Lexical Analysis - An Introduction
The purpose of the front end is to deal with the input language

- Perform a membership test: code $\in$ source language?
- Is the program well-formed (semantically)?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic
The Front End

Scanner

- Maps stream of characters into words
  - Basic unit of syntax
  - \( x = x + y \); becomes
    \[ <\text{id}, x> <\text{eq},=> <\text{id}, x> <\text{pl},+> <\text{id}, y> <\text{sc},;> \]
- Characters that form a word are its \textit{lexeme}
- Its \textit{part of speech} (or \textit{syntactic category}) is called its \textit{token type}
- Scanner discards white space & (often) comments

Speed is an issue in scanning
⇒ use a specialized recognizer
The Front End

Parser
- Checks stream of classified words (parts of speech) for grammatical correctness
- Determines if code is syntactically well-formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

We’ll come back to parsing in a couple of lectures
The Big Picture

- Language syntax is specified with *parts of speech*, not *words*
- Syntax checking matches *parts of speech* against a grammar

1. \( goal \rightarrow expr \)
2. \( expr \rightarrow expr \ op \ term \)
3. \( \mid \ term \)
4. \( term \rightarrow number \)
5. \( \mid \ id \)
6. \( op \rightarrow + \)
7. \( \mid - \)

\[ S = goal \]
\[ T = \{ \text{number, id, +, -} \} \]
\[ N = \{ \text{goal, expr, term, op} \} \]
\[ P = \{ 1, 2, 3, 4, 5, 6, 7 \} \]
The Big Picture

- Language syntax is specified with *parts of speech*, not *words*
- Syntax checking matches *parts of speech* against a grammar

1. \( \text{goal} \rightarrow \text{expr} \)
2. \( \text{expr} \rightarrow \text{expr} \ op \ \text{term} \)
3. \( \quad \mid \ \text{term} \)
4. \( \text{term} \rightarrow \text{number} \)
5. \( \quad \mid \ \text{id} \)
6. \( \text{op} \rightarrow + \)
7. \( \quad \mid - \)

\( S = \text{goal} \)
\( T = \{ \text{number, id, +, -} \} \)
\( N = \{ \text{goal, expr, term, op} \} \)
\( P = \{ 1, 2, 3, 4, 5, 6, 7 \} \)

No words here!

Parts of speech, not words!
The Big Picture

Why study lexical analysis?

- We want to avoid writing scanners by hand
- We want to harness the theory from classes like Automata Theory

Goals:

- To simplify specification & implementation of scanners
- To understand the underlying techniques and technologies

Represent words as indices into a global table

Specifications written as “regular expressions”
So let's start to study it

- **Purpose**: to group input characters (the source program text) into tokens

- **Examples of Tokens**
  - Operators: $= + > \{ == <>$
  - Keywords: *if* while for *int*
  - Identifiers: var1 returnval getinput
  - Numeric literals: 42 3.14159 0xF3
  - Character literals: ‘a’ ‘\n’ ‘\015’
  - String literals: “hello world!” “EECS 6083”

- **Examples of program text which is not a token**
  - White space: space, tab and end-of-lines
  - Comments
Lexical Analysis Example

for (count=1, count<10, count++)

```
for (count = 1, count < 10, count++)
```

for lparen Id ("count") assign_op Const(1) comma Id ("count")

- Functions of the lexical analyzer (scanner)
  - Partition input program into groups of characters corresponding to tokens
  - Attach the corresponding text attributes to tokens when appropriate
  - Eliminate white space and comments
  - Maintain line number
- Requires a formal notation for describing tokens (regular expressions)
- Automatic Scanner Generations tools
  - Lex, flex, Jlex
Describing Tokens

- In English: an excerpt from CoolAid
  - “Integers are non-empty strings of digits 0-9”
  - “Identifiers are strings (other than keywords) consisting of letters, digits, and the underscore character” “…;object identifiers begin with a lower case letter.”

- With regular expressions:
  - \[0|1|2|3|4|5|6|7|8|9\] \([0|1|2|3|4|5|6|7|8|9\]^*
  - \[a-z]\[0-9|a-z|A-Z|_\]^*
Regular Expression Review

- **ε**  The empty string a special string of length 0

- **Regular expression operators**
  - * |  Choice among alternatives (alternation operator)
  - * ·  Concatenation operator (may be omitted, r · s also written as rs)
  - * *  repetition or “closure”

- **Algebraic Properties**
  - * |  is commutative and associative
    - * r | s = s | r
    - * r | (s | t) = (r | s) | t
  - * Concatenation is associative
    - * (rs) t = r (st)
Regular Expression Basics (continued)

- Algebraic Properties (continued)
  - Concatenation distributes over $\mid$
    - $r(s\mid t) = rs \mid rt$
    - $(s\mid t)r = sr \mid tr$
  - $\varepsilon$ is the identity for concatenation
    - $\varepsilon r = r$
    - $r\varepsilon = r$
  - $^*$ is idempotent
    - $r^{**} = r^*$
    - $r^* = (r\varepsilon)^*$
Common Extensions

- r+ one or more of expression r, same as rr*
- rk k repetitions of r
  - \( r^3 = rrr \)
- [rsz] a character from the specified set, a short hand for alternation
  - \([abcdeg] = a \mid b \mid c \mid d \mid e \mid g\)
- \([^rsz]\) set consists of all characters not listed within the brackets
  - \([^\t\n] \)
- r-z range of characters
  - \([0-9a-z]\)
- r? Zero or one copy of expression (used for fields of an expression that are optional)
In Class Example:

- Regular Expression for Representing Months
- Examples of legal inputs
  - January represented as 1 or 01
  - October represented as 10
In Class Example:

- Regular Expression for Representing Months
  - Examples of legal inputs
    - January represented as 1 or 01
    - October represented as 10
  - First Try: `[0|1]|[0-9]` matches all legal inputs?
    - 01, 1, 02, … 11, 12 Yes!
  - matches only legal inputs?
    - 00, 18, 19 No!
In Class Example:

- Regular Expression for Representing Months
  - Second Try: \([1-9]|(0[1-9])|(1[0-2])\)

matches all legal inputs? Yes
1, 2, 3, …, 10, 11, 12, 01, 02, …, 09
matches only legal inputs? Yes
In class example:

- Regular Expression for Floating point numbers
  - Examples of legal inputs
    - 1.0, 0.2, 3.14159, -1.0, 2.7e8, 1.0E-6
In class example:

- Regular Expression for Floating point numbers
  - Building the regular expression
    - Assume
      \[ \text{digit} \rightarrow 0|1|2|3|4|5|6|7|8|9 \]
    - Handle simple decimals such as 1.0, 0.2, 3.14159
      \[ \text{digit}+.\text{digit}+ \]
    - Add an optional sign
      \[ (-|\varepsilon)\text{digit}+.\text{digit}+ \text{ or } -?\text{digit}+.\text{digit}+ \]

This is called a regular name, allows us to easily write down common patterns.

Assumes that a 0 is required before numbers less than 1 and does not prevent extra leading zeros, so numbers such as 0011.0 or 0003.14159 are legal.
In class example (continued):

- Handling the optional exponent
  - Format for the exponent
    \[(E|e)(+|-)?(digit+)\]
  - Adding it as an optional expression to the decimal part
    \[(-|\varepsilon)digit+.digit+((E|e)(+|-)?(digit+))?\]
Deterministic Finite Automata (DFA)

- Set of states (represented by circles) including a start state and one or more accepting states
- A transition (represented by an arrow labeled with a character, $c$) specifies the next state when the input is $c$.
- A finite automata is deterministic if the next state can be uniquely determined based on the current state and the current input character.
What strings are accepted by this DFA?

a, ab, aa, abb, aba
DFA (continued)

- Draw a DFA for the regular language over the alphabet \{a,b\}, the set of all strings containing an even number of b’s.

- Some examples
  - a
  - aaaa
  - bb
  - ababa
  - bbbbb
  - baaaaaaa
  - baaaaaaab

Zero b’s are considered an even number
DFA (continued)

- Draw a DFA for the regular language over the alphabet \{a,b\}, the set of all strings with an even number of b’s.

![DFA Diagram]

- DFA States:
  - State 1: Start state, on reading 'a' go to state 2.
  - State 2: On reading 'b' go to state 1.

- Transitions:
  - On reading 'a' from state 1, move to state 2.
  - On reading 'b' from state 2, move to state 1.

- The DFA accepts strings with an even number of 'b's.
Deterministic Finite Automata (DFA)

- A DFA for signed floating point numbers with no exponent.

Note that no error state is shown, when the next input character does not match any of the transitions out of the current state then an error occurs, except for the final or accepting state. The input alphabet is the ascii character set.
Non-Deterministic Finite Automata (NFA)

- **ε-transition** (represented by an arrow labeled with ε) specifies a transition that occurs without consuming any input. Can be thought of as a *match* of the empty string.
- A *non-deterministic* finite automata may have more than one transition out of the current state for a given input character.
What strings are accepted by this NFA?

$\epsilon$, $a$, $ab$, $aa$, $aba$
NFA (continued)

- Draw an NFA for the regular language over \( \{a,b\} \) of all strings \( a^*b^+ \)
- Some example strings are:
  - b, aaaaab, aabbbbbbbb
Draw an NFA for the regular language over \{a,b\} of all strings \(a^*b^+\).

Some example strings are:
- \(b\), \(aaaaab\), \(aabb\ldots\)