Lexical analysis

How to implement lexical analysis (without programming).

LEXICAL ANALYSIS

Specifying lexical analysis

Lexical analyser must recognize and process tokens in input:

1. Grammar:
   - Describes form of all valid tokens
   - Declarative
2. Recognizer:
   - Algorithm for recognizing specified tokens
3. Translation rules:
   - What to do when each token is found

Describing tokens

Lexical analysis views program as sequence of tokens.

Each token is sequence of characters.

Define program using grammar:

```
<program> ::= <token>*

<token> ::= <integer>
          | <real>
          | <identifier>
          | …

<integer> ::= <digit>+

<digit> ::= '0' | '1' | … | '9'

<real> ::= <integer> '.' <integer> <exponent>?
         | <integer> <exponent>

<exponent> ::= 'E' '-'? <integer>

<identifier> ::= …
```

In practice, omit first two rules and distinguish “token” nonterminal symbols from others:

```
<integer> ::= <digit>+

<real> ::= <integer> '.' <integer> <exponent>?
         | <integer> <exponent>

<identifier> ::= …

<digit> ::= '0'|'1'|…|'9'

<exponent> ::= 'E' '-'? <integer>
```

Now **some** nonterminal symbols are tokens. **All** terminal symbols are characters.

Types of grammar for lexical analysis

Regular grammar:

- Good for lexical analysis (only)
- Used by most lexical analyser generators

LL(*k*) grammar:

- More powerful than regular grammar
- Can also be used for syntax analysis
- Used by ANTLR lexical analyser generator

Regular expressions

Regular expressions over alphabet *S*:

ε			(empty string)
a	where *a* in *S*		(single string)
X1 X2			(*X1* followed by *X2*)
X1 | *X2*			(*X1* or *X2*)
*X1**			(zero or more repetitions of *X1*)
X+	means	*X X**	(1 or more repetitions of *X*)
X?	means	*X* | ε	(0 or 1 repetitions of *X*)

where *X1* and *X2* are regular expressions over *S*.

Regular expressions in lexical analysis

Tokens can be described by regular expressions over an alphabet of characters.

Examples:

'>=' operator: `'>' '='`

An integer: `('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')+`

Regular grammars for lexical analysis

Use regular grammar over alphabet of characters (terminal symbols are characters).

Each token is a nonterminal symbol.

May be other nonterminal symbols.

Example regular grammar for lexical analysis

```
<letter> ::= 'A'|'B'|…|'Z'|'a'|'b'|…|'z'

<digit> ::= '0'|'1'|…|'9'

<identifier> ::=
    <letter> (<letter> | <digit>)*

<integer> ::= <digit>+

<real> ::= <integer> '.' <integer> <exponent>?
         | <integer> <exponent>

<exponent> ::= 'E' '-'? <integer>

<less> ::= '<'

<lesseq> ::= '<' '='

<grtr> ::= '>'

<grtreq> ::= '>' '='

<equal> ::= '=' '='

<noteq> ::= '!' '='

<while> ::= 'w' 'h' 'i' 'l' 'e'
```

Building lexical analysers

Stages:

1. Convert regular grammar to *finite(-state) automaton*.
   See COMS11700.

   - FA is a recognizer (algorithm) for the specified language

     Decides whether input string is a valid token of the language.

2. If automaton is nondeterministic, convert to deterministic one (more efficient). See COMS11700.
   https://www.khanacademy.org/computer-programming/deterministic-finite-automata-constructor/4851098534805504

3. Convert automaton to an executable program (e.g., C or Java).

4. Add translation rules.

Token types and values

When token is recognized, return *token type* and *token value*:

- Operators, keywords, etc. have no value.

  E.g.: `(less)`, `(while)`

- Numbers, identifiers, etc. have value:
  - text string: “23”, “3.142”, “count”, *or*
  - numerical value, pointer to symbol table, etc.

How to distinguish keywords from identifiers?

- special productions to handle each keyword before checking for identifier

  E.g.: `<while> → 'w' 'h' 'i' 'l' 'e'`

- *or* look up alphanumeric strings in symbol table

- *or* look up recognized tokens in special table

Lex/Flex

Parser generator that converts regular grammar to a C function:

`yylex():` returns next token read from standard input.

Lex/Flex program structure

%{ *C declarations* %}
regular grammar definitions
%%
translation rules
%%
auxiliary C procedures

Lex translation rules

p_1 $\{action_1\}$

p_2 $\{action_2\}$

...

- Each p_i is a regular expression
- Each $action_i$ is C code

Examples:

```
{ws}          {}
{while}       {return(WHILE);}
{identifier}  {return(IDENTIFIER);}
{integer}     {return(INTEGER);}
```

The `yylex()` function repeatedly:

- Reads longest sequence of characters from standard input that matches one of the p_i
- If more than one matching p_i, use the *first* one
- Execute $action_i$ (might return)
- Returns token type as function value
- Returns token value in global variable `yylval`

```
real     {integer}\.{integer}{opt_exp}
%%
{real}
         {sscanf(yytext,"%lf",&yylval);
          return(REAL_NUMBER);}
%%
```

LL(*k*) lexical analysis

LL(*k*) grammars can also be used for lexical analysis.
Again, terminal symbols are characters. E.g.:

```
<integer> ::= <digit><int1>

<int1> ::= <int1> <digit>
<int1> ::=

<real> ::=  <integer> '.' <integer>
<real> ::=  <integer> '.' <integer> <exponent>
<real> ::=  <integer> <exponent>

<identifier> ::= …

<less> ::= '<'

<lesseq> ::= '<' '='

<digit> ::= '0'
…
<digit> ::= '9'

<exponent> ::= 'E' <integer>
<exponent> ::= 'E' '-' <integer>
```

Productions may be abbreviated using `|`, `*`, `+`, `?` as usual.

LL(*k*) grammars

A grammar is LL(*k*) if the next *k* terminal symbols *predict* a unique production.

So for lexical analysis:

> the right side of each production defining a nonterminal must begin with a different sequence of *k* characters.

Lookahead

Example grammar is not LL(1) because:

- Two productions for `<exponent>` begin with `'E'`
- Two productions for `<token>` begin with `'<'`

(Partial) solution: increase lookahead to 2.

Left recursion

Example grammar is not LL(k) for any k because definition of `<int1>` uses left recursion

```
<int1> → <int1> <digit>
<int1> →
```

Solution: replace left recursion by right recursion:

```
<int1> → <digit> <int1>
<int1> →
```

Left factoring

Example grammar is still not LL(*k*) for any *k* because four productions for `<token>` begin with a digit:

```
<integer> → <digit> <int1>

<real> →  <integer> '.' <integer>
<real> →  <integer> '.' <integer> <exponent>
<real> →  <integer> <exponent>
```

Solution: left factoring.

```
<number> →  <digit> <int1> <number1>

<number1> →
<number1> → '.' <digit> <int1> <number2>
<number1> → <exponent>

<number2> →
<number2> → <exponent>

<int1> → <digit> <int1>
<int1> →
```

ANTLR grammar definition

Uses usual EBNF syntax except:

- Colon ‘:’ in productions
- Nonterminals in upper case
- Terminals (characters) in quotes: ' for characters and strings
- Tokens distinguished from other nonterminal symbols: others are marked “`fragment`”

Example: extract from ANTLR version of grammar:

```
LESS        : '<' ;
LESSEQ      : '<=' ;
GRTR        : '>' ;
GRTREQ      : '>=' ;
NUMBER      : INT ( ( '.' INT ( EXPONENT )? | EXPONENT ) )? ;

fragment
EXPONENT    : 'e' ('-')? INT ;

fragment
INT         : ('0'..'9')+ ;
```

ANTLR translation rules

- Semantic actions and semantic predicates
- Enclosed in {braces}
- Embedded in ANTLR grammar at appropriate places

Syntax analysis

Simple method of syntax analysis:

top-down parsing by recursive descent.

Top-down and bottom-up parsing

Top-down (LL) parsers:

- Produce leftmost derivation.
- Work from top (root) of parse tree downwards.
- Decide early (by first *k* symbols) which production to use.

Bottom-up (LR) parsers:

- Produce rightmost derivation.
- Work from bottom (leaves) of parse tree upwards.
- Decide late which production to use by keeping track of all.

Top-down parsing

Basic idea:

- Try to expand start symbol using some production that matches input string.
- Try to expand each nonterminal symbol in right side of production, etc.
- In general, requires backtracking.

$$E \rightarrow M$$
$$E \rightarrow E + M$$
$$M \rightarrow F$$
$$M \rightarrow M * F$$
$$F \rightarrow x$$
$$F \rightarrow y$$
$$F \rightarrow (E)$$

E.g., input = “(x)+y”

$$\underline{E} \Rightarrow_1 \underline{M} \Rightarrow_1 \underline{F} \Rightarrow_3 (\ \underline{E}\ ) \Rightarrow_1 (\ \underline{M}\ ) \Rightarrow_1 (\ \underline{F}\ ) \Rightarrow_1 (\mathtt{x})$$

Backtrack!

$$\underline{E} \Rightarrow_2 \underline{E} + M \Rightarrow_1 \underline{M} + M \Rightarrow_1 \underline{F} + M \Rightarrow_3 (\ \underline{E}\ ) + M \Rightarrow_1$$
$$(\ \underline{M}\ ) + M \Rightarrow_1 (\ \underline{F}\ ) + M \Rightarrow_1 (\mathtt{x}) + \underline{M} \Rightarrow_1 (\mathtt{x}) + \underline{F} \Rightarrow_2 (\mathtt{x})+\mathtt{y}$$

To avoid backtracking, we need *predictive* parsing: predict which production to use by looking at the next terminal symbol(s) from the input. This is possible if we use LL(k) grammars.

Example: translate previous grammar to LL(k) form:

$$
\begin{aligned}
E &\rightarrow M \\
E &\rightarrow E + M \\
M &\rightarrow F \\
M &\rightarrow M * F \\
F &\rightarrow x \\
F &\rightarrow y \\
F &\rightarrow (\, E \,)
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
E &\rightarrow M\, E' \\
E' &\rightarrow +\, E \\
E' &\rightarrow \\
M &\rightarrow F\, M' \\
M' &\rightarrow *\, M \\
M' &\rightarrow \\
F &\rightarrow x \\
F &\rightarrow y \\
F &\rightarrow (\, E \,)
\end{aligned}
$$

Recursive descent top-down parsing

LL(1) parser can be written simply as a set of recursive functions:

$$E \rightarrow M\ E'$$
$$E' \rightarrow +\ E$$
$$E' \rightarrow$$
$$M \rightarrow F\ M'$$
$$M' \rightarrow *\ M$$
$$M' \rightarrow$$
$$F \rightarrow x$$
$$F \rightarrow y$$
$$F \rightarrow (\ E\ )$$

```
void E() { M(); E1(); }

void E1() {
  if (next == '+')
    { skip('+'); E(); }
}

void M() { F(); M1(); }

void M1() {
  if (next == '*')
    { skip('*'); M(); }
}

void F() {
  if (next == 'x') { skip('x'); }
  else if (next == 'y') { skip('y'); }
  else if (next == '(')
    { skip('('); E(); skip(')'); }
}
```

```
void skip(int ch) {
  if (next == ch)
    { next = in.read(); }
  else { error(); }
}
```

How does it work?

- Use current terminal symbol(s) to choose production
- Then skip matching terminal symbols and recursively parse nonterminal symbols in production
- How to use terminal symbol to decide which production to use?