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

exam – **Compiler Construction** – in4020
June 30, 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.** A lexical analyzer transforms a stream of tokens into an AST (Abstract Syntax Tree). true/false

**2.**

```
S  →  a | B
B  →  Bb | b
```

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

**3.** A lexical analyzer generated by **lex** is essentially a recursive-descent scanner true/false

**4.** The regular expressions **a?+** and **a+?** describe the same set of strings. 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 reduce item. true/false

**6.** Lexical analyzers based on an FSA (Finite State Automaton) 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
- When dividing a program text into tokens each character is inspected only once. true/false

**7.** The FSA constructed by a lexical analyzer generator is usually represented by a **transition table**.
- To reduce storage requirements, the transition table can be compressed by a technique known as **row displacement**. true/false

**8.** A **top-down** parser can handle LL($k$) grammars. true/false

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

**10.** 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 most* 1 token from the input stream. true/false

**11.**

```
S  →  A | x
A  →  aAb | x
```

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

**12.** When constructing a top-down parser we need to compute the FIRST sets of all production alternatives. The FIRST set of an alternative $\alpha$, FIRST($\alpha$), contains all terminals $\alpha$ can start with; if $\alpha$ can produce the empty string $\epsilon$, this $\epsilon$ is included in FIRST($\alpha$).

```
S  →  Ab
A  →  Aa | ϵ
```

- FIRST(S) contains 3 elements. true/false

**13.**

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

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

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

**15.** 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($\alpha$). 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.** 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/false

**18.** 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 inherited attributes. true/false

**19.** 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.
- If $N$ has $i$ inherited and $s$ synthesized attributes, the minimum number of dependencies is $i + s$. true/false

**20.** The evaluation of attribute grammars is a complex task. The most general approach is to perform **multiple visits** over the AST. By restricting the evaluation rules simpler evaluation schemes can be used.
- **L-attributed grammars** can be evaluated in a *single* visit by a top-down parser. true/false

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

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

**23.**

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

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

**24.** **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 $\{\texttt{i}\}$. true/false

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

**28.** The **weighted register allocation** optimization for simple code generation targeting a register machine evaluates the *heaviest* subtree of an AST node first.
- This optimization can only be used for associative and commutative operators (e.g., $+$ and $*$) since the order in which operands are evaluated is changed. true/false

**29.** **Graph coloring** is a register allocation technique that operates on multiple procedures at once to arrange for arguments to be passed in interference-free registers. true/false

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

**31.** 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

**32.**

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

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

If `a` and `b` are live on entry and dead on exit, and `x` is dead on entry and live on exit, the **register interference graph** associated with the basic block above includes an edge between `a` and `m`. true/false

**33.**

```
{
   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

**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 produces code blocks (one per ladder sequence) in the *reverse* order. 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.
- To handle forward references, the assembler must make two passes over the input stream. true/false

**36.** The relocation information present in an object file includes a list of entry points and unresolved references.
- This information is needed by the **linker** when combining multiple object files into a single executable. true/false

**37.** Low-level memory management requires the programmer to explicitly deallocate (free) memory blocks. For this purpose the memory allocator operates on chunks, which include some administrative part and a block of user data.
- The administrative part includes a pointer to the next free chunk. true/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/false

**39.** A down side of a **reference counting** garbage collector is that it cannot reclaim cyclic datastructures. true/false

**40.** To avoid a **mark-and-scan** garbage collector running out of stack space, when recursively marking live data, the **pointer-reversal** technique by Schorr & Waite can be employed.
- The space overhead is limited to a few bits (pointer-count field) per node. 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'.
- Subsequently, all mark bits in the 'from space' are cleared to ensure proper operation in the next invocation. 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 **implicit** type conversion whose validity cannot be determined at compile time, so a check is generated that will be evaluated at run time. true/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 structural equivalence, variables `a` and `b` have the same type. true/false

**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 values and inserts a reference coercion whenever a location is needed. true/false

**45.** When handling an object-oriented language, the compiler must maintain a method table for each class.
- If the language supports **static binding**, the compiler must generate a runtime representation of the method table (i.e. dispatch table) for each class. true/false

**46.** Supporting **polymorphic typing** in an object-oriented language, requires **pointer supertyping** in the implementation (i.e., generated code or runtime support). true/false

**47.** 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

**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 **dynamic link** in an activation record of a routine is needed on exit to retrieve the address of the instruction where to continue in the caller (who invoked this routine). true/false

**50.** In many programming languages routines may be passed as parameters. If routines may be nested, the compiler cannot simply pass the address of a routine, but must pass a routine descriptor holding the address as well as the static link.
- In addition the activation records of the enclosing routines must be allocated on the stack. true/false

**51.** 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). true/false

**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 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 4 entries. true/false

**53.** The functional programming language Haskell includes so-called syntactic sugar like **pattern matching** and **list comprehension**, which raises the abstraction level for the programmer, but complicates the task of the compiler writer.
- Pattern matching is dealt with by a source-to-source translation in the front-end of the compiler and does not require explicit handling in the back-end. 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.

```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f xs)
```

The (inferred) type definition

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

is correct. true/false

**55.**

```
apply f x = f x
```

The higher-order function `apply` is strict in its second argument (`x`). 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.

```
map f [] = []
map f (x:xs) = f x : map f xs
```

- The short-circuit optimization saves the construction of one application node per invocation of `map`. 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 rule. 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 default action of the basic, unoptimized Prolog interpreter discussed in the Modern Compiler Design (MCD) book is to attach clauses to the top-most left-most goal on the goal list stack.
- In a program with *N* clauses the interpreter will stack *N* new goal lists. true/false

**60.** When compiling logic programs to C, a concept called **list procedures** can be used to effectively implement backtracking. List functions invoke a function, passed as a parameter, for every computed value.
- In this approach, the basic unification function is a list procedure too. true/false