Faculty of Information Technology and Systems
Delft University of Technology

exam – **Compiler Construction** – in4020
August 29, 2003 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. A parser transforms a stream of tokens into an AST (Abstract Syntax Tree). true/false

2.

S	$\rightarrow$	a | B
B	$\rightarrow$	Bb | ϵ

The non-terminal S is left recursive. true/false

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

- This is a shift item. true/false

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

5. A lexical analyzer *generator* automatically construct a FSA (Finite State Automaton) that recognizes tokens.
- The generator is driven by a **regular description** true/false

6. Lexical analyzers based on a FSA efficiently match multiple token descriptions concurrently to the input stream. Nevertheless, input characters my 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 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/false

8. A **recursive descent** parser is based on a PDA (Push Down Automaton). true/false

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

10. A **bottom-up** parser creates the nodes in the AST in pre-order. true/false

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

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

13. The dynamic conflict resolvers of the LLgen parser generator can be used to resolve all LL(1) conflicts including left recursion. true/false

14. The following two items

$$\mathsf{A} \rightarrow \mathsf{P} \bullet \mathsf{Q}$$
$$\mathsf{B} \rightarrow \mathsf{P}\ \mathsf{Q} \bullet$$

can coexist in an LR item set. true/false

15.

$$\begin{array}{lcl} \mathsf{S} & \rightarrow & \mathsf{A} \mid \mathsf{xb} \\ \mathsf{A} & \rightarrow & \mathsf{aAb} \mid \mathsf{x} \end{array}$$

This is an LALR(1) grammar. true/false

16. 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).

$$\begin{array}{lcl} \mathsf{S} & \rightarrow & \mathsf{A\ B} \\ \mathsf{A} & \rightarrow & \epsilon \mid \mathsf{aA} \\ \mathsf{B} & \rightarrow & \mathsf{b} \mid \mathsf{bB} \end{array}$$

- FIRST(S) contains 2 elements. 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 a grammar with N *production rules*, cycles can be detected dynamically by performing (at most) N evaluation traversals of the AST. true/false

18. The evaluation of attribute grammars is a complex task. One approach is to perform **multiple visits** over the AST.
- The number of visits needed to evaluate all attributes can be determined at compile-time when generating the parser. 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.
- A *synthesized* attribute can depend on another *synthesized* attribute. 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. The (context) information that can be determined with **simple symbolic interpretation** can be obtained with a single visit of the (threaded) AST. true/false

22. **Data-flow equations** operate with IN, OUT, GEN, and KILL sets. true/false

23. **Data-flow equations** can be solved efficiently by using bitwise boolean instructions (AND, OR, etc.). true/false

24. **Symbolic interpretation** performs a stack-based simulation of the program at compile-time. true/false

25. 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/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. Code generation consists of three tasks:

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

- Instruction selection must be performed first because register allocation operates on basic blocks, not individual instructions. true/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 (depth-first) traversal of the AST, possibly introducing temporaries when running out of registers. true/false

29. The **weighted register allocation** optimization for simple code generation targeting a register machine saves at most 1 register per complete expression. 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 **memory spilling**. true/false

31.

```
{
   x = a*a - 2*a*b + b*b;
   y = a*a + 2*a*b + b*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

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/false

33. **Graph coloring** is a register allocation technique that operates at *individual* basic blocks. true/false

34. 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

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 *internal* 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. 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 previous chunk, for example, when coalescing adjacent blocks. 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 **mark-and-scan** garbage collector can reclaim cyclic datastructures. true/false

40. A **reference counting** garbage collector marks chunks to avoid looping on cyclic data structures when (recursively) freeing cells with a zero reference count. 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 stores a forwarding pointer in the original chunk in from-space. true/false

42. Type checking is complicated by **coercions** and **casts**, which converse (user-defined) data from one type into another.
- A coercion is an **explicit** type conversion specified by the programmer. true/false

43. 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. true/false

44. In a language that supports **overloading** an identifier may refer to different (function) types depending on the context.

```
function foo() :  int is return 11;
function foo() :  real is return 13.0;

function bar() :  real is return foo() / 2;
```

- The type checker must be prepared to operate on sets (of types). true/false

45. When handling an object-oriented language, the compiler must take care of **pointer supertyping** (to handle polymorphism) and **pointer subtyping** (to handle dynamic binding).

```
class A {
   field a1;
   method m1();
}

class B extends A {
   field b1;
   method m1();
   method m2();
}
```

- If the language supports only **single inheritance**, the effects of supertyping and subtyping is that of a type cast only. true/false

46. 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

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. When executing code on a stack machine, the size of an activation record is fixed during the lifetime of a routine invocation. true/false

49. The administrative part of an activation record contains space to save machine registers when calling another routine.
- In a **caller-saves** scheme, the compiler emits code to save all registers at routine entrance. true/false

50. 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

51. When generating code for a while statement, the compiler may use the following naive scheme

```
test_label:
  Boolean control code( condition, No_label, end_label)
  Code statement( statement)
  GOTO test_label;
end_label:
```

- An optimized scheme can save one (conditional) jump instruction per loop iteration. true/false

52. When generating code for a case statement, the compiler may decide to use a **jump table** to ensure that the selection during program execution takes constant time.
- The length of the jump table equals the number of case labels. true/false

53. 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/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.

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

The (inferred) type definition

```
twice ::  (a -> 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.

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

- The short-circuit optimization saves the construction of one application node per invocation of `twice`. true/false

57. The implementation of logic programming is based on the concept of unification. For efficiency the unification process uses **backtracking** to undo bindings of logic variables.
- Backtracking avoids the creation of a fresh copy of a program clause on each unification. true/false

58. An interpreter for logic programs maintains a goal list stack capturing all alternatives that could potentially satisfy the user query.
- All the goals in one list (row) on the stack must be satisfied, before the interpreter can report a result. true/false

59. **List procedures** invoke a function, passed as a parameter, for every "computed value". When compiling logic programs to C such list procedures can be used to effectively implement backtracking.
- The computed values are then successful unifications, that is, bindings of logic variables to terms. true/false

60. To speed up the performance of logic programs, the programmer may use the cut operator (!) that avoids backtracking over the goals before it in the goal-list.
- When generating (nested) list procedures, the cut induces a (non-local) jump to the end of the clause procedure in which it appears. true/false