TDDB44 Compiler Construction Exam 17/12-2009 Example Solutions

3. *a.* The wider scope available to these compilers allows better code generation (e.g. smaller code size, faster code) compared to the output of one-pass compilers, at the cost of higher compiler time and memory consumption. In addition, some languages cannot be compiled in a single pass, as a result of their design. One can also usually construct a more modular compiler using multiple passes.

b.
i.) One intermediate representation is transformed into the next representation which is closer to the actual target code. That is, intermediate representations early in the compilation process are closer in nature to the original source language while intermediate representations late in the compilation process are closer in nature to the target language. Code generation is continuous lowering of the intermediate representation towards target code.

ii.) **Pros:** Analysis/Optimization engines can work in the most appropriate level of abstraction. Clean separation of compiler phases.
Cons: Framework gets larger and slower.

4.
a. LL- and LR-parsers have a valid prefix property i.e. discover the error when the substring being analyzed together with the next symbol do not form a prefix of the language.

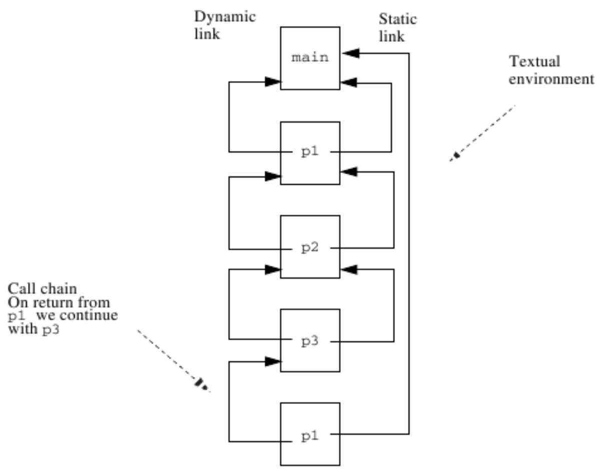
Example:
A = B + C * D THEN . . . ELSE . . .
When the next symbol is THEN we discover the error (the keyword IF is missing).

b. On discovering an error, a parser may perform local corrections on the remaining input.
Example: replace a comma with a semicolon or delete an extraneous semicolon.

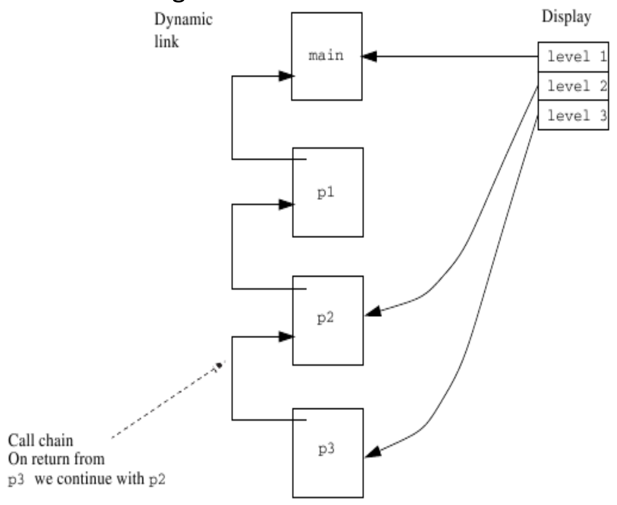
c. Algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible.

Example:
A = B + C * D THEN . . . ELSE . . .
Inserting the keyword IF is a minimum distance repair.

10.
Static link:
The static link is a pointer to the most recent activation record for the textually surrounding block. It is used to find non-local variables, etc... This method is practical and uses little space. With deeply nested procedures it will be slow.



Display:
Display is a table with pointers (addresses) to the most recent activation record for the textually surrounding block. The display can be stored in the activation records. Display is faster than static link for deep nesting, but requires more space. Display can be slightly slower than static link for very shallow nesting.



11.
a. A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor. Each functional unit is not a separate CPU core but an execution resource within a single CPU such as an arithmetic logic unit, a bit shifter, or a multiplier. Run-time scheduling by instruction dispatcher.

The VLIW approach executes operations in parallel based on a fixed schedule; compiler (or assembler-level programmer) must determine the schedule statically. So VLIW is harder to generate code for.

In superscalar designs, the number of execution units is invisible to the instruction set. Each instruction encodes only one operation. For most superscalar designs, the instruction width is 32 bits or fewer. In contrast, one VLIW instruction encodes multiple operations; specifically, one instruction encodes at least one operation for each execution unit of the device.

b. In computer architecture, a branch predictor is a digital circuit that tries to guess which way a branch (e.g. an if-then-else structure) will go before this is known for sure. The purpose of the branch predictor is to improve the flow in the instruction pipeline. Branch predictors are crucial in today's pipelined microprocessors for achieving high performance. This is because we want to take the most likely branch and thus reduce the number of instructions we will have to discard. Branch-instructions force the pipeline to restart and thus reduce performance. Worse on deeply pipelined superscalar processors.

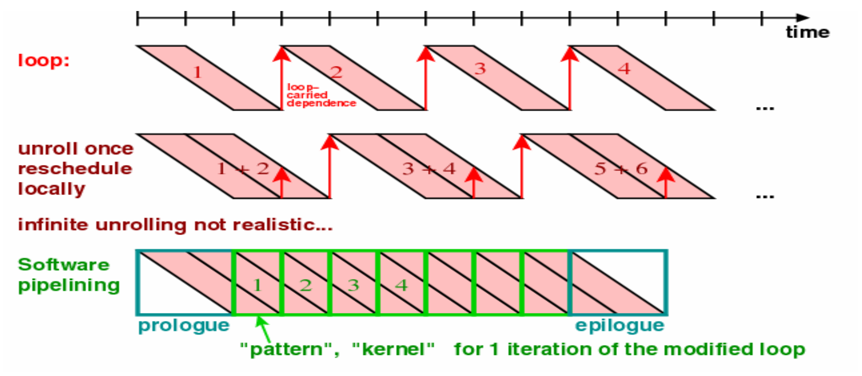
```
if (d==0) d=1;
if (d=1) ---

BNEZ R1, L1 ; branch b1 (d!=0)
DADDU R1,R0,#1 ; d==0, so d=1
L1: DADDIU R3,R1,#-1
BNEZ R3,L2 ; branch b2 (d!=1)
```

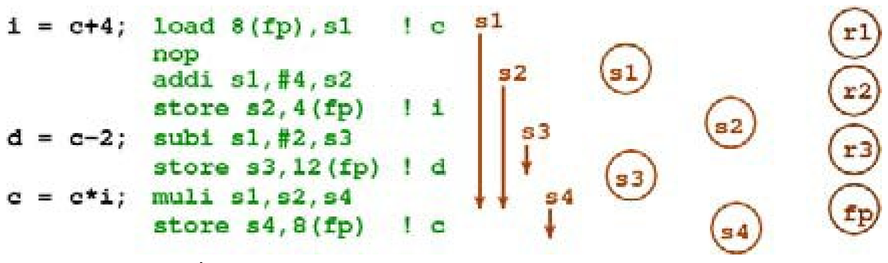
Possible Execution Sequence for the code fragment;

Initial d	d==0?	B1	d before b2	d==1?	b2
0	yes	not taken	1	yes	not taken
1	no	taken	1	yes	not taken
2	no	taken	2	no	taken

c. Software pipelining is a technique used to optimize loops, in a manner that parallels hardware pipelining. Software pipelining is a type of out-of-order execution, except that the reordering is done by a compiler (or in the case of hand written assembly code, by the programmer) instead of the processor.


time
loop:
1
2
3
4
loop-carried dependence
...
unroll once reschedule locally
1 + 2
3 + 4
5 + 6
...
infinite unrolling not realistic...
Software pipelining
1
2
3
4
prologue
epilogue
"pattern", "kernel" for 1 iteration of the modified loop

d. A variable is being defined at a program point if it is written (given a value) there. A variable is used at a program point if it is read (referenced in an expression) there. A variable is live at a point if it is referenced there or at some following point that has not (may not have) been preceded by any definition. A variable is reaching a point if an (arbitrary) definition of it, or usage (because a variable can be used before it is defined) reaches the point. A variable's live range is the area of code (set of instructions) where the variable is both alive and reaching.



```
i = c+4;  load 8(fp),s1    ! c
          nop
          addi s1,#4,s2
          store s2,4(fp)   ! i
d = c-2;  subi s1,#2,s3
          store s3,12(fp)  ! d
c = c*i;  muli s1,s2,s4
          store s4,8(fp)   ! c
```

Live ranges = red arrows.