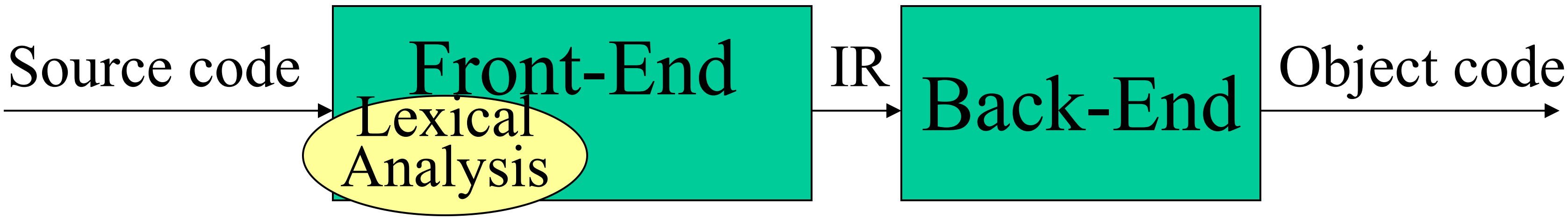
Lecture 3: Introduction to Lexical Analysis



(from last lecture) Lexical Analysis:

- reads characters and produces sequences of tokens.

Today's lecture:

Towards automated Lexical Analysis.

The Big Picture

First step in any translation: determine whether the text to be translated is well constructed in terms of the input language. Syntax is specified with parts of speech - syntax checking matches parts of speech against a grammar.

In natural languages, mapping words to part of speech is idiosyncratic.

In formal languages, mapping words to part of speech is syntactic:

- based on denotation
- makes this a matter of syntax
- reserved keywords are important

What does lexical analysis do?

Recognises the language's parts of speech.

Some Definitions

- A <u>vocabulary (alphabet)</u> is a finite set of <u>symbols</u>.
- A <u>string</u> is any finite sequence of symbols from a vocabulary.
- A <u>language</u> is any set of strings over a fixed vocabulary.
- A <u>grammar</u> is a finite way of describing a language.
- A context-free grammar, G, is a 4-tuple, $G=(S,N,T,P)$, where:
  - S: starting symbol
  - N: set of non-terminal symbols
  - T: set of terminal symbols
  - P: set of production rules
- A language is the set of all terminal productions of G.
- Example (thanks to Keith Cooper for inspiration):
  - S=CatWord; N={CatWord}; T={miau};
  - P={CatWord $\rightarrow$ CatWord miau | miau}

Example

(A simplified version from Lecture2, Slide 6):

$S=E;\ N=\{E,T,F\};\ T=\{+,*,(,),x\}$

$P=\{E\rightarrow T|E+T,\ T\rightarrow F|T*F,\ F\rightarrow (E)|x\}$

By repeated substitution we derive *sentential forms*:

$\underline{E} \Rightarrow \underline{E}+T \Rightarrow \underline{T}+T \Rightarrow \underline{F}+T \Rightarrow x+\underline{T} \Rightarrow x+\underline{T}*F \Rightarrow x+\underline{F}*F$
$\Rightarrow x+x*\underline{F} \Rightarrow x+x*x$

This is an example of a *leftmost derivation* (at each step the leftmost non-terminal is expanded).

To recognise a valid sentence we reverse this process.

- Exercise: what language is generated by the (non-context free) grammar:
  $S=S;\ N=\{A,B,S\};\ T=\{a,b,c\};$
  $P=\{S\rightarrow abc|aAbc,\ Ab\rightarrow bA,\ Ac\rightarrow Bbcc,\ bB\rightarrow Bb,\ aB\rightarrow aa|aaA\}$
  (for the curious: read about Chomsky's Hierarchy)

Why all this?

- Why study lexical analysis?
  - To avoid writing lexical analysers (scanners) by hand.
  - To simplify specification and implementation.
  - To understand the underlying techniques and technologies.
- We want to specify **lexical patterns** (to derive tokens):
  - Some parts are easy:
    - $WhiteSpace \rightarrow blank \mid tab \mid combination\_of\_blank\_and\_tab$
    - Keywords and operators (if, then, =, +)
    - Comments (/* followed by */ in C, // in C++, % in latex, ...)
  - Some parts are more complex:
    - Identifiers (letter followed by - up to *n* - alphanumerics…)
    - Numbers

We need a notation that could lead to an implementation!

Regular Expressions

Patterns form a regular language. A regular expression is a way of specifying a regular language. It is a formula that describes a possibly infinite set of strings.

(*Have you ever tried* **`ls [x-z]*`** *?*)

Regular Expression (RE) (over a vocabulary V):

- ε is a RE denoting the empty set $\{\varepsilon\}$.
- If $a \in$ V then a is a RE denoting $\{a\}$.
- If r_1, r_2 are REs then:
  - r_1* denotes zero or more occurrences of r_1;
  - $r_1 r_2$ denotes concatenation;
  - $r_1 \mid r_2$ denotes either r_1 or r_2;
- **Shorthands**: *[a-d]* for $a \mid b \mid c \mid d$; r^+ for rr^*; $r?$ for $r \mid \varepsilon$

Describe the languages denoted by the following REs

a; *a* | *b*; *a**; *(a | b)**; *(a | b)(a | b)*; *(a*b*)**; *(a | b)*baa*;

(*What about* **`ls [x-z]*`** *above? Hmm… not a good example?*)

Examples

- $integer \rightarrow (+ \mid - \mid \varepsilon)\ (0 \mid 1 \mid 2 \mid \ldots \mid 9)+$
- $integer \rightarrow (+ \mid - \mid \varepsilon)\ (0 \mid 1 \mid 2 \mid \ldots \mid 9)\ (0 \mid 1 \mid 2 \mid \ldots \mid 9)*$
- $decimal \rightarrow integer.(0 \mid 1 \mid 2 \mid \ldots \mid 9)*$
- $identifier \rightarrow [a\text{-}zA\text{-}Z]\ [a\text{-}zA\text{-}Z0\text{-}9]*$

- Real-life application (perl regular expressions):
  - `[+-]?(\d+\.\d+|\d+\.|\.\d+)`
  - `[+-]?(\d+\.\d+|\d+\.|\.\d+|\d+)([eE][+-]?\d+)?`

    (for more information read: **`% man perlre`**)

(Not all languages can be described by regular expressions. But, we don't care for now).

Building a Lexical Analyser by hand

Based on the specifications of tokens through regular expressions we can write a lexical analyser. One approach is to check case by case and split into smaller problems that can be solved *ad hoc*. Example:

```
void get_next_token() {
  c=input_char();
  if (is_eof(c)) { token ← (EOF,"eof"); return}
  if (is_letter(c)) {recognise_id()}
  else if (is_digit(c)) {recognise_number()}
       else if (is_operator(c))||is_separator(c))
            {token ← (c,c)}  //single char assumed
            else {token ← (ERROR,c)}
  return;
}
...
do {
  get_next_token();
  print(token.class, token.attribute);
} while (token.class != EOF);
```

Can be efficient; but requires a lot of work and may be difficult to modify!

Building Lexical Analysers "automatically"

Idea: try the regular expressions one by one and find the longest match:

```
set (token.class, token.length) ←(NULL, 0)
// first
find max_length such that input matches T1→RE1
   if max_length > token.length
        set (token.class, token.length) ←(T1, max_length)
// second
find max_length such that input matches T2→RE2
   if max_length > token.length
        set (token.class, token.length) ←(T2, max_length)
…
// n-th
find max_length such that input matches Tn→REn
   if max_length > token.length
        set (token.class, token.length) ←(Tn, max_length)
// error
if (token.class == NULL) { handle no_match }
```

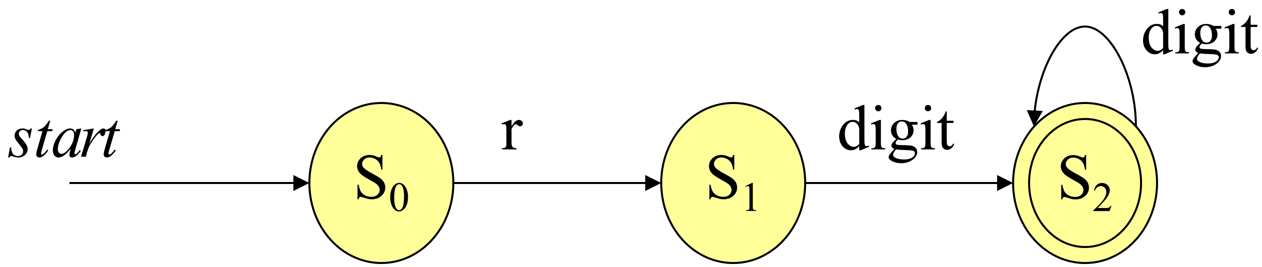
Disadvantage: linearly dependent on number of token classes and requires restarting the search for each regular expression.

We study REs to **automate** scanner construction!

Consider the problem of recognising register names starting with r and requiring at least one digit:

Register $\rightarrow$ *r (0|1|2|…|9) (0|1|2|…|9)** (or, *Register* $\rightarrow$ *r Digit Digit**)

The RE corresponds to a **transition diagram**:



Depicts the actions that take place in the scanner.

- A circle represents a state; S0: start state; S2: final state (double circle)
- An arrow represents a transition; the label specifies the cause of the transition.

A string is accepted if, going through the transitions, ends in a final state (for example, r345, r0, r29, as opposed to a, r, rab)

Towards Automation (finally!)

An easy (computerised) implementation of a transition diagram is a **<u>transition table</u>**: a column for each input symbol and a row for each state. An entry is a set of states that can be reached from a state on some input symbol. E.g.:

state	‘r’	digit
0	1	-
1	-	2
2(final)	-	2

If we know the transition table and the final state(s) we can build directly a recogniser that detects acceptance:

```
char=input_char();
state=0;    // starting state
while (char != EOF) {
    state ← table(state,char);
    if (state == ‘-’) return failure;
    word=word+char;
    char=input_char();
}
if (state == FINAL) return acceptance; else return failure;
```

The Full Story!

The generalised transition diagram is a **<u>finite automaton</u>**. It can be:

- **Deterministic**, DFA; as in the example
- **Non-Deterministic**, NFA; more than 1 transition out of a state may be possible on the same input symbol: think about: *(a | b)* abb*

Every regular expression can be converted to a DFA!

<u>Summary</u>: an introduction to lexical analysis was given.

<u>Next time</u>: More on finite automata and conversions.

<u>Exercise:</u> Produce the DFA for the RE (Q: what is it for?):

$$Register \rightarrow r\ ((0|1|2)\ (Digit|\varepsilon)\ |\ (4|5|6|7|8|9)\ |\ (3|30|31))$$

<u>Reading</u>: <u>Aho2</u>, Sections 2.2, 3.1-3.4. <u>Aho1</u>, pp. 25-29; 84-87; 92-105. <u>Hunter</u>, Chapter 2 (too detailed); Sec. 3.1 -3.3 (too condensed). <u>Grune</u> 1.9; 2.1-2.5. <u>Cooper</u>, Sections 2.1-2.3