Lexical Analysis (Scanning)

Translates a stream of characters to a stream of tokens

\[ f \circ \circ \omega = a + b \bar{a} (2, \bar{\omega} \omega) ; \]

<table>
<thead>
<tr>
<th>Token</th>
<th>Lexemes</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQUALS</td>
<td>=</td>
<td>an equals sign</td>
</tr>
<tr>
<td>PLUS</td>
<td>+</td>
<td>a plus sign</td>
</tr>
<tr>
<td>ID</td>
<td>a foo bar</td>
<td>letter followed by letters or digits</td>
</tr>
<tr>
<td>NUM</td>
<td>0 42</td>
<td>one or more digits</td>
</tr>
</tbody>
</table>

Lexical Analysis

Goal: simplify the job of the parser.
Scanners are usually much faster than parsers.
Discard as many irrelevant details as possible (e.g., whitespace, comments).
Parser does not care that the identifier is “supercalifragilisticexpialidocious.”
Parser rules are only concerned with tokens.

Describing Tokens

**Alphabet:** A finite set of symbols
Examples: \{ 0, 1 \}, \{ A, B, C, \ldots, Z \}, ASCII, Unicode

**String:** A finite sequence of symbols from an alphabet
Examples: \( \varepsilon \) (the empty string), Stephen, \( \alpha \beta \gamma \)

**Language:** A set of strings over an alphabet
Examples: \( \emptyset \) (the empty language), \{ 1, 11, 111, 1111 \}, all English words, strings that start with a letter followed by any sequence of letters and digits

Operations on Languages

Let \( L = \{ \varepsilon, wo \} \), \( M = \{ \text{man, men} \} \)

**Concatenation:** Strings from one followed by the other
\( LM = \{ \text{man, men, woman, women} \} \)

**Union:** All strings from each language
\( L \cup M = \{ \varepsilon, wo, \text{man, men} \} \)

**Kleene Closure:** Zero or more concatenations
\( M^* = \{ \varepsilon, M, MM, MMM, \ldots \} = \{ \varepsilon, \text{man, men, manman, manmen, menman, menmen, manmanmen, manmenman, manmenman, manmenman, \ldots} \} \)

Kleene Closure

The asterisk operator (*) is called the Kleene Closure operator after the inventor of regular expressions, Stephen Cole Kleene, who pronounced his last name “CLAY-nee.”

His son Ken writes “As far as I am aware this pronunciation is incorrect in all known languages. I believe that this novel pronunciation was invented by my father.”

Regular Expressions over an Alphabet \( \Sigma \)

A standard way to express languages for tokens.

1. \( \varepsilon \) is a regular expression that denotes \( \{ \varepsilon \} \)
2. If \( a \in \Sigma \), \( a \) is an RE that denotes \( \{ a \} \)
3. If \( r \) and \( s \) denote languages \( L(r) \) and \( L(s) \),
   - \( (r)(s) \) denotes \( L(r) \cup L(s) \)
   - \( (r)s \) denotes \( \{ tu : t \in L(r), u \in L(s) \} \)
   - \( r^* \) denotes \( \cup_{i=0}^{\infty} L^i (L^0 = \{ \varepsilon \} \) and \( L^i = LL^{i-1} \))

Regular Expression Examples

<table>
<thead>
<tr>
<th>RE</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( { a } )</td>
</tr>
<tr>
<td>( (a)b )</td>
<td>( { aa, ab, ba, bb } )</td>
</tr>
<tr>
<td>( a^* )</td>
<td>( { \varepsilon, a, aa, aaa, aaaa, \ldots } )</td>
</tr>
<tr>
<td>( (a)b^* )</td>
<td>( { \varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \ldots } )</td>
</tr>
<tr>
<td>( a[a^*b] )</td>
<td>( { a, b, ab, aab, aaab, aaaaab, \ldots } )</td>
</tr>
</tbody>
</table>
Specifying Tokens with REs

Typical choice: \( \Sigma = \text{ASCII characters, i.e.,} \), \( \{ , !, \#, \ldots, 0, 1, \ldots, 9, \ldots, A, \ldots, Z, \ldots, \} \)

letters: AB \cdots \{a \cdots z\}
digits: 01 \cdots 9

identifier: letter ( letter | digit )

Implementing Scanners Automatically

Regular Expressions (Rules)

Nondeterministic Finite Automata

Subset Construction

Deterministic Finite Automata

Tables

The Language induced by an NFA

An NFA accepts an input string \( x \) iff there is a path from the start state to an accepting state that "spells out" \( x \).

Translating REs into NFAs

Regular Expressions (Rules)

Example: translate \((a|b)*abb\) into an NFA

Simulating NFAs

Problem: you must follow the "right" arcs to show that a string is accepted. How do you know which arc is right?

Solution: follow them all and sort it out later.

"Two-stack" NFA simulation algorithm:
1. Initial states: the \( \epsilon \)-closure of the start state
2. For each character \( c \),
   - New states: follow all transitions labeled \( c \)
   - Form the \( \epsilon \)-closure of the current states
3. Accept if any final state is accepting

Nonstochastic Finite Automata

"All strings containing an even number of 0's and 1's"

\[
\begin{array}{ccc}
\text{state} & \epsilon & 0 \\
A & - & B \\
B & - & A \\
C & - & D \\
D & - & C \\
\end{array}
\]

Start state \( s_0 : \epsilon \)

Set of accepting states \( F : \{ A \} \)

Simulating an NFA: \(-aabb, \text{Start}\)

Simulating an NFA: \(-aabb, \epsilon\)-closure
Simulating an NFA: $aabb$

Simulating an NFA: $aabb$, $\epsilon$-closure

Simulating an NFA: $aa\cdot bb$

Simulating an NFA: $aa\cdot bb$, $\epsilon$-closure

Simulating an NFA: $aab\cdot b$

Simulating an NFA: $aab\cdot b$, $\epsilon$-closure

Simulating an NFA: $aabb$

Simulating an NFA: $aabb$, Done

Deterministic Finite Automata

Restricted form of NFAs:
- No state has a transition on $\epsilon$
- For each state $s$ and symbol $a$, there is at most one edge labeled $a$ leaving $s$.

Differs subtly from the definition used in COMS W3261 (Sipser, Introduction to the Theory of Computation)

Very easy to check acceptance: simulate by maintaining current state. Accept if you end up on an accepting state. Reject if you end on a non-accepting state or if there is no transition from the current state for the next symbol.
Deterministic Finite Automata

ELSE: "else" ;
ELSEIF: "elseif" ;

Building a DFA from an NFA

Subset construction algorithm
Simulate the NFA for all possible inputs and track the states that appear.
Each unique state during simulation becomes a state in the DFA.

Subset construction for \((a|b)^*abb\) (1)

Subset construction for \((a|b)^*abb\) (2)

Subset construction for \((a|b)^*abb\) (3)

Subset construction for \((a|b)^*abb\) (4)

Subset Construction
An DFA can be exponentially larger than the corresponding NFA.
\(n\) states versus \(2^n\)
Tools often try to strike a balance between the two representations.
ANTLR uses a different technique.

The ANTLR Compiler Generator
Language and compiler for writing compilers
Running ANTLR on an ANTLR file produces Java source files that can be compiled and run.
ANTLR can generate
- Scanners (lexical analyzers)
- Parsers
- Tree walkers
ANTLR File for a Simple Scanner

```java
class CalcLexer extends Lexer;

LPAREN : '(' ;   // Rules for punctuation
RPAREN : ')' ;
STAR : '*' ;
PLUS : '+' ;
SEMI : ';' ;

protected
    DIGIT : '0'..'9' ;   // Any character between 0 and 9

INT : (DIGIT)+ ;   // One or more digits

WS : (' ' | '	' | '
' | '') ;   // Whitespace
    { $setType(Token.SKIP); } ;   // Action: ignore
```

ANTLR Specifications for Scanners

- Rules are names starting with a capital letter.
- A character in single quotes matches that character.
- A string in double quotes matches the string.
- A vertical bar indicates a choice:
- Asterisk and plus match “zero or more,” “one or more.”

Free-Format Languages

Typical style arising from scanner/parser division

Program text is a series of tokens possibly separated by whitespace and comments, which are both ignored.

- Keywords (if while)
- Punctuation (, ( +)
- Identifiers (foo bar)
- Numbers (10 -3.14159e+32)
- Strings (“A String”)

Python

The Python scripting language groups with indentation

```python
i = 0
while i < 10:
    i = i + 1
    print i  # Prints 1, 2, ..., 10

i = 0
while i < 10:
    i = i + 1
print i   # Just prints 10
```

This is succinct, but can be error-prone.

How do you wrap a conditional around instructions?

Syntax and Language Design

Does syntax matter? Yes and no

More important is a language's semantics—its meaning.

The syntax is aesthetic, but can be a religious issue.

But aesthetics matter to people, and can be critical.

Verbosity does matter: smaller is usually better.

Too small can be a problem: APL is a compact, cryptic language with its own character set (!)

```apl
E←A TEST B;L
L←0.5
E←((A×A)+B×B)+L
```

Syntax and Language Design

Some syntax is error-prone. Classic FORTRAN example:

```fortran
DO 5 I = 1,25 ! Loop header (for i = 1 to 25)
DO 5 I = 1.25 ! Assignment to variable DO5I
```

FORTRAN 77

FORTRAN 77 is not free-format. 72-character lines:

```
100 IF(IN .EQ. 'Y' .OR. IN .EQ. 'y' .OR. $ IN .EQ. 'T' .OR. IN .EQ. 't') THEN

```

When column 6 is not a space, line is considered part of the previous.

Fixed-length line works well with a one-line buffer.

Makes sense on punch cards.

FORTRAN Specifications

- Question mark makes a clause optional.
- PERSON : ("wo")? 'm' ('a'|'e') 'n' ;
  (Matches man, men, woman, and women.)
- Double dots indicate a range of characters:
  - DIGIT : '0'..'9';
  - Asterisk and plus match “zero or more,” “one or more.”
  - ID : LETTER (LETTER | DIGIT)* ;
  - NUMBER : (DIGIT)+ ;

```java
ANTLR Specifications

- DIGIT : '0'..'9';
- Any character between 0 and 9
- INT : (DIGIT)+ ;
- One or more digits
```
**Parsing**

Objective: build an abstract syntax tree (AST) for the token sequence from the scanner.

\[ 2 \times 3 + 4 \Rightarrow \frac{4}{\frac{2}{3}} \]

Goal: discard irrelevant information to make it easier for the next stage.

Parentheses and most other forms of punctuation removed.

**Grammars**

Most programming languages described using a context-free grammar.

Compared to regular languages, context-free languages add one important thing: recursion.

Recursion allows you to count, e.g., to match pairs of nested parentheses.

Which languages do humans speak? I'd say it's regular: I do not not not not not not not not not not understand this sentence.

**Languages**

Regular languages \((t \text{ is a terminal}):\)

\[ A \rightarrow t_1 \ldots t_n B \]

\[ A \rightarrow t_1 \ldots t_n \]

Context-free languages \((P \text{ is terminal or a variable}):\)

\[ A \rightarrow P_1 \ldots P_n \]

Context-sensitive languages:

\[ \alpha_1 A \alpha_2 \rightarrow \alpha_1 B \alpha_2 \]

“\( B \rightarrow A \) only in the ‘context’ of \( \alpha_1 \ldots \alpha_2 \)”

**Issues**

Ambiguous grammars

Precedence of operators

Left- versus right-recursive

Top-down vs. bottom-up parsers

Parse Tree vs. Abstract Syntax Tree

**Ambiguous Grammars**

A grammar can easily be ambiguous. Consider parsing

\[ 3 - 4 \times 2 + 5 \]

with the grammar

\[ e \rightarrow e + e | e - e | e \times e | e / e | N \]

**Operator Precedence and Associativity**

Operator Precedence

Defines how “sticky” an operator is.

\[ 1 \times 2 + 3 \times 4 \]

* at higher precedence than +:

\[ (1 \times 2) + (3 \times 4) \]

+ at higher precedence than *:

\[ 1 \times (2 + 3) \times 4 \]

**Associativity**

Whether to evaluate left-to-right or right-to-left

Most operators are left-associative

\[ 1 - 2 - 3 - 4 \]

left associative

\[ 1 - (2 - (3 - 4)) \]

right associative
Fixing Ambiguous Grammars

Original ANTLR grammar specification

```plaintext
expr  :  expr '+'  expr |
      |  expr '-'  expr |
      |  expr '*'  expr |
      |  expr '/'  expr |
      |  NUMBER        
;
```

Ambiguous: no precedence or associativity.

Assigning Precedence Levels

Split into multiple rules, one per level

```plaintext
expr  :  expr '+'  expr |
      |  expr '-'  expr |
      |  expr '*'  expr |
      |  expr '/'  expr |
      |  NUMBER        
;
```

Assigning Associativity

Make one side or the other the next level of precedence

```plaintext
expr  :  expr '+'  term |
      |  expr '-'  term |
      |  term         
;
```

Parsing Context-Free Grammars

There are $O(n^3)$ algorithms for parsing arbitrary CFGs, but most compilers demand $O(n)$ algorithms.

Fortunately, the LL and LR subclasses of CFGs have $O(n)$ parsing algorithms. People use these in practice.

```
expr  :  expr '+'  term |
      |  expr '-'  term |
      |  term         
;
term  :  term '*'  term |
      |  term '/'  term |
      |  atom         
;
atom  :  NUMBER        
```

Still ambiguous: associativity not defined

Parsing LL(k) Grammars

LL: Left-to-right, Left-most derivation

```plaintext
stmt  :  'if'  expr  'then'  expr  |
      |  'while'  expr  'do'  expr |
      |  expr         
;
```

Implementing a Top-Down Parser

```plaintext
stmt() {
    switch (next-token) {
    case IF:arnation;)
    switch (next-token) {
        case IF: match(IF); expr(); match(THEN); expr(); break;
        case WHILE: match(WHILE); expr(); match(DO); expr(); break;
        case NUMBER or LPAR: expr(); match(COLEQ); expr(); break;
    }
}
```

Writing LL(k) Grammars

Cannot have left-recursion

```plaintext
expr  :  expr '+'  term |
      |  term         
;
```

becomes

```
AST expr() {
    switch (next-token) {
    case NUMBER : expr(); /* Infinite Recursion */
    }
}
```

Writing LL(1)Grammars

```
expr  :  ID '(' expr ')' |
      |  ID '=' expr |
      | ID          
;
```

becomes

```
expr() {
    switch (next-token) {
    case ID: match(ID); match(LPAR); expr(); match(RPAR); break;
    case ID: match(ID); match(EQUALS); expr(); break;
    }
}
```

Eliminating Common Prefixes

Consolidate common prefixes:

```
expr  :  expr '+'  term |
      |  expr '-'  term |
      |  term         
;
```

becomes

```
expr  :  expr ('+' term | '-' term ) |
      |  term         
;
```
Eliminating Left Recursion

Understand the recursion and add tail rules

\[
\begin{align*}
\text{expr} & : \text{expr} ('+\text{ term} | '-\text{ term} ) \\
& | \text{term} \\
\end{align*}
\]
becomes

\[
\begin{align*}
\text{expr} & : \text{term} \text{ exprt} ; \\
\text{exprt} & : '+' \text{ term} \text{ exprt} \\
& | '-' \text{ term} \text{ exprt} \\
& | /* \text{ nothing */}
\end{align*}
\]

Using ANTLR's EBNF

ANTLR makes this easier since it supports * and -:

\[
\begin{align*}
\text{expr} & : \text{expr} '+' \text{ term} \\
& | \text{expr} '-' \text{ term} \\
& | \text{term} ;
\end{align*}
\]
becomes

\[
\begin{align*}
\text{expr} & : \text{term} ('+\text{ term} | '-\text{ term})* ;
\end{align*}
\]

The Dangling Else Problem

Who owns the else?

\[
\begin{align*}
\text{if (a) if (b) c(); else d();}
\end{align*}
\]

Grammars are usually ambiguous; manuals give disambiguating rules such as C's:

As usual the “else” is resolved by connecting an else with the last encountered elseless if.

The Dangling Else Problem

Some languages resolve this problem by insisting on nesting everything.

E.g., Algol 68:

\[
\begin{align*}
\text{if a < b then a else b fi};
\end{align*}
\]

“fi” is “if” spelled backwards. The language also uses do-od and case-esac.

The Dangling Else Problem

Statement separators/terminators

C uses ; as a statement terminator.

\[
\begin{align*}
\text{if (a<b) printf("a less");}
\text{else }
\text{
  printf("b"); printf(" less");}
\}
\]

Pascal uses ; as a statement separator.

\[
\begin{align*}
\text{if a < b then writeln('a less')}
\text{else begin}
\text{  write('a'); writeln(' less')}
\text{end}
\]

Pascal later made a final ; optional.

Rightmost Derivation

1:  \( e \rightarrow t + e \)
2:  \( e \rightarrow t \)
3:  \( t \rightarrow \text{Id} * t \)
4:  \( t \rightarrow \text{Id} \)

A rightmost derivation for \( \text{Id} \rightarrow \text{Id} + \text{Id} \):

\[
\begin{align*}
\text{if } t + \text{Id} \\
\text{if } t + \text{Id} \\
\text{if } \text{Id} \text{Id} \\
\text{if } \text{Id} \text{Id} \\
\end{align*}
\]

Basic idea of bottom-up parsing: construct this rightmost derivation backward.

Bottom-up Parsing
Shift-reduce Parsing

1: \( e \rightarrow t + e \) 
2: \( e \rightarrow t \) 
3: \( t \rightarrow Id * t \) 
4: \( t \rightarrow Id \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id + Id + Id</td>
<td>( t \rightarrow t + e )</td>
<td>shift</td>
</tr>
<tr>
<td>Id + Id + Id</td>
<td>( e \rightarrow e )</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>Id + Id + Id</td>
<td>( t \rightarrow Id * t )</td>
<td>shift</td>
</tr>
<tr>
<td>Id + Id + Id</td>
<td>( t \rightarrow Id )</td>
<td>reduce (3)</td>
</tr>
<tr>
<td>Id + Id + Id</td>
<td></td>
<td>reduce (2)</td>
</tr>
<tr>
<td>Id + Id + Id</td>
<td></td>
<td>reduce (1)</td>
</tr>
</tbody>
</table>

LR Parsing

1: \( e \rightarrow t + e \) 
2: \( e \rightarrow t \) 
3: \( t \rightarrow Id * t \) 
4: \( t \rightarrow Id \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id + Id + Id + Id</td>
<td>( t \rightarrow Id )</td>
<td>shift</td>
</tr>
<tr>
<td>Id + Id + Id + Id</td>
<td>( e \rightarrow e )</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>Id + Id + Id + Id</td>
<td>( t \rightarrow Id * t )</td>
<td>shift</td>
</tr>
<tr>
<td>Id + Id + Id + Id</td>
<td>( t \rightarrow Id )</td>
<td>reduce (3)</td>
</tr>
<tr>
<td>Id + Id + Id + Id</td>
<td></td>
<td>reduce (2)</td>
</tr>
<tr>
<td>Id + Id + Id + Id</td>
<td></td>
<td>reduce (1)</td>
</tr>
</tbody>
</table>

LR Parsing

1: \( e \rightarrow t + e \) 
2: \( e \rightarrow t \) 
3: \( t \rightarrow Id * t \) 
4: \( t \rightarrow Id \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id + Id + Id</td>
<td>( t \rightarrow Id )</td>
<td>shift</td>
</tr>
<tr>
<td>Id + Id + Id</td>
<td>( e \rightarrow e )</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>Id + Id + Id</td>
<td>( t \rightarrow Id * t )</td>
<td>shift</td>
</tr>
<tr>
<td>Id + Id + Id</td>
<td>( t \rightarrow Id )</td>
<td>reduce (3)</td>
</tr>
<tr>
<td>Id + Id + Id</td>
<td></td>
<td>reduce (2)</td>
</tr>
<tr>
<td>Id + Id + Id</td>
<td></td>
<td>reduce (1)</td>
</tr>
</tbody>
</table>

Constructing the SLR Parse Table

The states are places we could be in a reverse-rightmost derivation. Let's represent such a place with a dot.

1: \( e \rightarrow t + e \) 
2: \( e \rightarrow t \) 
3: \( t \rightarrow Id * t \) 
4: \( t \rightarrow Id \)

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>( t \rightarrow t )</td>
</tr>
<tr>
<td>s2</td>
<td>( e \rightarrow e )</td>
</tr>
<tr>
<td>s3</td>
<td>( t \rightarrow Id * t )</td>
</tr>
<tr>
<td>s4</td>
<td>( t \rightarrow Id )</td>
</tr>
<tr>
<td>s5</td>
<td>( t \rightarrow Id )</td>
</tr>
<tr>
<td>s6</td>
<td>( e \rightarrow e )</td>
</tr>
<tr>
<td>s7</td>
<td>( t \rightarrow t )</td>
</tr>
</tbody>
</table>

The Punchline

This is a tricky, but mechanical procedure. The parser generators YACC, Bison, Cup, and others (but not ANTLR) use a modified version of this technique to generate fast bottom-up parsers.

You need to understand it to comprehend error messages:

Shift/reduce conflicts are caused by a state like 
Reduce/reduce conflicts are caused by a state like 
\( t \rightarrow Id * t \) 
\( t \rightarrow Id \) 
\( e \rightarrow t + e \) 
\( e \rightarrow t + e \)