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

exam – **Compiler Construction** – in4020
January 17, 2008 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. **Context handling** is part of the compiler **back-end** as its tasks (e.g., type checking) are depending on the characteristics of the target machine (e.g., word size). true/false

2.

```
S  →  b | A
A  →  Aa | a
```

This grammar is ambiguous. true/false

3. The regular expressions **(a+a)*a** and **a+a*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. A lexical analyzer *generator* automatically constructs an FSA (Finite State Automaton) that recognizes tokens.
- The generator is driven by a **regular description** true/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**.

```
integer → (•[0-9])+
```

- This is a reduce item. true/false

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 is exactly one "recognizing" row in the table. true/false

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

9. A **top-down** parser can handle LL(1) grammars. true/false

10. A **recursive descent** parser consists of a set of (recursive) routines, each routine corresponding closely to a rule in the grammar.
- An empty production (N $\rightarrow \epsilon$) 'translates' to a routine returning true without inspecting any token. true/false

11. 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 3 elements. true/false

12. The **yacc** parser generator 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

13. The stack used in a bottom-up parser contains an alternating sequence of states and grammar symbols (terminals and non-terminals).
- The depth of the stack is at most two times the number of terminals in the grammar. true/false

14. An LR(0) bottom-up parser uses a combined GOTO and ACTION table. true/false

15. 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(A) contains 2 elements. true/false

16. The following set

S $\rightarrow$ • A x {$}

A $\rightarrow$ • a {x}

A $\rightarrow$ • a A {x}

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 parent node of N is responsible for evaluating the values of N's synthesized attributes. 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 another *inherited* attribute. true/false

19. The **IS-SI graph** associated with a non-terminal N captures the (indirect) dependencies between the inherited and synthesized attributes of N. 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 *first* 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. The (context) information that can be determined with **simple symbolic interpretation** can be obtained with a single visit of the (threaded) AST. true/false

23. **Full symbolic interpretation** performs at least two traversals of the AST. true/false

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.

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

- The KILL set for this assignment statement is empty. true/false

25. A **recursive** interpreter operates on a threaded AST. true/false

26. To handle user-defined datatypes an **iterative** interpreter operates with **self identifying data**. true/false

27. In general code generation consists of three tasks:

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

- When targeting a *stack-machine*, however, register allocation becomes void, which simplifies code generation considerably. true/false

28. **Simple code generation** considers one AST node at a time.
- Child nodes (if any) are visited first, before emitting the code of an AST node itself. true/false

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

30. 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

31. When generating code at the basic block level, a **dependency graph** is used instead of an AST.
- The number of nodes in a dependency graph exceeds that of the corresponding AST as the sharing of intermediate values through local variables must be made explicit. true/false

32.

```
{  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 4 registers are needed for the basic block above. true/false

33. A **peephole optimizer** is a pre-processing phase in a compiler's back end that re-organizes the AST to ease the task of the code generator. true/false

34. When generating code at the basic block level, the dependency graph must be converted to target code. The **late evaluation** heuristic combines instruction selection and instruction ordering and operates by identifying **ladder sequences** within a basic block.
- It uses temporary registers to store intermediate values shared by ladder sequences originating at different roots 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 lists all entry points of the *external* 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 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/false

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 all 'free' memory is available as one big chunk. true/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/false

40. A **two-space copying** garbage collector can reclaim cyclic datastructures. true/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 remembers the original location in the from-space by storing a "source" pointer in the new chunk in to-space. 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 name 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. 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.
- Each object created during the execution of a program contains a pointer to the dispatch table. 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();
}
```

- The dispatch table for class `C` includes *four* entries, since the original methods `m1` (defined in `A`) and `m2` (defined in `B`) must be available for supertyping. true/false

47. To handle scope-resolution operators in an object-oriented language (e.g., `::` in C++), the compiler includes a static link in each object. true/false

48. 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. true/false

49. 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

50. For a programming language that supports nested routines the activation records generated by the compiler include a reference to the symbol table for looking up identifiers in outer scopes. 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 assignment operator can be translated into an efficient n-byte memory block copy with n set to the length of the (extended) record. true/false

53. The **offside-rule** of the functional programming language Haskell defines the (implicit) ending of syntactic constructs like equations and where clauses (local definitions).
- The offside-rule can be conveniently handled by the lexical analyzer inserting explicit markers into the token stream. true/false

54. 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. true/false

55. Strictness analysis is important for generating efficient code.

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

- The function `take` is strict in its first argument, but when the first two equations are reversed `take` is strict in its second argument. 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.

```
twice f x = f (f x)
```

- The short-circuit optimization saves the construction of two application nodes per invocation of `twice`. 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.

```
parent(koen, allard).
```

- 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), X \= Y, !.
```

- The query "`?- foo(mies,X).`" will produce two answers (bindings). true/false

59. 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

60. The key to fast execution of logic programs is the efficient implementation of unification through backtracking. The Warren Abstract Machine (WAM) is a very popular intermediate code for Prolog compilers.
- The WAM contains specialized functions for handling (unifying) the first *logic variable* of a program clause. true/false