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

exam – **Compiler Construction** – in4020
June 24, 2004 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:
- A **true** statement must be marked as **Juist** (**J**); a **false** statement must be marked as **Onjuist** (**O**).
- 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** (naam) and **student-ID** (studentnummer), and to **sign** the form (handtekening).

---

1. The symbol table is used to pass information only between consecutive stages (lexical analysis, semantics analysis, etc.) in the compiler pipeline. true/false

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

3. A lexical analyzer transforms a stream of characters into a stream of tokens. true/false

4. The regular expressions **(a**|**b)+** and **a+**|**b+** describe the same set of strings. true/false

5. A lexical analyzer generated by **lex** is essentially a PDA (Push Down Automaton). true/false

6. 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/false

7. 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` $\rightarrow$ `(`$\bullet$`[0-9])+`

- This is a reduce item. true/false

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

9. The error handling mechanism of the **yacc** parser generator pushes the input stream back when inserting 'missing' tokens. 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	$\rightarrow$	ϵ | bB

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

12. Grammars with LL(1) conflicts can be made LL(1) by applying left-factoring, substitution, and left-recursion removal.
- Left-factoring takes care of FIRST/FIRST conflicts. true/false

13.

S	$\rightarrow$	A | xb
A	$\rightarrow$	aAb | x

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

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

15. The LR(1) parsing technique reduces a handle (the righthand side of a production N $\rightarrow \alpha$) only when the current input token is an element of FOLLOW(N). true/false

16. The following two items

$A \rightarrow P \bullet P$

$B \rightarrow Q \bullet Q$

can coexist in an LR 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. 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 a *synthesized* attribute. true/false

19.

```
Number(SYN value) →
  DigitSeq(base, value) BaseTag(base)
  ATTRIBUTE RULES:
    SET value TO DigitSeq.value;
```

The dependency graph associated with the attribute evaluation rule for `Number` contains 2 dependencies (arrows). 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 *last* visit. true/false

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

22. **Full symbolic interpretation** processes one basic block at the time. true/false

23. 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

24.

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

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

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

26. A **recursive** interpreter is constructed as a set of routines that implement the semantics of each type of AST node.
- the number of routines depends on the specific source program being interpreted. true/false

27. Code generation consists of three tasks:

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

- For generating "optimal" code these three tasks must be considered at once, because they dependent on each other. true/false

28. **Simple code generation** considers one AST node at the time. 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.
- 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

30. 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. true/false

31. 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

32. Register allocation by **graph coloring** uses a register interference graph.
- If a node in the graph is connected to N other nodes, then at at least $N + 1$ registers (colors) are needed. true/false

33.

```
{  double quads = a*a + b*b;
   double cross_prod = 2*a*b;

   x = quads - cross_prod;
   y = quads + cross_prod;
}
```

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

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

35. An **assembler** transforms an object file into a list of assembler instructions. 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 the size of the chunk to be able to identify the beginning of the next adjacent chunk, for example, when scanning the heap for a free chunk. 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, the free space may still be divided over multiple (non-adjacent) chunks. true/false

39. A **two-space copying** garbage collector can reclaim cyclic datastructures. true/false

40.

```
VAR p, q:  pointer;
...
p := q;
```

A compiler implementing garbage collection based on **reference counting** must generate code that first increments the count associated with `q`, then decrements the count associated with `p`, and finally sets `p` to `q`. true/false

41. A **mark-and-scan** garbage collector visits all chunks only *once* (per run) when reclaiming free chunks. true/false

42. Type checking is complicated by **coercions** and **casts**, which converts (user-defined) data from one type into another.
- A coercion is an **implicit** type conversion that must be handled by the compiler. true/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/false

44. To handle overloaded identifiers the type checker maintains a scoped symbol table. true/false

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

47. 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/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/false

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/false

50. 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 at routine entrance. true/false

51. When generating code for **boolean control expressions** that direct the flow of control, the compiler can make effective use of (conditional) jumps by emitting code that simply jumps to the "right" basic block identified through true and false labels.

```
if (i>0 || j>0) { ...  }
```

- The runtime evaluation of the condition in the above if-statement results in at least 1 jump. 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.

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

The higher-order function `twice` is strict in its first argument (`f`). 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.

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

The (inferred) type definition

```
sum :: [Int] -> Int
```

is correct. true/false

55. The implementation of (lazy) functional languages is based on **graph reduction**. The high-level concept of function application is explicitly represented in the form of binary application nodes (`@`) with a function and argument pointer.
- This explicit representation eases the support of pattern matching. true/false

56. Strictness analysis is important for generating efficient code.

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

- Since function `drop` is strict in its first argument, the expression `n-1` can be evaluated immediately. 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. 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. true/false

59. An interpreter for logic programs maintains a goal list stack capturing all alternatives that could potentially satisfy the user query.
- The action of the interpreter depends on the topmost leftmost goal only true/false

60. The Warren Abstract Machine (WAM) was specifically designed to handle the lazy evaluation semantics of Prolog. true/false