Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

exam – **Compiler Construction** – in4020
June 29, 2009 09.00 - 10.30

This exam (8 pages) consists of 60 True/False questions.
Your score will be computed as: max(0, #correct - 30).
It is **not** allowed to consult the book, handouts, or any other notes.

Instructions for filling in the answer sheet:
- You may use a **pencil** (erasures are allowed) or a **pen** (blue or black, **no** red, **no** strike outs).
- Fill in the boxes **completely**.
- Answer **all** questions; there is no penalty for guessing.
- Do not forget to fill in your **Name** and **Student Number**, and to **sign** the form.

---

1. In a CFG (Context Free Grammar) the set of terminal and non-terminal symbols must be disjoint. true/false

2.

```
S  →  if C then S else S
```

This grammar is ambiguous. true/false

3. The regular expressions **(a+a)** and **(a)+(a)** describe the same set of strings. true/false

4. In a hand-written lexical analyzer the so-called **dot** marks the beginning of the next token or comment in the input buffer. true/false

5. A lexical analyzer *generator* automatically constructs an FSA (Finite State Automaton) that recognizes tokens. The generator is driven by a **regular description**.

```
dot          →  [.]
digit        →  [0-9]
integer      →  digit+
fixed_point  →  integer? dot integer
```

- The (minimal) FSA for accepting a `fixed_point` number consists of 3 states. true/false

6. When generating a lexical analyzer from a token description, the item sets (states) are constructed by two types of "moves": character moves and ϵ moves.
- ϵ moves do not consume any input character, but merely expand non-basic items with repetition and composition operators. true/false

7. Dotted items (T $\rightarrow \alpha \bullet \beta$) record which part of a token has already been matched. There are two kinds of basic items: **shift items** and **reduce items**.

`identifier` $\rightarrow \bullet$ `letter (letter_digit_underscore)`$\star$

- This is a shift item. true/false

8. A **recursive-descent** parser evaluates the nodes in the AST (Abstract Syntax Tree) in post-order. true/false

9. A **top-down** parser can handle LL(k) grammars. true/false

10. When constructing a top-down parser we need to compute the FIRST sets of all production alternatives. The FIRST set of an alternative α, FIRST(α), contains all terminals α can start with; if α can produce the empty string ϵ, this ϵ is included in FIRST(α).

S	$\rightarrow$	Ab
A	$\rightarrow$	Aa $\mid \epsilon$

- FIRST(S) contains 2 elements. true/false

11. Automatically generated top-down parsers are built upon a PDA (Push Down Automaton) driven by a prediction table.
- An entry in the prediction table specifies for every non-terminal / token combination to which state the PDA should move. true/false

12. Grammars with LL(1) conflicts can be made LL(1) by applying left-factoring, substitution, and left-recursion removal.
- Substitution takes care of FOLLOW/FOLLOW conflicts. true/false

13. The stack used in a bottom-up parser contains an alternating sequence of states and grammar symbols. true/false

14. An LR(0) bottom-up parser uses a combined GOTO and ACTION table. true/false

15. The SLR(1) parsing technique reduces a handle (the righthand side of a production N $\rightarrow \alpha$) when the current input token is an element of FOLLOW(N). true/false

16. The following set

S $\rightarrow \bullet$ A x {\$}

A $\rightarrow \bullet$ a {\$}

A $\rightarrow \bullet$ a A {\$}

is a valid LR(1) item set true/false

17. In an **attribute grammar** a node N in the AST may be annotated with two kinds of attributes: **inherited** and **synthesized** attributes.
- The inherited attributes are, by convention, drawn on the left of node N in the AST. true/false

18.

```
DigitSeq( INH base, SYN value) →
  DigitSeq(base, value) Digit(base, value)
  ATTRIBUTE RULES:
    SET value TO DigitSeq.value*Base + Digit.value;
```

The dependency graph associated with the attribute evaluation rule for `DigitSeq` contains 5 dependencies (arrows). true/false

19. The evaluation of attribute grammars is a complex task. The most general approach is to perform **multiple visits** over the AST. By restricting the evaluation rules simpler evaluation schemes can be used.
- **S-attributed grammars** can be evaluated in a *single* visit by a bottom-up parser. true/false

20. When dealing with attribute grammars it is important to detect cycles in the evaluation rules to prevent the evaluator to loop endlessly.
- By limiting ourselves to **L-attributed grammars**, the problem of cycle detection becomes irrelevant because they simply cannot occur. true/false

21. An AST can be **threaded** with a *single* visit. true/false

22. Several manual methods exist to annotate an AST with information aiding code generation.
- **Simple symbolic interpretation** can be used to perform **constant propagation** on a threaded AST. true/false

23.

$$\texttt{IN(N)} = \cup_{\texttt{M} \in \texttt{predecessor[N]}} \texttt{OUT(M)}$$
$$\texttt{OUT(N)} = \texttt{IN(N)} \setminus \texttt{KILL(N)} \cup \texttt{GEN(N)}$$

Forward data-flow equations can be solved by a basic closure-algorithm using the above inference rules. true/false

24. **Live analysis** can be performed by solving a set of data flow equations backwards. Setting up the set of equations boils down to generating the appropriate GEN and KILL sets for the individual statements in the data flow graph.

```
M[i] = v;
```

- The KILL set for this assignment statement must be set to $\{\texttt{i}\}$. true/false

25. A **recursive** interpreter is constructed as a set of routines that implement the semantics of each type of AST node.
- When interpreting a single expression from a user program, a routine may be invoked multiple times. true/false

26. To handle user-defined datatypes a **recursive** interpreter operates with **self identifying data**. true/false

27. **Simple code generation** considers one AST node at a time.
- When generating code for an identifier node, the compiler must distinguish a local variable from a global variable since they are addressed differently in the target code (i.e. assembly). true/false

28. Simple code generation for a register machine requires the specification of a target register in which the value of a (sub)tree should be computed as well as the set of available registers.
- By enforcing the convention that all registers with a number greater or equal than the target register are available, explicit (set) operations for recording the status of individual registers are avoided. true/false

29. The **weighted register allocation** optimization for simple code generation balances the weights of subtrees by spilling operands of the *heaviest* subtree to memory. true/false

30. When generating code at the basic block level, a **dependency graph** is used instead of an AST.
- The number of nodes in a dependency graph exceeds that of the corresponding AST as the sharing of intermediate values through local variables must be made explicit. true/false

31. **Graph coloring** is a register allocation heuristic that *usually* finds the minimal number of registers needed for a procedure. true/false

32.

```
{  int n = a*b + 1;
   int m = 2*(n+7);

   x = (n+m+a)/3;
}
```

If `a` and `b` are live on entry and dead on exit, and `x` is dead on entry and live on exit, the **register interference graph** associated with the basic block above includes an edge between `a` and `n`. true/false

33. When generating code at the basic block level, the dependency graph must be converted to target code. The **late evaluation** heuristic combines instruction selection and instruction ordering and operates by identifying **ladder sequences** within a basic block.
- It uses temporary registers to store intermediate values shared by ladder sequences originating at different roots true/false

34. Performing **common subexpression elimination** on a dependency graph requires the identification of nodes with the same operator and operands.
- A naive approach checks each of the n nodes against all others, resulting in a linear time $O(n)$ algorithm. true/false

35. An **assembler** transforms a list of symbolic machine instructions into a binary object file. The main complexity is translating the references to symbolic names (labels, function names, etc.) into machine addresses.
- The **relocation table** is part of the object file, and lists all entry points of the *external* symbols. true/false

36. A **linker** combines multiple object files into a single executable object.
- After resolving all cross references, the linker runs a **peephole optimizer** to remove shared addresses from the combined object code. true/false

37. Low-level memory management requires the programmer to explicitly deallocate (free) memory blocks. The basic memory allocator (i.e. without free lists) operates on chunks, which include an administrative part and a block of user data.
- The status (free/in-use) of each block is recorded by a single bit (flag) in the administrative part. true/false

38. To counter memory fragmentation adjacent free chunks should be coalesced into a single chunk. Instead of trying to coalesce a chunk whenever it is released (freed) by the user, malloc() initiates a single coalesce-sweep of the complete heap when it cannot find a chunk of the right size.
- After the sweep, the free space may still be divided over multiple (non-adjacent) chunks. true/false

39. A **reference counting** garbage collector can reclaim cyclic datastructures. true/false

40. To avoid a **mark-and-scan** garbage collector running out of stack space, when recursively marking live data, the **pointer-reversal** technique by Schorr & Waite can be employed.
- The space overhead is limited to a few bits (pointer-count field) per node. true/false

41. A **two-space copying** garbage collector only visits the live data when switching between from- and to-space. true/false

42. When a language supports mutually recursive types, the compiler must be prepared to handle **forward references** to, yet, undeclared type identifiers.
- To handle these forward references all types in a compilation unit are collected in a separate **name space**. true/false

43. Type checking is complicated by **coercions** and **casts**, which convert (user-defined) data from one type into another.
- A cast is an **implicit** type conversion that must be handled by the compiler. true/false

44. There are two important notions of type equivalence: **name equivalence** and **structural equivalence**.

```
VAR a : ARRAY [Integer] OF Integer;
VAR b : ARRAY [Integer] OF Integer;
```

- In a language with name equivalence, variables `a` and `b` have the same type. true/false

45. When translating an object-oriented language to C, the compiler generates a set of routines, one for each method of a class.
- A method with *N* parameters translates into a C routine with *N+1* parameters; the additional parameter points to the object on which the routine is invoked. true/false

46. Pointer supertyping and subtyping is complicated by **multiple inheritance**.

```
class A {           class B {
   field a1;           field b1;
   method m1();        method m2();
}                   }

class C extends A, B {
   field c1;
   method m1();
   method m2();
}
```

- The dispatch table for class `C` includes *four* entries, since the original methods `m1` (defined in `A`) and `m2` (defined in `B`) must be available for supertyping. true/false

47. When handling an object-oriented language, the compiler must maintain a method table for each class. If the language supports **dynamic binding**, the compiler must generate a runtime representation of the method table (i.e. dispatch table) for each class.
- Each object created during the execution of a program contains a pointer to the dispatch table. true/false

48. The **dynamic link** in an activation record of a routine is needed on exit to retrieve the return address, i.e. the location in the caller where computation should be resumed. true/false

49. The administrative part of an activation record contains space to save machine registers when calling another routine.
- In a **callee-saves** scheme, the compiler emits code to save *all* registers at routine entrance. true/false

50. In many programming languages routines may be passed as parameters. If routines may be nested, the compiler cannot simply pass the address of a routine, but must pass a routine descriptor holding the address as well as the static link.
- In addition the activation records of the enclosing routines must be allocated on the stack. true/false

51. When generating code for a case statement, the compiler may decide to use a **jump table** to ensure that the selection at runtime takes constant time.

```
switch (foo) {
   case 's':  return aap;
              break;
   case 'o':  return noot;
              break;
   case 'r':  return mies;
              break;
   case 't':  return teun;
              break;
}
```

- The jump table for the above case statement has 4 entries. true/false

52. When generating code for **boolean control expressions** that direct the flow of control, the compiler can make effective use of (conditional) jumps by emitting code that simply jumps to the "right" basic block identified through true and false labels.
- In the case of a degenerated if statement, i.e. the else part is missing, only one label will be emitted. true/false

53. The functional programming language Haskell includes so-called syntactic sugar like **pattern matching** and **list comprehension**, which raises the abstraction level for the programmer, but complicates the task of the compiler writer.
- List comprehension can be conveniently handled by the lexical analyzer replacing symbols (e.g., `[` and `]`) with calls to higher-order library functions. true/false

54.

```
twice f x = f (f x)
```

The higher-order function `twice` is strict in its second argument (`x`). true/false

55. One of the additional complexities a compiler for a functional language faces is the need to infer polymorphic types of user-defined functions.

```
sum [] = 0
sum (x:xs) = x + sum xs
```

The (inferred) type definition

```
sum :: [Int] -> Int
```

is correct. true/false

56. The implementation of (lazy) functional languages is based on a **graph reducer** that repeatedly

1) finds a redex (reducible expression) with functor f,
2) instantiates the right-hand-side of f, and
3) updates the root of the redex.

- In step 2) the graph reducer may invoke itself recursively to evaluate strict arguments of f. true/false

57. Given the following Prolog relations (clauses)

```
bar(aap).
bar(noot).
bar(mies).
foo(X,Y) :- bar(Y), bar(X).
```

- The query "`?- foo(mies,X).`" will produce the binding "X = mies" as the first answer. true/false

58. An interpreter for logic programs maintains a goal list stack capturing all alternatives that could potentially satisfy the user query.
- All the goals in one list (row) on the stack must be satisfied, before the interpreter can report a result. true/false

59. The default action of the basic, unoptimized Prolog interpreter discussed in the Modern Compiler Design (MCD) book is to attach clauses to the top-most left-most goal on the goal list stack.
- In a program with *N* clauses the interpreter will stack *N* new goal lists. true/false

60. **List procedures** invoke a function, passed as a parameter, for every "computed value". When compiling logic programs to C such list procedures can be used to effectively implement backtracking.
- The computed values are then successful unifications, that is, bindings of logic variables to terms. true/false