Compilers

Spring 2022

Intermediate Representation and Symbol Tables

Sample Exercises and Solutions

Prof. Pedro C. Diniz

Faculdade de Engenharia da Universidade do Porto
Informatics Engineering Department
pedrodiniz@fe.up.pt

Problem 1: Abstract Syntax Tree Representation

Consider the sequence of input statements described below (left) for scalar integer a, b and c, as well as for and double valued variable f. Assume the CFG grammar depicted below (right).

```
c = a + b;
f = (1.0 * c) * (a+b);
```

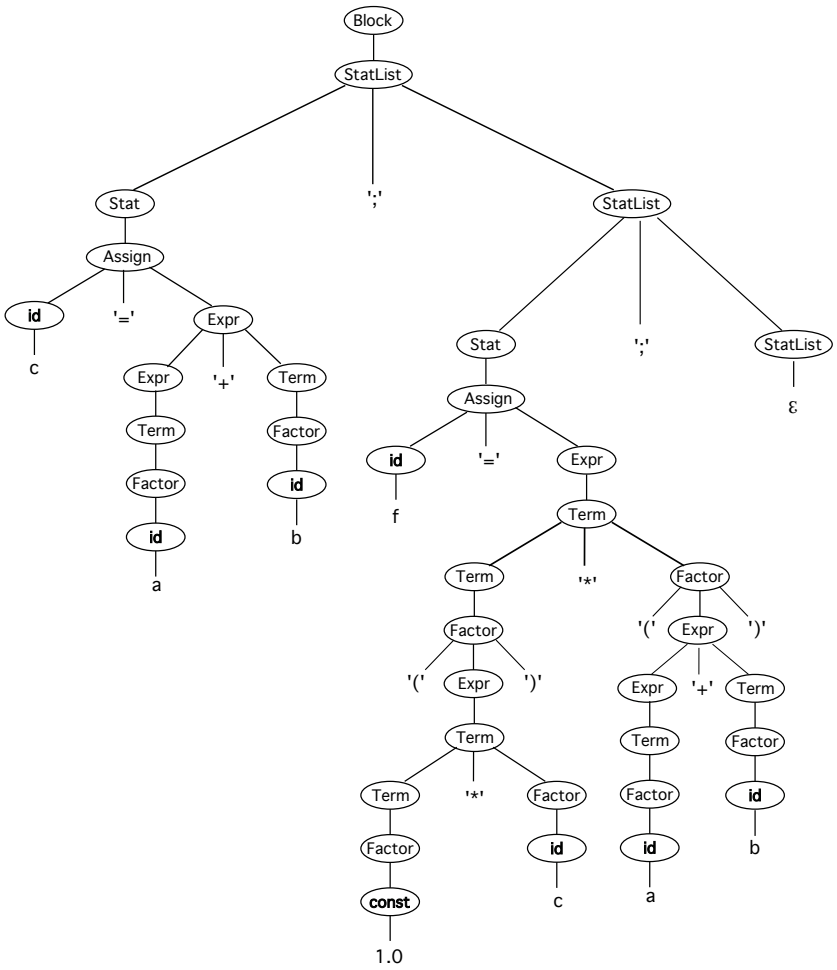
```
Block    → StatList
StatList → Stat ';' StatList
         → ε
Stat     → Assign
         → ...
Assign   → id '=' Expr
Expr     → Expr '+' Term
Expr     → Expr '-' Term
Expr     → Term
Term     → Term '*' Factor
Term     → Term '/' Factor
Term     → Factor
Factor   → '(' Expr ')'
Factor   → id
Factor   → const
```

For this sequence of statements and CFG determine the following:

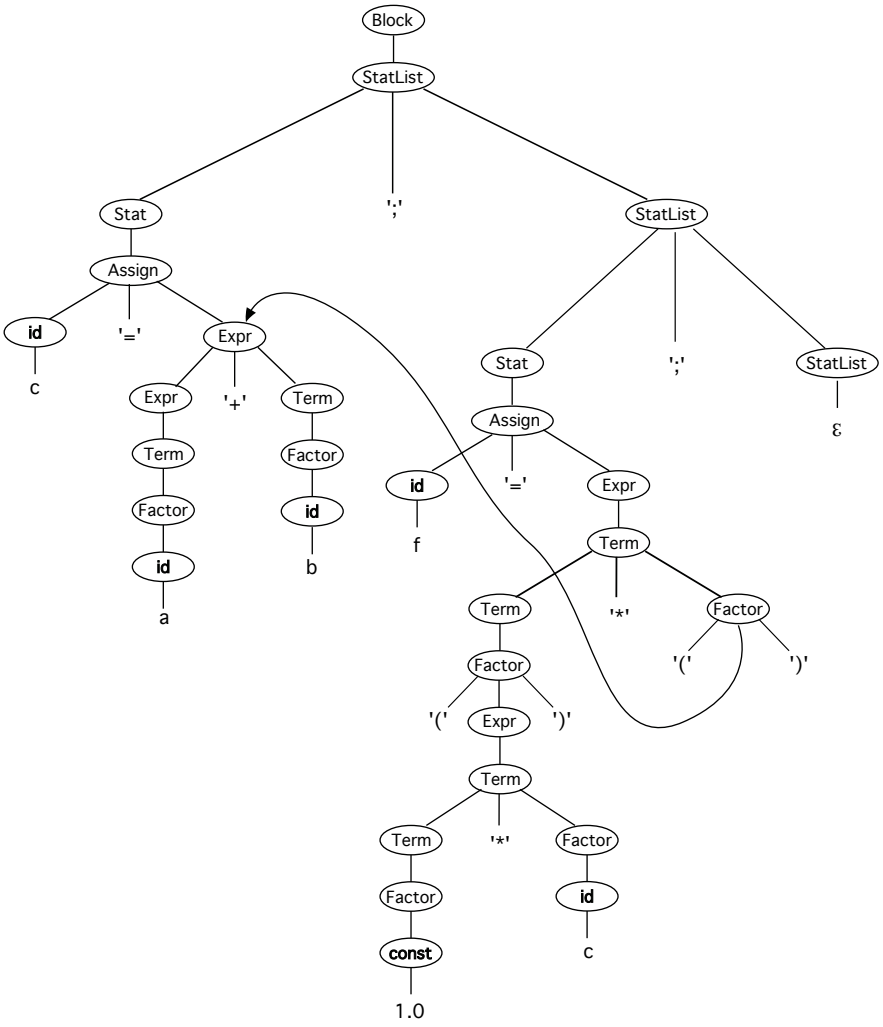
a) Draw a abstract syntax tree for the statements above.
b) Convert the AST representation into a DAG by observing the common expression (a+b) in statements.
c) When generating code there is a need to traverse a tree and emit code for the various expressions in the AST. For the case of the DAG the implementation must save in the tree the information that a give node has had its code emitted and saved in a temporary variable. Outline a code generation scheme that can reuse the computation in these common subexpressions in the code. Make sure you address the connectness of the transformation in case one of the variables involved in the comon expression is modified.

Solution:

a) The figure on the right depicts the parse tree for the two statements provided.



b) The only difference here is that the sub-expression "`a+b`" in the `Factor` sub-tree of the second assignment statement is "linked" to the first occurrence of that same sub-expression.


Block
StatList
Stat
';'
StatList
Assign
id
'='
Expr
Stat
';'
StatList
c
Expr
'+'
Term
Assign
ε
Term
Factor
id
'='
Expr
Factor
id
id
Term
f
id
b
Term
'*'
Factor
a
Term
Factor
'('
')'
'('
Expr
')'
Term
Term
'*'
Factor
Factor
id
const
c
1.0

c) In terms of code generation the recognition of a common sub-expression will allow a later stage of the compile to reuse the effort in the evaluation of the common sub-expression. The key observation is that between the first and the second evaluation of the sub-expression neither of its operands has changed its value, i.e., there was no assignment to either "`a`" or "`b`". As such the reuse of the same computation (via a temporary register for example) is a correct transformation.

A way to implement this code generation "optimization" is to associate an attribute with the tree node corresponding to the common sub-expression indicating the intermediate variable where the value of the sub-expression has been saved. The snippet of 3-address code below illustrates this concept (as well as the code without using this transformation on the right) where is can be noted the "distance" in terms of instructions between the instructions that evaluate the expression "`a+b`" and the reuse of the value during the computation of the value to be assigned to the variable "`f`".

```
t1 = a + b
c  = t1
t2 = 1.0
t3 = t2 * c
t4 = t3 * t1
f  = t4
```

```
t1 = a + b
c  = t1
t2 = 1.0
t3 = t2 * c
t4 = a + b
t5 = t3 * t4
f  = t5
```

As can be seen without using this transformation the intermediate code requires an addition temporary variable (in some setting leading to a higher number of required registers) and also an addition addition operation.

Problem 2: Symbol Table Organization

For the PASCAL code below answer the following questions:

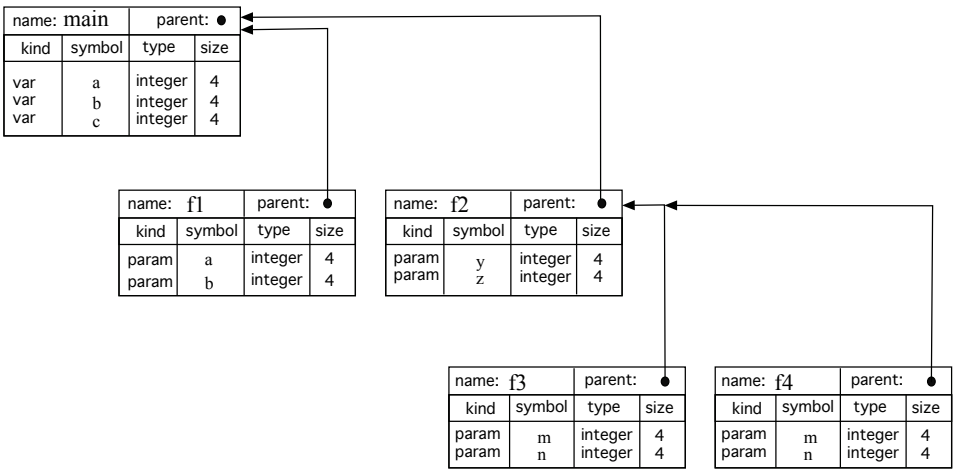
```
01:   procedure main
02:     integer a, b, c;
03:     procedure f1(a,b);
04:       integer a, b;
05:       call f2(b,a);
06:       end;
07:     procedure f2(y,z);
08:      integer y, z;
09:      procedure f3(m,n);
10:       integer m, n;
11:       end;
12:      procedure f4(m,n);
13:       integer m, n;
14:       end;
15:      call f3(c,z);
16:      call f4(c,z);
17:      end;
18:   ...
19:   call f1(a,b);
20:   end;
```

a) Draw the symbol tables for each of the procedures in this code (including `main`) and show their nesting relationship by linking them via a pointer reference in the structure (or record) used to implement them in memory. Include the entries or fields for the local variables, arguments and any other information you find relevant for the purposes of code generation, such as its type and location at run-time.

b) For the statement in line 15 what are the specific instance of the variables used in this statement the compiler needs to locate? Explain how the compiler obtains the data corresponding to each of these variables table.

Solution:

a) The figure below depicts the internal data and relative organization of the symbol tables related to the various functions and `main` procedure in this program.



b) For the statement in line 15 the symbol "`c`" refers to the scalar variable in the `main` procedure, whereas the symbol "`z`" refers to the scalar variable in the `f2` procedure. The compiler uncovers which procedure variable or parameter a given symbol corresponds to by traversing the tree of symbol tables up to the "root" in this case the symbol table of the `main` procedure.