TDDB44 Compiler Construction – Exam 16/12 – 2006 Example Solutions

---

1. a. ((0|1)*00(0|1)*1) **|** ((1|01)*(0|ε))

b. **Start** : ε -> A, ε -> E **A** : 0 -> A, 0 -> B, 1 -> A **B** : 0 -> C **C** : 0 -> C, 1 -> C, 1 -> D
D : **E** : ε -> G, 0 -> F, 0 -> G, 1 -> E **F** : 1 -> E **G** :
(D and G are final states and missing arcs goes to an error state)

c. **Start** : 0 -> A, 1 -> Start **A** : 0 -> B, 1 -> Start **B** : 0 -> B, 1 -> C **C** : 0 -> B, 1 -> C
(Start, A and C are final states)

d. {palindromes over {a,b}}, {balanced strings of parentheses}, etc.

---

2. a. **+** Better code generation, smaller code, more efficient code, more expressive languages possible (some languages need multi-pass: for instance with function usage before function being defined), etc... **–** Longer compilation time, more memory consumption, etc...

b. Scanner (input: symbols, output: tokens), Parser (input: tokens, output: abstract syntax tree and/or error messages), Semantic Analysis (input: abstract syntax tree, output: error messages, abstract syntax tree), Optimization (input: abstract syntax tree, output: (optimized) abstract syntax tree), Intermediary Code Generation (input: abstract syntax tree, output: intermediary code), Code Generation (input: intermediary code, output: executable code (perhaps assembly)). The scanner creates tokens (specified by the language specification); the parser asks the scanner for tokens and parses the program and determines whether it follows the language specification or not and if it does creates an abstract syntax tree; the semantic analysis perform type checking, etc.; the optimization part optimizes the abstract syntax tree; the intermediary code generation creates an internal form closer to the final code; and the code generation generates the final code (this is typically architecture specific, one has to consider memory issues, etc.).

---

3. a. *S -> aS | aX X -> XbX | d* (See also lecture 5.)

Left factoring: *S -> aS' S' -> S | X* Left recursion: *X -> dX' X' -> bXX' | ε*

```
procedure S
begin
if token == a then
  scan(); S2();
else error(…);
end if;
end procedure;
```

```
procedure S2  // S'
begin
if token == a then
  S();
else if token == d then
  X();
else error(…);
end if;
end procedure;
```

```
procedure X
begin
if token == d then
  scan(); X2();
else error(…);
end if;
end procedure;
```

```
procedure X2 // X'
begin
if token == b then
  scan(); X(); X2();
end if;
end procedure;
```

```
procedure main
scan(); S();
if not eof then error(…);
end main;
```

b. For simplicity reasons, over-kill.

---

4. *A -> xAyAz | xBzAx | ε B -> yBzBx | yBxBz | ε* (See also lecture 7.)

S -> .A$
A -> . xAyAz
A -> . xBzAx
A -> .

FOLLOW(A) = {$, x, y, z}
Shift / Reduce conflict and x in FOLLOW(A)
=> **not SLR(1)**
Conflicts => **not LR(0)**
Not SLR(1) => **not LR(0)**

---

5. An activation record is a data structure, typically allocated as a contiguous block of memory locations on the stack that contains all information required for executing a procedure/function call: call arguments, return address of the caller, link to the caller's activation record (i.e., dynamic link) and, where applicable, to the statically enclosing procedure's/function's activation record; saved registers of the caller, local variables of the callee, and (if the procedure returns a value, in other words it is a function) a memory location where the return value can be stored. [As an optimization to save memory accesses, some of these values (e.g., a small number of arguments and the return value) may, under certain conditions, be passed in special registers instead.] Languages that do not support recursion may statically pre-allocate one activation record for every procedure/function in a predetermined memory area since there can be at most one activation of a procedure at any time. Languages that do not have procedures/functions do not need activation records at all.

6.
Hash table: 1. j, k, link to 5 **|** 2. i, link to 6 **|** 3. (empty) **|** 4. main, link to 2
Symbol table:
1. int k, hash link is null.
2. main(), hash link is null.
3. int i, hash link is null.
4. int j, hash link 1.
5. int k, hash link 4.
6. int i, hash link 3. (sym_pos)
Block table: 1. Link to 1 in symbol table. **|** 2. Link to 2. **|** 3. Link to 4. **|** 4. Link to 6 (current_scope).

In each iteration of the for-loop, a scope is opened, an integer variable i is added to the symbol table and then the scope is closed. Opening and closing a scope involves, among other things, changing the block table and the current_scope pointer. When searching for k in the assignment i = k * 2; we simply do a hash and receive an entry into the symbol table. If we don't find the correct variable we follow hash links, but in this case we found the right variable k at first try. We can enumerate the variables in a block by going from block_table[current_scope] to sym_pos.

---

7. What we want:
p: <Quadruples for stmt>
  <Quadruples for temp := expr>
 (JEQZ, p, temp, 0)

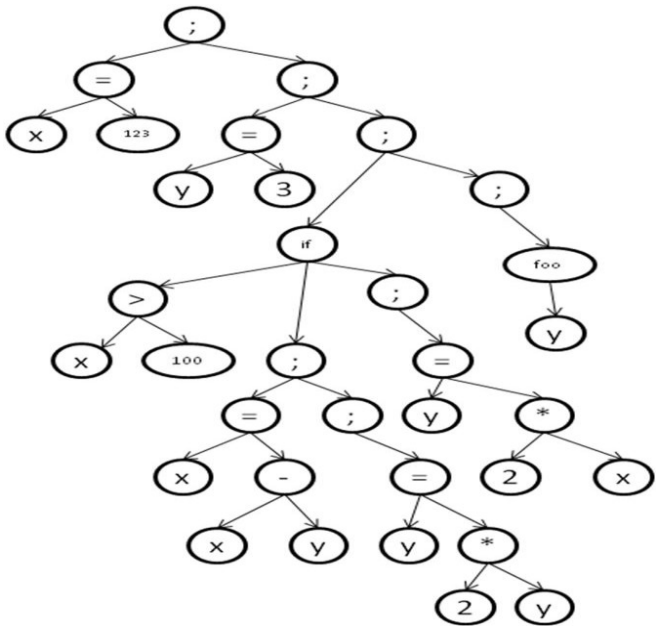
Rewrite the grammar and add semantic actions:

<rep-smt> -> <rep> <stmt> UNTIL <expr> { GEN(JEQZ, <rep>.quad, <expr>.addr, 0); }

<rep> -> REPEAT { <rep>.quad = currentquad + 1; }

---

8. a. Abstract syntax tree:



Quadruples:

q_iload	**123**	**-**	**$1**
q_iassign	**$1**	**-**	**x**
q_iload	**3**	**-**	**$2**
q_iassign	**$2**	**-**	**y**
q_iload	**100**	**-**	**$3**
q_igt	**x**	**$3**	**$4**
q_jmpf	**1**	**$4**	**-**
q_iminus	**x**	**y**	**$5**
q_iassign	**$5**	**-**	**x**
q_iload	**2**	**-**	**$6**
q_imult	**$6**	**y**	**$7**
q_iassign	**$7**	**-**	**y**
q_jmp	**2**	**-**	**-**
q_labl	**1**	**-**	**-**
q_iload	**2**	**-**	**$8**
q_imult	**$8**	**X**	**$9**
q_iassign	**$9**	**-**	**y**
q_labl	**2**		
q_param	**y**	**-**	**-**
q_call	**Foo**	**1**	**-**

Postfix code:

x 123 =
y 3 =
x 100 >
L1 JEQZ
x x y - =
y 2 y * =
L2 JUMP
L1: y 2 x * =
L2: y foo

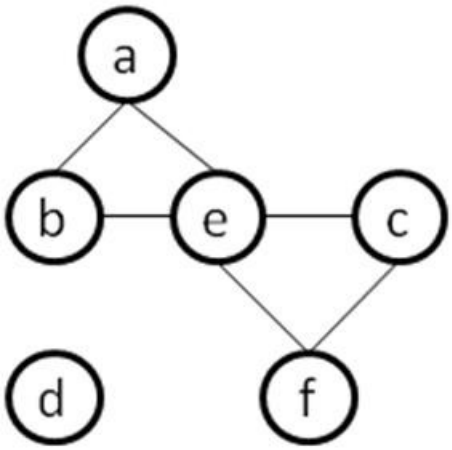
b. Basic blocks:

--
1.
2. Link to 3.
--
3.
4.
5.
6. Link to 7 and 13.
--
7.
8. Link to 1 and 10.
--
10.
11.
12. Link to 3.
--
13.
--

9.

a. Branch instructions (might) force the pipeline to restart and thus reduce performance. This is worse on deeply superscalar processors. Branch prediction determines whether a conditional branch in the instruction flow of a program is likely to be taken or not. Example: early implementations of SPARC and MIPS (two of the first commercial RISC architectures) did trivial branch prediction, they always predicted that a branch (or unconditional jump) would not be taken, so they always fetched the next sequential instruction and only when the branch or jump was evaluated did the instruction fetch pointer get set to a non-sequential address.

b. Here we only draw the live range interference graph for the entire fragment:



We need 3 colors to color the graph and thus at least 3 registers. (For full points one should also draw the live ranges.)

c.
From a compiler (software) engineering point of view, it is easier to handle these tasks in separate phases, such that each can be developed and tested separately. However, this separation leads to "artificial" constraints on the code; the phase run first constrains the other one. For instance, doing register allocation first may introduce "false" data dependences that constrain the instruction scheduler. Vice versa, scheduling instructions first may lead to excessive register need, which in turn may require spilling, and this spill code must be scheduled again.