Introduction to Lexical Analysis

Scanning and Regular Expressions

Lexical Analysis

Definition:

- reads characters and produces sequences of tokens.

Target:

- Towards automated Lexical Analysis.

First Step

- 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

Lexical Analysis

What does lexical analysis do?

Recognises the language's parts of speech.

Goals of Lexical Analysis

- Convert from physical description of a program into sequence of of **tokens**.
  - Each token represents one logical piece of the source file – a keyword, the name of a variable, etc.
- Each token is associated with a **lexeme**.
  - The actual text of the token: “137,” “int,” etc.
- Each token may have optional **attributes**.
  - Extra information derived from the text – perhaps a numeric value.
- The token sequence will be used in the parser to recover the program structure.

Choosing Tokens

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
    cout << k << endl;
}
```

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
    cout << k << endl;
}
```

```
for        {
int        }
<<         ;
=          <
(          [
)          ]
++
```

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
    cout << k << endl;
}
```

```
for            {
int            }
<<             ;
=              <
(              [
)              ]
++

Identifier
IntegerConstant
```

Choosing Good Tokens

- Very much dependent on the language.
- Typically:
  - Give keywords their own tokens.
  - Give different punctuation symbols their own tokens.
  - Group lexemes representing identifiers, numeric constants, strings, etc. into their own groups.
  - Discard irrelevant information (whitespace, comments)

Scanning is Hard

- FORTRAN: Whitespace is irrelevant

```
DO 5 I = 1,25
DO 5 I = 1.25
```

Thanks to Prof. Alex Aiken

Scanning is Hard

- FORTRAN: Whitespace is irrelevant

```
DO 5 I = 1,25
 DO5I  = 1.25
```

Thanks to Prof. Alex Aiken

Scanning is Hard

- FORTRAN: Whitespace is irrelevant

```
DO 5 I = 1,25
 DO5I  = 1.25
```

- Can be difficult to tell when to partition input.

Thanks to Prof. Alex Aiken

Scanning is Hard

- C++: Nested template declarations

```
vector<vector<int>> myVector
```

Thanks to Prof. Alex Aiken

Scanning is Hard

- C++: Nested template declarations

```
vector < vector < int >> myVector
```

Thanks to Prof. Alex Aiken

Scanning is Hard

- C++: Nested template declarations

```
(vector < (vector < (int >> myVector)))
```

Thanks to Prof. Alex Aiken

Scanning is Hard

- C++: Nested template declarations

```
(vector < (vector < (int >> myVector)))
```

- Again, can be difficult to determine where to split.

Thanks to Prof. Alex Aiken

Scanning is Hard

- PL/1: Keywords can be used as identifiers.

Scanning is Hard

- PL/1: Keywords can be used as identifiers.

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF
```

Thanks to Prof. Alex Aiken

Scanning is Hard

- PL/1: Keywords can be used as identifiers.

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF
```

Scanning is Hard

- PL/1: Keywords can be used as identifiers.

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF
```

- Can be difficult to determine how to label lexemes.

Thanks to Prof. Alex Aiken

Challenges in Scanning

- How do we determine which lexemes are associated with each token?
- When there are multiple ways we could scan the input, how do we know which one to pick?
- How do we address these concerns efficiently?

Some Definitions

- A vocabulary (alphabet) is a finite set of symbols.
- A string is any finite sequence of symbols from a vocabulary.
- A language is any set of strings over a fixed vocabulary.
- A grammar 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*.

Cat Language

- Example:

  S=CatWord;

  N={CatWord};

  T={miau};

  P={CatWord $\rightarrow$ CatWord miau | miau}

Example:

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

➔Use left most derivation

To derive the expression: X + X * X.

Validation

- To recognise a valid sentence we reverse this process.

Exercise:

- what language is generated by the (non-context free) grammar:

  $S = S;$

  $N = \{A, B, S\};$

  $T = \{a, b, c\};$

  $P = \{S \rightarrow abc \mid aAbc,$

  $Ab \rightarrow bA,$

  $Ac \rightarrow Bbcc,$

  $bB \rightarrow Bb,$

  $aB \rightarrow aa \mid aaA\}$

  (for the curious: read about Chomsky's Hierarchy)

Why study lexical analysis?

- To avoid writing lexical analysers (scanners) by hand.
- To simplify specification and implementation.
- To understand the underlying techniques and technologies.

Why study lexical analysis?

- We want to specify **lexical patterns** (to derive tokens):
  - Some parts are easy:
    - *WhiteSpace* $\rightarrow$ *blank | tab | WhiteSpace blank | WhiteSpace 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.

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 | r_2 denotes either r_1 or r_2;

Regular Expressions

- **Shorthands**:
  - *[a-d]* for *a | b | c | d*;
  - r^+ for rr^*;
  - *r?* for $r \mid \varepsilon$

Operator Precedence

- Regular expression operator precedence is

$$(R)$$

$$R^*$$

$$R_1R_2$$

$$R_1 \mid R_2$$

- So **ab*c|d** is parsed as **((a(b*))c)|d**

Simple Regular Expressions

- Suppose the only characters are **`0`** and **`1`**.
- Here is a regular expression for strings containing **`00`** as a substring:

(0 | 1)*00(0 | 1)*

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings containing **00** as a substring:

(0 | 1)*00(0 | 1)*

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings containing **00** as a substring:

(0 | 1)*00(0 | 1)*

11011100101
0000
11111011110011111

Simple Regular Expressions

- Suppose the only characters are **`0`** and **`1`**.
- Here is a regular expression for strings containing **`00`** as a substring:

(0 | 1)*00(0 | 1)*

11011100101
0000
111110111100111111

Simple Regular Expressions

- Suppose the only characters are **`0`** and **`1`**.
- Here is a regular expression for strings of length exactly four:

Simple Regular Expressions

- Suppose the only characters are **`0`** and **`1`**.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

Simple Regular Expressions

- Suppose the only characters are **`0`** and **`1`**.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

0000
1010
<u>1111</u>
1000

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are **`0`** and **`1`**.
- Here is a regular expression for strings of length exactly four:

(0|1){4}

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are **`0`** and **`1`**.
- Here is a regular expression for strings of length exactly four:

(0|1){4}

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are **`0`** and **`1`**.
- Here is a regular expression for strings that contain at most one zero:

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings that contain at most one zero:

$$1^*(0 \mid \varepsilon)1^*$$

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings that contain at most one zero:

1*(0 | ε)1*

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings that contain at most one zero:

1*(0 | ε)1*

11110111
111111
0111
0

Simple Regular Expressions

- Suppose the only characters are **`0`** and **`1`**.
- Here is a regular expression for strings that contain at most one zero:

1*(0 | ε)1*

11110111

111111

0111

0

Simple Regular Expressions

- Suppose the only characters are **`0`** and **`1`**.
- Here is a regular expression for strings that contain at most one zero:

1*0?1*

11110111
111111
0111
0

Applied Regular Expressions

- Suppose our alphabet is **a**, @, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

aa* (.aa*)* @ aa*.aa* (.aa*)*

Applied Regular Expressions

- Suppose our alphabet is **a**, @, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

aa* (.aa*)* @ aa*.aa* (.aa*)*

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, @, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

aa* (.aa*)* @ aa*.aa* (.aa*)*

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, @, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

aa* (.aa*)* @ aa*.aa* (.aa*)*

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, @, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

aa* (.aa*)* @ aa*.aa* (.aa*)*

cs143@cs.stanford.edu

first.middle.last@mail.site.org

barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

$$a^+ (.aa^*)^* @ aa^*.aa^* (.aa^*)^*$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, @, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

$$\mathbf{a}^+ \quad (.\mathbf{a}^+)^* \quad @ \quad \mathbf{a}^+.\mathbf{a}^+ \quad (.\mathbf{a}^+)^*$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, @, and **.**, where **a** represents “some letter.”
- A regular expression for email addresses is

$$a^{+} \quad (.a^{+})^{*} \quad @ \quad a^{+}.a^{+} \quad (.a^{+})^{*}$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

$$\mathbf{a}^+ \quad (\mathbf{.a}^+)^* \quad @ \quad \mathbf{a}^+ \quad (\mathbf{.a}^+)^+$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose our alphabet is **a**, @, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

$$a^+(.a^+)^*@a^+(.a^+)^+$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?(0|1|2|3|4|5|6|7|8|9)*(0|2|4|6|8)

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?(0|1|2|3|4|5|6|7|8|9)*(0|2|4|6|8)

42
+1370
-3248
-9999912

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?(0|1|2|3|4|5|6|7|8|9)*(0|2|4|6|8)

42
+1370
-3248
-9999912

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?[0123456789]*[02468]

42
+1370
-3248
-9999912

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?[0-9]*[02468]

42
+1370
-3248
-9999912

Regular Expressions

Describe the languages denoted by the following REs:

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

Examples

- *integer* $\rightarrow$ *(+ | – | ε) (0 | 1 | 2 | … | 9)+*
- *integer* $\rightarrow$ *(+ | – | ε) (0 | (1 | 2 | … | 9) (0 | 1 | 2 | … | 9)*)*
- *decimal* $\rightarrow$ *integer.(0 | 1 | 2 | … | 9)**
- *identifier* $\rightarrow$ *[a-zA-Z] [a-zA-Z0-9]**

- 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 }
```

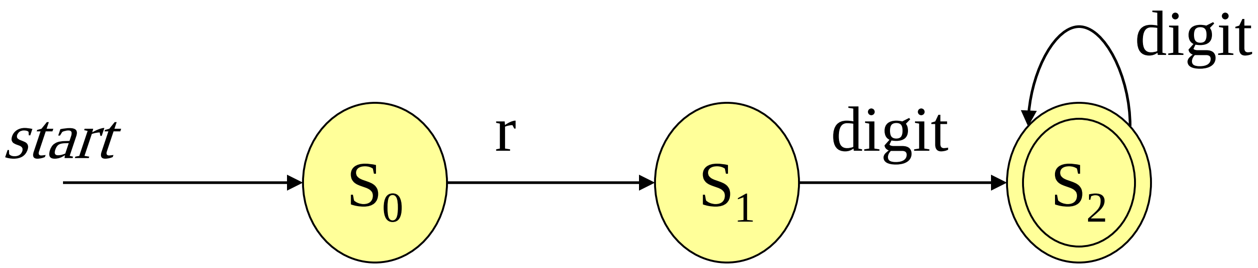
Disadvantage: linearly dependent on number of token classes and requires restarting the search for each regular expression.

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 **transition table**: 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;
```

DFA & NFA

The generalised transition diagram is a **finite automaton**. 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!