Scanning/tokenizing

- Scanning is the process of reading the input characters from the source and transforming them into a stream of recognized tokens
- Scanners can be auto-generated from a grammar (as we're doing with lex in the labs) or crafted by hand
- Hand crafted scanners can be more efficient, and require relatively few updates if the source language is stable
- Auto-generated scanners can be easily/automatically produced, so may be preferable if development speed is more important than run-time efficiency

Microsyntax

- language rules regarding tokenization are a small subset of overall language rules, sometimes called a microsyntax
- As characters are read, they are aggregated into words or tokens, relying on the microsyntax to identify the end of a token or the transition to a new token
- Some tokens will be identified by patterns, others as specific/exact keywords (may be identified by keyword dictionary, or encoded as part of microsyntax)
- Scanners generally implemented to run in time proportional to the number of input characters

Recognizing tokens

- Patterns for a token can be described with a regular expression, and recognized with a corresponding finite state machine
- Given FSMs for each token type, we will eventually build one overall FSM to identify all token types
- That FSM can be optimized and transformed to actual scanner code, either automatically or by hand (we'll look at both approaches)

Finite state machines

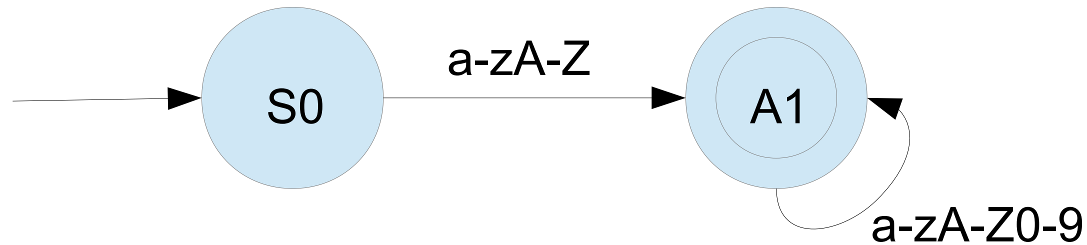
- A recognized alphabet
- A unique start state (for no characters read yet)
- A set of states (matching patterns of characters “en route” to an accept state)
- A transition function: if in state A and we see character C, identifies which state we should switch to
- A subset are accepting states: where we have read a valid token (which accept state tells us which token type)

Example

- FSM to recognized identifiers that begin with an alpha, followed by any number of alphanumerics
- Note we want the fsm to continue reading/aggregating chars as long as we stay in an accept state
- If token types overlap (e.g. Identifiers and keywords) the code at the accept state can be adjusted to return the highest priority token type (e.g. the keyword)

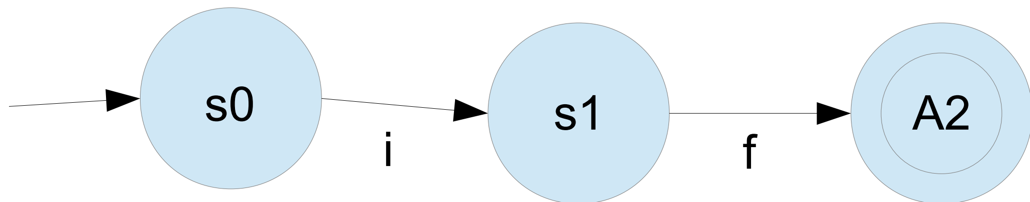
Example FSM

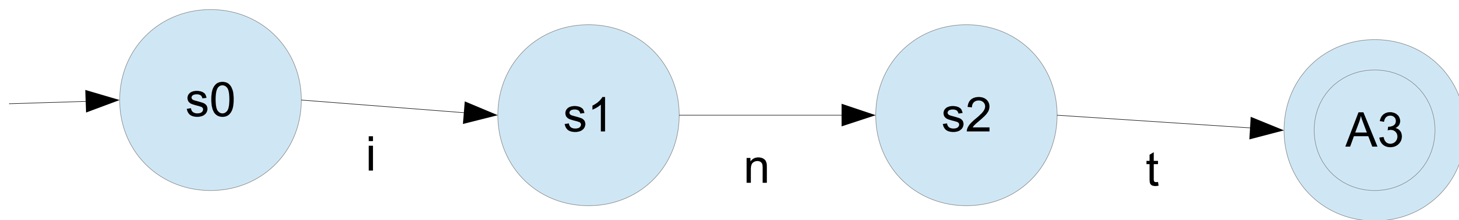
- Assuming any other transitions go to reject state



Example: more token types

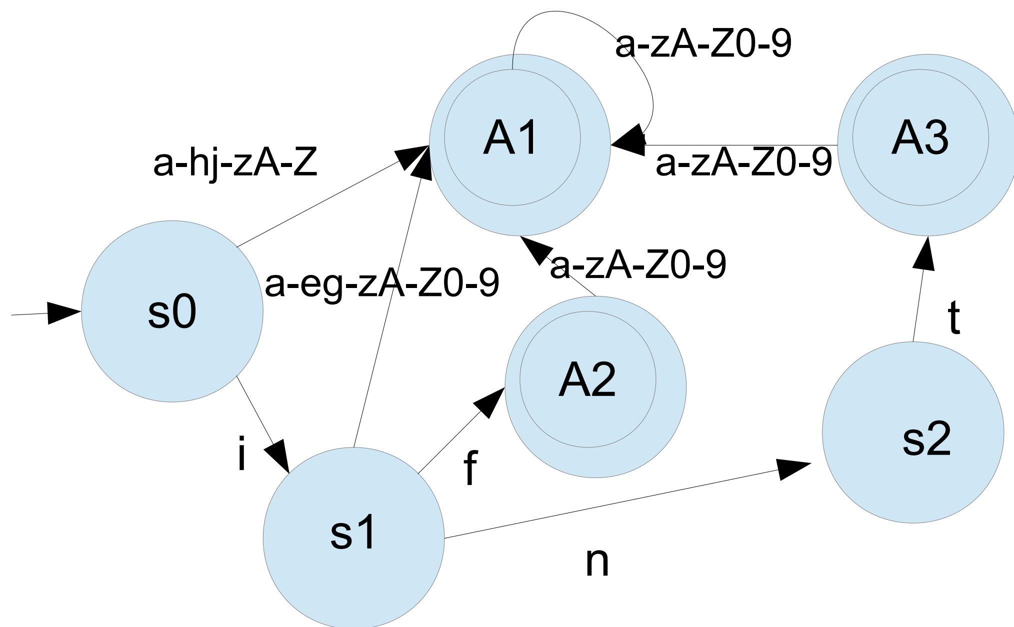
- FSMs to recognize keywords **if** and **int**





Example: combined FSM

- Recognizes all three token types



Simplistic conversion to code

```
state = s0
c = readChar()
while ( not(delimiter(c)) && (state != reject)) {
   if ((state = s0) && (c == 'i')) state = s1
   else if ((state == s0) && (alphanum(c)) state = s2
   ... etc
   else state = reject
   c = readChar()
}
// if end in A1, A2, or A3 we found a token
```