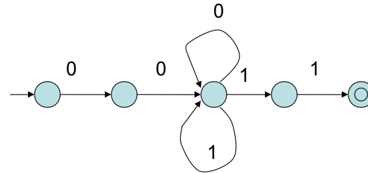
Solutions Exam Compiler Construction—April 1st 2015

1. **lexical analysis**[20 points]

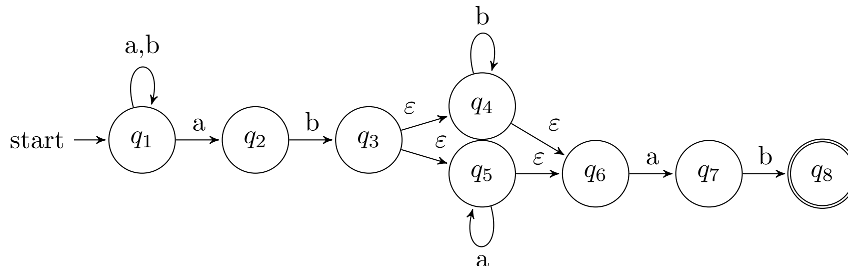
(a) [3 points] Consider the following NFA over the alphabet $\Sigma = \{0, 1\}$. Give a regular expression for the language that is accepted by this NFA.



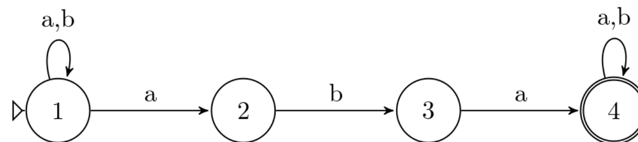
Answer: Clearly, the NFA accepts the regular expression $00(0|1)^*11$.

(b) [7 points] Consider the language L consisting of all strings over the alphabet $\Sigma = \{a, b\}$ that can be produced by the regular expression $(a|b)^*ab(b^*|a^*)ab$. Draw a non-deterministic finite state automaton (NFA) that accepts L.

Answer: Of course, several solutions are possible. It is probably easiest to use an automaton that makes use of ε-transitions.



(c) [10 points] Consider the following non-deterministic finite state automaton (NFA) over the alphabet $\Sigma = \{a, b\}$.



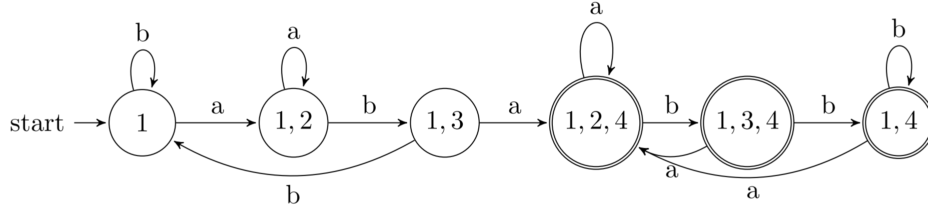
Construct an equivalent deterministic finite state automaton (DFA) for this NFA.

Answer: To convert an NFA into a DFA, we need to perform a power set construction:

state	input	new state
{1}	a	{1,2}
{1}	b	{1}
{1,2}	a	{1,2}
{1,2}	b	{1,3}
{1,3}	a	{1,2,4}
{1,3}	b	{1}

state	input	new state
{1,2,4}	a	{1,2,4}
{1,2,4}	b	{1,3,4}
{1,3,4}	a	{1,2,4}
{1,3,4}	b	{1,4}
{1,4}	a	{1,2,4}
{1,4}	b	{1,4}

In conclusion, we found the following DFA:



2. **Grammars**[25 points]

(a)[5 points] Consider the following grammar, in which upper case letters denote the non-terminals and lower case letters the terminals:

$$
\begin{array}{lcl}
S & \rightarrow & A\ S \\
S & \rightarrow & \epsilon \\
A & \rightarrow & A\ b \\
A & \rightarrow & a\ A\ b \\
A & \rightarrow & a\ b
\end{array}
$$

Show that this grammar is *ambiguous.*

Answer: It is enough to show that there exists a string that can be derived with two different parse trees. For example, the string *aabbb* can be obtained with two different derivations:

$$
\begin{array}{lcl}
S & \Rightarrow & A\ S \Rightarrow a\ A\ b\ S \Rightarrow a\ A\ b\ b\ S \Rightarrow a\ a\ b\ b\ b\ S \Rightarrow a\ a\ b\ b\ b \\
S & \Rightarrow & A\ S \Rightarrow A\ b\ S \Rightarrow a\ A\ b\ b\ S \Rightarrow a\ a\ b\ b\ b\ S \Rightarrow a\ a\ b\ b\ b
\end{array}
$$

(b)[5 points] Consider the following grammar for arithmetic expressions (which contains only one nonterminal E):

$$
\begin{array}{lcl}
E & \rightarrow & E\ \texttt{+}\ E \\
E & \rightarrow & E\ \texttt{-}\ E \\
E & \rightarrow & E\ \texttt{*}\ E \\
E & \rightarrow & E\ \texttt{/}\ E \\
E & \rightarrow & E\ \texttt{\^{}}\ E) \\
E & \rightarrow & (\ E\ ) \\
E & \rightarrow & \texttt{number}
\end{array}
$$

This grammar is clearly ambiguous. Convert this grammar into an unambiguous grammar that accepts the same language. Moreover, the parse tree of an expression should be such that it corresponds with the normal rules for arithmetic: the addition, subtraction, multiplication, and division operators are left-associative while the exponentiation operator (`^`) is right-associative.

Answer: The grammar is converted into the standard grammar with terms and factors:

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
E & \rightarrow & E + T & E & \rightarrow & F \\
E & \rightarrow & E - T & F & \rightarrow & X \\
E & \rightarrow & T & F & \rightarrow & F \mathbin{\hat{}} X \\
T & \rightarrow & T * F & F & \rightarrow & F \mathbin{\hat{}} E \\
T & \rightarrow & T \,/\, F & X & \rightarrow & (\ E\ ) \\
E & \rightarrow & F & X & \rightarrow & \texttt{number}
\end{array}
$$

(c)[5 points] Consider the following grammar for strings containing parentheses.

$$
\begin{array}{ll}
S \rightarrow & S\ S \\
S \rightarrow & (\ S\ ) \\
S \rightarrow & (\ )
\end{array}
$$

Explain why this grammar is not LL(1), and convert it into an equivalent LL(1) grammar.

The grammar is clearly not LL(1), since the very first rule is left-recursive. It is not hard to see that this grammar can be converted in the following equivalent LL(1) grammar:

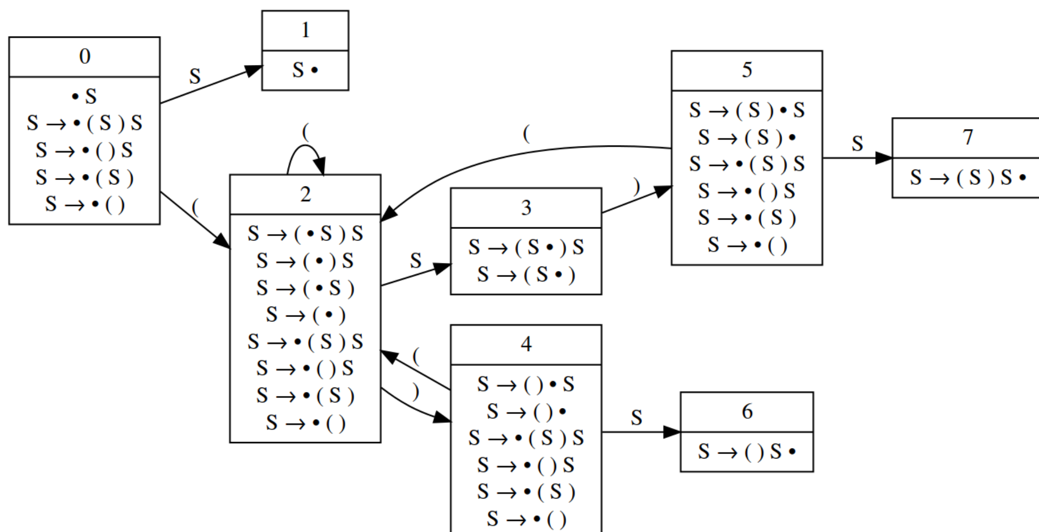
$$
\begin{array}{ll}
S \rightarrow & (\ T \\
T \rightarrow & S\ )\ U \\
T \rightarrow &)\ U \\
U \rightarrow & S \\
U \rightarrow & \varepsilon
\end{array}
$$

(d)[10 points] Consider the following variation of the grammar for strings containing parentheses.

$$
\begin{array}{ll}
S \rightarrow & (\ S\ )\ S \\
S \rightarrow & (\ )\ S \\
S \rightarrow & (\ S\ ) \\
S \rightarrow & (\ )
\end{array}
$$

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).

Answer: We start with the construction of the LR(0)-item sets (and the corresponding automaton or tables):



Note that this automaton does not use the augmented rule $S' \rightarrow S\ \$$. If a student introduced this rules, then this is of course perfectly fine.

From the figure, it is easy to see that there are two *shift-reduce* conflicts: one in state 4, and one in state 5. Hence, the grammar is not LR(0). We can also draw this conclusion from the LR(0) parse tables:

State	(	)	S
0	shift(2)		1
1	accept	accept	
2	shift(2)	shift(4)	3
3		shift(5)	
4	shift(2) reduce($S \rightarrow$ ( ))	reduce($S \rightarrow$ ( ))	6
5	shift(2) reduce($S \rightarrow$ (S))	reduce($S \rightarrow$ (S))	7
6	reduce($S \rightarrow$ ( ) S)	reduce($S \rightarrow$ ( ) S)	
7	reduce($S \rightarrow$ (S) S)	reduce($S \rightarrow$ (S) S)	

To check whether the grammar is SLR(1), we need to construct the SLR(1) parse table:

State	(	)	$	S
0	shift(2)			1
1			accept	
2	shift(2)	shift(4)		3
3		shift(5)		
4	shift(2)	reduce($S \rightarrow$ ( ))	reduce($S \rightarrow$ ( ))	6
5	shift(2)	reduce($S \rightarrow$ (S))	reduce($S \rightarrow$ (S))	7
6		reduce($S \rightarrow$ ( ) S)	reduce($S \rightarrow$ ( ) S)	
7		reduce($S \rightarrow$ (S) S)	reduce($S \rightarrow$ (S) S)	

Since there are no conflicts in this table, the grammar is SLR(1). Since the grammar is SLR(1), it is surely LR(1) (result of the inclusion relation $LR(0) \subset SLR(1) \subset LR(1)$).

3. **Recursive descent parsing**[20 points]
   Given is the following LL(1) grammar for strings over the alphabet $\Sigma = \{a, b, c, d\}$. The non-terminals are S (start symbol), A, B, C, and D. The terminals are the tokens `a`, `b`, `c`, and `d`. Note, that ε denotes the empty string:

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{c}\ S\ A\ \texttt{a} \\
S & \rightarrow & \texttt{d}\ D\ \texttt{a} \\
S & \rightarrow & A\ \texttt{b}\ \texttt{a} \\
A & \rightarrow & B\ \texttt{a}\ C \\
B & \rightarrow & \texttt{b} \\
B & \rightarrow & \varepsilon \\
C & \rightarrow & \texttt{c} \\
C & \rightarrow & \varepsilon \\
D & \rightarrow & \texttt{d}
\end{array}
$$

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

```
typedef enum {
  toka, tokb, tokc, tokd
} 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. The parser should print an error message if it detects a syntax error (using `syntaxError`).

Answer: It helps to make a function `expect(tokens tok)` that expects the token `tok`. If the current token mismatches, the routine prints a syntax error and aborts. The translation from grammar into recursive procedure calls is one-to-one:

```
void expect(tokens tok) {
  if (!accept(tok)) {
    syntaxError();
  }
}

void parseB() {
  /* B -> b */
  accept(tokb);
  /* B -> epsilon */
  return;
}

void parseC() {
  /* C -> c */
  accept(tokc);
```

```
  /* C -> epsilon */
  return;
}

void parseD() {
  /* D -> d */
  expect(tokd);
}
void parseA() {
  /* A -> B a C */
  parseB();
  expect(toka);
  parseC();
}

void parseS() {
  /* S -> c S A a */
  if (accept(tokc)) {
    parseS();
    parseA();
    expect(toka);
    return;
  }
  /* S -> d D a */
  if (accept(tokd)) {
    parseD();
    expect(toka);
    return;
  }
  /* S -> A b a */
  parseA();
  expect(tokb);
  expect(toka);
}

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

4. **Syntax directed translation and optimization**[15 points]
(a) [5 points] Consider the following C code fragment for computing the greatest common divisor of `x` and `y` (both variables are of type `int`):

```
while (x != y) {
  if (x >= y) {
    x = x - y;
  } else {
    y = y - x;
  }
}
```

Translate this code by hand, simulating syntax directed translation (without any optimizations), into intermediate code. You may introduce as many auxiliary variables as needed.

The intermediate code may only consist of assignments of the form `<operand1>=<operand2>` or `<operand1>=<operand2> <operator> <operand3>` (i.e. quadruples), labels, and conditional jumps. Here the operands may be a variable or a constant. The conditional jumps may only be of the form:

- `if eq(<operand>) goto <labX>` meaning `if (<operand> == 0) goto <labX>`.
- `if lt(<operand>) goto <labX>` meaning `if (<operand> < 0) goto <labX>`.

Note that the `<operand>` in a conditional jump must be a variable.

Answer: The requested intermediate code would be generated by the routine `genIRcode()` which is discussed during the lectures (and is also in the lecture slides).

```
lab1:
  t1 = x;
  t2 = y;
  t3 = t1  - t2;
  if eq(t3) goto lab2;
  t4 = x;
  t5 = y;
  t6 = t4 - t5;
  if lt(t6) goto lab 3;
  t7 = x;
  t8 = y;
  t9 = t7 - t8;
  x = t9;
  goto lab4;
lab3:
  t10 = y;
  t11 = x;
  t12 = y - x;
  y = t12;
lab4:
  goto lab1
lab2:
```

(b)[5 points] Consider the following intermediate codes in quadruple format (all variables are of type `int`):

assignments	translation
`a = b;`	`a = b;`
`c = d + a;`	`c = d + b;`
`d = e;`	`d = e;`
`c = a + d;`	`c = b + e;`
`c = a;`	`c = b;`
`a = a + d;`	`a = b + e;`
`b = d;`	`b = e;`
`a = c;`	`a = c;`
`d = d + 1;`	`d = e+1;`
`c = a + 1;`	`c = c + 1;`
`a = c;`	`a = c;`

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 line by line which actions the

compiler performs during this translation and which information is stored during this process.

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
`a = b;`	`a = b;`	(a,b)
`c = d + a;`	`c = d + b;`	(a,b)
`d = e;`	`d = e;`	(a,b), (d,e)
`c = a + d;`	`c = b + e;`	(a,b), (d,e)
`c = a;`	`c = b;`	(a,b), (c,b), (d,e)
`a = a + d;`	`a = b + e;`	(c,b), (d,e)
`b = d;`	`b = e;`	(b,e), (d,e)
`a = c;`	`a = c;`	(a,c), (b,e), (d,e)
`d = d + 1;`	`d = e+1;`	(a,c), (b,e)
`c = a + 1;`	`c = c + 1;`	(b,e)
`a = c;`	`a = c;`	(a,c), (b,e)

(c) [5 points] An algorithm that performs redundant/dead code elimination transforms the code on the left hand side of question (b) into:

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

Explain how this optimization works (line by line). What would be the resulting code if we apply this optimization to the code on the right hand side of question (b)?

Answer: A variable `x` is called *alive* if its value is being used as an operand later, but before `x` is assigned some new value. A variable `x` is called *dead* if its value is not being used, until `x` is assigned some new value. Assignments to dead variables can be removed.

If we apply this principle to the code on the left hand side of question (b), then we draw the following conclusions:

- `a=b;` In the next assignment, `a` is used as an operand. So, this assignment 'survives'.
- `c = d + a;` In the next line the variable `c` is not used, while a line later is gets overwritten. Therefore, we eliminate this line.
- `d = e;` The variable `d` is used in the next line. So, it 'survives'.
- `c = a + d;` In the next line the variable `c` gets overwritten. Therefore, we eliminate this line.
- `c = a;` The variable `c` is used 3 lines later, so it 'survives'.
- `a = a + d;` The variable `a` is not used in the next line, and a line later it gets overwritten. This line is eliminated.
- `b = d;` This is the last assignment to `b` in this basic block. So, it survives.
- `a = c;` The variable `a` is used two lines later. It survives.
- `d = d + 1;` This is the last assignment to `d` in this basic block. So, it survives.

- `c = a + 1;` This is the last assignment to `c` in this basic block. So, it survives.
- `a = c;` This is the last assignment to `a` in this basic block. So, it survives.

If we apply this process to the code in the right hand side of part (b), then the following lines 'survive':

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

5. **Memory organization: activation records** [10 points]
Consider the following code fragment:

```
int x = 42, y = 10;

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

int f(int a) {
  int b = 2*a;
  int c = x+y;
  /* location 1:  (after c=x+y, before return) */
  return g(a, b, c);
}

int main() {
  int a = f(x);
  /* location 3: after return from f(); before printf() */
  printf("%d\n", a);
  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, assume that the code generator does not use registers to store variables, nor are function results passed via registers. So, on function entry/exit there is no need to save registers. You may also assume that at the beginning of the function `main`, the stack is empty (even though this is not true in reality). Also, assume that the stack grows from low addresses to high addresses (so the base pointer of `main` is address 0).

Answer: Global variables are stored on the heap, so the variables `x` and `y` are 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.

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

(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`.

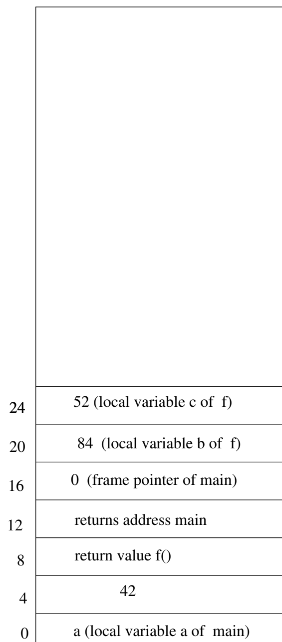
(h) Place the return value of `new` on the stack.

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

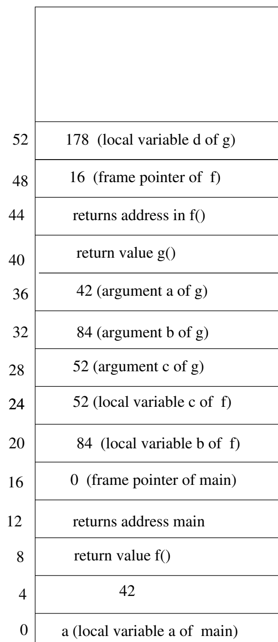
(j) Jump to the saved return address.

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

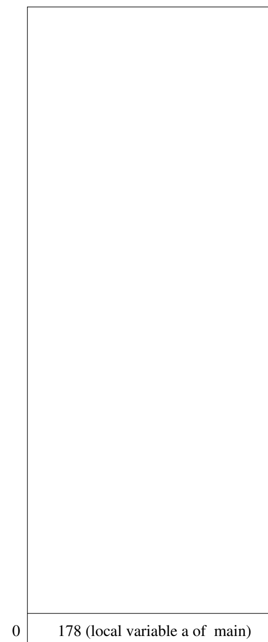
- location 1: We start with an empty stack (beginning of `main`), and push the local variable `a`. Next, the argument of the call `f(x)` is pushed: we push the value of `x` (i.e. 42). Next, we reserve space (4 bytes) for the return value of `f()`, and push the return address in `main` (i.e. location 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 52 (c), push 84 (b), push 42 (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 `d` 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 layout is given in figure (c).



(a)



(b)



(c)