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

exam – **Compiler Construction** – in4020
January 15, 2009 14.00 - 15.30

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. The symbol table is used to record information about global as well as local identifiers. **true**

2.

S → `if` C `then` S `else` S

This grammar is ambiguous. **false/true**

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

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

5. A lexical analyzer generated by **lex** is essentially an FSA (Finite State Automaton). **true**

6. 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` → `letter` (• `letter_digit_underscore`)*

- This is a shift item. **true**

7. A lexical analyzer *generator* automatically constructs an FSA that recognizes tokens. The generator is driven by a **regular description**.

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

- The (minimal) FSA for accepting a `number` consists of 4 states. **true**

8. A **bottom-up** parser creates the nodes in the AST (Abstract Syntax Tree) in pre-order. **false**

9. A **predictive parser** is a **top-down** parser. **true**

10. The **yacc** parser generator can handle LR(1) grammars. **false**

11. The stack used in a bottom-up parser contains an alternating sequence of states and terminals (tokens). **false**

12. 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	→	Ab
A	→	Aa | ϵ

- FIRST(S) contains 1 element. **false**

13. Grammars with LL(1) conflicts can be made LL(1) by applying left-factoring, substitution, and left-recursion removal.
- Left-factoring takes care of FIRST/FOLLOW conflicts. **false**

14. Automatically generated top-down parsers are built upon a PDA (Push Down Automaton) driven by a prediction table.
- The number of *rows* in the table is equal to the number of *terminals* in the grammar. **false**

15. SLR(1) parsers only reduce a production rule when the current input token is an element of the FOLLOW set of that rule.

S	→	AB
A	→	ϵ | aA
B	→	b | bB

- FOLLOW(S) contains zero elements. **false**

16. The following set

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

$\texttt{A} \rightarrow \bullet\ \text{a}\ \{\text{x,a}\}$

$\texttt{A} \rightarrow \bullet\ \text{A a}\ \{\text{x,a}\}$

is a valid LR(1) item set **true**

17. In an **attribute grammar** a node N in the AST may be annotated with two kinds of attributes: **inherited** and **synthesized** attributes.
- The parent node of N is responsible for evaluating the values of N's inherited attributes. **true**

18. When dealing with attribute grammars it is important to detect cycles in the evaluation rules to prevent the evaluator to loop endlessly. In the case of **dynamic** cycle detection, the AST is traversed multiple times until either all attributes have obtained a value, or a maximum number of traversals is reached (in which case a cycle can be reported).
- For a grammar with N *production rules*, cycles can be detected dynamically by performing (at most) N evaluation traversals of the AST. **false**

19.

```
Number(SYN value) →
  DigitSeq(base, value) BaseTag(base)
  ATTRIBUTE RULES:
    SET value TO DigitSeq.value;
```

The dependency graph associated with the attribute evaluation rule for `Number` contains just one dependency (arrow). **false**

20. The **late evaluation** partitioning heuristic determines the sets of attributes that should be evaluated in each visit of an ordered attribute grammar.
- It *starts* by determining the attributes that should be evaluated in the *last* visit. **true**

21. In a **threaded AST** the chain of successor nodes is essentially a post-order visit of the AST. **false**

22. **Full symbolic interpretation** processes one basic block at the time. **false**

23. **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 GEN set for this assignment statement must be set to $\{\texttt{i}\}$. **false**

24. **Data-flow equations** can be solved efficiently by using bitwise boolean instructions (AND, OR, etc.). **true**

25. The advantage of using an interpreter over a true compiler is that much better error checking can be performed.
- A **recursive** interpreter maintains a **shadow memory** in which properties of the program data (e.g., type, status) are maintained. **false**

26. A **recursive** interpreter is constructed as a set of routines that implement the semantics of each type of AST node.
- When processing an AST consisting of N nodes, the interpreter will invoke at most N^2 routines. **false**

27. **Simple code generation** considers one AST node at a time. **true**

28. The **weighted register allocation** optimization for simple code generation targeting a register machine evaluates the *heaviest* subtree of an AST node first.
- The weight of a tree is computed as the maximum of the weights of its subtrees. **false**

29. When generating code at the basic block level, a **dependency graph** is used instead of an AST.

```
{  int n;

   n = a+1;
   x = (b+c) * n;
   n = n+1;
   y = (b+c) * n;
}
```

- The dependency graph for the above basic block contains a node for the local variable `n`. **false/true**

30. Performing **common subexpression elimination** on a dependency graph requires the identification of nodes with the same operator and operands. A naive approach checks each node against all others, resulting in a quadratic time algorithm.
- When using a hash table (with a hash function based on operator and operands) all common nodes can be identified in linear time. **true**

31. When generating code at the basic block level, the dependency graph must be converted to target code. By identifying **ladder sequences**, instruction selection and instruction ordering can be performed efficiently in a single pass.
- The number of registers needed for a dependency graph equals the number of ladder sequences (without incoming dependencies) identified during code generation. **false**

32. Register allocation by **graph coloring** uses a register interference graph.
- If a node in the graph is connected to N other nodes, then at at least $N + 1$ registers (colors) are needed. **false**

33.

```
{  double quads = a*a + b*b;
   double cross_prod = 2*a*b;

   x = quads - cross_prod;
   y = quads + cross_prod;
}
```

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

34. A **peephole optimizer** replaces small sequences of symbolic instructions with more efficient sequences.
- Besides improving code quality, a peephole optimizer also reduces the complexity of code generators in *subsequent* stages of the compiler. **false**

35. An **assembler** combines (assembles) multiple files into a single executable object. **false**

36. The relocation information present in an object file includes a list of entry points and unresolved references.
- An entry point specifies the location of a text or data symbol relative to the start of the respective segment in the object file. **true**

37. Low-level memory management requires the programmer to explicitly allocate (malloc) and deallocate (free) memory blocks. The basic memory allocator (i.e. without free lists) operates on chunks, which include an administrative part (status flag + size) and a block of user data.
- Malloc() uses a simple first-fit policy and scans the heap from low to high for a free block that is larger than the requested size; if this fails it runs a procedure to coalesce adjacent free blocks and scans the heap once more. **true/false**

38. An advanced implementation of the `malloc()` routine maintains a table of free lists indexed by ($^2\log$) block size to speed up the allocation time of the basic, linear-search implementation.
- This improvement comes at the price of a slight overhead in memory, since the administrative part at the beginning of a chunk must be enlarged to accommodate a "next" pointer. **false**

39. A system based on **reference counting** is inherently free from memory fragmentation as chunks are freed as soon as they become garbage. **false**

40. A **mark-and-scan** garbage collector can reclaim cyclic datastructures. **true**

41. A **two-space copying** garbage collector copies all live data, reachable from the **root set**, from the 'from space' into the 'to space'.
- With each copy the collector stores a forwarding pointer in the original chunk in from-space. **true**

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

43. 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 **backpatch list**. **false**

44. To handle overloaded identifiers the type checker maintains a scoped symbol table. **false**

45. 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. **true**

46. When handling a dynamically-typed object-oriented language, the compiler must take care of **pointer subtyping** induced by method overriding. **true**

47. Overloaded/overridden identifiers in an object-oriented language influence the layout and size of the objects at runtime.

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

class C extends B {
   field a1;
   method m2();
   method m3();
}
```

- An object of class `C` includes storage for *three* data fields. **true**

48. To handle a routine call, the compiler emits code to allocate a new activation record and fill in various parts. Usually activation records are allocated on the call stack, and parameters are passed by evaluating and stacking the actual arguments in the *reverse* order.
- This reverse ordering is dictated by the x86 architecture featuring a stack that grows "downwards". **false**

49. The administrative part of an activation record contains space to save machine registers when calling another routine.
- In the **callee-saves** scheme, the entry code of the callee saves those registers that may be corrupted by the routine. **true**

50. To efficiently support nested routines a compiler includes a **static link** in each activation record.
- The static link is passed as an additional parameter whenever a routine at level n calls a routine at an outer level m ($m < n$). **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 6 entries. **true**

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.

```
if (i>0 && j>0) { ...  }
```

- The compiler emits two conditional jump instructions, but at most one is taken at runtime. **true**

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.
- For efficiency pattern matching is dealt with by the code generator, which translates the high-level case analysis into a compact jump table. **false**

54.

```
bar x 0 z = x
bar x y z = bar z x (y-1)
```

The function `bar` is strict in its first and second arguments (`x` and `y`). **true**

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

```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

The (inferred) type definition

```
foldr :: (a -> a -> a) -> a -> [a] -> a
```

is correct. **false**

56. An important optimization in a functional language compiler is to **short-circuit** the top-level function call at the right-hand side of an equation.

```
apply f x = f x
```

- The short-circuit optimization saves the construction of one application node per invocation of `apply`. **true**

57. Logic programming is based on named **relations** between terms, **facts** stating such relations, and **rules** for inferring new facts from established facts.

```
grandparent(koen, allard).
```

- This is a fact. **true**

58. Given the following Prolog relations (clauses)

```
bar(aap).
bar(noot).
bar(mies).

foo(X,Y) :- bar(Y), bar(X), X \= Y, !.
```

- The query "`?- foo(mies,X).`" will produce one answer (binding). **true**

59. An interpreter for logic programs maintains a goal list stack capturing all alternatives that could potentially satisfy the user query.
- The action of the interpreter depends on the topmost leftmost goal only **true**

60. When compiling logic programs to C, a concept called **list procedures** can be used to effectively implement backtracking.
- A list procedure receives a function pointer (`goal_tail_list`) as parameter that must be invoked for each "computed value" (successful unification). **true**