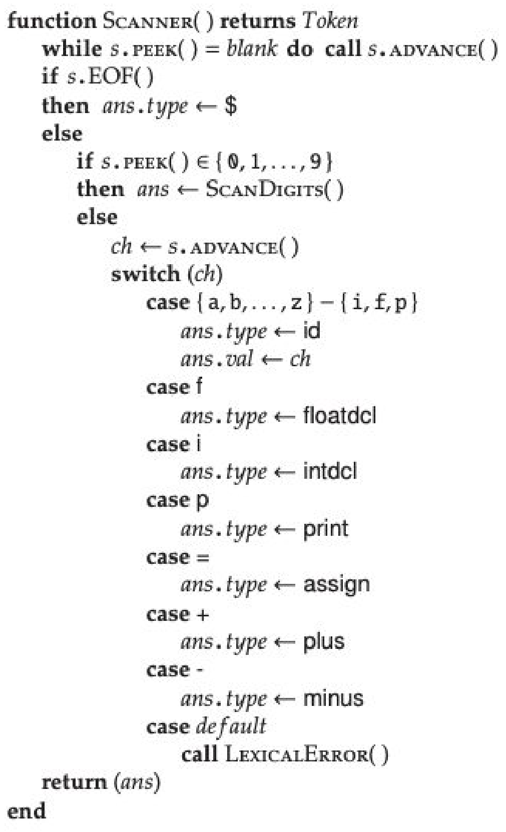
Individual Exercises - Lecture 3

1. Do Fisher et al. exercise 3,4,6 page 54-55 (exercise 3,8,9 page 82-84 in GE)
   3. Extend the ac scanner (Figure 2.5) in the following ways:



```
function SCANNER( ) returns Token
    while s.PEEK( ) = blank do call s.ADVANCE( )
    if s.EOF( )
    then ans.type ← $
    else
        if s.PEEK( ) ∈ { 0, 1, . . . , 9 }
        then ans ← SCANDIGITS( )
        else
            ch ← s.ADVANCE( )
            switch (ch)
                case { a, b, . . . , z } − { i, f, p }
                    ans.type ← id
                    ans.val ← ch
                case f
                    ans.type ← floatdcl
                case i
                    ans.type ← intdcl
                case p
                    ans.type ← print
                case =
                    ans.type ← assign
                case +
                    ans.type ← plus
                case -
                    ans.type ← minus
                case default
                    call LEXICALERROR( )
    return (ans)
end
```

Figure 2.5: Scanner for the ac language. The variable s is an input stream of characters.

(a) floatdcl can be represented as either f or float, allowing a more Java-like syntax for declarations.
Change case f -> case f, (“f” “l” “o” “a” “t”) without creating two case(“f”)
(alternative solution: case f: if(s.peek == ‘l’) consume(“loat”))
(Consume calls LexicalError() if s.advance() does not correspond to ‘l’ ‘o’ ‘a’ ‘t’

(b) An intdcl can be represented as either i or int
Change case i -> case i, (“i” “n” “t”)

(c) A num may be entered in exponential (scientific) form. That is, an ac num may be suffixed with an optionally signed exponent (1.0e10, 123e-22 or 0.31415926535e1).

```
function ScanDigits() returns token
   tok.val = " "
   while s.peek() ∈ {0, 1 .... 9} do
      tok.val = tok.val + s.advance()
   if s.peek() == "." or s.peek() == "e"
      tok.type = fnum
      if s.peek() == "."
          tok.val = tok.val + s.advance()
          while s.peek() ∈ {0, 1 .... 9} do
              tok.val = tok.val + s.advance()
      else if s.peek() == "e"
         tok.val = tok.val + s.advance()
         if s.peek() == "-"
            tok.val = tok.val + s.advance()
         while s.peek() ∈ {0, 1 .... 9} do
            tok.val = tok.val + s.advance()
   else
      tok.type = inum
   return (tok)
end
```

4. *Write a recursive descent parser for each of the non-terminals in figure 2.1*

1	Prog	$\rightarrow$	Dcls Stmts \$
2	Dcls	$\rightarrow$	Dcl Dcls
3		|	λ
4	Dcl	$\rightarrow$	floatdcl id
5		|	intdcl id
6	Stmts	$\rightarrow$	Stmt Stmts
7		|	λ
8	Stmt	$\rightarrow$	id assign Val Expr
9		|	print id
10	Expr	$\rightarrow$	plus Val Expr
11		|	minus Val Expr
12		|	λ
13	Val	$\rightarrow$	id
14		|	inum
15		|	fnum

Figure 2.1: Context-free grammar for ac.

```
Procedure Prog()
      call Dcls()
      call Stmts()
      call MATCH(ts, $)
end
```

```
Procedure Dcls()
      if ts.peek() = floatdcl or ts.peek() = intdcl
      then
            call Dcl()
            call Dcls()
      else
            if ts.peek() = $ OR ts.peek() = id OR ts.peek() = print
            then
                  /* Do nothing*/
            else
                  call ERROR()
end
```

```
Procedure Dcl()
      if ts.peek() = floatdcl
      then
            call MATCH(ts, floatdcl)
            call MATCH(ts, id)
      else
            if ts.peek() = intdcl
            then
                  call MATCH(ts, intdcl)
                  call MATCH(ts, id)
            else
                  call ERROR()
end
```

```
Procedure Stmts()
      if ts.peek() = id or ts.peek() = print
      then
            call Stmt()
            call Stmts()
      else
            if ts.peek() = $
            then
```

```
                  /* Do nothing*/
            else
                  call ERROR()
end
```

```
Procedure Stmt()
      if ts.peek() = id
      then
            call MATCH(ts, id)
            Call MATCH(ts, assign)
            Call Val()
            Call Expr()
      else
            if ts.peek() = print
            then
                  call MATCH(ts, print)
                  call MATCH(ts, id)
            else
                  call ERROR()
end
```

```
Procedure Expr()
      if ts.peek() = plus
      then
            call MATCH(ts, plus)
            call Val()
            call Expr()
      else
            if ts.peek() = minus
            then
                  call MATCH(ts, minus)
                  call Val()
                  call Expr()
end
```

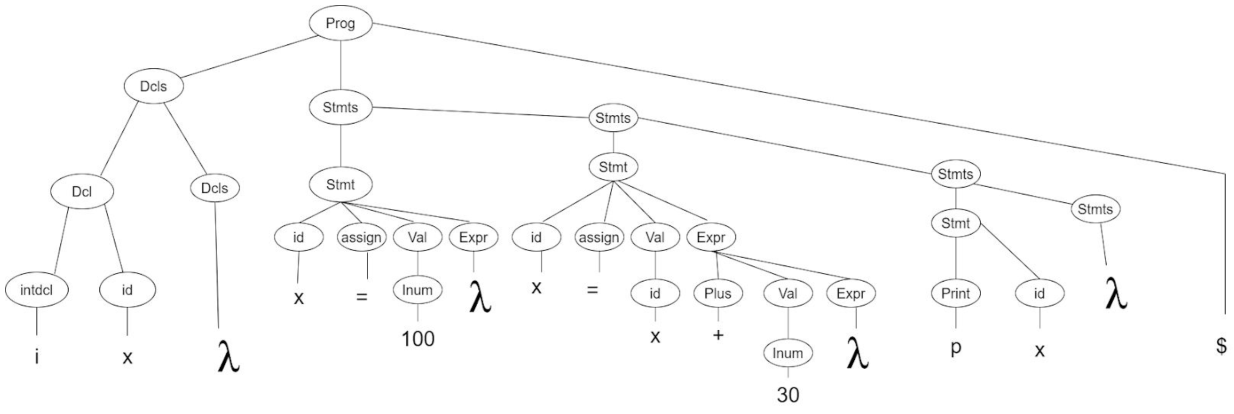
```
Procedure Val()
      if ts.peek() = id
      then
            call MATCH(ts, id)
      else
            if ts.peek() = inum
            then
                  call MATCH(ts, inum)
            else
                  if ts.peek() = fnum
                  then
                        call MATCH(ts, fnum)
                  else
                        call ERROR()
end
```

6. Variables are considered uninitialized after they are declared in some programming languages. In ac a variable must be given a value in an assignment statement before it can be correctly used in an expression or print statement. Suggest how to extend ac's semantic analysis (Section 2.7) to detect variables that are used before they are properly initialized.

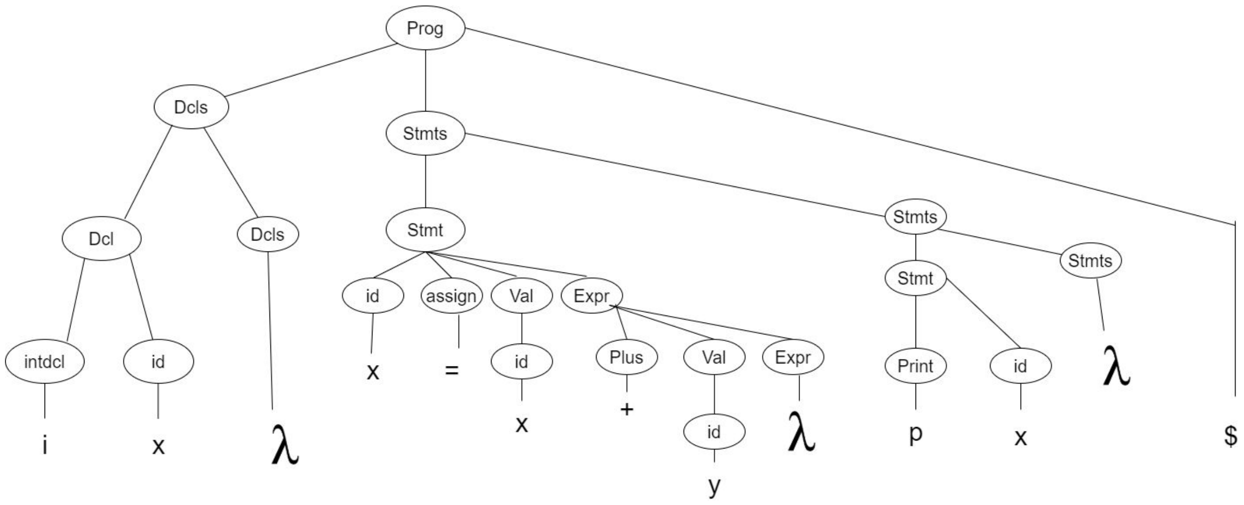
Extend the symbol table, such that it contains information whether a variable has been initialized or not. In this way it is possible to distinguish between declared and initialized.

2. *Based on the grammar in Figure 2.1 construct the parse tree for each of the following:*

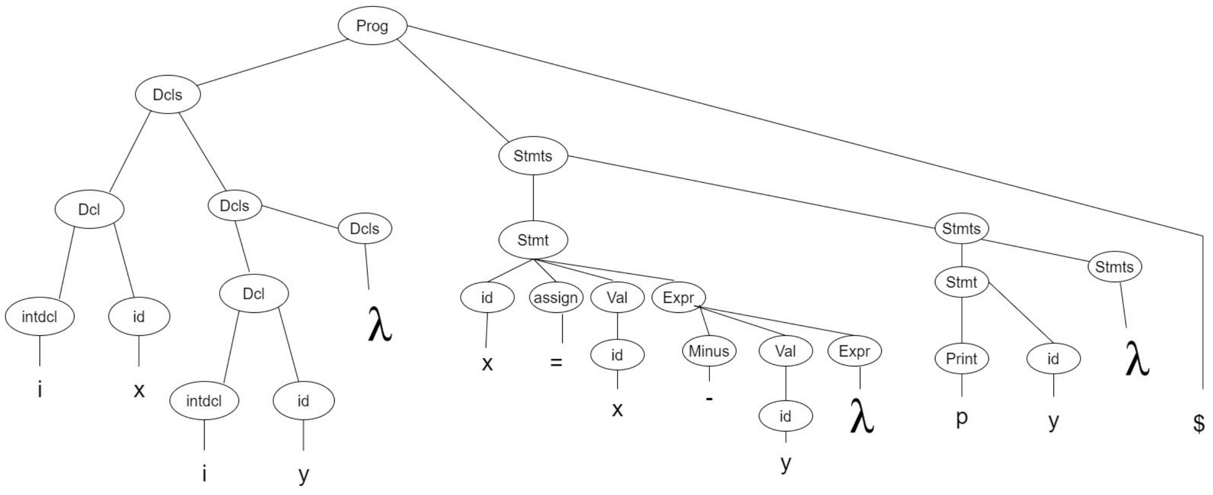
a) i x x = 100 x = x + 30 p x



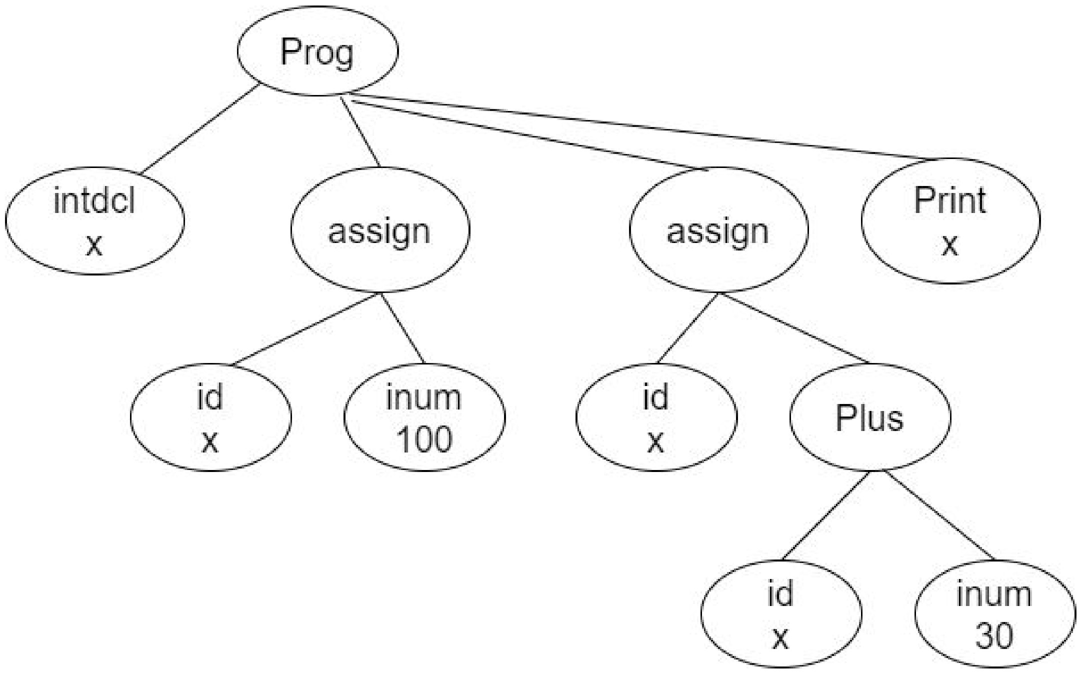
b) i x x = x + y p x



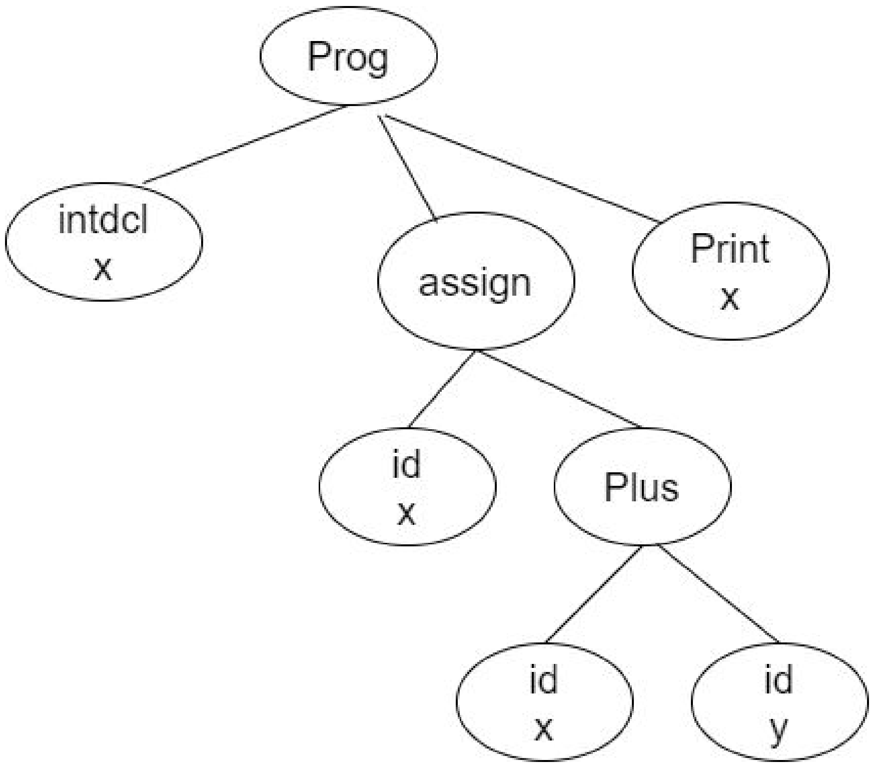
c) i x i y x = x - y p y



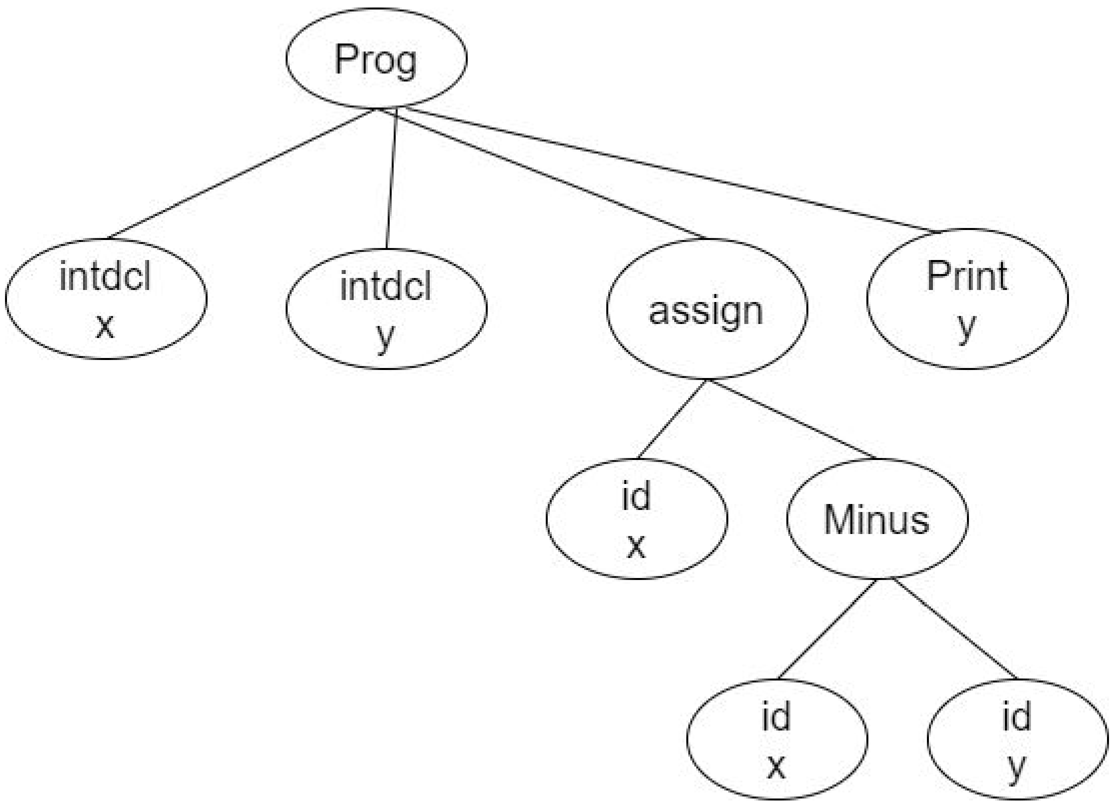
3. *For each of the programs in exercise 2:*
   a) Construct the AST
   i x x = 100 x = x + 30 p x



   i x x = x + y p x



i x i y x = x - y p y



b) According to the definition of ac, is the input semantically correct? If not, what changes to the ac language would render the program semantically correct

For “i x x = x + y p x” y is not declared before it is used. The ac language requires an identifier to be declared prior to use.

For “i x i y x = x - y p y” y and x is not initialized before use. The ac language requires an identifier to be initialized prior use.

c) With ac changes in place that make the program semantically correct, show the code that would be generated from these programs

i x x = 100 x = x + 30 p x

Code	*Source*	*Comments*
100 *sx* *0 k*	*x = 100*	*Push 100 on stack* *Pop stack, storing the popped value in reg x* *Reset precision to integer*
lx *30* *+*	*x = x + 30*	*Push value of reg x on stack* *Push 30 on stack* *Pop 2 elements from stack and push the sum*

sx		*Store top stack value in x*
lx *p* *si*	*p x*	*Push value of the x register* *Print the top-of-stack value* *Pop the stack by storing into the i register*

i x i y x = 30 y = 30 x = x + y p x

Code	Source	Comments
100 Sy 0k	y = 30	Push 30 on stack *Pop stack, storing the popped value in reg y* *Reset precision to integer*
30 sx 0 k	x = 30	Push 30 to stack Pop and store top of stack in reg x Set precision to Integer
Lx Ly + sx 0k	x = x +y	Push value of reg x on stack Push value of reg y on stack Pop 2 elements from stack and push the sum Store top stack value in x Reset precision to integer
Lx P si	p x	Push value of reg x Print the top-of-stack value Pop the stack by storing into the i register

i x i y x = 30 y = 30 x = x - y p y

Code	Source	Comments
30 sx 0 k	x = 30	Push 30 to stack Pop and store top of stack in reg x Set precision to Integer

30 sy 0 k	y = 30	Push 30 to stack Pop and store top of stack to reg y Set precision to integer
lx ly - sx	x = x -y	Push value of reg x to stack Push value of reg y to stack Pop two elements from stack and push difference Pop stack and store in reg x
ly p	p y	Push value of reg y to the stack Pop and print

4. *(optional but recommended) Follow the studio associated with Crafting a Compiler Chapter 2: A Simple Compiler http://www.cs.wustl.edu/~cytron/cacweb/Chapters/2/studio.shtml You can find the source zip file in the General_Course_Material directory https://www.moodle.aau.dk/pluginfile.php/154451/mod_folder/content/0/Studio_Code.zip?forcedownload=1*
   For this a virtual machine will be provided with the necessary tools pre installed.

Group Exercises - Lecture 3

The following exercises are best done as group discussions:

```
 1  Prog  → Dcls  Stmts  $
 2  Dcls  → Dcl  Dcls
 3        | λ
 4  Dcl   → floatdcl  id
 5        | intdcl  id
 6  Stmts → Stmt  Stmts
 7        | λ
 8  Stmt  → id  assign  Val  Expr
 9        | print  id
10  Expr  → plus  Val  Expr
11        | minus  Val  Expr
12        | λ
13  Val   → id
14        | inum
15        | fnum
```

Figure 2.1: Context-free grammar for ac.

1. Do Fisher et al. Exercise 1,2,8,9,10,12 page 54-55 (exercise 4,5,6,10,12,13 on page 82-84 in GE)

   1. The CFG shown in Figure 2.1 defines the syntax of ac programs. Explain how this grammar enables you to answer the following questions.
   (a) Can an ac program contain only declarations (and no statements)?
   Yes, Using Rule 7, the nonterminal Stmts can derive to the null string. (Fisher,page 33, figure 2.1)
   (b) Can a print statement precede all assignment statements?
   Yes, The rules for Stmts impose no constraints on the ordering of a sequence of instances of the Stmt.

   2. Sometimes it is necessary to modify the syntax of a programming lan- guage. This is done by changing the CFG that the language uses. What changes would have to be made to ac's CFG (Figure 2.1) to implement the following changes?
   (a) All ac programs must contain at least one statement.

   Prog -> Dcls Stmt Stmts $

   (b) All integer declarations must precede all float declarations

   Prog -> IntDcls FloatDcls Stmts $

   FloatDcls-> floatdcl FloatDcls

| λ

IntDcls -> intdcl IntDcls

| λ

(c) The first statement in any ac program must be an assignment statement.

Prog -> Dcls Assignment Stmts $

Assignment -> id assign Val Expr

Stmt -> Assignment | print id

8. The grammar for ac shown in Figure 2.1 requires all declarations to precede all executable statements. In this exercise, the ac language is extended so that declarations and executable statements can be interspersed. However, an identifier cannot be mentioned in an executable statement until it has been declared.

(a) Modify the CFG in Figure 2.1 to accommodate this language extension.

Checking that a variable is declared before use cannot be enforced in the CFG. This is part of the semantic analysis and should be done using the symbol table. We can, however, change the CFG to allow for Dcls and Stmt to be interspersed.

```
Prog -> Lines $
Lines -> Dcl Lines
        | Stmt Lines
        | λ
Dcl -> floatdcl id
        | intdcl id
Stmt -> id assign Val Expr
        | print id
Expr -> plus Val Expr
        | minus Val Expr
        | λ
Val -> id
        | inum
        | fnum
```

(b) Discuss any revisions you would consider in the AST design for ac.

Link 'id' nodes to their corresponding declaration node

(c) Discuss how semantic analysis is affected by the changes you envision for the CFG and the AST.

Since Dcls and Stmts can be interspersed i now not only have to check if the variable is declared, but also when it was declared. This can for example be done using a symbol table with a column for when a given variable was declared or assigned.

9. The abstract tree design for ac uses a single node to represent a print operation (see Figure 2.9). Consider an alternative design in which the print operation always has a single id child that represents the variable to be printed. What are the design and implementation issues associated with the two approaches?
The existing design is based on the definition of a print command in ac, which only allows specification of a single name for which the corresponding value is to be printed. The implementation that follows is simple because all of the information regarding a print statement resides in a single **abstract syntax tree** (AST) node. The alternate approach suggested in this exercise would more easily accommodate a language change that allowed a list of names or constant values as part of a single print statement. The implementation would be more complicated because traversal of a list of items to print would be required, even if there was only a single item on that list.

10**.** *The code in Figure 2.10 examines an AST node to determine its effect on the symbol table. Explain why the order in which nodes are visited does or does not matter with regard to symbol-table construction.*
Since the result of a call to LookupSymbol does not depend on the order in which symbols are entered into the symbol table, the order in which declaration nodes are processed does affect the lookup process.
If a symbol is declared twice, the error message in EnterSymbol will be triggered regardless of the order in which the declarations are processed, although the error message may be attached to the first declaration of a symbol with more than one declaration if declarations are processed out of order.

12. The last fragment of code generated in Figure 2.15 pops the dc stack and stores the resulting value in register i.

(a) Why was register i chosen to receive the result?

Register **i** is used because **i** is a keyword, which can never be used for a variable *(b) Which other registers could have been chosen without causing any problems for code that might be generated subsequently?*

Register **f** or **p** could be chosen, as they are keywords

2. (Optional) Do the group exercise part of the studio associated with Crafting a Compiler Chapter 2: A Simple Compiler
   For this a virtual machine will be provided with the necessary tools pre installed.

3. (Optional) Extend the ac grammar to allow parentheses in expressions i.e. (4-3) + (2-1). How would this affect the ac compiler?

```
Prog → Dcls Stmts $

Dcls → Dcl Dcls
       | λ

Dcl → floatdcl id
       | intdcl id

Stmts → Stmt Stmts
       | λ

Stmt → id assign Val Expr
       | id assign '(' Val Expr ')' Expr
       | print id

Expr → plus Val Expr
       | plus '(' Val Expr ')' Expr
       | minus Val Expr
       | minus '(' Val Expr ')' Expr
       | λ

Val → id
       | inum
       | fnum
```

The compiler would apply precedence according to the parenthesis