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

exam – **Compiler Construction** – in4020
January 18, 2007 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. The language generated by a CFG is unambiguous. true/false

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

4. For convenience lexical analyzers should read the complete input program into memory. 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**.

```
letter          →  [a-zA-Z]
digit           →  [0-9]
letter_or_digit →  letter | digit
identifier      →  letter letter_or_digit*
```

- The (minimal) FSA for accepting an `identifier` consists of 2 states. true/false

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**.

```
fixed point → ([0-9])* • '.'([0-9])+
```

- This is a reduce item. true/false

7. 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

8. A **top-down** parser creates the nodes in the AST (Abstract Syntax Tree) in post-order. true/false

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

10. A **recursive descent** parser consists of a set of (recursive) routines, each routine corresponding closely to a rule in the grammar.
- An empty production (N $\rightarrow \epsilon$) 'translates' to a routine returning true without inspecting any token. true/false

11. 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 non-terminals in the grammar. true/false

12.

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

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

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

```
S  →  A B
A  →  a | aA
B  →  ϵ | bB
```

- FOLLOW(A) contains 2 elements. true/false

14. The following two items

A → x • B

B → • y

can coexist in an LR item set. true/false

15. The stack used in a bottom-up parser contains an alternating sequence of states and grammar symbols (terminals and non-terminals).
- The depth of the stack is at most two times the number of non-terminals in the grammar. true/false

16. The LR(1) parsing technique reduces a handle (the righthand side of a production N $\rightarrow \alpha$) when the current input token is an element of FIRST(N). 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 child nodes of N are responsible for evaluating the values of N's inherited attributes. true/false

18. The **dependency graph** associated with the attribute evaluation rule of some production rule ($N \rightarrow \alpha$) captures how values flow from one attribute to another, hence, which attribute depends on which others.
- If N has i inherited and s synthesized attributes, the maximum number of dependencies is $i \times s$. true/false

19. The **IS-SI graph** associated with a non-terminal N captures the (indirect) dependencies between the inherited and synthesized attributes of N. true/false

20. 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

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. **Full symbolic interpretation** performs at least two traversals of the AST. true/false

23.

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

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

24. **Live analysis** can be 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 KILL set for this assignment statement must be set to $\{\texttt{i}\}$. true/false

25. An **iterative** interpreter is faster than a **recursive** interpreter.
- The advantage of a recursive interpreter is that it supports a larger class of grammars. true/false

26. To handle user-defined datatypes a **recursive** interpreter operates with self identifying data. Each data value is represented as a pointer to a type descriptor and (a set of) sub values.
- The recursive interpreter stores the type descriptors in a so-called **shadow memory**. true/false

27. Code generation consists of three tasks:

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

- When targetting modern, super-scalar processors the second task becomes void because of the multi-ported register file implemented in such processors. true/false

28. **Simple code generation** considers one AST node at the time.
- When the target is a *stack* machine, the compiler emits one instruction per AST node that will leave "its" value on top of the stack at execution time of the compiled program. true/false

29. 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

30. When a register allocator runs out of registers, it can free a register by temporarily storing the value in memory.
- To support recursive functions, this memory must be allocated in an activation record at runtime. true/false

31. Register allocation by graph coloring uses a **register interference graph**.
- Two nodes in the graph are joined by an edge when the live ranges of the values they represent overlap. 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 `x`. true/false

33. 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. true/false

34. A **peephole optimizer** replaces small sequences of symbolic instructions with more efficient sequences.
- Besides improving code quality, a peephole optimizer allows for simpler code generators in earlier stages of the compiler. 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 *internal* symbols. true/false

36. A **linker** combines multiple object files into a single executable object.
- For each object file the linker creates a relocation table capturing all resolved references. 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. 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 reclaims nodes as soon as they become garbage (except for cyclic structures). true/false

40. In the first phase of a **mark-and-scan** garbage collector all nodes reachable from the root set are marked as live.
- The root set contains pointers to heap objects stored in global data as well as in activation records on the stack. 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. To handle multiple names spaces (e.q., type names, variable/procedure names) the type checker maintains a scoped symbol table. 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 name 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. 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

46.

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

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

- When translating a call of method `m2` on an object `c` of type `C`, the compiler must do pointer supertyping as in `(*(c->disp_table[1]))(C2B(c))` to convert the self-pointer to the appropriate type. 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. The **static link** in an activation record of a nested routine is needed to reference variables defined in enclosing (outer) scopes.
- If a routine at level n refers to a variable at level 1, the compiler emits code to follow $n-1$ static links regardless of the number of activation records on the stack at runtime. true/false

49. The administrative part of an activation record contains space to save machine registers when calling another routine.
- In a **caller-saves** scheme, the compiler emits code to save all registers holding live values before issuing a routine call. true/false

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

51. Field alignment requirements may cause the compiler to insert gaps in a record representation.
- The equality comparison operator can be translated into an efficient n-byte string comparison with n set to the length of the (extended) record. true/false

52. 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 1:  return bar;
            break;
   case 3:  return bar*2;
            break;
   case 5:  return bar+3;
            break;
   default: return 0;
            break;
}
```

- The jump table for the above case statement has 4 entries. 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.
- Pattern matching is dealt with by a source-to-source translation in the front-end of the compiler and does not require explicit handling in the back-end. 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.

```
zip [] ys = []
zip xs [] = []
zip (x:xs) (y:ys) = [x,y] : zip xs ys
```

The (inferred) type definition

```
zip :: [a] -> [a] -> [[a]]
```

is correct. 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.

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

- The short-circuit optimization saves the construction of two application nodes per invocation of `sum`'s second equation. true/false

57. Given the following Prolog relations (clauses)

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

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

- The query "`?- foo(aap,B).`" will produce the binding "B = noot" 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.
- The interpreter always operates on the topmost goal list. 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 *M* facts and *N* rules the interpreter will stack *N* new goal lists. true/false

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/false