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

exam – **Compiler Construction** – in4303
April 4, 2011 14.00 - 15.30

This exam (7 pages) consists of 52 True/False questions.
Your score will be computed as: max(0, #correct - 26).
It is **not** allowed to consult the book, handouts, or any other notes.

Instructions for filling in the answer sheet:
- 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** and **Student Number**, and to **sign** the form.

1.

```
S  →  a | B
B  →  Bb | b
```

The non-terminal `B` is right recursive. true/false

2. The symbol table is used to record information about global identifiers; the information about local identifiers is stored in the AST (Abstract Syntax Tree). true/false

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

4. In a hand-written lexical analyzer the so-called **dot** marks the beginning of the next token in the input buffer. true/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**.

`identifier → letter (letter_digit_underscore)* •`

- This is a shift item. true/false

6. A lexical analyzer *generator* automatically constructs an FSA (Finite State Automaton) that recognizes tokens. The generator is driven by a **regular description**.

```
dot          →  [.]
digit        →  [0-9]
integer      →  digit+
fixed_point  →  integer?  dot integer
number       →  integer | fixed_point
```

- The (minimal) FSA for accepting a `number` includes 2 *accepting* states. true/false

7. 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

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

9. The **yacc** parser generator can handle LALR(1) grammars. true/false

10.

```
S  →  A | x
A  →  aAb | x
```

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

11. **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/false

12.

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

This grammar can be made LL(1) by applying **substitution**. true/false

13. 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  →  AB
A  →  ε | aA
B  →  ε | bB
```

- FIRST(`S`) contains 3 elements. 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  →  AB
A  →  ϵ | aA
B  →  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 can contain the empty string ϵ. true/false

16. The following set

$\texttt{S} \rightarrow \bullet\ \texttt{A}\ \text{x}\ \{\$\}$

$\texttt{A} \rightarrow \bullet\ \text{a}\ \{\$\}$

$\texttt{A} \rightarrow \bullet\ \text{a}\ \text{A}\ \text{a}\ \{\$\}$

is a valid LR(1) item set true/false

17. In an **attribute grammar** a node N in the AST may be annotated with two kinds of attributes: **inherited** and **synthesized** attributes.
- The synthesized attributes are, by convention, drawn on the left of 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 *starts* by determining the attributes that should be evaluated in the *last* visit. true/false

21. When **threading** an AST it might be necessary to introduce additional (fork) nodes to ensure that each language construct has a single entry point. true/false

22. **Full symbolic interpretation** performs at least one traversal of the AST. true/false

23.

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

Forward data-flow equations can be solved by a basic closure-algorithm using the above inference rules. true/false

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

25. 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

26. The advantage of using an interpreter over a true compiler is that much better error checking can be performed.
- An **iterative** interpreter maintains a **shadow memory** in which properties of the program data (e.g., type, status) are maintained. 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 depend on each other. true/false

28. **Simple code generation** considers one AST node at a 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. Performing **common subexpression elimination** on a dependency graph requires the identification of nodes with the same operator and operands. A naive approach checks each node against all others, resulting in a quadratic time algorithm.
- When using a hash table (with a hash function based on operator and operands) all common nodes can be identified in linear time. true/false

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

31. 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

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

33. 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. true/false

34.

```
{
   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 4 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. true/false

37. Low-level memory management requires the programmer to explicitly allocate (malloc) and deallocate (free) memory blocks. The basic memory allocator (i.e. without free lists) operates on chunks, which include an administrative part (status flag + size) and a block of user data.
- Malloc() uses a simple first-fit policy and scans the heap from low to high for a free chunk that is large enough (chunk size $\leq$ requested block size + admin size); if this fails it runs a procedure to coalesce adjacent free chunks and scans the heap once more. 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. A **mark-and-scan** garbage collector can reclaim cyclic datastructures. true/false

40. A **two-space copying** garbage collector can reclaim cyclic datastructures. 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 structures.
- The space overhead in each data structure is linear in the number of pointer fields in it. 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 structural equivalence, variables `a` and `b` have the same type. true/false

44. In a language that supports **overloading** an identifier may refer to different (function) types depending on the context.
- If the type checker cannot resolve an identifier due to separate compilation, a runtime check needs to be generated. true/false

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. true/false

46. Pointer supertyping and subtyping is complicated by **multiple inheritance**.

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

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

- An object of class `C` includes *two* pointers into the dispatch table for class `C`. 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. 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/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 holding live values before issuing a routine call. true/false

50. In some languages routines may be passed as parameters and returned as values.
- If routines may *not* be nested (as in ANSI C) the compiler can simply pass/return the address of the routine. 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 'a':  return aap;
              break;
   case 'n':  return noot;
              break;
   case 'm':  return mies;
              break;
}
```

- The jump table for the above case statement has 3 entries. true/false

52. Field alignment requirements may cause the compiler to insert gaps in a record representation.
- The equality comparison operator can be translated into an efficient n-byte string comparison with n set to the length of the (extended) record. true/false