Scanning & Parsing

- **Scanner** (also known as *lexer*) breaks the program up into individual words with little interest in structure
  - **Input**: A sequence of characters
  - **Output**: tokens from the language
  - Separate from parsing for historical reasons, but also for optimization and convenience

- **Parser** takes the parts, interprets them *structurally* based on the BNF/EBNF grammar specification
  - **Input**: tokens
  - **Output**: parse (or syntax) tree, giving the essential logical structure of the program

Syntax Specification

- Consider statements in a language of variable assignments, arithmetic, and primitive I/O operations, read and write (text, p. 74):
  
  \[
  \begin{align*}
  stmt & \rightarrow id := expr \mid read \mid write expr \\
  expr & \rightarrow term \text{ term} \text{ tail} \\
  term & \rightarrow add\_op \text{ term} \text{ tail} \mid \epsilon \\
  factor & \rightarrow (expr) \mid id \mid number \\
  add\_op & \rightarrow + \mid - \\
  mult\_op & \rightarrow * \mid / \\
  
  \end{align*}
  \]

- A sample program fragment, with comments, could be e.g.: 

```plaintext
// print difference of 2 inputs
read A
read B
diff := A - B
write diff
```
Scanning Language Tokens

- The first part of the compilation (and interpretation) process is to take the text of a possible program and break it into individual legal tokens.
- At this stage, we are not concerned with syntactic correctness (that is the job of the next stage, the parser).
- We only want to find and label all valid linguistic components.
- Thus, for the following bit of 'code' in Java, the scanner would simply identify 6 legal tokens:
  
  ```java
  while if a = = 3
  ```
- 2 keywords, 2 assignment operators, an identifier, and a number.
- We don’t care yet that this is not syntactically valid.

Identifying Tokens (Tokenization)

- Like overall syntax, the tokenization process begins with a formal specification of legal 'words', including active code and comments.
- Part of scanning process is stripping out comments to simplify the later stages of scanning and further compilation.
- The scanner will take possible code and return a listing of all tokens, along with their role.
- Sometimes, these are effectively identical.
- For basic operators, the label (assign, plus, etc.) tells us everything we need to know about what it actually looks like.
  
  (Every plus sign is the same...)

Scanning Tokens

- The scanner will take possible code and return a listing of all tokens, along with their role.
- Sometimes, these are effectively identical.
- For basic operators, the label (assign, plus, etc.) tells us everything we need to know about what it actually looks like.
  
  (Every plus sign is the same...)

Scanning Tokens

- The scanner will take possible code and return a listing of all tokens, along with their role.
- Other times, a token can have many possible forms.
- For each id, number, or comment, the scanner will identify which of the three it is, and return that text, along with the appropriate label.
Scanning Tokens

- The scanner will take possible code and return a listing of all tokens, along with their role.
- Other times, a token can have many possible forms.

Reserved Words

- This language has only two special words: read and write.
- These must be distinguished from identifiers, which can otherwise combine any letters we wish.

Reserved Words & Keywords

- Some languages/programmers use reserved word and keyword interchangeably, but they are distinct concepts:
  1. A reserved word is one that can never be used for purposes of identifiers and the like (it belongs only to the language itself, and is ‘set aside’)
  2. A keyword is one that has special semantics in some contexts (acts as an operation or the like)

Reserved Words & Keywords

- Language designers can choose whichever keywords and reserved words that they like.
- In Java, for instance, every keyword is reserved.
- Each of these has special semantic meaning, changing how a program executes, and none can be used as an identifier.
- However, the reverse is not true: there are two Java reserved words that are not actually used as keywords.
- These are set aside for possible future use: they are not actually part of the language and do nothing, but still can’t be used.
- This allows new versions of Java to start using these as keywords, while preserving backwards compatibility of existing code.
Reserved Words & Keywords

- While Java reserves every keyword, this is not always true.
- Some languages allow keywords to be used as identifiers, so long as syntactic context can determine which are which.
- In FORTRAN and ALGOL 68, for instance, ‘if’ is a keyword, but can also be used as an identifier in the right contexts.

```
FORTRAN:
if = 1
```

- An assignment to a variable called ‘if’ is legal.

```
ALGOL 68:
if if eq 0 or if eq 1 then 2 fi
```

- Using variable ‘if’ (2nd and 3rd occurrences) in a conditional is legal.

(Note: Most ALGOL implementations then use e.g. bold-face or ALL CAPS for true keywords as an aid to programmers)

Redefining Operators and Keywords

- In Java, basic language components can’t be changed.
- In other languages, like C++, operators like << can be overloaded.
- Other languages are more flexible.

- In Smalltalk-80 only 6 basic keywords exist:
  ```
  true, false, nil, self, super, thisContext
  ```

- However there are ways to redefine even some of these!
  ```
  true become: false
  false become: true
  ```

- In PL/I there are no reserved words at all, and types and variable names are disambiguated only by context:
  ```
  INTEGER REAL;
  REAL INTEGER;
  ```

Generating Scanners Automatically

- Based on a given token grammar, we can always write a scanner for the language in an ad hoc fashion.
- A nicer approach is to do so programmatically.
- One version of this uses discrete finite automata (DFA).
- Finite state machines represented by directed graphs with:
  1. A single designated starting vertex (state of the machine).
  2. Directed edges labeled with language symbols (transitions).
  3. One of more designated stopping vertices (accept states).

- A string w is accepted by a given DFA if we can begin in the starting vertex and reach some stopping vertex, following edges one symbol at a time from w.
DFAs and Regular Expressions

An example DFA, where the start state ($S_0$) is labeled and the accept state ($S_3$) is indicated by a double circle:

- **Question:** What language (set of strings) will be accepted by this finite state machine? That is, what would a string have to look like to get us from $S_0$ to $S_3$?
- **Answer:** any string containing '001'

Building DFAs for a Grammar

- For every DFA, there exists a regular grammar with the same language, and vice-versa
- Generating a DFA that corresponds to a given grammar is not so simple, however
- In practice, algorithms to build a DFA data-structure for scanning strings work in an indirect fashion:
  1. Given a grammar, $G$, build a non-deterministic representation: an NFA, $N$
  2. Convert $N$ to a DFA, $D$
  3. Optimize the DFA $D$ to reduce its size as much as practical (step 2 can increase the state-space of the automaton exponentially if we are not careful!)

A DFA to Scan the Calculator Language

Given a program in our language, the DFA will:
1. Ignore all comments (returning to Start after the end of any such line/block)
2. Reach an accept state on any legal token (after which it will either terminate or return to Start if more text remains)
This Week

- **Topic:** Scanning/Parsing
- **Read:** Text, 2.2–2.3
- **Homework 01:** Posted to D2L last Friday
  - **Due:** Friday, 22 February, 5:00 PM on D2L
- **Office Hours:** Wing 210
  - Monday, 8:45 AM – 9:45 AM
  - Wednesday: 9:00 AM – 10:30 AM
  - Tuesday: 3:00 PM – 4:00 PM
  - Thursday: 2:00 PM – 3:00 PM
  - Friday: 9:00 AM – 10:30 AM

Monday, 11 Feb. 2019  Programming Languages (CS 421/521)