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

exam – **Compiler Construction** – in4303
January 28, 2010 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. The number of nodes in a parse tree is greater than or equal to the number of nodes in the corresponding AST (Abstract Syntax Tree). **true**

2.

```
S  →  a | B
B  →  bB | b
```

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

3. A rule of thumb is that two characters belong to the same token if inserting white space changes the meaning of the program, hence, tokens have a minimum length of two. **false**

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

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 4 *accepting* states. **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` $\rightarrow$ `letter (letter_digit_underscore)*` $\bullet$

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

7. 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 may be more than one "recognizing" row in the table. **true**

8. A **recursive descent** parser can handle left-recursive grammars. **false**

9. 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 *terminals* in the grammar. **true**

10. Making a grammar with a FIRST/FIRST conflict LL(1) requires the application of **left factoring**. **true**

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

12. The stack used in a bottom-up parser contains an alternating sequence of states and terminal symbols. **false**

13. An LR(0) bottom-up parser uses a combined GOTO and ACTION table. **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	$\rightarrow$	AB
A	$\rightarrow$	ϵ | aA
B	$\rightarrow$	b | bB

- FOLLOW(S) contains 1 element. **true**

15. The following set

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

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

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

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

16. **Yacc** contains built-in support for handling ambiguous grammars resulting in shift-reduce conflicts.
- By default these conflicts are solved by performing the reduce action. **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 right of node N in the AST. **true**

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 nodes, cycles can be detected dynamically by performing (at most) N evaluation traversals. **false**

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 two dependencies (arrows). **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 *IS-SI graph* to compute the partitioning. **true**

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

22. Symbolic interpretation performs a stack-based simulation of the program at compile-time.
- For **simple** symbolic interpretation the information maintained with each stack element is limited to binary (on/off) properties. **false**

23.

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

Backward data-flow equations can be solved by a basic closure-algorithm using the above inference rules. **true**

24. **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.

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

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

25. An **iterative** interpreter is faster than a **recursive** interpreter.
- The advantage of a recursive interpreter is that it can handle left-recursive grammars. **false**

26. 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**

27. Code generation consists of three tasks:

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

- Register allocation is performed last as it deals with the most critical resource as far as execution speed is concerned. **false**

28. **Simple code generation** considers one AST node at a time.
- When the target is a *stack* machine, the code can be generated in one 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.
- The weight of a tree is computed as the maximum of its subtrees plus one. **false**

30. 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**

31. 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.
- register allocation is not need as all ladder sequences can reuse the same register, known as the accumulator register. **false**

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

33.

```
{  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 `x`. **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 replacement sequence must be shorter than the original. 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 captures all unsolved references to external symbols. **true**

36. A **linker** combines multiple object files into a single executable object. **true**

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.
- The administrative part is write-protected to avoid users modifying it when (accidentally) writing past the end of a block. **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 efficient implementation of the `malloc()` routine maintains a table of free lists indexed by ($^2\log$) block size.
- This way memory blocks can be allocated in constant time (on average). **true**

39. 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.
- By grouping all pointer fields, the administration overhead is limited to a single counter, that is, the space overhead per object is constant. **true**

40. A **reference counting** garbage collector is unable to reclaim cyclic datastructures. **true**

41. In the second phase of a **mark-and-scan** garbage collector the complete heap is scanned to reclaim the un-marked, garbage chunks.
- While scanning, the mark bits of the live chunks are cleared as a preparation for the next run of the garbage collector. **true**

42. Type checking is complicated by **coercions** and **casts**, which convert (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 me, you : RECORD
                  name: STRING;
                  id : INTEGER;
              END;
```

- In a language with name equivalence, variables `me` and `you` have the same type. **true**

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 locations and inserts a *dereference* coercion whenever a value is needed. **true**

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 storage for *three* data fields. **true**

46. If an object-oriented language provides **polymorphic typing** in combination with **static binding**, then the compiler can infer for every method call what the actual type of the object is. **false**

47. 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**

48. The **static link** in an activation record of a nested routine is needed to reference variables defined in enclosing (outer) scopes. **true**

49. 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 function and procedure calls in the same way (even though procedures do not return a value). **false**

50. The administrative part of an activation record contains space to save machine registers when calling another routine.
- In the **caller-saves** scheme, the entry code of a routine starts by saving those registers that will be used (corrupted) by it. **false**

51. Field alignment requirements may cause the compiler to insert gaps in a record representation.
- The assignment operator can be translated into an efficient n-byte memory block copy with n set to the length of the (extended) record. **true**

52. 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 'b':  return aap;
              break;
   case 'u':  return noot;
              break;
   case 'g':  return mies;
              break;
}
```

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