Structure of a Scanner

How does a scanner generation tool like `flex` actually work?

1. An NDFA is generated from each regular expression. Final states are marked according to which rule is used.
2. These NDFAs are combined into a single NDFA.
3. The big NDFA is converted into a DFA.
4. How are final states marked?
5. The final DFA is minimized for efficiency. The DFA is usually represented in code with a state-character array.

Look-ahead in scanners

The “maximal munch” rule says to always return the longest possible token.

But how can the DFA tell if it has the maximal munch?

Usually, just stop at a transition from accepting to non-accepting state. This requires one character of look-ahead.

Is this good enough for any set of tokens?
Parsing

Parsing is the second part of syntax analysis.

We use grammars to specify how tokens can combine. A parser uses the grammar to construct a parse tree with tokens at the leaves.

**Scanner:** Specified with regular expressions, generates a DFA

**Parser:** Specified with context-free grammar, generates a . . .

Generalize or Specialize?

Parsing a CFG deterministically is hard: requires lots of computing time and space.

By (somewhat) restricting the class of CFGs, we can parse much faster.

For a program consisting of $n$ tokens, we want $O(n)$ time, using a single stack, and not too much look-ahead.

Parsing Strategies

**Top-Down Parsing:**
- Constructs parse tree starting at the root
- “Follow the arrows” — carry production rules forward
- Requires predicting which rule to apply for a given nonterminal.
- LL: Left-to-right, Leftmost derivation

**Bottom-Up Parsing:**
- Constructs parse tree starting at the leaves
- “Go against the flow” — apply reduction rules backwards
- Requires
- LR: Left-to-right, Rightmost derivation
Parsing example

Simple grammar

\[
S \rightarrow T \ T \\
T \rightarrow \text{aa} \\
T \rightarrow \text{bb}
\]

Parse the string aabb, top-down and bottom-up.

Top-down parsing

1. Initialize the stack with $S$, the start symbol;
2. while stack and input are both not empty do
3. if top of stack is a terminal then
   4. Match terminal to next token
4. else
5. Pop nonterminal and replace with r.h.s. from a derivation rule
6. Accept iff stack and input are both empty

Make choice on Step 6 by “peeking” ahead in the token stream.

LL(1) Grammars

A grammar is LL(1) if it can be parsed top-down with just 1 token’s worth of look-ahead.

Example grammar

\[
S \rightarrow T \ T \\
T \rightarrow \text{ab} \\
T \rightarrow \text{aa}
\]

Is this grammar LL(1)?
Common prefixes

The common prefix in the previous grammar causes a problem.
In this case, we can “factor out” the prefix:

LL(1) Grammar

\[
S \rightarrow T \ T \\
T \rightarrow a \ X \\
X \rightarrow b \\
X \rightarrow a
\]

Left recursion

The other enemy of LL(1) is left recursion:

\[
S \rightarrow \text{exp} \\
\text{exp} \rightarrow \text{exp} + \text{NUM} \\
\text{exp} \rightarrow \text{NUM}
\]

- Why isn’t this LL(1)?
- How could we “fix” it?

Handling Errors

How do scanning errors occur?
How can we handle them?

How do parsing errors occur?
How can we handle them?

“Real” scanners/parsers also tag everything with filename & line number to give programmers extra help.