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

exam – **Compiler Construction** – in4020
July 6, 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. A parser transforms a stream of characters into a stream of tokens. true/false

2.

```
S  →  a S | Sb | c
```

This grammar is ambiguous. true/false

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

4. 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 just one state. true/false

5. 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 do not consume any input character, but merely expand non-basic items with repetition and composition operators. 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**.

`integer` $\rightarrow$ `([0-9])+` $\bullet$

- This is a shift item. true/false

7. Lexical analyzers based on an 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 **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. true/false

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

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

11. The stack used in a bottom-up parser contains an alternating sequence of terminal and non-terminal symbols. true/false

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

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/false

14.

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

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

15. 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 FOLLOW(N). true/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 2 elements. 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 *inherited* attributes of each particular node N in the AST. 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.
- An *inherited* attribute can depend on a *synthesized* attribute. true/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). 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. **Simple symbolic interpretation** performs a single traversal of the AST. true/false

22. **Full 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}. 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. **Data-flow equations** can be solved efficiently by combining the effects of subsequent statements inside a basic block. true/false

25. 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. 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: 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.
- If the target is a *register* machine, the code can be generated in one traversal of the AST, possibly introducing temporaries when running out of registers. true/false

29. 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/false

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

31. Register allocation by **graph coloring** uses a register interference graph.
- If a node in the graph is connected to N other nodes, then at at least $N + 1$ registers (colors) are needed. true/false

32.

```
{
   x = a*a - 2*a*b + b*b;
   y = a*a + 2*a*b + b*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/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/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** 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.). true/false

36. A **linker** combines multiple object files into a single executable object. 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 conversion from block-pointer to chunk-pointer depends on the chunk size stored in the administrative part. true/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, the free space may still be divided over multiple (non-adjacent) chunks. true/false

39. A **mark-and-scan** garbage collector can reclaim cyclic datastructures. true/false

40. To avoid a **two-space copying** garbage collector running out of stack space, when recursively marking live data, the **pointer-reversal** technique by Schorr & Waite can be employed. true/false

41. A **mark-and-scan** garbage collector visits all chunks only *once* (per run) when reclaiming free chunks. 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. 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 name equivalence, variables `a` and `b` have the same type. true/false

44. To handle overloaded identifiers the type checker maintains a scoped symbol table. true/false

45. 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 {
   field a1;
   method m1();
}

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

- If the language supports only **single inheritance**, the effects of supertyping and subtyping is that of a type cast only. true/false

46. 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/false

47. 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

48. When executing code on a stack machine, the size of an activation record is fixed during the lifetime of a routine invocation. 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 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. true/false

51. When generating code for a while statement, the compiler may use the following naive scheme

```
test_label:
  Boolean control code( condition, No_label, end_label)
  Code statement( statement)
  GOTO test_label;
end_label:
```

- An optimized scheme can save one (conditional) jump instruction per loop iteration. 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.

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

The (inferred) type definition

```
twice :: (a -> b) -> a -> b
```

is correct. true/false

54.

```
apply f x = f x
```

The higher-order function `apply` is strict in its second argument (`x`). true/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 pattern matching. true/false

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. true/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 a fresh copy of a program clause on each unification. true/false

58. 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 = aap" as the first answer. true/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/false

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. true/false