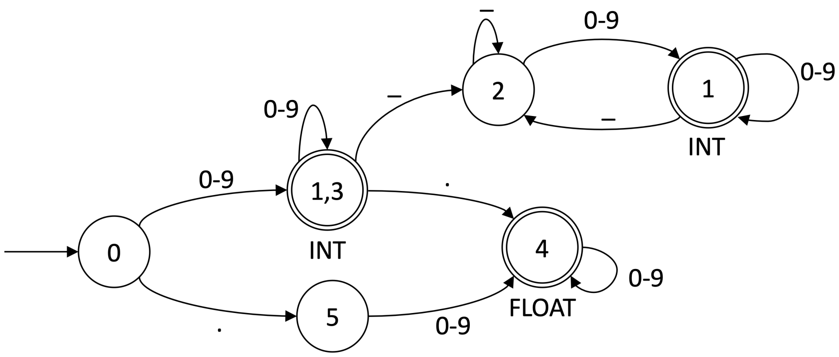
c) (3p)

A DFA for FLOAT



d) (4p)

The combined DFA



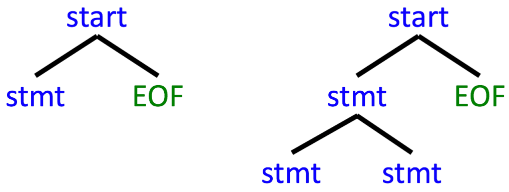
e) (2p)

An INT token for the lexeme `"375_97"` will be output. The scanner will need to read two more characters to decide this: `"_."`

2 Grammars

a) (5p)

For example, the sentence **EOF** can be derived in the following two ways:



b) (5p)

The FOLLOW set for **stmt** is

```
{ "{", "}", ID, EOF }
```

To prove that each of these terminals are in the FOLLOW set, we construct derivations from the start symbol that show that each of them can follow directly after a **stmt** symbol. We can, for example, construct the following derivation:

```
start ⇒ stmt EOF ⇒ "{" stmt "}" EOF
```

Here we see that a **stmt** symbol can be followed by **EOF** and by **"}"** .

To see that also **"{"** and **ID** can follow a **stmt** symbol, we can construct, for example, the following derivations:

```
start ⇒ stmt EOF ⇒ stmt stmt EOF ⇒ stmt ID "=" NUM ";" EOF
```

and

```
start ⇒ stmt EOF ⇒ stmt stmt EOF ⇒ stmt "{" stmt "}" EOF
```

c) (5p)

An equivalent grammar on EBNF form, with as few nonterminals and production alternatives as possible:

```
start → stmt* EOF
stmt → "{" stmt* "}" | ID "=" NUM ";"
```

d) (5p)

Equivalent LL(1) grammar:

p_0 : `start → stmtlist EOF`

p_1 : `stmtlist → stmt stmtlist`

p_2 : `stmtlist →` ϵ

p_3 : `stmt → "{" stmtlist "}"`

p_4 : `stmt → ID "=" NUM ";"`

We can check that this grammar is LL(1) by constructing the LL(1) table:

	"{"	"}"	ID	"="	";"	NUM	EOF
start	p0		p0				p0
stmtlist	p1	p2	p1				p2
stmt	p3		p4				

Since there is no conflict, the grammar is LL(1).

3 Abstract grammars and analysis

a) (5p)

Abstract grammar

```
Program ::= FuncDecl*;
abstract Stmt;
abstract Expr;
FuncDecl ::= <ID> Param* FuncDecl* Stmt*;
Param ::= <ID>;
Assignment:Stmt ::= IdUse Expr;
Returnstmt:Stmt ::= Expr;
CallStmt:Stmt ::= Call;
Call : Expr ::= IdUse Expr*;
IdUse : Expr ::= <ID>;
```

b) (5p)

Attribute grammar computing `FuncCall.levelsOut()`:

```
inh int FuncDecl.level();
inh int FuncCall.level();
eq Program.getChild().level() = 0;
eq FuncDecl.getChild().level() = level()+1;

syn int FuncCall.levelsOut() {
  if (decl() != null) {
    return level() - decl().level();
  }
  return -1;
}
```

c) (5p)

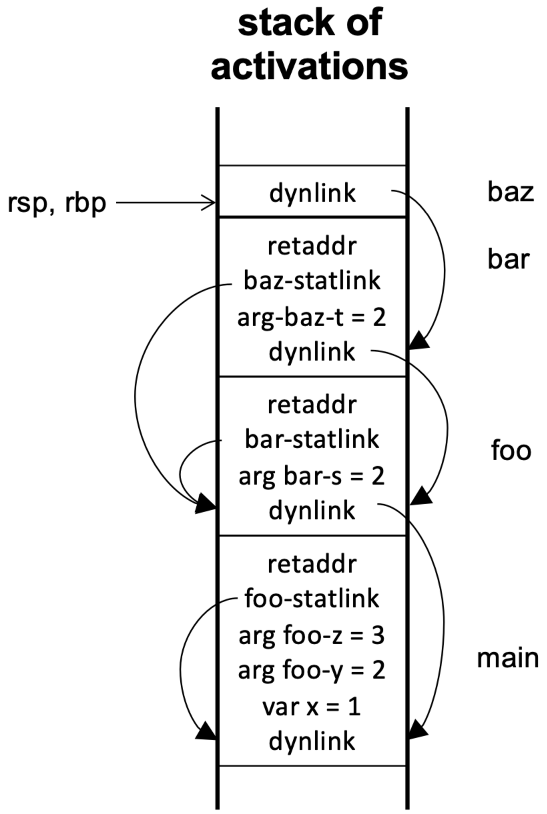
Attribute grammar computing `FuncCall.decl)`:

```
syn FuncDecl FuncCall.decl() = lookup(getIdUse().getID());
inh FuncDecl FuncCall.lookup(String s);

eq Program.getChild().lookup(String s) {
  for (FuncDecl d : getFuncDecls())
    if (d.getID().equals(s)) return d;
  return null;
}

inh FuncDecl FuncDecl.lookup(String s);
eq FuncDecl.getChild().lookup(String s){
  for (FuncDecl d : getFuncDecls())
    if (d.getID().equals(s)) return d;
  return lookup(s);
}
```

4 Runtime systems

a) (8p)

Stack:



b) (2p)

The frame pointer points to the start of the `baz` frame. Using the frame pointer, the static link for `baz` can be accessed (2 words down) and followed to the start of the `foo` frame. From this point, the static link for `foo` can be accessed (2 words down), and followed to the start of the `main` frame. From there, the local variable x is found 1 word up.