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

exam – **Compiler Construction** – in4020
June 30, 2005 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. The symbol table is used to pass tokens between the lexical analyzer and the parser in a compiler pipeline. true/false

2.

```
S  →  a S | Sb | c
```

This grammar is ambiguous. true/false

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

4. 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 shift item. 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 FSA for accepting an `identifier` consists of just one state. true/false

6. Lexical analyzers based on an FSA efficiently match multiple token descriptions concurrently to the input stream.
- Each input character is inspected only once. true/false

7. The FSA constructed by a lexical analyzer generator is usually represented by a **transition table**.
- To reduce storage requirements, the transition table can be compressed by a technique known as **row displacement**. true/false

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

9. An SLR(1) bottom-up parser uses a combined GOTO and ACTION table. true/false

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

11.

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

This grammar can be made LL(1) by applying **left-recursion removal**. 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. The stack used in a bottom-up parser contains an alternating sequence of states and terminal symbols. true/false

14. The following two items

$A \rightarrow P \bullet x$

$B \rightarrow P\ x \bullet$

can coexist in an LR item set. 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 at least one element. true/false

16.

```
S  →  A | x
A  →  aAb | x
```

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

17. Attribute grammars are specifically designed to handle the complex semantics of modern programming languages like functional and logic languages. 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.
- A *synthesized* attribute can depend on an *inherited* attribute. true/false

19. 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 **S-attributed grammars**, the problem of cycle detection becomes irrelevant because they simply cannot occur. 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 only considers *synthesized* attributes, since the *inherited* attributes can always be evaluated in the first visit. true/false

21. **Simple symbolic interpretation** performs a post-order 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.

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

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 {v}. true/false

25. An **iterative** interpreter maintains a **shadow memory** in which properties of the program data (e.g., type, status) are maintained. true/false

26. A **recursive** interpreter is constructed as a set of routines that implement the semantics of each type of AST node.
- when interpreting a single expression from a user program, a routine may be invoked multiple times. true/false

27. **Simple code generation** considers one basic block at the time. true/false

28. The **weighted register allocation** optimization for simple code generation targeting a register machine evaluates the *lightest* subtree of an AST node first too free up registers for the evaluation of the other subtrees. true/false

29. **Graph coloring** is a register allocation heuristic that *usually* finds the minimal number of registers needed for a procedure. true/false

30.

```
{  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 `x` and `cross_prod`. true/false

31.

```
{
   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

32. 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

33. 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

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 includes all entry points of the internal symbols. true/false

36. A **linker** combines multiple object files into a single executable object.
- for each object file, it needs to process both the internal and the external references. 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. 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.
- When returning a block to its specific free list, the `Chunk pointer.free` flag is set to `true`. true/false

39.

```
VAR p, q:  pointer;
...
p := q;
```

A compiler implementing garbage collection based on **reference counting** must generate code that first decrements the count associated with the object pointed to by `p`, then increments the count associated with `q`, and finally sets `p` to `q`. 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. 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.
- by grouping all pointer fields, the administration overhead is limited to a single counter, that is, the space overhead is constant. true/false

42. There are two important notions of type equivalence: **name equivalence** and **structural equivalence**.

```
TYPE node = RECORD
              key: INTEGER;
              left, right: POINTER TO node;
            END;
VAR root : node;
VAR tmp : node;
```

- In a language with structural equivalence, variables `root` and `tmp` have the same type. true/false

43. Type checking is complicated by **coercions** and **casts**, which convert (user-defined) data from one type into another.
- A cast is an **implicit** type conversion that must be handled by the compiler. 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 values and inserts a reference coercion whenever a location is needed. 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 a pointer to a method table holding *two* entries. 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. 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+2* parameters. One additional parameter (`this`) points to the object on which the routine is invoked; the other (`parent`) points to the object – if any – it inherits from. true/false

48. The administrative part of an activation record contains space to save machine registers when calling another routine.
- In the **callee-saves** scheme, the entry code of the callee saves those registers that may be corrupted by the routine. true/false

49. In some programming languages routines may be returned as values. If routines may be nested, the compiler may not simply return the address of a routine, but must return 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

50. For a programming language that *forbids* the use of nested routines the activation records generated by the compiler contain both a dynamic and a static link. true/false

51. When generating code for a case statement, the compiler may decide to use a *symbol table* to ensure that the selection during program execution takes constant time. 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 -> a) -> a -> a
```

is correct. true/false

54. 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.
- List comprehension 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

55.

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

The list-processing function `take` is strict in both its arguments. 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.

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

- The short-circuit optimization saves the construction of one application node per invocation of `apply`. true/false

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

```
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
```

- This is a fact. 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. 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 $n * m$ times. true/false

60. The compilation of Prolog to C discussed in the Modern Compiler Design (MCD) book makes use of **list procedures** and **nested functions**.

```
foo(X,Y) :- bar(Y), bar(X), X \= Y.
```

- A rule with a body consisting of N goals translates into a C function with $N - 1$ nested functions handling subgoals $2 \cdots N$. true/false