Faculty of Information Technology and Systems
Delft University of Technology

exam – **Compiler Construction** – in4020
June 26, 2003 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 narrow compiler is a one-pass compiler. **false**

2. A lexical analyzer transforms a stream of tokens into an AST (Abstract Syntax Tree). **false**

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

4. The transition table in a lexical analyzer records for each state (row) which token, if any, is recognized in that state.
- For each token there is exactly one "recognizing" row in the table. **false**

5. 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` $\rightarrow$ `(•[0-9])+`

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

6. Lexical analyzers based on a FSA (Finite State Automaton) efficiently match multiple token descriptions concurrently to the input stream.
- Each input character is inspected only once. **false**

7. A lexical analyzer generated by **lex** is essentially a FSA. **true**

8. A **top-down** parser creates the nodes in the AST in post-order. **false**

9. The actions (shift, reduce) in a SLR(1) parser depend on a lookahead symbol (current input token). **true**

10. An LR(0) bottom-up parser uses a combined GOTO and ACTION table. **false**

11. Automatically generated top-down parsers are built upon a PDA (Push Down Automaton) driven by a prediction table.
- An empty entry in the table signals a syntax error. **true**

12. A grammar with a FIRST/FIRST conflict can be made LL(1) by *only* applying **left factoring**, that is, no substitution and left-recursion removal are needed. **false**

13. The following two items

$\mathsf{A} \rightarrow \bullet\ \mathrm{x}$

$\mathsf{B} \rightarrow \mathrm{x}\ \bullet$

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

14.

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

This grammar contains a shift-reduce conflict. **true**

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. **false**

16. 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	$\rightarrow$	b | bB

- FIRST(S) contains 3 elements. **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 *inherited* attributes of each particular node N in the AST. **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 4 dependencies (arrows). **false**

19. The **IS-SI graph** associated with the attribute evaluation rule for some production rule ($N \rightarrow \alpha$) captures the (indirect) dependencies between the inherited and synthesized attributes of N. **true**

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 *dependency graph* to compute the partitioning. **true**

21. **Simple symbolic interpretation** performs a pre-order traversal of the AST. **false**

22. **Data-flow equations** can be solved both forwards (e.g., constant propagation) and backwards (e.g., live analysis). **true**

23. The (context) information that can be determined with **full symbolic interpretation** can be obtained with a single visit of the (threaded) AST. **false**

24. **Symbolic interpretation** operates with IN, OUT, GEN, and KILL sets. **false**

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.
- An **iterative** interpreter is (much) faster than a **recursive** interpreter. **true**

26. 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**

27. Code generation consists of three tasks:

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

- These three tasks may be carried out independently without compromising the quality of the generated code. **false**

28. **Simple code generation** considers one AST node at the time.
- When the target is a *stack* machine, the code can be generated in one (depth-first) traversal of the AST. **true**

29. The **weighted register allocation** optimization for simple code generation targeting a register machine evaluates the *heaviest* subtree of an AST node first. **true**

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**

31. **Graph coloring** is a register allocation technique that *always* finds the minimal number of registers needed for a procedure. **false**

32.

```
{  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 4 registers are needed for the basic block above. **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.
- If the target machine supports arithmetic operations with one operand in memory (e.g., `Add_Mem a, Rb`), then the code for a ladder sequence requires only one register accumulating the result. **true**

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** 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 lists all entry points of the *external* symbols. **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 *backward* references (to labels, functions, etc.). **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 *one* flag for marking the chunk as free/in-use. **true**

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 blocks can be *freed* efficiently compared to using a single list. **false**

39. A **reference counting** garbage collector can reclaim cyclic datastructures. **false**

40. 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 identifying the root set. **true**

41. A **mark-and-scan** garbage collector visits all chunks *twice* (per run) when reclaiming free chunks. **false**

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

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**

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**.
- Entries in the type table may refer to the symbol table, and vice versa. **true**

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**

46. When handling an object-oriented language, the compiler must take care of **pointer supertyping** (to handle polymorphism) and **pointer subtyping** (to handle dynamic binding).

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

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

- If the language supports **multiple inheritance**, the effects of supertyping and subtyping is that of a type conversion only; no actual (generated) code is involved. **false**

47. Compared to a traditional imperative programming languages, the presence of classes in an object-oriented language complicates the task of the compiler since an additional scope mechanism must be handled.
- The resolution of names in the presence of inheritance requires the compiler to know the *dynamic* type of each object reference. **false**

48. For a programming language that supports nested routines the activation records generated by the compiler contain both a dynamic and a static link. **true**

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**

50. 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 routines with a variable number of arguments. **true**

51. 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.
- A disadvantage of a jump table is its space overhead when the number of case entries is small and the range (or reach) is large. **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.
- Irrespective of the complexity of the boolean expression, the compiler has to generate at most two new labels. **true**

53.

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

The higher-order function `twice` is strict in its first argument (`f`). **true**

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

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

The (inferred) type definition

```
sum ::  [a] -> Int
```

is correct. **false**

55. The implementation of (lazy) functional languages is based on **graph reduction**. The high-level concept of function application is explicitly represented in the form of binary application nodes (@) with a function and argument pointer.
- This explicit representation eases the support of higher-order functions and currying. **true**

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.

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

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

57. 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 one new *logic variable* (copy) on each unification. **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 leftmost goal. **true**

59. 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**

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 *logic variable* of a program clause. **false**