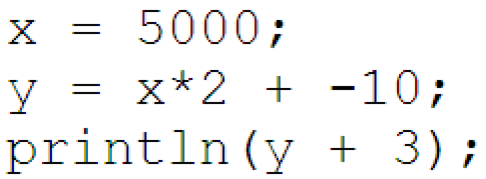
Chapter 12

S1--A Simple Compiler

Source language



```
x = 5000;
y = x*2 + -10;
println(y + 3);
```

Source language

```
x = +y;            ← unary plus not legal
x = x + -y;        ← unary minus not legal
```

However, constants can be signed. For example, the following statements are legal:

```
x = +5;
x = x + -20;
```

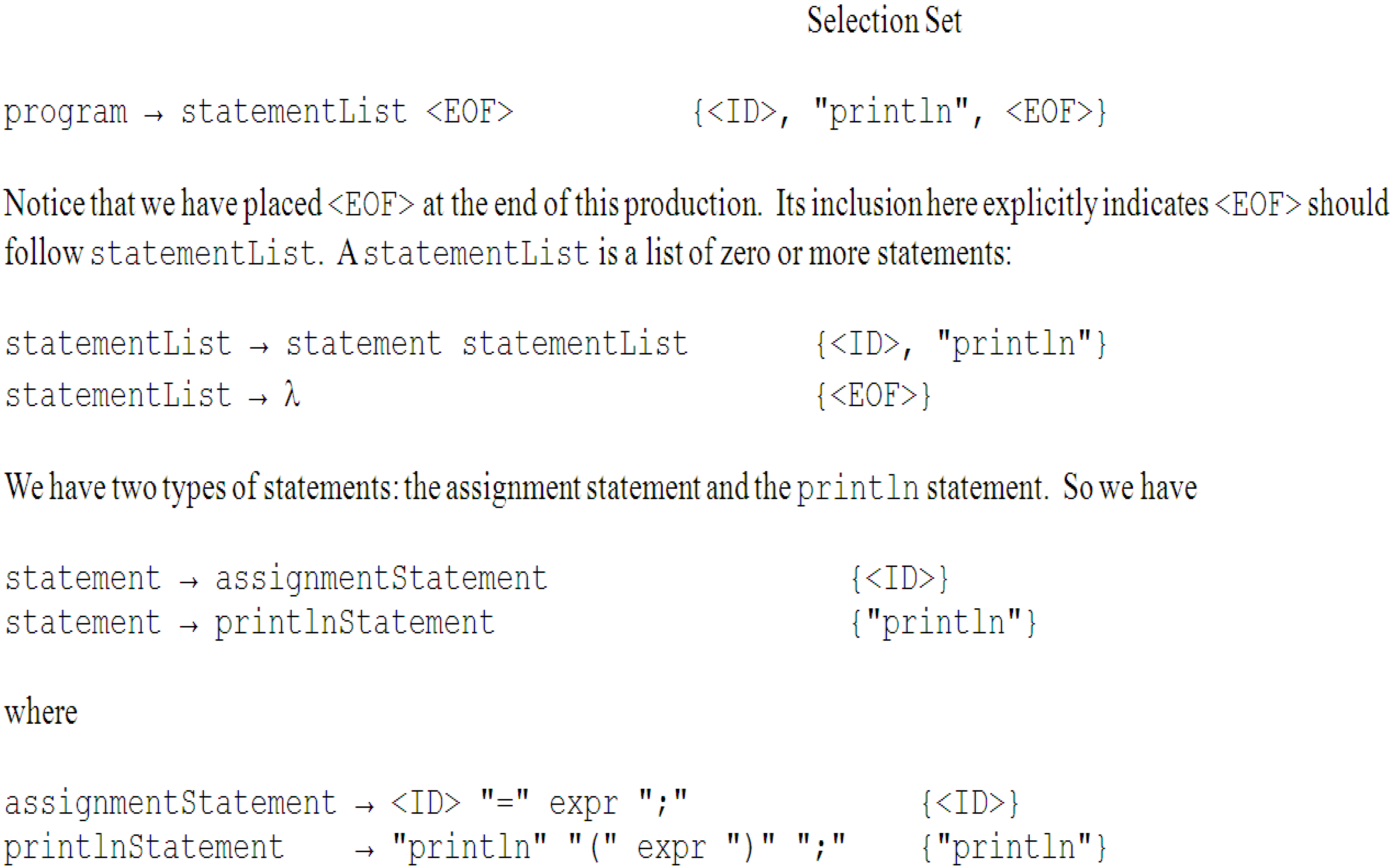
The `println` statement must have exactly one argument. Thus, the following statements are all legal:

```
println(5);
println(5 + 20);
println(y);
println(x + y + -3);
```

but these statements are illegal:

```
println();         ← null argument list not legal
println(x, y);     ← more than one argument not legal
```

Grammar



```
                                                  Selection Set

program → statementList <EOF>              {<ID>, "println", <EOF>}
```

Notice that we have placed <EOF> at the end of this production. Its inclusion here explicitly indicates <EOF> should follow statementList. A statementList is a list of zero or more statements:

```
statementList → statement statementList      {<ID>, "println"}
statementList → λ                            {<EOF>}
```

We have two types of statements: the assignment statement and the println statement. So we have

```
statement → assignmentStatement                   {<ID>}
statement → printlnStatement                      {"println"}
```

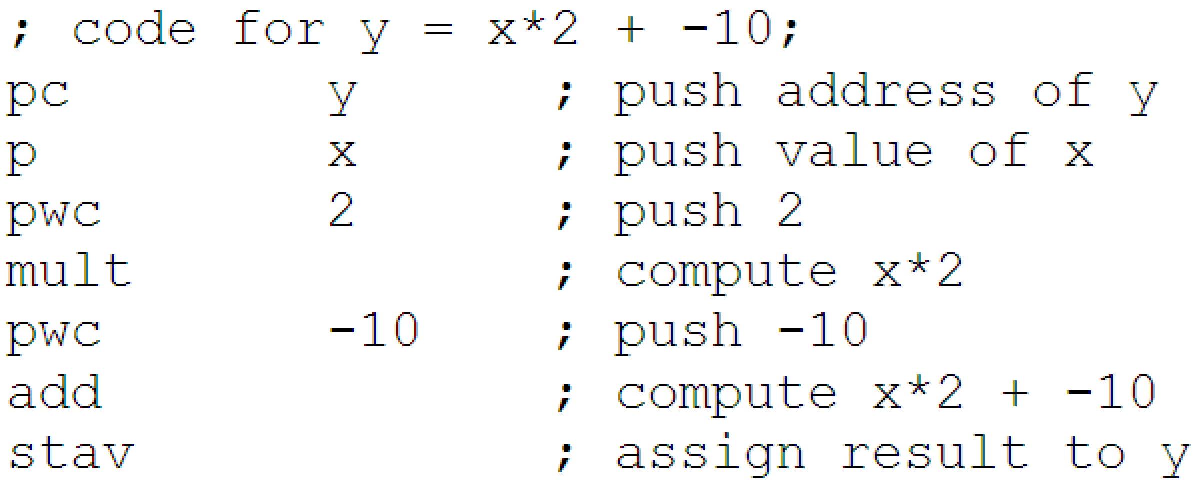
where

```
assignmentStatement → <ID> "=" expr ";"           {<ID>}
printlnStatement    → "println" "(" expr ")" ";"  {"println"}
```

Grammar

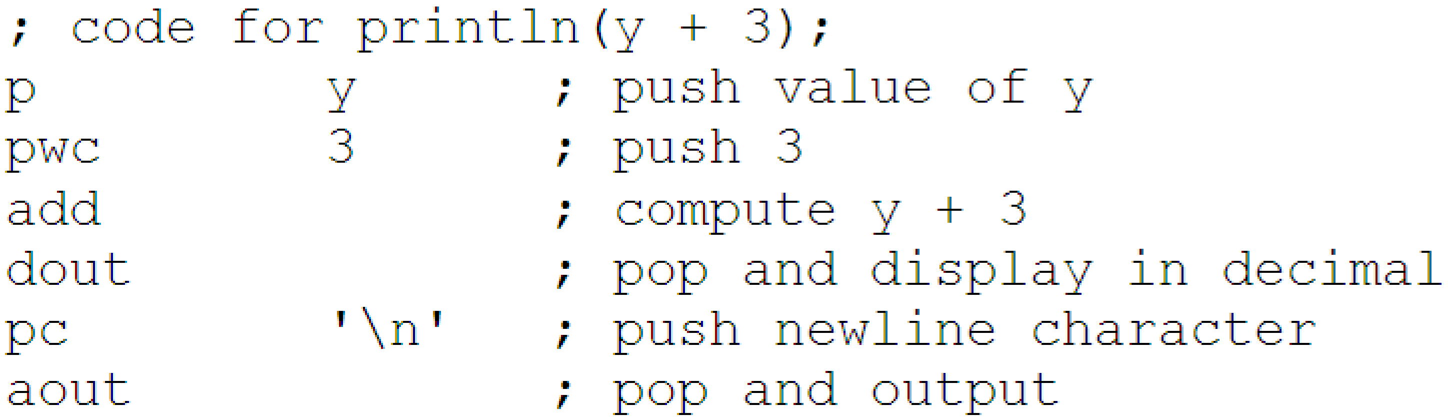
```
expr       → term termList            {"(", "-", <UNSIGNED>, <ID>}
termList   → "+" term termList        {"+"}
termList   → λ                        {")", ";"}
term       → factor factorList        {"(", "-", <UNSIGNED>, <ID>}
factorList → "*" factor factorList    {"*"}
factorList → λ                        {")", ";", "+"}
factor     → <UNSIGNED>               {<UNSIGNED>}
factor     → "+" <UNSIGNED>           {"+"}
factor     → "-" <UNSIGNED>           {"-"}
factor     → <ID>                     {<ID>}
factor     → "(" expr ")"             {"("}
```

Target language



```
; code for y = x*2 + -10;
pc        y      ; push address of y
p         x      ; push value of x
pwc       2      ; push 2
mult             ; compute x*2
pwc       -10    ; push -10
add              ; compute x*2 + -10
stav             ; assign result to y
```

Target language



```
; code for println(y + 3);
p         y      ; push value of y
pwc       3      ; push 3
add              ; compute y + 3
dout             ; pop and display in decimal
pc        '\n'   ; push newline character
aout             ; pop and output
```

Code generator

```
cg.emit("mult");
cg.emit("pc", t.image);
```

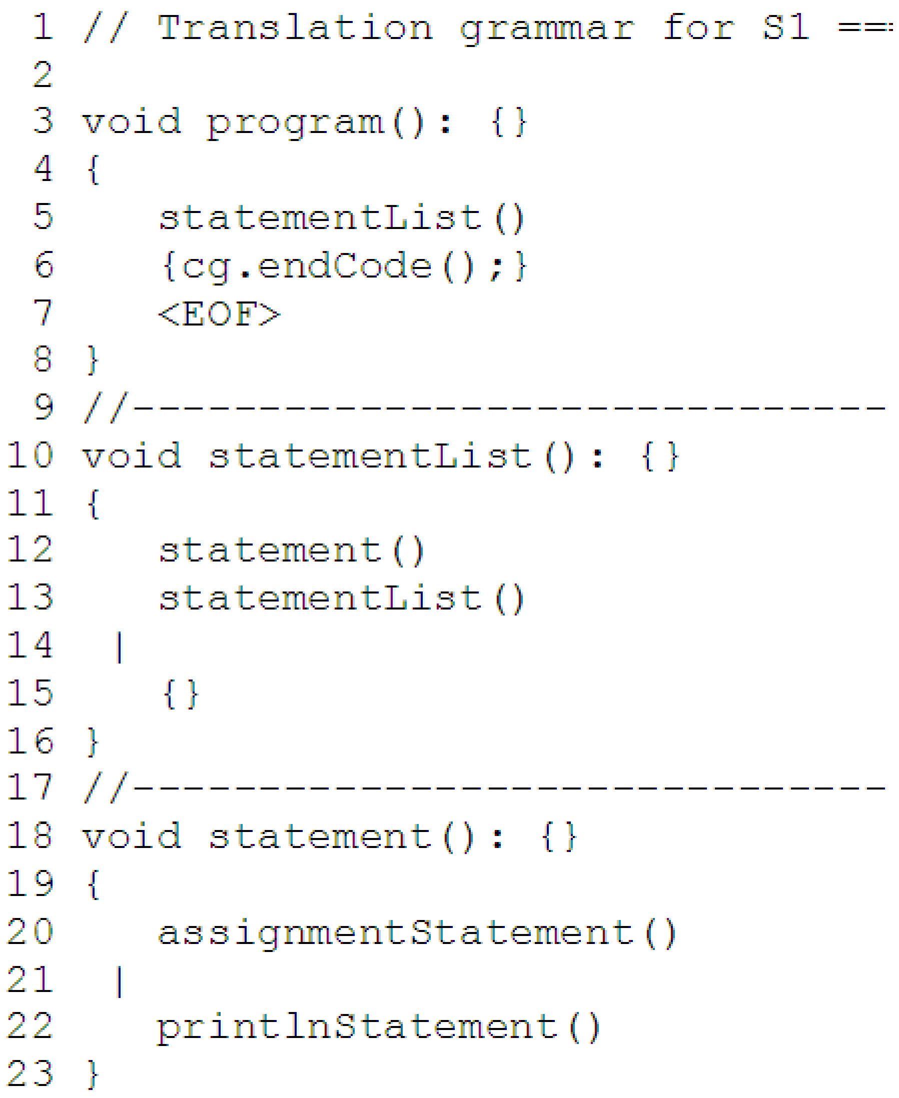
output

```
mult
pc    x
```

Token class

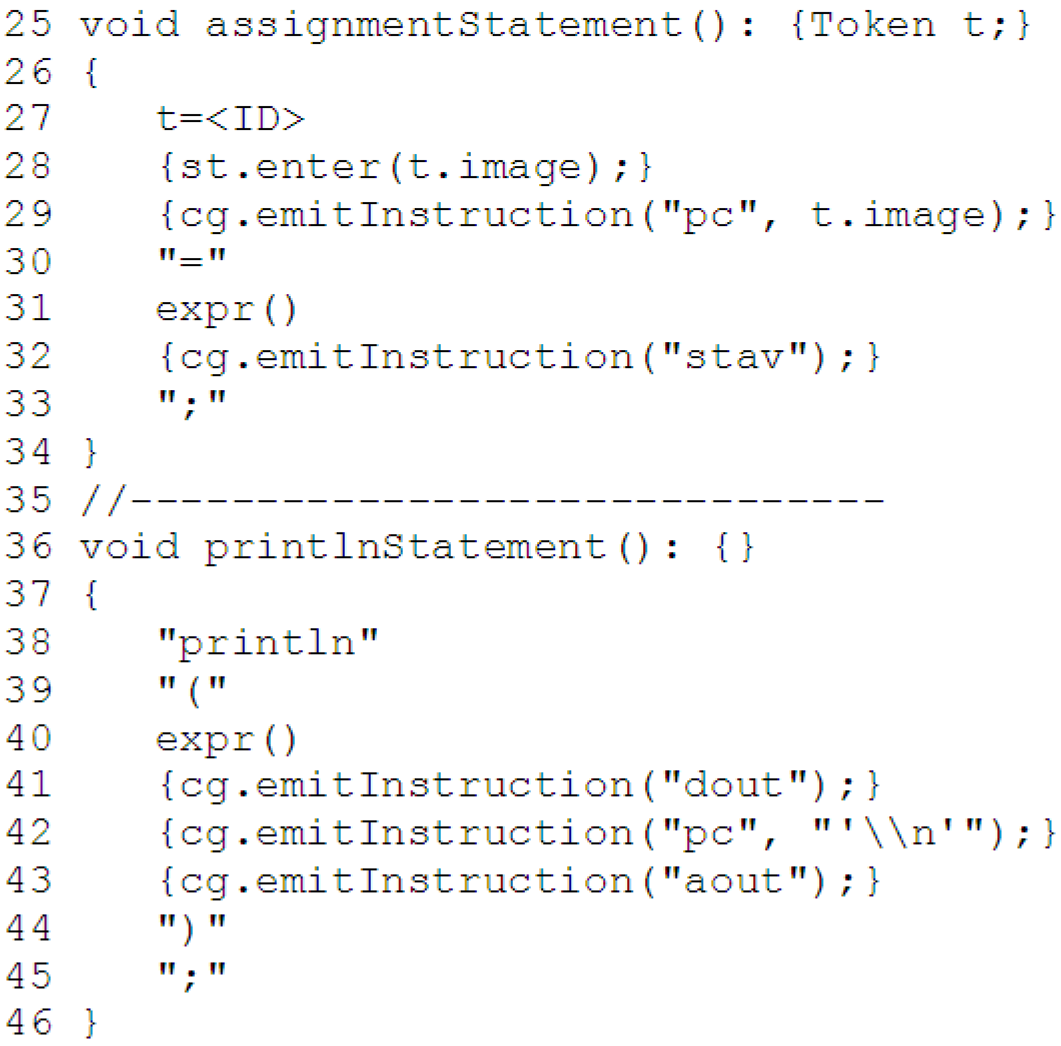
```
class Token
{
  // integer that identifies kind (i.e., category) of token
  public int kind;

  // location of token in source program
  public int beginLine, beginColumn, endLine, endColumn;

  // String consisting of characters that make up token
  public String image;

  // link to next Token object
  public Token next;
}
```

Translation grammar



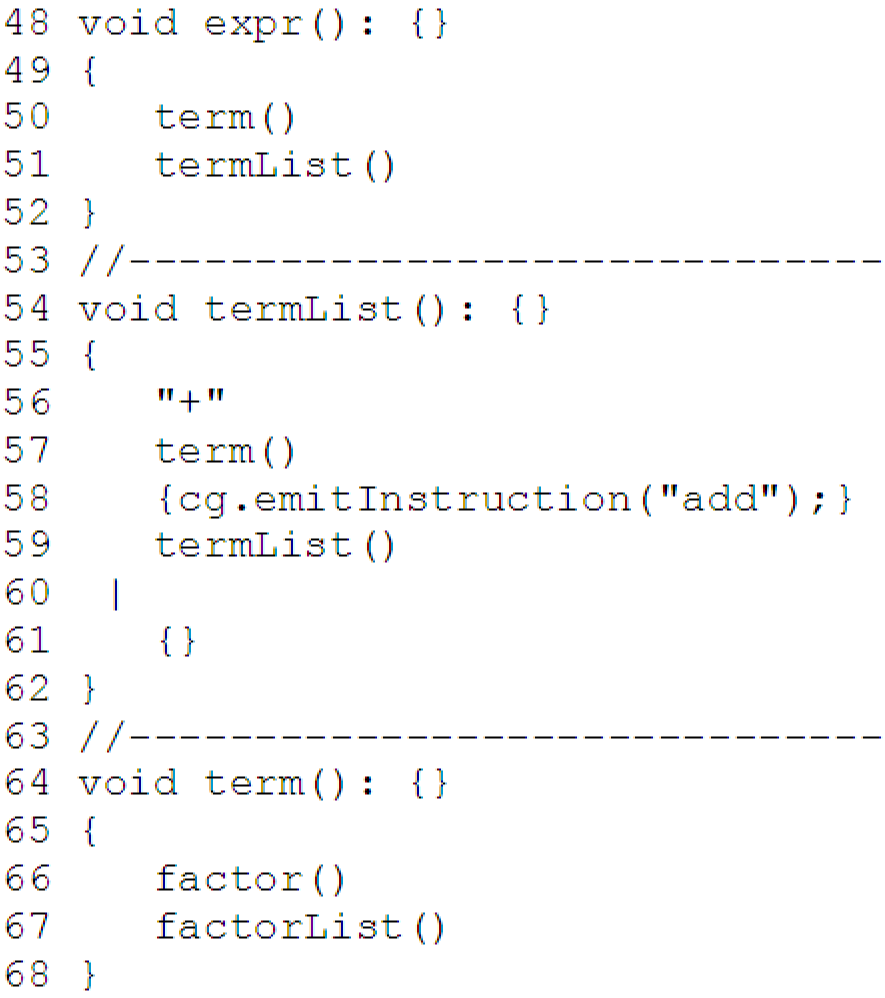
```
// Translation grammar for S1 ==

void program(): {}
{
   statementList()
   {cg.endCode();}
   <EOF>
}
//--------------------------------
void statementList(): {}
{
   statement()
   statementList()
 |
   {}
}
//--------------------------------
void statement(): {}
{
   assignmentStatement()
 |
   printlnStatement()
}
```

Translation grammar



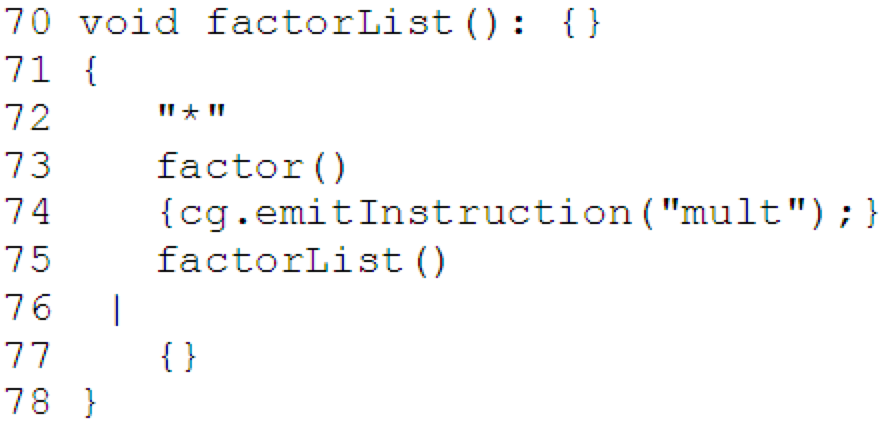
```
void assignmentStatement(): {Token t;}
{
   t=<ID>
   {st.enter(t.image);}
   {cg.emitInstruction("pc", t.image);}
   "="
   expr()
   {cg.emitInstruction("stav");}
   ";"
}
//--------------------------------
void printlnStatement(): {}
{
   "println"
   "("
   expr()
   {cg.emitInstruction("dout");}
   {cg.emitInstruction("pc", "'\\n'");}
   {cg.emitInstruction("aout");}
   ")"
   ";"
}
```

Translation grammar



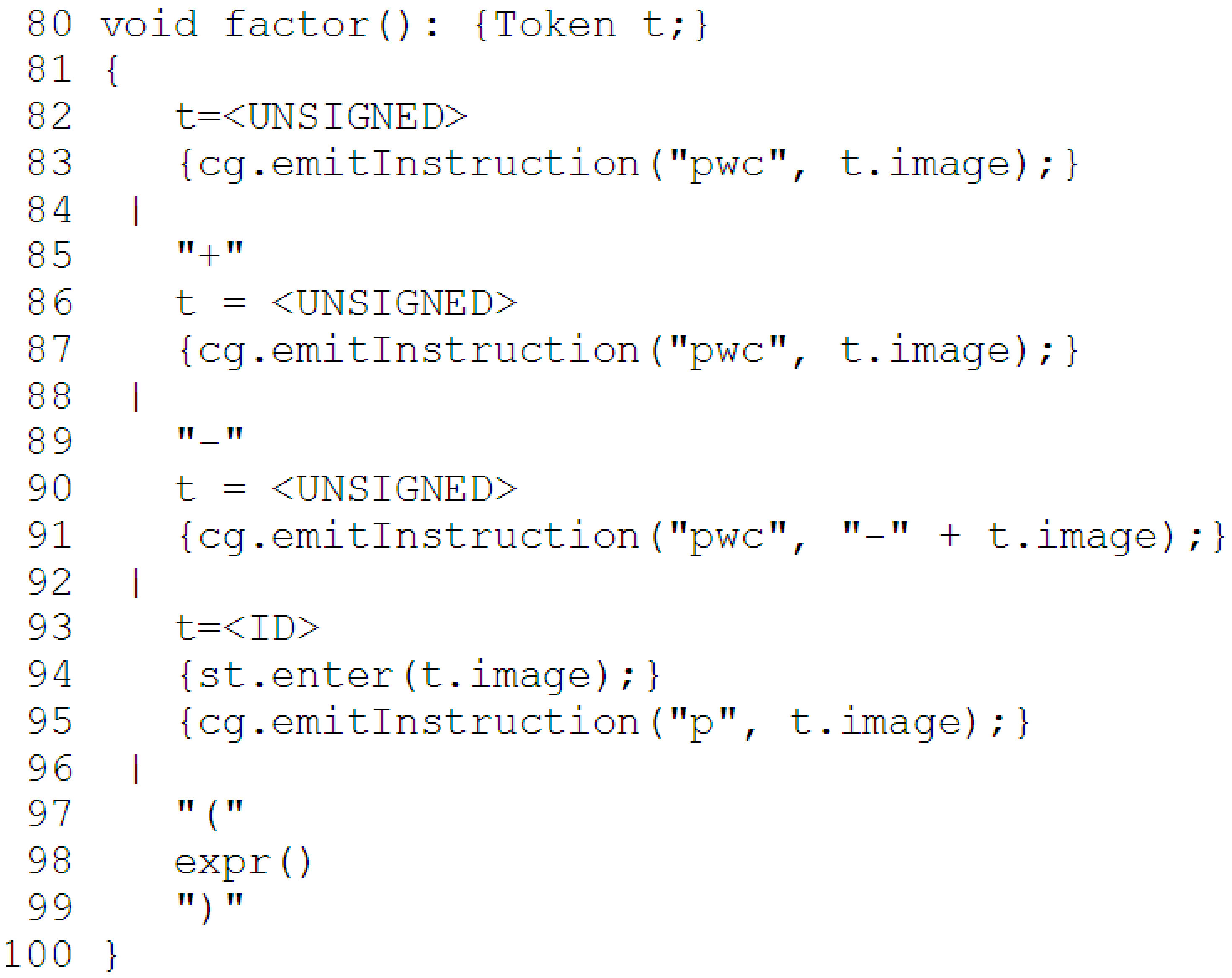
```
void expr(): {}
{
   term()
   termList()
}
//------------------------------
void termList(): {}
{
   "+"
   term()
   {cg.emitInstruction("add");}
   termList()
 |
   {}
}
//------------------------------
void term(): {}
{
   factor()
   factorList()
}
```

Translation grammar



```
void factorList(): {}
{
   "*"
   factor()
   {cg.emitInstruction("mult");}
   factorList()
 |
   {}
}
```

Translation grammar



```
void factor(): {Token t;}
{
   t=<UNSIGNED>
   {cg.emitInstruction("pwc", t.image);}
 |
   "+"
   t = <UNSIGNED>
   {cg.emitInstruction("pwc", t.image);}
 |
   "-"
   t = <UNSIGNED>
   {cg.emitInstruction("pwc", "-" + t.image);}
 |
   t=<ID>
   {st.enter(t.image);}
   {cg.emitInstruction("p", t.image);}
 |
   "("
   expr()
   ")"
}
```

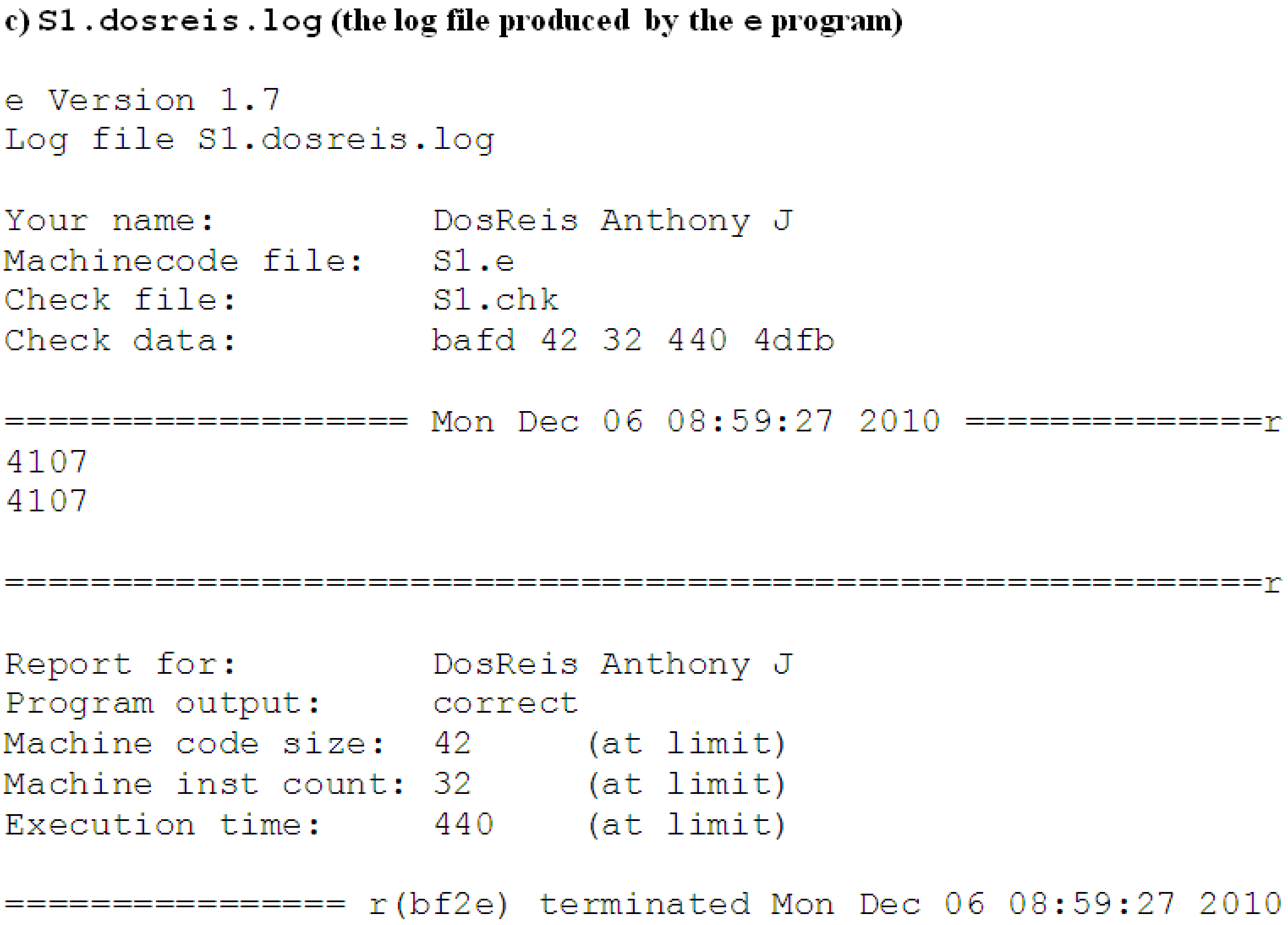
S1 compiler

S1.txt

Trying out S1

```
javac S1.java
java S1 S1
a S1.a
e S1 /c
```

Log file



c) S1.dosreis.log (the log file produced by the e program)

```
e Version 1.7
Log file S1.dosreis.log

Your name:          DosReis Anthony J
Machinecode file:   S1.e
Check file:         S1.chk
Check data:         bafd 42 32 440 4dfb

=================== Mon Dec 06 08:59:27 2010 ==============r
4107
4107

===========================================================r

Report for:         DosReis Anthony J
Program output:     correct
Machine code size:  42     (at limit)
Machine inst count: 32     (at limit)
Execution time:     440    (at limit)

================ r(bf2e) terminated Mon Dec 06 08:59:27 2010
```

Extending S1

- Set kind field of keyword tokens.
- Read 1 character beyond end of every token.
- Debug your token manager first.
- Use the correct selection sets.
- Do forget required `break` statements.
- Call `consume` method as required.
- Interpret translation grammar correctly:

  t = <UNSIGNED>   translation grammar

  t = currentToken;   Java code