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

exam – **Compiler Construction** – in4020
January 19, 2006 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.

```
S  →  a S | Sa | c
```

This grammar is ambiguous. true/false

2. A lexical analyzer transforms a stream of characters into a stream of tokens. true/false

3. For convenience lexical analyzers should read the complete input program into memory. true/false

4. The regular expressions **a+|b+** and **(a|b)+** describe the same set of strings. 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 handling an `identifier` includes 2 *accepting* 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.
- character moves handle basic patterns. 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**.

`fixed point` $\rightarrow$ `([0-9])* '.'(• [0-9])+`

- This is a shift item. true/false

8. An **LR** parser inspects the tokens in the input stream from right to left. true/false

9. A **top-down** parser can handle any LR(1) grammar. true/false

10. A parser generated by **yacc** is essentially a PDA (Push Down Automaton). true/false

11. A **bottom-up parser** can handle grammars with **left recursive** non-terminals. true/false

12. A **recursive descent** parser consists of a set of (recursive) routines, each routine corresponding closely to a rule in the grammar.
- when a routine returns true (i.e. it successfully matched a production) it has consumed at least 1 token from the input stream. true/false

13.

S	$\rightarrow$	A | x
A	$\rightarrow$	aAb | x

This grammar contains a reduce-reduce conflict. true/false

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

S	$\rightarrow$	A B
A	$\rightarrow$	ϵ | aA
B	$\rightarrow$	b | bB

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

16. The following two items

$A \rightarrow P \bullet x$

$B \rightarrow Q \bullet x$

can coexist in an LR item set. true/false

17. In an **attribute grammar** a node N in the AST (Abstract Syntax Tree) 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 synthesized attributes. 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 **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.
- **L-attributed grammars** can be evaluated in a *single* visit by a top-down parser. true/false

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

22. **Full symbolic interpretation** performs at least one traversal of the 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 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 = v + 1;
```

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

27. Code generation consists of three tasks: instruction selection, register allocation, and instruction ordering.
- For generating "optimal" code these three tasks must be considered at once, because they dependend on each other. 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. The **weighted register allocation** optimization for simple code generation targeting a register machine evaluates the *heaviest* subtree of an AST node first. true/false

30. When a register allocator runs out of registers, it can free a register by temporarily storing the value in memory.
- This technique is known as **register spilling**. true/false

31. 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;
}
```

- During the construction of a dependency graph all local variables in the basic block (e.g., `n`) are removed from the original AST. true/false

32. 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

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 `a` and `quads`. 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, a register allocator based on **graph coloring** will determine that 3 registers are needed for the basic block above. 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 includes entry points of the internal 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. For this purpose the memory allocator operates on chunks, which include some administrative part and a block of user data.
- The administrative part includes the size of the chunk to be able to identify the beginning of the next adjacent chunk, for example, when scanning the heap for a free chunk. 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. true/false

39. To reduce the memory overhead of a **two-space copying** garbage collector the **pointer-reversal** technique by Schorr & Waite can be employed to embed the call stack inside the nodes' data. 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.
- When marking a node n, the garbage collector *first* sets the live flag(bit) of n, and *then* calls itself repeatedly to mark the child nodes. true/false

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 remembers the original location in the from-space by storing a "source" pointer in the new chunk in to-space. true/false

42. Type checking is complicated by **coercions** and **casts**, which convert (user-defined) data from one type into another.
- A coercion is an **explicit** type conversion specified by the programmer. true/false

43. In a language that supports **overloading** an identifier may refer to different (function) types depending on the context.

```
function foo() : int   is return 11;
function foo() : real   is return 13.0;

function bar() : real   is return foo() / 2;
```

- The type checker must be prepared to operate on sets (of types). true/false

44. 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 **type table**. true/false

45. 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 storage for just *two* data fields. true/false

46. To handle scope-resolution operators in an object-oriented language (e.g., `::` in C++), the compiler includes a static link in each object. 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.
- Whenever an object binds to a new type the pointer to its dispatch table is updated accordingly. true/false

48. The **static link** in an activation record of a routine is needed on exit to restore the frame pointer to that of the caller of the routine. 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 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 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 5 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.

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

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

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

```
map f [] = []
map f (x:xs) = f x :  map f xs
```

The (inferred) type definition

```
map :: a -> b -> [a] -> [b]
```

is correct. true/false

54.

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

The function `foo` is strict in its first argument (`x`) only. true/false

55. Strictness analysis is important for generating efficient code.

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

- Although function `take` is strict in its first argument, the expression `n-1` may not be evaluated immediately since the list constructor (`:`) is lazy in its tail argument. true/false

56. 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. 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. 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 *M* new goal lists. true/false

59. The implementation of logic programming is based on the concept of unification. For efficiency the unification process uses **backtracking** to undo bindings of logic variables.
- Backtracking avoids the creation of a fresh copy of a program clause on each unification. 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