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

exam – **Compiler Construction** – in4020
July 5, 2007 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:
- 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  →  if C then R
R  →  S | S else S
```

This grammar is ambiguous. **true**

2. In a CFG (Context Free Grammar) the set of terminal and non-terminal symbols may overlap. **false**

3. A lexical analyzer transforms a stream of characters into a stream of tokens.
- The tokens are stored in the symbol table for further processing by the parser. **false**

4. The regular expressions **a+a** and **a*a** describe the same set of strings. **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 (minimal) FSA for handling an `identifier` includes 1 *accepting* state. **true**

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 → ([0-9])+ •`

- This is a shift item. **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**

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

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

10. The **yacc** parser generator can handle LALR(1) grammars. **true**

11. 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. **false**

12. Automatically generated top-down parsers are built upon a PDA (Push Down Automaton) driven by a prediction table.
- An empty entry in the table signals a syntax error. **true**

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	$\rightarrow$	A | B
A	$\rightarrow$	ϵ | aA
B	$\rightarrow$	b | bB

- FIRST(S) contains 2 elements. **false**

14. The stack used in a bottom-up parser contains an alternating sequence of states and grammar symbols. **true**

15. The following two items

$$\mathtt{A} \rightarrow \mathtt{P} \bullet \mathtt{Q}$$

$$\mathtt{B} \rightarrow \mathtt{P}\ \mathtt{Q} \bullet$$

can coexist in an LR item set. **false**

16. 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. **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 child nodes of N are responsible for evaluating the values of N's inherited attributes. **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 another *synthesized* attribute. **true**

19. 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**

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**

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. **false**

22. **Simple symbolic interpretation** performs a pre-order traversal of the AST. **false**

23. Several manual methods exist to annotate an AST with information aiding code generation.
- **Full symbolic interpretation** can be used to perform **constant propagation** on a threaded AST. **true**

24. **Data-flow equations** can be solved efficiently by combining the effects of subsequent statements inside a basic block. **true**

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

26. A **recursive** interpreter is constructed as a set of routines that implement the semantics of each type of AST node.
- When processing an AST consisting of N nodes, the interpreter will invoke at most N routines. **false**

27. Code generation consists of three tasks:

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

- These three tasks may be carried out independently without compromising the quality of the generated code. **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**

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

30. A **peephole optimizer** replaces small parts of the AST with more efficient subtrees. **false**

31. When generating code at the basic block level, a **dependency graph** is used instead of an AST.
- The nodes in the graph represent the basic blocks, and the edges capture the ordering dependencies that must be obeyed at execution time. **false**

32. 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 overlap. **true**

33.

```
{  int n;

   n = a+1;
   x = (b+c) * n;
   n = n+1;
   y = (b+c) * n;
}
```

If `a`, `b`, and `c` 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. **false**

34. When generating code at the basic block level, the dependency graph must be converted to target code.
- After identifying **ladder sequences**, instruction selection and instruction ordering can be performed efficiently in a *single* pass. **true**

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 lists all entry points of the *external* symbols. **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**

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. 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.
- The advantage is that blocks can be *freed* efficiently compared to using a single list. **false**

39. A **reference counting** garbage collector may need to invoke itself when reclaiming a user-defined datastructure such as a list or tree.
- each invocation is triggered by a reference count dropping to zero. **true**

40. During a run of a **mark-and-scan** garbage collector all free (garbage) nodes are coalesced into one big chunk. **false**

41. A **two-space copying** garbage collector copies all live data, reachable from the **root set**, from the 'from space' into the 'to space'.
- With each copy the collector stores a forwarding pointer in the original chunk in from-space. **true**

42. 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. **false**

43. A variable name (identifier) can be used in two ways: 1) to denote a value (rvalue), and 2) to denote a location (lvalue). It is the type checker's task to determine which use is allowed in which context.
- The type checker may insert coercions from lvalue to rvalue when processing the AST. **true**

44. A straightforward way to handle multiple names spaces (e.q., type names, variable/procedure names) is to maintain a separate symbol table for each name space. **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 just *two* data fields. **false**

46. 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.
- Each object created during the execution of a program contains a copy of the dispatch table. **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+1* parameters; the additional parameter points to the parent class to handle inheritance. **false**

48. The **static 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. **false**

49. The administrative part of an activation record contains space to save machine registers when calling another routine.
- In a **callee-saves** scheme, the compiler emits code to save all registers holding live values before issuing a routine call. **false**

50. 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**

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 1:  return bar;
            break;
   case 2:  return bar*2;
            break;
   case 3:  return foo+bar;
            break;
   case -1: return 0;
            break;
}
```

- The jump table for the above case statement has 5 entries. **true**

52. 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**

53. One of the additional complexities a compiler for a functional language faces is the need to infer polymorphic types of user-defined functions.

```
drop 0 xs = xs
drop n (x:xs) = drop (n-1) xs
```

The (inferred) type definition

```
drop :: ([Int] -> [a]) -> [a]
```

is correct. **false**

54.

```
foo 0 y z = z
foo x y z = foo (y-1) z x
```

The function `foo` is strict in its first argument (`x`) only. **true**

55. The implementation of (lazy) functional languages is based on a **graph reducer** that repeatedly

1) finds a redex (reducible expression) with functor f,
2) instantiates the right-hand-side of f, and
3) updates the root of the redex.

- In step 2) the graph reducer may invoke itself recursively to evaluate strict arguments of f. **true**

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.

```
sum [] = 0
sum (x:xs) = x + sum xs
```

- The short-circuit optimization saves the construction of two application nodes per invocation of `sum`'s second equation. **true**

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

```
grandparent(koen, allard).
```

- This is a fact. **true**

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 = mies" as the first answer. **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. **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 3 goals translates into a C function with 3 nested functions, one for each goal. **false**