Where are we?

1. Define the tokens for the language using regular expressions
   • Natural representation for tokens
   • But difficult to produce a scanner from REs
2. Convert the regular expressions into non-deterministic finite state automata (NFA)
   • Straightforward conversion
   • Can produce a scanner from NFA, but an inefficient one
3. Convert the NFA into deterministic finite state automata (DFA)
   • Straightforward conversion
4. Convert the DFA into an efficient scanner implementation
Why convert to DFAs?

- Because
  - they are equivalent in power to NFAs
  - they are deterministic, which makes them a terrific basis for an efficient implementation of a scanner
NFA => DFA

- Basic problem
  - NFA can choose among alternative paths
    - either $\varepsilon$ transitions or
    - multiple transitions from a state with the same label
  - But a DFA cannot have this kind of choice

- Solution: subset construction
  - In the newly constructed DFA, each state represents a set of states in the NFA, all of which the NFA might be in during its traversal
Subset construction algorithm

_initial step_

- Create start state of new DFA
  - Label it with the set of NFA states that can be reached by ε transitions
    - That is, without consuming any input
    - Think of it as all possible start states in the NFA, since there could be more than one given ε transitions

- Then process this new start state
  - Details in a couple of slides
In groups (and stay there!)

< 1 minute: start state of the new DFA?

Example from
Crafting a Compiler,
Fischer & LeBlanc
Subset construction algorithm

processing a state

- To process a state $S$ in the new DFA with label \( \{s_1, \ldots, s_n\} \)
- For each symbol $x$ in the alphabet
  - Compute the set $T$ of NFA states reached from any of the NFA states $s_1, \ldots, s_n$ by an $x$ transition followed by any number of $\varepsilon$ transitions
  - If $T$ is not empty
    - If there is already a DFA state with $T$ as a label, add a transition labeled $x$ from $S$ to $T$
    - Otherwise create a new DFA state labeled $T$, add a transition labeled $x$ from $S$ to $T$, and then process $T$
Same groups:
apply the algorithm
Subset construction algorithm

*defining final states*

- After the algorithm terminates
- Mark every DFA state as final if *any* of the NFA states in its label is final
Same Groups:
mark final states
Subset construction: notes

- It is provable that this works and produces an equivalent DFA
- This activity can be automated
- Question: What can be said about the number of states in the DFA relative to the NFA?
  - In theory? In practice?
Minimizing DFAs

- There is also an algorithm for minimizing the number of states in a DFA.
- Given an arbitrary DFA, one can find a unique DFA with a minimum number of states that is equivalent to the original DFA.
  - Except for a renaming of the states.
  - Essentially, try to merge states.
Constructing scanners from DFAs

- Use a table-driven scanner
- Write disciplined procedures that encode the DFA
- We’ll talk about both (the first briefly)
- The second approach is used in the PL/0 compiler
  - Because it’s generally easier to handle a few practical issues (and it may be faster)
Table-driven scanner

- Represent the DFA as an adjacency matrix
  - One row per state
  - One column per character in the alphabet
  - Entry is state to transition to

- Mechanically walk the input, taking appropriate transitions
  - Rules for termination remain unchanged

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1,2}</td>
<td>{3,4,5}</td>
<td></td>
</tr>
<tr>
<td>{3,4,5}</td>
<td>{5}</td>
<td>{4,5}</td>
</tr>
<tr>
<td>{4,5}</td>
<td>{5}</td>
<td>{5}</td>
</tr>
<tr>
<td>{5}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Approach 2: procedural

- Define a procedure for each state in the DFA
- Use conditionals to check the input character and then make the appropriate transition
- A transition is a call to the procedure for the next state

```plaintext
procedure {3,4,5} begin
    if nextChar() == 'a'
        call {5}
    elsif nextChar() == 'b'
        call {4,5}
    else
        reject("no transition out of this state")
    end
```
The heart of the PL/0 scanner

*it’s not quite as clean (but it’s not bad!)*

Token ::= Id | Integer | Keyword | Operator | Punct

- Where’s the DFA?
- How come five kinds of tokens and only three branches?
PL/0’s GetIdent method

*In groups, answer these questions*

- Is PL/0 case-sensitive?
- What does `SearchReserved` return?

```cpp
token* scanner::getident() {
    char ident[maxidlength+1];
    int lengthofid = 0;
    while (isalnum(currentch)) {
        ident[lengthofid] = tolower(currentch);
        lengthofid ++;
        getch();
    }
    ident[lengthofid] = '\0';
    return searchreserved(ident);
}
```
**PL/0’s `GetInt` method**

```cpp
Token* Scanner::GetInt() {
    int integer = 0;
    while (isdigit(CurrentCh)) {
        integer = 10 * integer + (CurrentCh - '0');
        GetCh();
    }
    return new IntegerToken(integer);
}
```
PL/0’s GetPunct method

Token* Scanner::GetPunct() {
    Token* T;
    switch (CurrentCh) {
        case ':':
            GetCh();
            if (CondReadCh('=')) {
                T = new Token(GETS);
            } else {
                T = new Token(COLON);
            }
            break;
        case '<':
            GetCh();
            if (CondReadCh('=')) {
                T = new Token(LEQ);
            } else if (CondReadCh('>')){
                T = new Token(NEQ);
            } else {
                T = new Token(LSS);
            }
            break;
        ...
    }
    return T;
}
A few other notes about the scanner

- There is a `Scanner` class
  - There is only one instance of this class
  - This is an example of the *Singleton* design pattern

- The high-level structure we showed has the scanner scan before the parser parses
  - Study the compiler to figure out what really happens

- Make sure (for this and all other phases) to read the interface (the `.h` file) very, very carefully
Language design issues (lexical)

- Most languages are now free-form
  - Layout doesn’t matter
  - Use whitespace to separate tokens, if needed
  - Alternatives include
    - Fortran, Algol68: whitespace is ignored
    - Haskell: use layout to imply grouping

- Most languages now have reserved words
  - Cannot be used as identifiers
  - Alternative: PL/0 has keywords that are treated specially only in certain contexts, but may be used as identifiers, too
Objectives: next lectures

- Understand the theory and practice of parsing
- Describe the underlying language theory of parsing (CFGs, etc.)
- Understand and be able to perform top-down parsing
- Understand bottom-up parsing