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

exam – **Compiler Construction** – in4020
January 20, 2005 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:
- A **true** statement must be marked as **Juist** (**J**); a **false** statement must be marked as **Onjuist** (**O**).
- 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** (naam) and **student-ID** (studentnummer), and to **sign** the form (handtekening).

---

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

2.

S	→	a | B
B	→	Bb | ϵ

The non-terminal B is left recursive. true/false

3. A lexical analyzer transforms a stream of characters into a stream of tokens.
- The tokens are stored in the symbol table for further processing by the parser. true/false

4. A lexical analyzer *generator* automatically constructs a FSA (Finite State Automaton) that recognizes tokens.
- The generator is driven by a CFG true/false

5. The regular expressions **a***|**b*** and **(a**|**b)*** describe the same set of strings. 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 take care of wildcards (i.e., '.') matching any character. true/false

7. Lexical analyzers based on a FSA efficiently match multiple token descriptions concurrently to the input stream. Nevertheless, input characters may be inspected more than once, due to the automaton overshooting the end of the token while looking for a possible longer token
- The complexity (O) of dividing a program text into tokens is *worse than* linear in the length of that text. true/false

8. A **predictive parser** is a **top-down** parser. true/false

9. A **top-down parser** can handle grammars with **left recursive** non-terminals. true/false

10. The **yacc** parser generator can handle LALR(1) grammars. true/false

11. **Yacc** contains built-in support for handling ambiguous grammars resulting in shift-reduce conflicts.
- By default these conflicts are solved by performing the reduce action. true/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	$\rightarrow$	A B
A	$\rightarrow$	ϵ | aA
B	$\rightarrow$	ϵ | bB

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

14. Automatically generated top-down parsers are built upon a PDA (Push Down Automaton) driven by a prediction table.
- a production rule with N alternatives occupies N rows in the table. true/false

15. When constructing an LR(1) parser we record for each item exactly in which context it appears, which resolves many conflicts present in SLR(1) parsers based on FOLLOW sets.
- The look-ahead set associated with an LR(1) item can contain the empty string ϵ. true/false

16.

```
term  →  IDENTIFIER
      |  IDENTIFIER '[' expression ']'
```

This grammar can be made LL(1) by applying **substitution**. true/false

17. In an attribute grammar each production rule ($N \rightarrow \alpha$) has a corresponding **attribute evaluation rule** that describes how to compute the values of the *synthesized* attributes of each particular node N in the AST (Abstract Syntax Tree). true/false

18. The, possibly indirect, dependencies between the inherited and synthesized attributes of a non-terminal N are captured in a single **dependency graph**. 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.
- **L-attributed grammars** can be evaluated by a bottom-up parser. true/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 uses the *IS-SI graph* to compute the partitioning. true/false

21. When **threading** an AST it might be necessary to introduce additional (join) nodes to ensure that each language construct has a single exit point. true/false

22. **Symbolic interpretation** can be carried out both forwards (e.g., constant propagation) and backwards (e.g., live analysis). true/false

23.

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

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

24. **Live analysis** can be determined 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.

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

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

25. An **iterative** interpreter operates on a threaded AST. true/false

26. A **recursive** interpreter is constructed as a set of routines that implement the semantics of each type of AST node.
- collectively, the set of routines implement a depth-first traversal of the AST. true/false

27. Code generation consists of three tasks:

1. instruction selection
2. register allocation
3. instruction ordering

- The last task (instruction ordering) is usually performed by the assembler since it depends on the lay-out of the object file. true/false

28. **Simple code generation** considers one AST node at the 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

29. 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 its subtrees; if two subtrees have equal weight then 1 more register is needed. true/false

30. When generating code at the basic block level, a **dependency graph** is used instead of an AST.
- The edges (dependencies) between the nodes in the graph correspond with the threading pointers in the original AST. true/false

31. **Graph coloring** is a register allocation technique that operates on a complete procedure at a time. true/false

32. Register allocation by graph coloring uses a **register interference graph**.
- A single variable will be represented by multiple nodes in the graph if it is used to store multiple values with different live ranges. 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.

```
{  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 `a` and `cross_prod`. 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 captures all unsolved references to external symbols. true/false

36. An **assembler** can be considered a small compiler since it transforms a source language (assembly) into a less abstract target language (binary object code).
- A one-pass assembler (narrow compiler) needs a **backpatch list** to handle forward references (to labels, functions, etc.). true/false

37. Low-level memory management requires the programmer to explicitly deallocate (free) memory blocks. For this purpose the memory allocator operates on chunks, which include some administrative part and a block of user data.
- The administrative part includes a pointer to the next free chunk. true/false

38. In general programs (users) operate on a restricted set of data types, hence, most dynamically allocated memory blocks are of the same size. Therefore, an advanced implementation of the `malloc()/free()` routines maintains a table of free lists indexed by ($^2\log$) block size.
- The advantage is that adjacent blocks can be coalesced efficiently in constant time. true/false

39. A **reference counting** garbage collector is unable to reclaim cyclic datastructures. true/false

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

41. When supporting garbage collection, a compiler may assist by generating self-descriptive chunks that indicate if and where pointers are present in user-defined data structure.
- the space overhead in each data structure is linear in the number of pointer fields in it. true/false

42. A straightforward way to handle multiple names spaces (e.q., type names, variable/procedure names) is to maintain a separate symbol table for each name space. true/false

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

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

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

44. The process of type checking is complicated by the dual use of a variable name: it can be used to denote a value (rvalue) or a location (lvalue). It is useful to make this difference explicit.
- The type checker then treats all variable names as locations and inserts a *dereference* coercion whenever a value is needed. true/false

45. To support inheritance in an object-oriented language, the compiler must generate code that includes a pointer to a method table in each object. true/false

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

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

class B extends A {
   field a1;
   field a2;
   method m1();
   method m2();
}
```

- An object of class `B` includes a pointer to a method table holding *three* entries. true/false

47. 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

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 convenient for handling function and procedure calls in the same way (even though procedures do not return a value). true/false

49. The **dynamic link** in an activation record of a nested routine is needed to reference variables defined in enclosing (outer) scopes. true/false

50. The implementation of a *lazy* functional programming language requires that all activation records are allocated on the heap. true/false

51. 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 runtime evaluation of the condition in the above if-statement results in at most 1 jump. true/false

52. When generating code for a case statement, the compiler may decide to use a **jump table** to ensure that the selection during program execution takes constant time.
- The jump table holds the text addresses of all labeled statements. true/false

53. The **offside-rule** of the functional programming language Haskell defines the (implicit) ending of syntactic constructs like equations and where clauses (local definitions).
- The offside-rule can be conveniently handled by the lexical analyzer inserting explicit markers into the token stream. true/false

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

```
append [] ys = ys
append (x:xs) ys = x :  append xs ys
```

The (inferred) type definition

```
append :: [a] -> [a] -> [a]
```

is correct. true/false

55.

```
take 0 xs = []
take n [] = []
take n (x:xs) = x : take (n-1) xs
```

The list-processing function `take` is strict in its first argument (`n`) only. true/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.

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

- The short-circuit optimization saves the construction of two application nodes per invocation of `twice`. true/false

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

```
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
```

- This is a rule. true/false

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 two answers (bindings). 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. The compilation of Prolog to C discussed in the MCD book makes use of **list procedures** and **nested functions**.

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

- A rule with a body consisting of 3 goals translates into a C function with 3 nested functions, one for each goal. true/false