2019 "Alt"prüfung Lösungen

1

Q: In how many different states can a computer with 1GB of memory be?

A: 1GB of memory = $\begin{cases} 10^9 \text{ bits} & GB = \text{gigabyte} \\ 2^{30} \text{ bits} & GB = \text{gibibyte} \end{cases}$ ← difference: 1KB := 1024 vs 1000 bit

each bit can be in two states. State was defined as an assignment of 0 or 1 to each bit. There are 2^{10^9} or $2^{2^{30}}$ possible assignments depending on how we define GB.

The exact formula is 2^n, where n is the number of bits of memory the machine has.

Q: How does the machine distinguish between code and data?

A: [Usually,] Memory is divided into static and dynamic memory. Program code and some of its data live in the static part which is again divided into a code (text) and data segment. The data segment only holds global and static local variables, the code segment holds instructions and is read-only. The dynamic part holds data allocated at runtime, the static part is fixed at compiletime.

The program break separates the static and dynamic part. Within the static part, the CPU has a special register called the program counter that holds the memory address of the next instruction to be executed; i.e. a pointer to some place in the code segment.* The program counter tells the machine that whatever address it is currently pointing to is the next instruction to be executed and is updated to the address of the next instruction after execution (taking into account branch instructions etc.) The PC is how the machine distinguishes code and data.

* that is bootstrapped to the address of the first instruction of the program.

Q: Can the same word sometimes be code and sometimes data?

In theory yes since there's nothing stopping the program counter from pointing to some data address and the semantics of the word at that address is not inherently given, so the CPU can just load and interpret the word as instruction, giving it meaning as code.

In practice however this is not usually done, only words in the read-only code segment (hence never treated as data) are executed.

There is another way in which the same word can be code and data in practice however. When compiling some program the program and the executable output file are data while the compiler is code. When executing the output, it becomes code.

When compiling a compiler with itself, it becomes code and data at the same time: e.g.

```
./selfie -c selfie.c -o selfie
```

② Q: Why do most computers encode information in binary? Why is it a popular format?

Othe options are unary, octal, decimal, hexadecimal. Unary results in exponentially longer words compared to other bases ≥ 1 while the word length difference is a constant growing logarithmically with alphabet size when comparing binary to bases ≥ 2. Arithmetics is easy in binary and becomes increasingly cumbersome although more succinct as the alphabet grows.

We conclude that unary neccessitates massive amounts of memory while bases larger than 2 offer diminishing returns and increasing complexity. Furthermore current semiconductor based computers lend themselves to the binary format.

[Careful: Might also ask about two's complement!]

Q: Convert $123_{(10)}$ to binary. Provide the number and formula. Also do it the other way around.

Use repeated division by two, noting down the sequence of remainders, until reaching zero; then reverse the sequence of remainders.

For the other direction, given a k-digit binary number $a_k a_{k-1} \ldots a_1 a_0$, compute:

$$\sum_{i=0}^{k} a_i \cdot 2^i$$

123	61	1
61	30	1
30	15	0
15	7	1
7	3	1
3	1	1
1	0	1

$1111011_{(2)}$

Q: How are characters and strings encoded in bits?

ASCII is a binary encoding for characters. It maps characters to exactly one bit sequence of length 8 each according to the UTF-8 standard. Internally, the machine only manages bit sequences, mapping them to and from characters as necessitated by user in/output.

[In selfie,] Strings are contiguous chains of those 8-bit sequences representing characters in memory. The end of these chains is marked with the "null character" (which is just the numerical value 0). Succinctly said, strings are stored <u>contiguously</u> at <u>increasing addresses</u> in memory and <u>null-terminated</u>.

There are other ways to store strings with various tradeoffs, however.

(3)

Q: Define FSMs.

A FSM M is defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ with

$Q :=$ set of states

$\Sigma :=$ input alphabet

$\delta : Q \times \Sigma \to Q$ = transition function

$q_0 \in Q :=$ initial state

$F \subseteq Q :=$ set of accepting states

Intuitively, we start in the initial state q_0 and deterministically transition to other states as defined by δ for each symbol of the input word until we reach the end of input, at which point we will be in some state either in F or not in F. We accept or reject the input depending on that.

Q: Name their key advantage

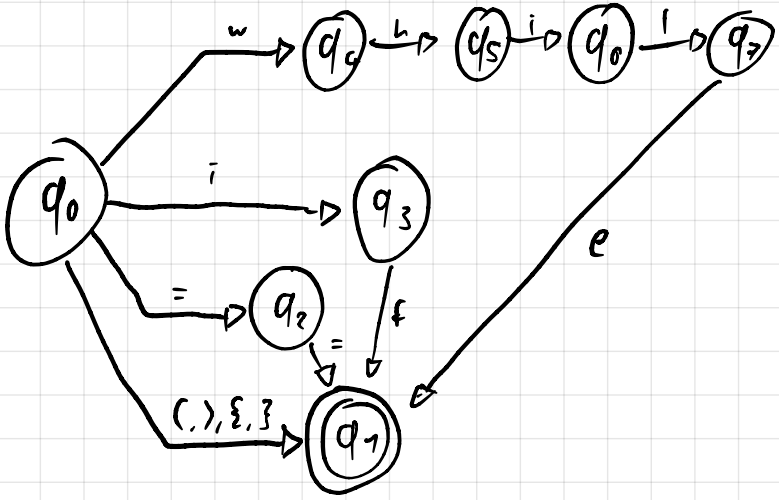
They can recognize regular expressions while being simple and efficient to implement with if/else.

This makes them a good choice for implementing scanners in compilers. Scanners tokenize the input character stream (they transform it into a sequence of encoded symbols that the parser, the next stage of the compiler, understands).

Q: Provide a FSM recognizing if, while, (,), {, }, == as symbols as a graph and as C* code.

For the graphical FSM, any transitions that are not given lead to a dead error state.

For the C* code, define get_character(), find_next_character() etc. as in selfie.c




```
if (find_next_character() != EOF) {
  if (character = 'w') {
    get_character();
    if (character = 'h') {
      ...
        symbol = SYM_WHILE;
        get_character();  return;   } return 1;
  } else if (character = 'i') {
    get_character();
    if (character = 'f') {
      symbol = SYM_IF;
      get_character();  return;   } return 1;
  } else if (character = '(') {
```

```
        symbol = SYM_LPARENTHESIS;
        get_character(); return;
    } else if (character = '}') {
        symbol = SYM_RBRACE;
        get_character(); return;
    } else if (character = '=') {
        get_character();
        if (character = '=') {
            symbol = SYM_EQUALITY;
            get_character(); return;
        }
    }
}

symbol = SYM_EOF; // for lack of a better
                  error code
```

4

Q: Define a PDA

A PDA is a FSM that, additionally, has a stack that is read from and written to on every transition. ε-writes and ε-transitions are allowed. Deterministic PDAs are more powerful than FSMs as they can recognize context-free languages while still being easy and efficient to implement with recursion.

Q: Provide C* code recognizing a while statement in C*

```
while = "while" "(" expression ")"
        ( statement | "{" { statement } "}" ).
```

Note: EBNF

"..." = literal

(...) = block

[...] = optional block

| = intra block OR

Code:

```
if (symbol == SYM_WHILE) {
  get_symbol();
  if (symbol == SYM_LPARENTHESIS) {
    get_symbol();
    compile_expression();
    if (symbol == SYM_RPARENTHESIS) {
      get_symbol();
      if (symbol == SYM_LBRACE) {
        while (is_not_rbrace_or_eof())
          compile_statement();
        if (symbol == SYM_RBRACE) {
```

```
                get_symbol();
                return 1;
            }
        }
    }
}
return 0;
```

Q: How do compilers manage names of procedures and global/local variables?

With a symbol table, often implemented as a hashmap, that maps the name of the variable to its address in memory as well as some other metainformation. For procedures, the address is left to be zero, but the return type for instance is stored.

In selfie, there are multiple symbol tables used to implement scope as well as mapping. There is a local, library and a global symbol table. When looking up a name, they are referenced in the listed order until a hit is found. This way, local variables have priority over global variables and library procedures have priority over global procedures.

Q: Name the data structure, its operations and their usage.

The local table is a linked list and only holds variables. The library table only holds procedures and is a LL as well. The global table is a hashmap because O(n) lookups as with LLs are slow with a table as big as the global table.

Both a hashmap and a LL can store a variable number of elements. In a LL, elements are wrapped in containers that are "linked" together meaning each has a pointer to the next. This is the reason why searches are O(n).

Operations + usage:

- insert - when a new variable/procedure has been parsed
- search - to look up information about vars/procedures or to find out whether they have been declared or defined at all
- getters/setters for member variables of list/map items
  - to look up information, to update information (e.g. when a previously declared procedure or variable is finally defined)

Q: What's the difference between a declaration and a definition?

Declaration: Allocating space, giving it a type and labeling it with some name which then becomes an identifier. For procedures, the "type" is the return type and parameter type(s).

Definition: Actually filling that previously allocated space with data

Q: What does a computer do upon declaring and defining a variable and procedure?

Variables:

Declaration: Allocate as much memory as required by the type. Map name to allocated memory address

Definition: Fill allocated space with data corresponding to the definition.

Procedures:

Declaration: Create global symbol table entry introducing the name for the procedure globally. Create local ST filling it with entries for the procedure's parameters

Definition: Fill local ST with any local variables declared in the body. Generate code for:
- prologue preparing the machine for executing the procedure (callee)
- the actual procedure
- epilogue preparing the machine for returning to the caller

↑ this refers to the stack and registers.

⑥ Q: What's the memory layout for code generated by most compilers?