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

exam – **Compiler Construction** – in4020
January 8, 2004 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. A parser transforms a stream of characters into a stream of tokens. **false**

2.

```
S  →  a | A
A  →  Aa | a
```

This grammar is ambiguous. **true**

3. The regular expressions **(a*|b)?** and **a*|b?** describe the same set of strings. **true**

4. In a hand-written lexical analyzer finding the end of a token is straightforward since tokens are separated by white space (spaces, tabs, etc.). **false**

5. A lexical analyzer generated by **lex** is essentially a recursive-descent scanner **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**.

```
integer → ([0-9])+ •
```

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

7. A lexical analyzer *generator* automatically constructs a FSA (Finite State Automaton) that recognizes tokens. The generator is driven by a **regular description**.
- A description with n tokens results in a FSA with n accepting states. **false**

8. 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 consuming the current input token and returning true regardless of its (token) value. **false**

9. A **predictive parser** is a **bottom-up** parser. **false**

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

11. The actions (shift, reduce) in an LR(0) parser depend on a lookahead symbol (current input token). **false**

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

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

14. The following two items

$$\mathsf{A} \rightarrow \mathsf{P} \bullet \mathsf{Q}$$

$$\mathsf{B} \rightarrow \mathsf{P} \bullet \mathsf{Q}$$

can coexist in an LR item set. **true**

15. **Yacc** contains built-in support for handling ambiguous grammars resulting in shift-reduce conflicts.
- By default these conflicts are solved by performing the shift action. **true**

16. The error handling mechanism of the **LLgen** parser generator pushes the input stream back when inserting 'missing' tokens. **true**

17. 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**

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.
- An *inherited* attribute can depend on another *inherited* attribute. **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 3 dependencies (arrows). **false**

20. The evaluation of attribute grammars is a complex task. The most general approach is to perform **multiple visits** over the AST (Abstract Syntax Tree). 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**

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

22. **Simple symbolic interpretation** can be used to perform **constant propagation** on a threaded AST. That is, each use of a variable can be annotated with an item of the information set {UNKNOWN, <the value>, ANY}. **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**

24. **Data-flow equations** can be solved efficiently by combining the effects of subsequent statements inside a basic block. **true**

25. The advantage of using an interpreter over a true compiler is that much better error checking can be performed, but at the expense of slower execution.
- A **recursive** interpreter is faster than an **iterative** interpreter. **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**. **false**

27. In general code generation consists of three tasks:

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

- When targeting a *stack-machine*, however, register allocation becomes void, which simplifies code generation considerably. **true**

28. **Simple code generation** considers one basic block at the time. **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**

30. The **weighted register allocation** optimization for simple code generation targeting a register machine evaluates the *lightest* subtree of an AST node first. **false**

31. When generating code at the basic block level, a **dependency graph** is used instead of an AST.
- The advantage of using the dependency graph is that it captures only the essential (data) dependencies between values (variables, subexpressions, etc.). **true**

32. 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 are disjoint. **false**

33.

```
{
   y = a*a + b*b;
   b = 2*a*b;
   x = y + b;
   y = y - b;
}
```

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**

34. 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**

35. 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 earlier stages of the compiler. **true**

36. An **assembler** combines (assembles) multiple files into a single executable object. **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 is located after the user data at the end of the chunk. **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 all ‘free’ memory is available as one big chunk. **false**

39. A **reference counting** garbage collector reclaims nodes as soon as they become garbage (except for cyclic structures). **true**

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**

41. A **two-space copying** garbage collector copies all live data, reachable from the **root set**, from the ‘from space’ into the ‘to space’.
- The compiler assists the collector by updating the root set at every pointer manipulation in the user program. **false**

42. To handle multiple names spaces (e.q., type names, variable/procedure names) the type checker maintains a scoped symbol table. **false**

43. 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 structural equivalence, variables `a` and `b` have the same type. **true**

44. A variable name (identifier) can be used in two ways: 1) to denote a value (rvalue), and 2) to denote a location (lvalue). It is the type checker's task to determine which use is allowed in which context.
- The type checker may insert coercions from rvalue to lvalue when processing the AST. **false**

45. Scope-resolution operators in an object-oriented language (e.g., `::` in C++) are handled at compile time and do not depend on information available only at runtime (like pointers in an object). **true**

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

47. Overloaded 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 two entries (`B.m1` and `B.m2`), and storage for three fields (`A.a1`, `B.a1` and `B.a2`). **true**

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

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 holding live values before issuing a routine call. **false**

50. The compiler handles routine invocations by generating code to allocate a new activation record.
- If *nested* routines may be passed as parameters (but not returned as values) the activation records cannot be allocated on the call stack, but must be allocated on the heap instead. **false**

51. Field alignment requirements may cause the compiler to insert gaps in a record representations.
- The equality comparison operator can be translated into an efficient n-byte string comparison with n set to the length of the (extended) record. **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 least 1 jump. **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.

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

The (inferred) type definition

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

is correct. **false**

54.

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

The function `foo` is strict in its first and third arguments (`x` and `z`). **false**

55. The implementation of *lazy* functional languages is based on representing a (delayed) computation as a graph of some nodes.

```
ones = 1 : ones
```

- The compiler translates the infinite list of ones into a single graph node. **true**

56. Strictness analysis is important for generating efficient code.

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

- Since function `take` is strict in its first argument, the expression `n-1` can be evaluated immediately. **false**

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

```
parent(koen, allard).
```

- This is a rule. **false**

58. The implementation of logic programming is based on the concept of unification, which binds logic variables to terms.
- In a program with n facts and m rules, a logic variable can be successfully unified at most n times. **false**

59. When compiling logic programs to C, a concept called **list procedures** can be used to effectively implement backtracking. List functions invoke a function, passed as a parameter, for every computed value.
- In this approach, the basic unification function is a list procedure too. **true**

60. The key to fast execution of logic programs is the efficient implementation of unification through backtracking. The Warren Abstract Machine (WAM) is a very popular intermediate code for Prolog compilers.
- The WAM contains specialized functions for handling (unifying) the *first* argument of a program clause. **true**