Answers exam Compiler Construction—4 april 2014

1. *Consider the following code fragment:*

```
int x;

int g(int a, int b) {
  int c = a + b;
  /* location 2 */
  return c;
}

void f(int a, int b, int c) {
  int d = 2*a;
  int e = 3*b + c;
  /* location 1 */
  x = g(d, e);
}

int main() {
  f(15, 2, 6);
  /* location 3 */
  printf("%d\n", x);
  return 0;
}
```

The program is executed. Make a sketch of the memory layout (heap + stack of activation records) when execution of the program reaches the three marked locations. Assume that there are no optimizations at all, so parameters are not passed via registers. Moreover, on function entry/exit there is no need to save registers.

Answer: Global variables are stored on the heap, so the variable `x` is stored on the heap (so, not on the stack).
As far as the stack layout is concerned, this is the procedure for calling a function `new()` from a function `old()`[1]

(a) Push the arguments for `new` in reverse order.

(b) Reserve stack space for the return value of the function `new`.

(c) Push the return address (i.e. code pointer/program counter back to `old`).

(d) Push the frame pointer value of `old` (to save it).

(e) Use the old stack pointer as frame pointer of `new`.

(f) Reserve stack space for the local variables of `new`.

(g) Execute the code of `new`.

(h) Place the return value of `new` on the stack.

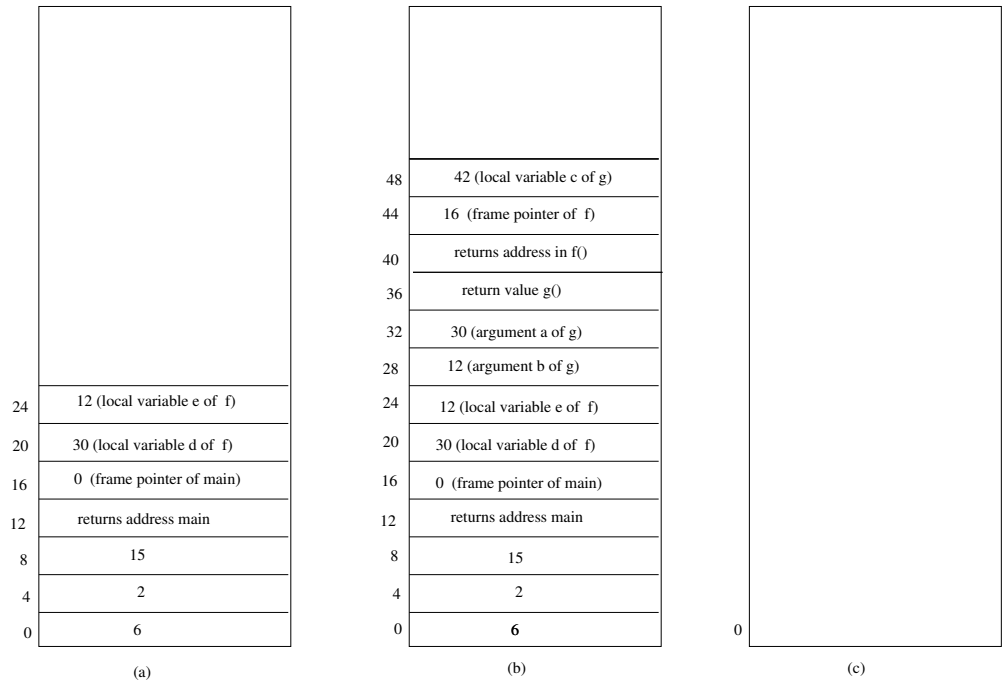
(i) Restore the `old` frame pointer value.

(j) Jump to the saved return address.

If we apply this procedure to the program, we find the following stack layouts:

[1] In the lecture slides, a more complicated procedure is presented due to saving of registers.

- location 1: We start with an empty stack (beginning of `main`). The arguments of the call `f(15,2,6)` are pushed in reverse order: `push 6; push 2; push 15`. Next, we push the return address in `main` (i.e. location 3). The next step is to push (save) the frame pointer of main (which is 0). We reserve two locations for the local variables of `f()`, and execute the first two assignments of the function `f()`. So, the stack layout at location 1 is given in figure (a).
- location 2: we start from the situation of location 1. We push the arguments for the function `g()` in reverse over: `push 12 (b), push 30 (a)`. Next, we reserve space (4 bytes) for the return value of `g()`, and push the return address in `f()`. The next step is to push (save) the frame pointer of `f` (which is 16). We reserve one location for the local variable `c` of `g()`, and execute the assignment in the function `g()`. So, the stack layout at location 2 is given in figure (b).
- location 3: this location is actually easy. It simply is the situation after popping the activation records of `g()` and `f()`. The stack is therefore empty (see figure (c)).

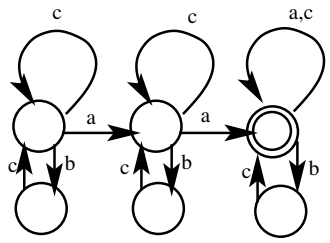


6

2. **(a)** *Consider the language L consisting of all strings w over the alphabet $\Sigma = \{a, b, c\}$ such that w contains at least two times a symbol a, and each occurrence of a symbol b is directly followed by a symbol c.*

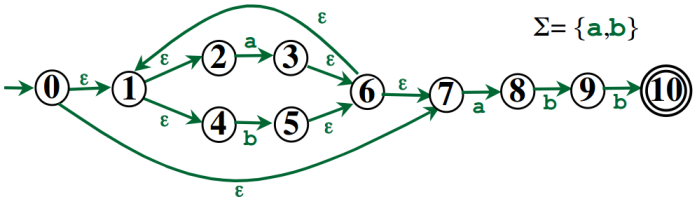
   Give a regular expression that describes the language L, and draw a deterministic finite state automaton (DFA) that accepts L.

   The following DFA accepts the language L:

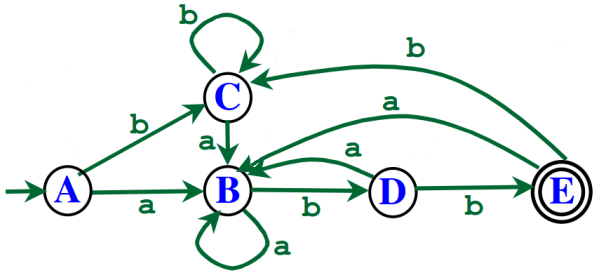


From the DFA we can see that L is defined by the regular expression $(bc|c)^*a(bc|c)^*a(a|bc|c)^*$.

(b) *Consider the following non-deterministic finite state automaton (NFA).*



The above NFA is equivalent to the following deterministic finite state automaton (DFA). Show all steps of the conversion process that converts the NFA into the DFA.



We start by computing the first state: $A = \varepsilon\text{-closure}(\{0\}) = \{0, 1, 2, 4, 7\}$. Next, we compute the transition table, and construct new states on-the-fly.

From	symbol	To	New state
A	a	$\varepsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$	B
A	b	$\varepsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\}$	C
B	a	$\varepsilon\text{-closure}(\{3, 8\}) = B$	
B	b	$\varepsilon\text{-closure}(\{5, 9\}) = \{1, 2, 4, 5, 6, 7, 9\}$	D
C	a	$\varepsilon\text{-closure}(\{3, 8\}) = B$	
C	b	$\varepsilon\text{-closure}(\{5\}) = C$	
D	a	$\varepsilon\text{-closure}(\{3, 8\}) = B$	
D	b	$\varepsilon\text{-closure}(\{5, 10\}) = \{1, 2, 4, 5, 6, 7, 10\}$	E
E	a	$\varepsilon\text{-closure}(\{3, 8\}) = B$	
E	b	$\varepsilon\text{-closure}(\{5\}) = C$	

3. *Consider the following grammar, in which upper case letters denote the non-terminals , and lower case letters the terminals:*

$$\begin{array}{lcl} S & \to & A \\ S & \to & B \\ A & \to & a\ A \\ A & \to & a\ B \\ B & \to & B\ b \\ B & \to & c \end{array}$$

(a) *For all non-terminals, determine the First and Follow sets. Explain why this grammar is not LL(1).*

Non-terminal	First set	Follow set
S	$\{a, c\}$	$\emptyset$
A	$\{a\}$	$\emptyset$
B	$\{c\}$	$\{b\}$

The grammar is not $LL(1)$ since there are two LL(1)-problems:

- The symbol a is in the intersection of the first sets of the rules $A \to a\ A$ and $A \to a\ B$.
- The rule $B \to B\ b$ is left-recursive.

(b) *Convert the above grammar into an equivalent LL(1) grammar.*

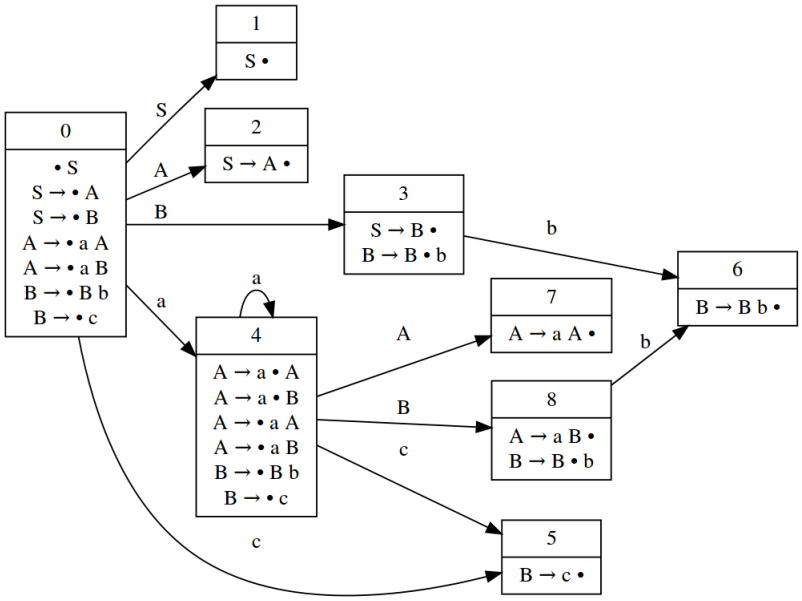
We start by eliminating the left-recursion on B. It can produce only strings of the form c, cb, cbb, $cbbb$, i.e. cb^*. We can produce such strings by introducing an auxiliary non-terminal B' and replacing the rules of B by the new rules $B \to c\ B'$, $B' \to b\ B'$, and $B' \to \varepsilon$.

The conflict on A is simply resolved by left-factoring the common prefix a, so we replace the rules for A by the rules $A \to aA'$, $A' \to A$, and $A' \to B$. In conclusion, we found the following equivalent LL(1)-grammar:

$$\begin{array}{lcl} S & \to & A \\ S & \to & B \\ A & \to & a\ A' \\ A' & \to & A \end{array} \qquad \begin{array}{lcl} A' & \to & B \\ B & \to & c\ B' \\ B' & \to & b\ B' \\ B' & \to & \varepsilon \end{array}$$

(c) *Compute the LR(0)-item sets of the given grammar.*

The LR(0)-item sets are depicted in the following automaton:



(d) *Is the grammar LR(0), SLR(1), LR(1)? In case of conflicts be sure to identify them clearly (you do not have to solve them).*
From the automaton it is clear that there are two shift/reduce conflicts: one in state 3, and one in state 8. So, the grammar is not LR(0).

This can also be seen from the parse table (but not asked for):

State	a	b	c	S	A	B
0	shift(4)		shift(5)	1	2	3
1	accept	accept	accept			
2	reduce($S \to A$)	reduce($S \to A$)	reduce($S \to A$)			
3	reduce($S \to B$)	shift(6) reduce($S \to B$)	reduce($S \to B$)			
4	shift(4)		shift(5)		7	8
5	reduce($B \to$ c)	reduce($B \to$ c)	reduce($B \to$ c)			
6	reduce($B \to B$ b)	reduce($B \to B$ b)	reduce($B \to B$ b)			
7	reduce($A \to$ a A)	reduce($A \to$ a A)	reduce($A \to$ a A)			
8	reduce($A \to$ a B)	shift(6) reduce($A \to$ a B)	reduce($A \to$ a B)			

However, if we try to construct the SLR(1) table, then we find:

State	a	b	c	$	S	A	B
0	shift(4)		shift(5)		1	2	3
1				accept			
2				reduce($S \to A$)			
3		shift(6)		reduce($S \to B$)			
4	shift(4)		shift(5)			7	8
5		reduce($B \to$ c)		reduce($B \to$ c)			
6		reduce($B \to B$ b)		reduce($B \to B$ b)			
7				reduce($A \to$ a A)			
8		shift(6)		reduce($A \to$ a B)			

We see that the conflicts are resolved. Hence the grammar is SLR(1).

Since the grammar is SLR(1) and SLR(1)$\subset$LR(1), we are sure that the grammar is LR(1).

4. *Given is the following LL(1) grammar for expressions (non-terminals start with an upper case letter, terminals are completely in lower case, and ε denotes the empty string):*

$$
\begin{array}{lcl}
E & \to & T\ Etail \\
Etail & \to & plus\ T\ Etail \\
Etail & \to & minus\ T\ Etail \\
Etail & \to & \varepsilon \\
T & \to & F\ Ttail \\
Ttail & \to & times\ F\ Ttail \\
Ttail & \to & divide\ F\ Ttail \\
Ttail & \to & \varepsilon \\
F & \to & leftpar\ E\ rightpar \\
F & \to & number \\
F & \to & identifier
\end{array}
$$

The terminal symbols of this grammar are represented by the enumeration type `tokens`*:*

```
typedef enum {
```

```
  divide, identifier, minus, number, plus, times, leftpar, rightpar
} tokens;
```

A global variable `currentToken` *and the function* `accept` *are used for interfacing with the lexer (you don't have to write the lexer, you may assume that* `yylex()` *exists, but you are only allowed to call it via* `accept`*). You can initialize* `currentToken` *using the function* `initParser`*.*

```
tokens currentToken;

void initParser() {
  currentToken = yylex();
}

int accept(tokens tok) {
  if (currentToken == tok) {
    currentToken = yylex();
    return 1;
  }
  return 0;
}
```

There is also a void function `syntaxError` *available, that prints an error message and aborts:*

```
void syntaxError(){
  printf("Syntax error: abort\n");
  exit(EXIT_FAILURE);
}
```

Write a Recursive Descent Parser *for the above grammar for expressions. The parser should print an error message if it detects a syntax error (using* `syntaxError`*).*

The translation of the rules into recursive routines is straight-forward:

```
void E();

void F() {
  if (accept(leftpar)) {
    E();
    if (!accept(rightpar)) {
      syntaxError();
    }
  }
  if (accept(number) || accept(identifier)) return;
  syntaxError();
}

void Ttail() {
  if (accept(times) || accept(divide)) {
    F();
    Ttail();
    return;
  }
}

void T() {
  F();
  Ttail();
}
```

```
void Etail() {
  if (accept(plus) || accept(minus)) {
    T();
    Etail();
    return;
  }
}

void E() {
  T();
  Etail();
}

void parser() {
  initParser();
  E();
}
```

5. **(a)** *Consider the following two code fragments (in which all variables are of type* `int`*):*

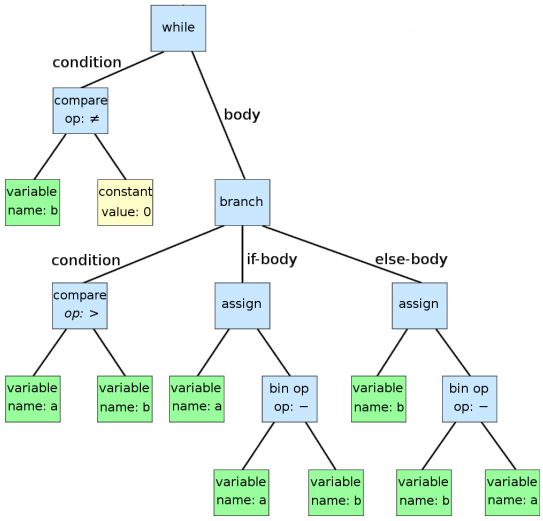
```
b = a;                    b = a;
c = b + 1;                c = a + 1;
d = b;                    d = a;
b = d + c;                b = a + c;
b = d;                    b = a;
```

A compiler that uses copy propagation *translates the code fragment on the left hand side into the equivalent code fragment on the right hand side. Explain step by step which actions the compiler performs during this translation.*

Answer: For copy propagation, a table with equivalences is needed. This table is updated after each assignment, and can be used to replace operands by copies:

assignments	translation	equivalences
`b = a;`	`b = a;`	(b,a)
`c = b + 1;`	`c = a + 1;`	(b,a)
`d = b;`	`d = a;`	(b,a), (d,a)
`b = d + c;`	`b = a + c;`	(d,a)
`b = d;`	`b = a;`	(b,a), (d,a)

(b) *The following figure shows an Abstract Syntax Tree (AST) of the Euclidean algorithm for finding the greatest common divisor of the variables* `a` *and* `b`*.*



Give (pseudo-)code for a recursive algorithm `void genIRcode(ASTtree tree)` *that translates ASTs like the above AST into an intermediate code representation that uses quadruples (i.e. assignments with a left hand side, and a right hand side with at most 2 operands and an operator), and conditional jumps (*`if (variable) goto label`*). You do not need to specify the AST data structure: you may use pseudo-instructions like "*`if node is a while-node {...}`*". You may assume that all variables are of type* `int`*, and that there is an infinite amount of temporary variables* `t0, t1, ...` *that you can use without declaration.*

Answer: The routine `genIRcode()` can be found in the lecture slides.

(c) *What is the code that would be generated by* `genIRcode()` *if we apply it to the AST of the Euclidean algorithm?*

Answer:

```
lab1:
  t1 = b;
  t2 = 0;
  t3 = t1 - t2;
  if (t3 == 0) goto lab2:
lab3:
  t4 = a;
  t5 = b;
  t6 = t4 - t5;
  if (t6 <= 0) goto lab4
  t7 = a;
  t8 = b;
  t9 = t7 - t8;
  a = t9;
  goto lab5
lab4:
  t10 = b;
  t11 = a;
  t12 = t10 - t11;
  b = t12;
lab5:
  goto lab1;
lab2:
```