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

exam – **Compiler Construction** – in4303
April 9, 2010 14.00 - 15.30

This exam (6 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 | ϵ
```

The non-terminal `B` is left recursive. **true**

2. A parser transforms a stream of tokens into an AST (Abstract Syntax Tree). **true**

3. The regular expressions **a(b|c)** and **ab|ac** 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 *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 only 1 *accepting* state. **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**.

```
identifier → • letter (letter_digit_underscore)*
```

- This is a reduce item. **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.
- ϵ moves do not consume any input character, but merely expand non-basic items with repetition and composition operators. **true**

8. A **recursive descent** parser can handle right recursive grammars. **true**

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

10. 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  →  A | B
A  →  ϵ | aA
B  →  b | bB
```

- FIRST(S) contains 3 elements. **true**

11. Automatically generated top-down parsers are built upon a PDA (Push Down Automaton) driven by a prediction table.
- The number of *columns* in the table is equal to the number of *non-terminals* in the grammar. **false**

12.

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

This grammar can be made LL(1) by applying **left factoring**. **false**

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

14. The SLR(1) parsing technique reduces a handle (the righthand side of a production N $\rightarrow \alpha$) when the current input token is an element of FIRST(α). **false**

15. The following set

S $\rightarrow$ • A x {$}

A $\rightarrow$ • a {x}

A $\rightarrow$ • a A a {x}

is a valid LR(1) item set **true**

16. An LALR(1) parser is more powerful, that is, can handle a larger class of grammars, than an LR(1) parser. **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 inherited attributes are, by convention, drawn on the right of node N in the AST. **false**

18. When dealing with attribute grammars it is important to detect cycles in the evaluation rules to prevent the evaluator to loop endlessly. In the case of **dynamic** cycle detection, the AST is traversed multiple times until either all attributes have obtained a value, or a maximum number of traversals is reached (in which case a cycle can be reported).
- For an AST with N attributes, cycles can be detected dynamically by performing (at most) N evaluation traversals. **true**

19.

```
FixedPoint(SYN value) →
  Number( dec) '.'  Number(frac)
  ATTRIBUTE RULES:
    SET value TO dec + frac/10;
```

The dependency graph associated with the attribute evaluation rule for `FixedPoint` contains just one dependency (arrow). **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. **false**

21. **Simple symbolic interpretation** performs a single traversal of the AST. **true**

22. **Full symbolic interpretation** performs an iterative stack-based simulation of the program at compile-time.
- To guarantee termination the information maintained with each stack element is limited to binary (on/off) properties. **false**

23. **Live analysis** can be performed 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.

```
M[i] = v;
```

- The GEN set for this assignment statement must be set to $\{$`i,v`$\}$. **true**

24. **Data-flow equations** can be solved efficiently by combining the effects of subsequent statements inside a basic block.
- this optimization allows the equations to be solved in a single pass of the control flow graph. **false**

25. To handle user-defined datatypes a **recursive** interpreter operates with **self identifying data**. **true**

26. An **iterative** interpreter operates on a threaded AST. **true**

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**

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

29. When generating code at the basic block level, a **dependency graph** is used instead of an AST.
- The edges (dependencies) between the nodes in the graph correspond with the threading pointers in the original AST. **false**

30. The **weighted register allocation** optimization for simple code generation evaluates the heaviest *basic block* first. **false**

31. **Graph coloring** is a register allocation technique that operates on a complete procedure at a time. **true**

32.

```
{  int n = a*b + 1;
   int m = 2*(n+7);

   x = (n+m+a)/3;
   y = (m+a)/2;
}
```

If `a` and `b` are live on entry and dead on exit, and `x` and `y` are dead on entry and live on exit, the **register interference graph** associated with the basic block above includes an edge between `a` and `y`. **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** repeatedly replaces a small sequence (window) of symbolic instructions with a more efficient sequence.
- when optimizing for code size, the number of instructions in the replacement sequence may be larger than in the original. **true**

35. The relocation information present in an object file includes a list of entry points and unresolved references.
- An entry point specifies the location of a text or data symbol relative to the start of the respective segment in the object file. **true**

36. 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.
- To handle forward references, the assembler maintains a **shadow memory** to annotate the various types of entry points (names) in the binary image. **false**

37. Low-level memory management requires the programmer to explicitly deallocate (free) memory blocks. The basic memory allocator (i.e. without free lists) operates on chunks, which include an administrative part and a block of user data.
- The length of a chunk equals the sum of the block length and the size of the administrative part. **true**

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 chunks. **true**

39. A **mark-and-scan** garbage collector visits all chunks *twice* per run. **false**

40. In the first phase of a **mark-and-scan** garbage collector all nodes reachable from the root set are marked as live.
- The root set contains node pointers stored in global data as well as in activation records on the stack. **true**

41. A **two-space copying** garbage collector is highly efficient as it only visits the live data (when switching between from- and to-space).
- This speed advantage comes at a price in memory footprint as the administration part of each chunk must include an additional field to store the forwarding pointer to the new location in to-space. **false**

42. Type checking is complicated by **coercions** and **casts**, which convert (user-defined) data from one type into another.
- A coercion is an **implicit** type conversion whose validity cannot be determined at compile time, so a check is generated that will be evaluated at run time. **false**

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. 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. **false**

45. Compared to 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**

46. When handling a **polymorphic** object-oriented language, the compiler must take care of **pointer supertyping**. **true**

47. Overloaded/overridden identifiers in an object-oriented language influence the layout and size of the objects at runtime.

```
class A {
   field a;
   method f();
}

class B extends A {
   field a;
   field b;
   method f();
   method g();
}
```

- An object of class `B` includes storage for just *two* data fields. **false**

48. The **dynamic link** in an activation record of a routine is needed on exit to restore the frame pointer to that of the caller of the routine. **true**

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

50. The administrative part of an activation record contains space to save machine registers when calling another routine.
- On machines with large register files, the **caller-saves** scheme is more efficient than the **callee-saves** scheme. **false**

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.
- The number of entries in the jump table equals the number of case labels. **false**

52. Field alignment requirements may cause the compiler to insert gaps in a record representation.
- Such gaps must be cleared (zeroed) explicitly at each record allocation and assignment (copy). **false**