Lecture Outline

- What is Tokenization?
- Another Aside on Writing Systems and Character Encoding
- Regular Expressions
- Finite State Automata
- Finite State Transducers
- Tokenization Revisited

Reading:

- Chapter 2 of Jurafsky and Martin
- Daniels, P. T. and Bright, W. eds, The World's Writing Systems. Oxford University Press, 1996.

Tokenization

- Typically the first step in the detailed processsing of any natural language text is **tokenization**.
- Tokenization is the separation of a text character stream into elementary chunks or segments or **tokens** – generally **words**, but also numbers, punctuation, etc.
- In addition to recognizing tokens, a tokenizer may associate a token class/type with each token. E.g.

  George – alpha-upperinitial
  1066 – numeric
  “ – punctuation

- Not always obvious what counts as one token. E.g, is can't
  - ◇ one token: can't
  - ◇ two tokens: can + 't
  - ◇ three tokens: can + ' + t

Tokenization Example

- Example:

  > *"Our profits fell 3.5% in the fourth quarter," Mr. David P. Sacks, hedge-fund manager at 3GPiracy Inc. admitted Friday. "Not what we'd hoped."*

  Breaking on whitespace alone yields:

  | *"Our* | *profits* | *fell* | *3.5%* | *in* | *the* | *fourth* | *quarter,"* | *Mr.* | *David* | *P.* | *Sacks,* | *hedge-fund* | *manager* | *at* | *3GPiracy* | *Inc.* | *admitted* | *Friday.* | *"Not* | *what* | *we'd* | *hoped."* |

  This includes units such as quarter," which are clearly not words. Prefer something like:

  | *"* | *Our* | *profits* | *fell* | *3.5* | *%* | *in* | *the* | *fourth* | *quarter* | *,"* | *Mr.* | *David* | *P.* | *Sacks* | *,* | *hedge-fund* | *manager* | *at* | *3GPiracy* | *Inc.* | *admitted* | *Friday* | *.* | *"* | *Not* | *what* | *we'd* | *hoped* | *."* |

  Still some issues:

  | *"* | *Our* | *profits* | *fell* | *3.5* | *%* | *in* | *the* | *fourth* | *quarter* | *,"* | *Mr.* | *David* | *P.* | *Sacks* | *,* | *hedge-fund* | *manager* | *at* | *3GPiracy* | *Inc.* | *admitted* | *Friday* | *.* | *"* | *Not* | *what* | *we'd* | *hoped* | *."* |

Aside on Writing Systems: Languages vs Scripts

- What tokenisation should be carried out is dependent on the **orthographic** (to do with writing) conventions adopted in the writing system or script being processed.
- A **script** is the system of graphical marks employed to record expressions of a language in visible form
- There is a many-to-many mapping between languages and scripts
  - ◇ many languages may share one script
    - e.g. most of the languages of Western Europe use the Roman (or Latin) script with some small variations (accents in French, some extra characters in Spanish, the Scandinavian languages)
  - ◇ some languages may have multiple scripts
    - Japanese – kanji (from Chinese); hiragana (for grammatical particles); katakana (for loan words from languages other than Chinese); arabic numerals
    - Chinese – traditional logograms; pinyin – romanised script

Types of Scripts

- Scripts may be classified along a number of dimensions
  - Directionality
    - left-to-right – e.g. English, Russian
    - right-to-left – e.g. Arabic, Hebrew
    - top-to-bottom – e.g. Chinese
    - bottom-to-top – e.g. Mongolian
    - boustrophedon ("ox-turning") – e.g. (some) ancient Greek
  - Historical derivation
  - Relationship between symbols and sounds
    - *phonological* systems show a clear relationship between sounds of the language and symbols
    - *non-phonological* systems do not
- Sound/symbol relationship generally agreed to be the best way to classify scripts
  - but no consensus among scholars as to the best set of sub-classifications

Types of Scripts – Non-phonological Systems

- **pictographic**
  - graphemes (pictographs/pictograms) provide recognisable picture of entities in the world
  - some reject pictograms as true writing systems, arguing that to count as writing system:
    "it must be possible to represent an utterance in such a way that that it can be recovered more or less exactly without the intervention of the utterer" (Daniels and Bright, 1996)
- **ideographic**
  - graphemes (ideographs/ideograms) no longer directly represent things in the world, but have an abstract or conventional meaning
  - may contain linguistic or phonological elements
  - e.g. early Sumerian, Hittite scripts

Types of Scripts – Non-phonological Systems (cont)

- **hieroglyphic** (= "sacred carving")
  - contains ideograms, phonograms (ideogram for "bee" + "r" = "beer" - rebus) for consonants, and determinatives (disambiguate words with multiple senses)
  - e.g. Egyptian, Mayan, Indus Valley
- **logographic**
  - graphemes (logographs/logograms) represent words/parts of words
  - some characters derive from earlier ideograms, others are phonetic elements, others are modified versions of base characters to indicate words of related meaning
  - e.g. Chinese, Japanese Kanji, mathematical/logical notations

Text Examples – Non-phonological Scripts

Hieroglyphics

Text Examples – Non-phonological Scripts (cont)

Simplified Modern Chinese

中国方面则表示，假如美国对中国实施贸易制裁，中国方面将采取报复。但是，白宫发言人麦科里说，他希望形势不至于发展到这一步。麦科里说，在我们的一个非常重要的贸易伙伴威胁要进行贸易战的时候，我们总是感到关注。我们认为，在经济上与中国保持广泛的接触符合中国和美国人民的共同利益。

Types of Scripts – Phonological Systems

- **syllabic**
  - graphemes correspond to spoken syllables, typically a vowel-consonant pair
  - e.g. classical Cypriot, Japanese katakana, Cherokee
- **alphabetic**
  - direct correspondence between graphemes and phonemes (sound units) of the language
  - e.g. Phonecian, Arabic, Hebrew, Greek, Roman
  - ideal is a one-to-one grapheme to phoneme relation, but this frequently fails (e.g. Spanish quite good; English notoriously bad)
    - pronunciation in spoken language evolves and written form does not keep pace
    - alphabet was "borrowed" from one designed for a different language
  - in some alphabets not all phonemes are represented graphemically
    - in Arabic marking of vowels using diacritics is optional – such languages are sometimes called consonantal alphabetic
    - in the Devanagari script (Hindi, Nepali, Marathi) consonants have a default vowel, which can be overridden by adding a diacritic – such languages are sometimes called syllabic alphabetic

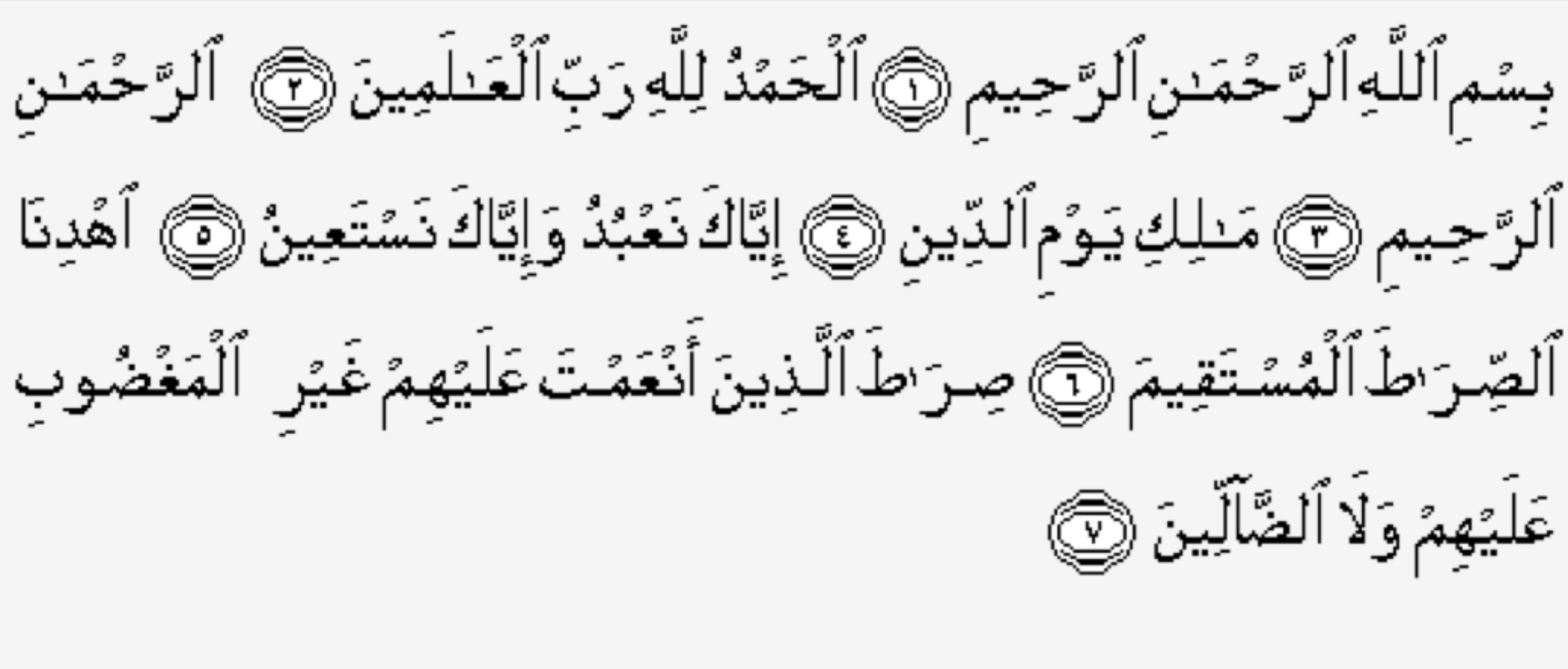
Text Examples – Phonological Scripts

Devanagari – Part of the "Alphabet" /Syllabary

अ a	आ ā	इ i	ई ī	उ u	ऊ ū
ऋ ṛ	ॠ ṝ	ऌ ḷ	ॡ ḹ		
ए e	ऐ ai	ओ o	औ au	अं aṃ	अः aḥ
क ka	ख kha	ग ga	घ gha	ङ ṅa	
च ca	छ cha	ज ja	झ jha	ञ ña	

Text Examples – Phonological Scripts

Arabic – Koran v. 1-7



Characters vs Glyphs

- A *character*/*grapheme* is the smallest component of written language that has a semantic value.
- A *glyph* is a representation of a character or characters, as they is rendered or displayed.
  - E.g. **A**, A, $\mathcal{A}$
    are different glyphs representing the Latin capital letter A
  - A repertoire of glyphs makes up a *font*
- The relationship between glyphs and characters can be complex
  - for any character there may be many glyphs representing it (in different fonts)
  - a single glyph may correspond to a number of characters. E.g. fi is typically a single glyph used to represent f i in typesetting English text
  - there may be arbitrariness – should é
    - be stored as two characters and rendered using two glyphs
    - be stored as two characters and rendered by one glyph
    - be stored as one character and rendered by one glyph

Characters vs Glyphs (cont)

- When deciding on the what the set of characters for a language are for purposes of representing them in a computer, it is extremely important to separate out characters from glyphs
  - the underlying representation of a text should contain the character sequence only
    - to support string search/matching, sorting, etc.
  - the final appearance of the text is the responsibility of the rendering process

Character Encoding

- Character encoding is necessary for the representation of texts in digital form
- Standards must be agreed if such data is to be widely shared
- Many standards have emerged over the years, even for single languages and languages as relatively straightforward as English (EBCDIC, ASCII, ISO 8859)
- Developing a single multilingual encoding standard will permit
  - ◇ processing of multiple languages in one document
  - ◇ reuse of software for products dealing with multiple languages (e.g. Web software)

Character Encoding (cont)

- **Unicode** is emerging as a global standard, though not without contention
- Key features of Unicode include:
  - 16-bit codes, with extensibility to 24-bits via surrogate pairs
  - 8-bit encoding version for backwards compatibility (UTF-8)
  - a commitment to encode plain text only
  - attempt to unify characters across languages, modulo support for existing standards
  - use of character property tables to associate additional information with characters
  - an encoding model which clearly separates:
    - the abstract character repertoire
    - their mapping to a set of integers
    - their encoding forms + byte serialization

Regular Expressions

To tokenize a text that

- is in a given character encoding
- adopts known orthographic conventions

we can write rules using **regular expression (RE)** pattern matching capabilities found in most modern programming languages (Java, Perl, Python)

Regular Expressions (cont)

A RE is a formal notation for characterising a set of strings.
RE's may be or contain:

- **basic** or **simple REs**: every character or sequence of characters is an RE that matches itself (with a few exceptions). For example:
  - the RE `a` matches `a`
  - the RE `foobar` matches `foobar`

Regular Expressions (cont)

- **metacharacters**: one of:

  `\ | ( ) [ { ^ $ * + ? .`

  These characters have special meanings.
  - ◇ the RE `.` matches any single character, So `.` matches `f`
  - ◇ `\` is used
    - to escape a metacharacter in an RE to allow metacharacters to matched. So the RE `\\` matches the string `\`
    - in conjunction with another character to specify a **metasymbol** with special meaning – see below

Regular Expressions (cont)

- **Character classes**: To match a character from a class of characters a RE can contain
  - ◇ a enumerated list of the characters to be matched in square brackets
    - the character class `[aeiou]` matches any of the five vowels
  - ◇ a range of characters to be matched in square brackets
    - the character class `[0-9]` matches any of the ten digits
    - the character class `[a-j]` matches the first ten lower case alphabetics
    - the character class `[a-zA-Z]` matches all alphabetics

  To match any character **not** in a class of characters a RE can contain a `^` at the start of the character class specification. E.g.
  - ◇ `[^xyz]` matches any single character **except** `x` or `y` or `z`

Regular Expressions (cont)

- **Metasymbols**: sequences of two or more characters with special meaning in REs, the first character of which is always a backslash. Metasymbols fall into two categories:
  - **special characters**: certain characters in a RE that cannot easily be typed. E.g.
    - `\n` (Newline), `\r` (Return), `\t` (Tab)
  - **character class shortcuts**: character classes used so frequently abbreviations have been created for them
    - `\d` (digit character) abbreviates `[0-9]`
    - `\w` (word character) abbreviates `[A-Za-z0-9_]`
    - `\s` (space character) abbreviates `[\f\t\n\r ]`
    - `\D`, `\W`, `\S` negate these classes; i.e. `\D` abbreviates `[^\d]`, `\W` abbreviates `[^\w]`, and `\S` abbreviates `[^\s]`

Regular Expressions (cont)

- **Disjunction/Grouping operators**.
  - The disjunction operator | permits the construction of disjunctive REs of the form `a | b` that match either `a` or `b`
  - The grouping operators `( )` placed around a complex RE allows it to be treated as a unit to which other operators may be applied. E.g.
    - `/gupp(y|ies)/` matches `guppy` and `guppies`
- **Quantifiers**: Specify how many times a preceding RE is to match.
  - Three basic quantifiers in REs:
    - `*` : match the preceding item 0 or more times
    - `+` : match the preceding item 1 or more times
    - `?` : match the preceding item 0 or 1 times

    Examples
    - `/foo+/` matches `foo`, `fooo`, `foooo`, etc.
    - `/(foo)+/` matches `foo`, `foofoo`, `foofoofoo`, etc.
  - Precise numbers of matches can be specified using numeric quantifiers in curly braces.
    - `/\d{5,10}/` matches from 5 to 10 digits
    - `/\d{5,}/` matches 5 or more digits
    - `/\d{5}/` matches exactly 5 digits

Regular Expressions (cont)

- **Anchors**: Force a pattern to match at particular points in the string
  - ◇ The beginning of a string is matched using ^
    - `/bert/` matches `bert` and `robert`
    - `/^bert/` matches `bert` but not `robert`
  - ◇ The end of a string is matched using the $ character in a regexp
    - `/bert/` matches `bertrand` and `robert`
    - `/bert$/` matches `robert` but not `bertrand`
  - ◇ \b (word-boundary anchor) anchors matching at the beginning or end of "words" (strings matched by \w+)
    - `/\bbert/` matches `to bert` but not `to robert`
    - `/bert\b/ matches robert was` but not `bertrand was`

    \B matches non-word boundaries – everywhere \b does not
    - `/\bbert\B/` matches `bertrand` but not `bert` or `roberto`

Regular Expressions (cont)

- **Capture and Backreference operators**
  - ◇ Capturing is done using parentheses ("()") – any substring of the target matched by a pattern segment in parentheses is remembered by the matcher
  - ◇ Back referencing is done using a backslash followed by an integer identifying which captured string is referred to

For example, suppose we wanted to find all words ending in a double-letter followed by `ing` or `ed` – e.g. `planned, hitting, running`, etc. but not `planed, ruined`, etc.
This task could be accomplished with the pattern

```
/\B([a-z])\1(ing|ed)\b/
```

- The first character is captured by the subpattern `([a-z])`
- The `\1` insists that this is followed by a second occurrence of what was captured in the preceding subpattern
- If `ing` or `ed` match, they are captured too, and are available via the backreference `\2` if needed

Regular Expressions: Examples

- Write a regular expression to match date expressions of the form DD/MM/YY or DD/MM/YYYY.

  `/[0-3]\d\/[01]\d\/(\d\d|\d\d\d\d)/`

  (a crude, first attempt – why is it inadequate?)
- Write a regular expression to match all strings which have both the word *grotto* and the word *raven* in them (but not, e.g. words like *grottos* that merely *contain* the word *grotto*). (Jurafsky and Martin, Exercise 2.1 (f))

  `/(.*\bgrotto\b.*\braven\b.*)|(.*\braven\b.*\bgrotto\b.*)/`

Finite State Automata

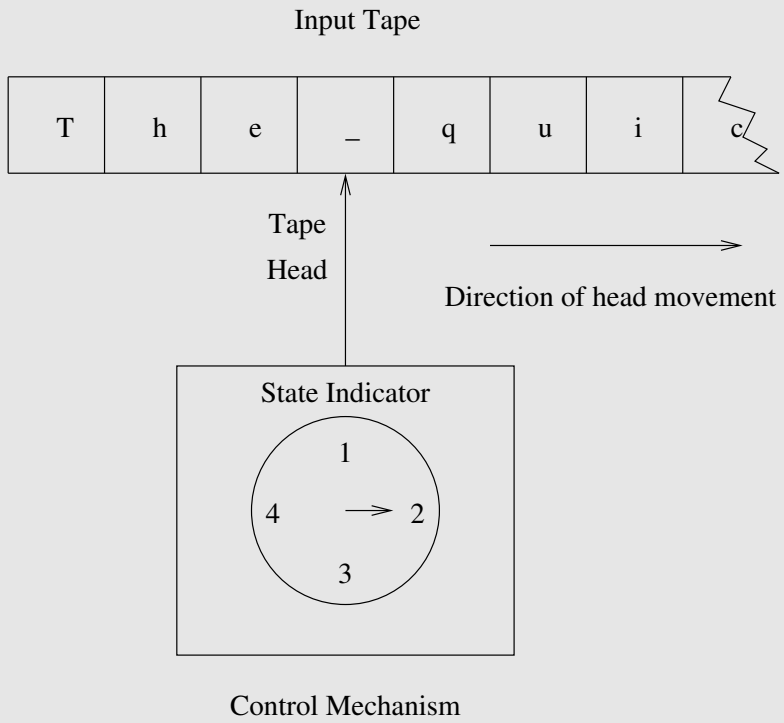
- From a formal language theory perspective a regular expression is an **automaton** or a **language acceptor**
  - the set of strings the regular expression matches is the language it accepts
- From formal language theory: regular expressions are equivalent to regular grammars and to finite automata (Kleene theorem)
- A **finite state automaton (FSA)** is an abstract device that
  - is always in one of a **finite** number of states
  - has exactly one initial state
  - has at least one accept state
  - is attached to an input stream from which symbols are read sequentially one at a time
  - is capable of switching to a new state (or staying in the current state) which is uniquely **determined** by the the current state and the symbol read from the input
  - has for every input symbol a defined successor state.

Finite State Automata (cont)

- The process of switching from one state another is called a **state transition**.
- The mechanism which decides, on the basis of the current state and the next input symbol, which new state to move to is called the **control mechanism**.
- The FSA starts in the initial state with its head at the beginning of the tape and proceeds until
  - it reaches an accept state or
  - it runs out of input or
  - it mets a symbol for which no transition is defined.

Finite State Automata (cont)

- A FSA may be visualised by imagining:
  - its input stream to be a tape
    - divided into cells each holding one symbol
    - with a fixed end at the left but extending indefinitely to the right
  - its control mechanism to be a clock face with a single hand indicating the current state
  - a read head positioned over the tape which reads a symbol, passes it to the control mechanism and is then repositioned over the next cell.

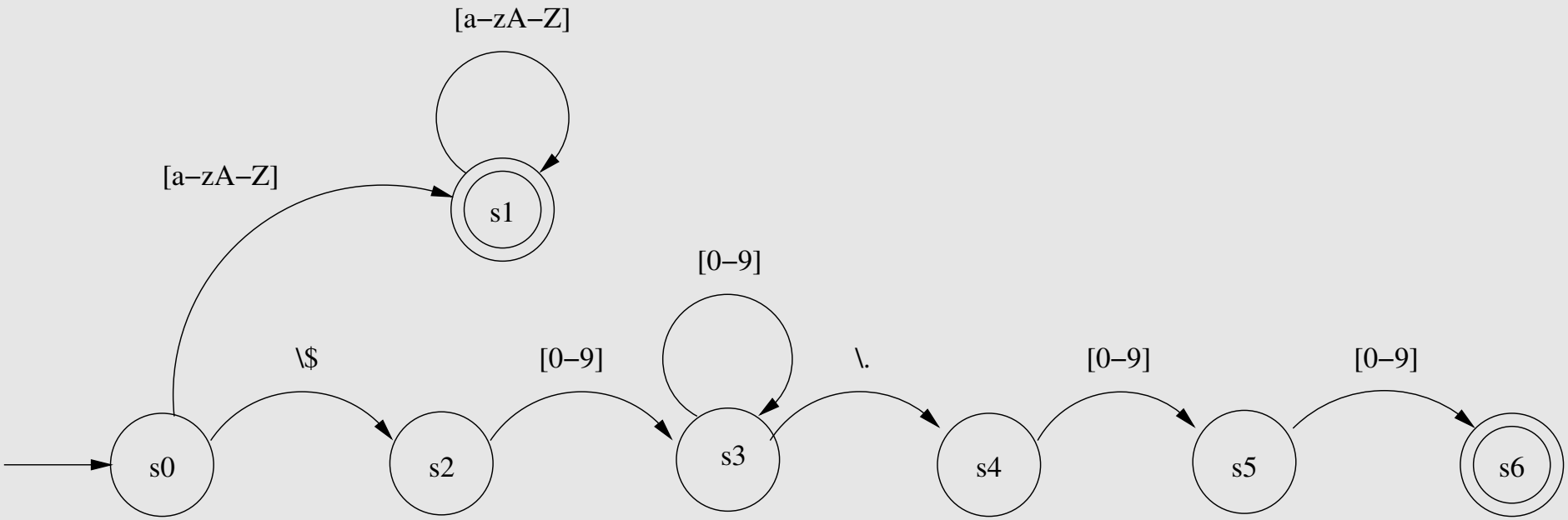


Finite State Automata: Transition Diagrams

- FSAs are frequently represented via **transition diagrams** which are graphs where
  - nodes represent states
  - arcs represent transitions
  - accept states are denoted by double circles
  - the unique start state is indicated by an incoming arrow with no state at its tail

Finite State Automata: Transition Diagrams (cont)

- Here is a transition diagram for an FSA that accepts two types of tokens:
  - ◇ strings of one or more alphabetic characters in lower or uppercase, e.g. `quick`
  - ◇ strings consisting of a dollar sign, one or more digits, a full stop and then exactly two more digits, e.g. `$512.35`



Finite State Automata: Transition Tables

- A transition diagram can easily be transformed into a **transition table** expressing the same information
- For example the previous transition diagram can equally well be represented via the transition table:

	Input			
State	[a-zA-Z]	[0-9]	$	.
0	1	–	2	–
1	1	–	–	–
2	–	3	–	–
3	–	3	–	4
4	–	5	–	–
5	–	6	–	–
6	–	–	–	–

This table tells us for each state and input symbol what the next FSA state is (– indicates transition not defined)

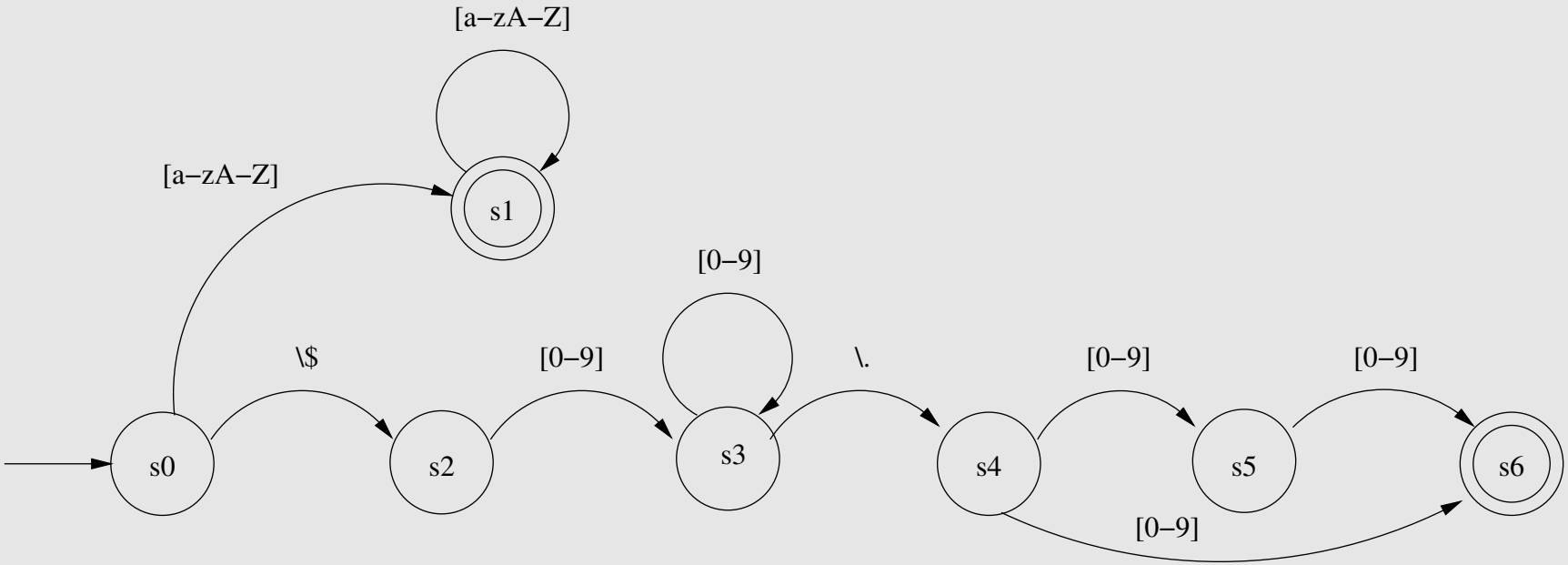
Finite State Automata: A Recognition Algorithm

- Given a transition table representation of a FSA it is easy to write an algorithm that implements the FSA (Jurafsky & Martin, 1st ed. p.37)

```
FSA-Recognize(tape,machine)
index ← beginning of tape
current_state ← initial state of machine
WHILE true do
    IF end of input THEN
        IF current_state is an accept state THEN
            RETURN accept
        ELSE
            RETURN reject
    ELSIF transition_table[current_state,tape[index]] is empty THEN
        RETURN reject
    ELSE
        current_state ← transition_table[current_state,tape[index]]
        index ← index + 1
ENDWHILE
```

Non-Deterministic Finite State Automata

- Suppose we want to extend our previous automaton to recognise dollar amounts with either one or two digits after the full stop (e.g. to recognise expressions like *$1.2 billion*).
- Can conveniently represent this in transition diagram as follows:



- Note, however, that there are now two routes out of s4 for the same input. I.e. when in s4 the automaton is faced with a choice. Such an automaton is called a **non-deterministic** finite state automaton.

Non-Deterministic Finite State Automata

- When testing to see if a non-deterministic FSA accepts a string there are three standard solutions to the problem of which path to pursue when confronted with a choice:
  - ◇ **Backup** Mark the state and position in the input where the choice arises and then pursue one path. If that path cannot be progressed then backup to the most recent choice point and try another path.
  - ◇ **Look-ahead** When confronted with a choice look ahead in the input to help decide which path to take.
  - ◇ **Parallelism** Pursue all alternatives in parallel.
- Algorithms for each of the above approaches exist but are considerably more complex than for a deterministic FSA. Jurafsky and Martin supply a state-space search algorithm for the backup version.

Non-Deterministic Finite State Automata (cont)

- **Theorem from automata theory**: For every non-deterministic FSA there is an equivalent deterministic FSA (one that accepts exactly the same language).
  There is a simple algorithm for converting an NFSA to the equivalent DFSA.
  - ◇ However, for an n-state NFSA the equivalent DFSA may have as many as 2^n states.

Finite State Transducers

- FSA's are a powerful conceptual tool for specifying sets of strings via a mechanism for recognising them.
- Frequently we do not wish simply to recognise strings; we want to record information about them or add information to them that another process can use.
- For example, we do not wish simply to recognize that a string consists of a sequence of tokens, we want to record the token boundaries for a subsequent process to utilise.
- We can view this problem as one of mapping one string (the input string) into another (the input string with token boundaries marked)
- An abstract computational device for performing this mapping is the **finite state transducer (FST)**.

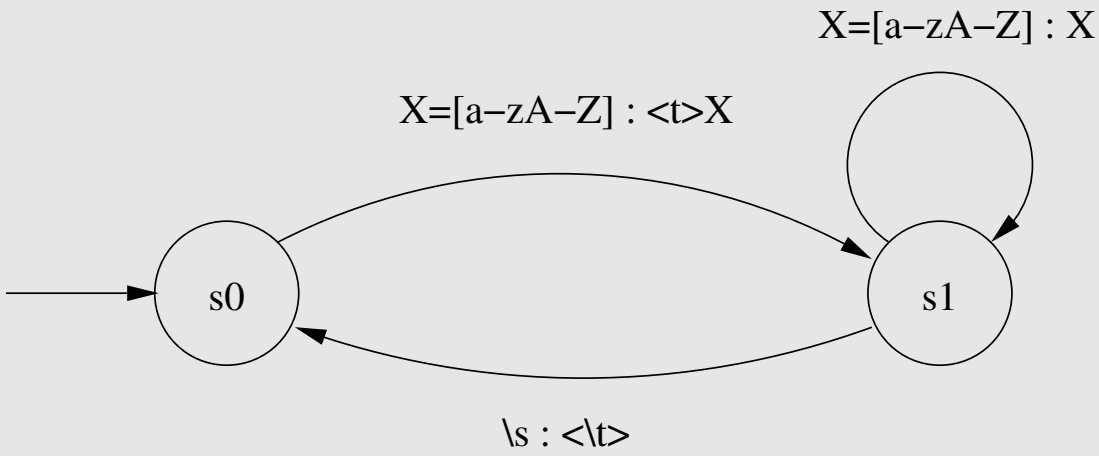
Finite State Transducers (cont)

- An FST may be thought of as a FSA where each arc specifies an output symbol that is produced when the arc is traversed.
- Thus in addition to consuming the symbols on an input tape, an FST may be thought of as writing symbols on an output tape.
- Transition diagrams for FST's are just like those of FSAs/NFSAs, except we now indicate input and output on the arcs using the notation `input : output`

Finite State Transducers (cont)

For example, here is a fragment of a FST:

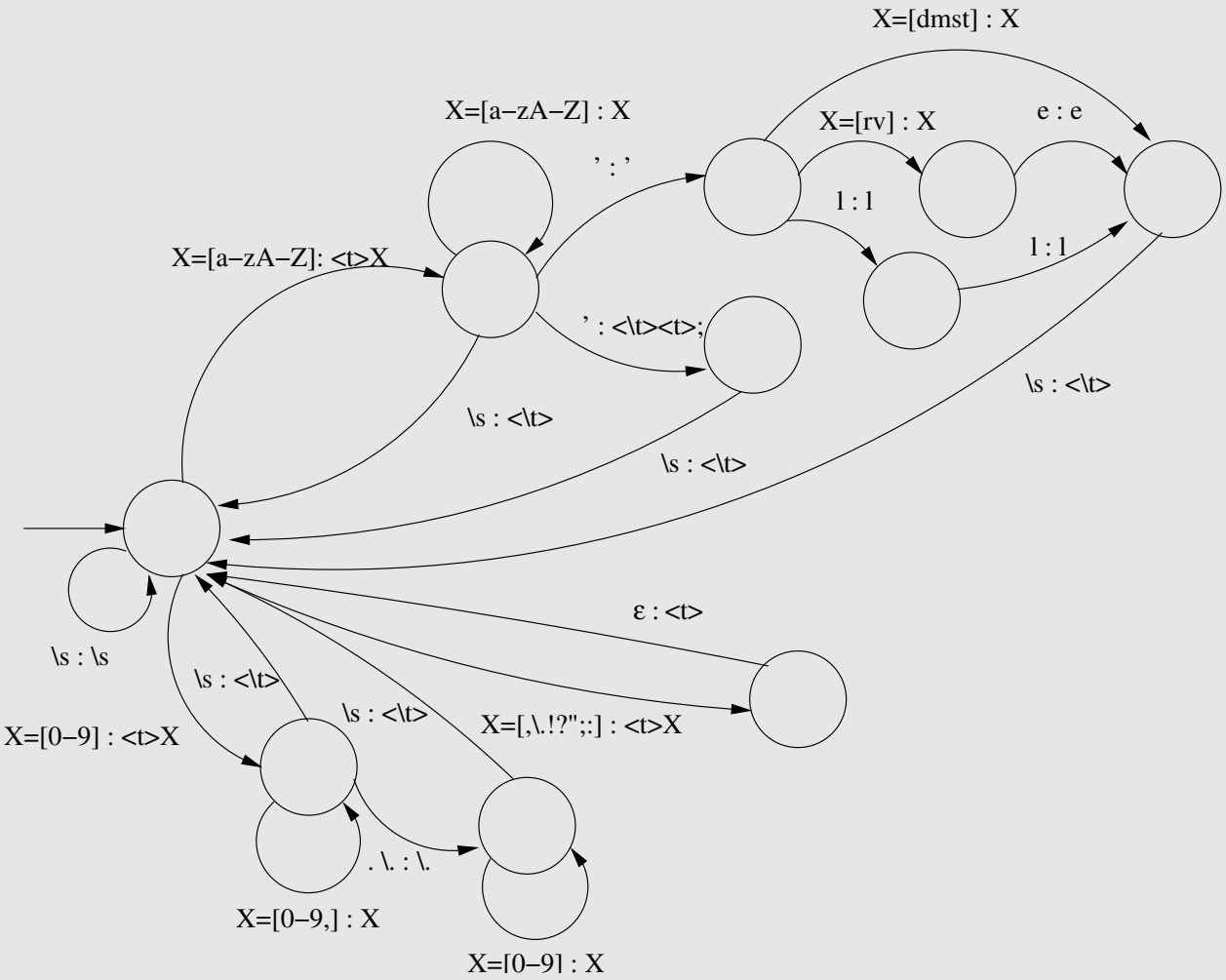
- We use RE notation to abbreviate the set of inputs that would be accepted on an arc.
- We adopt the additional convention of using a variable (`X`) which is set to the input value and then used to repeat it in the output



- This two state machine recognizes strings of white-space separated alphabetic characters and wraps them in XML-like `<t> ... <\t>` tags.

Tokenization Revisited

- FSTs are good mechanisms for designing and implementing NL tokenizers.
- Here is a simplified tokenizer for English text:



Tokenization Revisited

- A number of observations about this tokenizer:
  - ◇ It under-generates: It cannot handle mixed alpha-numerics or alpha strings containing hyphens (3GPiracy, hedge-fund).
  - ◇ It over-generates. For instance it recognises strings like John're.
  - ◇ It ignores case information.
  - ◇ It does not attempt to recognize more complex token types such as dates (e.g. 12/02/07).
  - ◇ It does not add token class information. It could be easily modified to do so.
  - ◇ It has an ϵ transition – one that consumes no input, but allows output to be produced.

Tokenization Revisited (cont)

- Also note this FST is non-deterministic. Thus, to implement it on a serial computer requires a backup or lookahead strategy.
  - ◇ One advantage of look-ahead for FST's is that output operations do not have to be postponed until a correct path through the automaton is discovered – arcs are only traversed when then are known to be on an acceptable path, so output can be generated as they are crossed.
  - ◇ This simple FST only requires look-ahead of 1. This can be compiled into the recognizer to eliminate any requirement for search at run-time. For longer look-aheads this may not be possible.

Summary

- The first step in computational processing of text is **tokenisation** – splitting the character stream into elementary chunks and possibly associating information with these chunks (e.g. token type, case, etc.).
- What/how tokenisation should be carried out depends on
  - the orthographic conventions of the writing system of the language
  - what further processing is to be carried out.
- Tokenisation is
  - relatively easy for languages like English, which use whitespace as a word delimiter
  - relatively challenging for languages like Chinese where there is no explicit word delimiter in the writing system (called the word segmentation problem)
- Finite state automata, more specifically finite state transducers, are good mechanisms to use to implement tokenisers.

Summary (cont)

- Tokenisation can generally be carried using **regular expressions (REs)**, facilities for which can be found in modern programming languages such as Java, Python and Perl.
- REs are a syntax for writing **finite state automata**
- More sophisticated tokenizers can be written using generalised **finite state transducers**, automata which emit outputs as well as accepting inputs.