# CS342: Organization of Prog Languages

## Topic 4: Syntax and Formal Languages

- Syntax Elements
- Scanning vs Parsing
- Lexical Analysis
- Syntactic Analysis
- Regular Languages
- Context-Free Languages
- The Chomsky Hierarchy
- Composition of Languages
- Character Sets
- Extra topic: Operator Precedence Parsing
- Lexical and Syntactic Structure of Scheme
  (using formalisms of the last lecture)

© 2000-2007 Stephen M. Watt.

# Synax Elements

A whirlwind tour of scanning, parsing and formal language theory.

## Scanning vs Parsing

We distinguish

- "lexical analysis" = "scanning"
  = grouping characters together into tokens or words

and

- "parsing" = "syntactic analysis"
  = grouping a linear sequence of tokens into a tree
  according to some rules.

# Lexical Analysis

- In: Sequence of characters in some character set `'a'`, `'é'`, '$\psi$'.
- Out: Sequence of tokens belonging to a fixed set of classes.
- *E.g.*

| | | |
|---|---|---|
| `1234` | $\longrightarrow$ | `INT` |
| `/* ... */` | $\longrightarrow$ | `COMMENT` |
| `hello` | $\longrightarrow$ | `ID` |
| `if` | $\longrightarrow$ | `IF` |

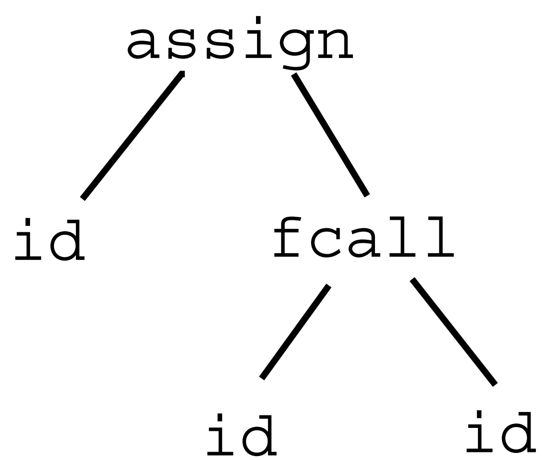
- The rules for the classes are language-specific, and can usually be described by a "regular language."

# Syntactic Analysis

- In: Sequence of tokens from some set of token classes.
- Out: Parse tree.
- *E.g.*

  `ID  ASSIGNOP  ID  LPREN  ID  RPREN`

  yields



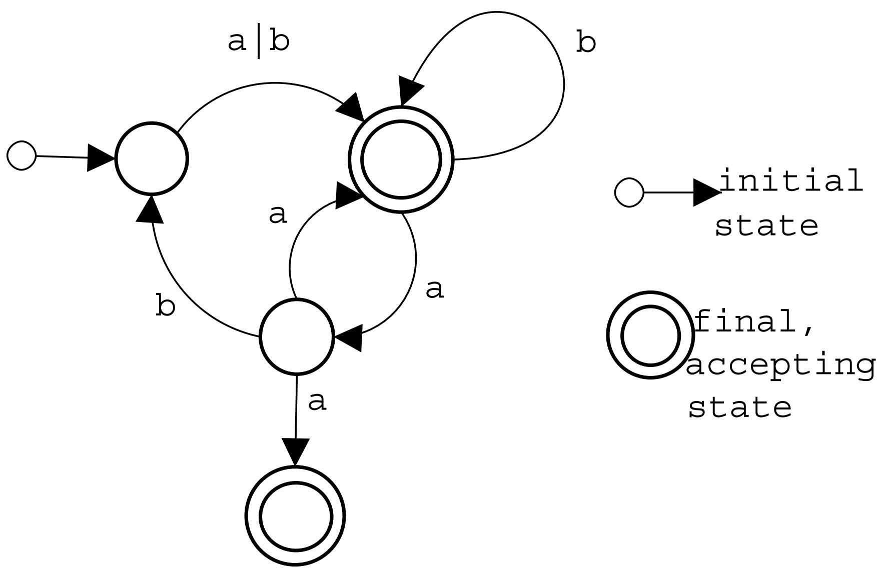

- The rules for making the trees are language-specific, and can usually be described by a "context-free language."

# REGULAR LANGUAGES

- Described by regular expressions

**a b** **a | b** **a *** **(ab|cd)***

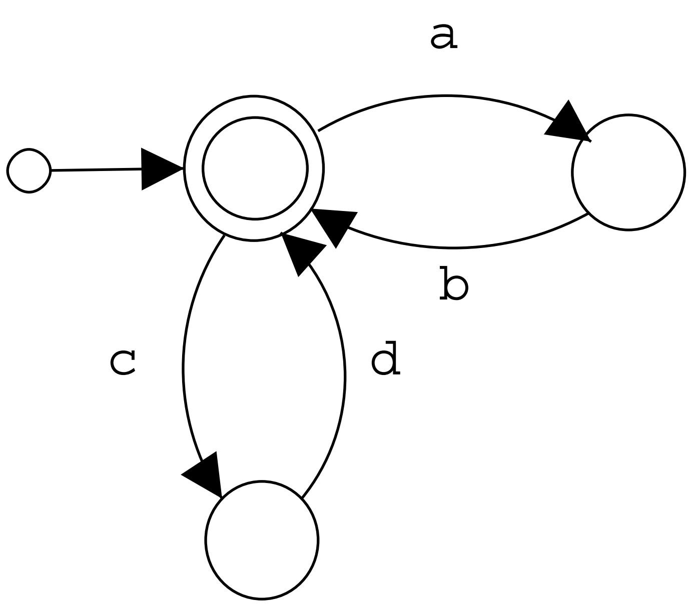
- Accepted by finite automata



# A regular expression and its finite automaton

- There is a correspondence between regular expressions and finite automata

`(ab|cd)*`



# Grammars for Regular Languages

- $\sum$, the *alphabet*. E.g. `{'a', 'b', 'c', ...}`
- $V$, the *variables*. E.g. {token, word, int, ...}
- $S$, the *start* symbol $\in V$. E.g. token
- $P$, the *productions* = rules, with the LHS $\in V$ and RHS $\in (\sum \cup V)^*$. E.g.

```
uppercase -> 'A' | 'B' | ... | 'Z'                (26 rules)
lowercase -> 'a' | 'b' | ... | 'z'                (26 rules)
digit  -> '0' | ... | '9'                         (10 rules)
letter -> uppercase | lowercase                   ( 2 rules)
int    -> digit   | digit int                     ( 2 rules)
word   -> letter  | letter word                   ( 2 rules)
token  -> int | word                              ( 2 rules)
```

Recursive rules are allowed, but the recursion must be either at the left or the right of the RHS in each instance

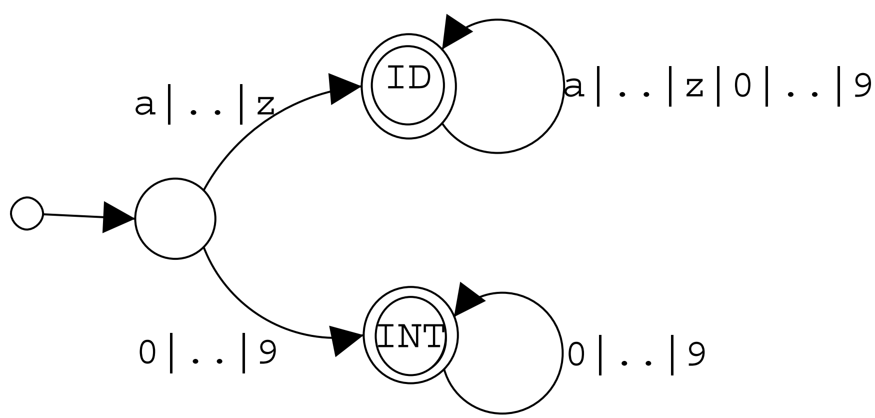
# Alternatively...

- Can specify rules using regular expressions on RHS, where each RHS uses only previously defined variables.
  I.e. rule for $v_i$ is a regular expression on $\sum \cup \{v_j | j < i\}$.
  E.g.

```
uppercase -> 'A' | 'B' | ... | 'Z'              (1 rule)
lowercase -> 'a' | 'b' | ... | 'z'              (1 rule)
digit  -> '0' | ... | '9'                       (1 rule)
letter -> uppercase | lowercase                 (1 rule)
int    -> digit digit*                          (1 rule)
word   -> letter letter*                        (1 rule)
token  -> int | word                            (1 rule)
```

# Accepting states classify tokens

- A scanner accepts or rejects an input, depending on whether it is in an accepting state when it reaches the end of the string.
- Accepting states can be labelled to *classify* the tokens accepted.




- Note: the categories of the token classes and the variables of the grammar need not have anything to do with each other.

# Tools for Scanners

- lex, flex, jflex
- take a grammar for a regular language
- produce a finite automaton as a program.

# CONTEXT-FREE LANGUAGES

- As before: $\sum$ alphabet, $V$ variables, $S \in V$ start, $P$ productions
- Rules (productions) are of the form $v \rightarrow \alpha$, where $v \in V$, $\alpha \in (\sum \cup V)^*$. Arbitrary recursion allowed in the RHS.
- **Example:** Well-nested parentheses.

$$\begin{array}{rcl} \Sigma & = & \{\, '(', \, ')' \,\} \\ V & = & \{E\} \\ S & = & E \\ P & = & \left\{ \begin{array}{l} E \rightarrow \text{nothing} \\ E \rightarrow '(' \; E \; ')' \\ E \rightarrow E \; E \end{array} \right\} \end{array}$$

- **Example:** Arithmetic expressions. $\sum = \{'(', ')', '+', '-', '*', \mathtt{ID}, \mathtt{INT}\}$, $V = \{Expr, Sum, Product, Factor\}$, $S = Expr$, $P = \{$

$$\begin{array}{rcl} Expr & \rightarrow & Sum \\ Sum & \rightarrow & Product \; '+' \; Sum \mid Product \; '-' \; Sum \mid Product \\ Product & \rightarrow & Factor \; '*' \; Product \mid Factor \\ Factor & \rightarrow & \mathtt{ID} \mid '(' \; Expr \; ')' \} \end{array}$$

# Pushdown Automata

- Context-free languages are recognized by "Pushdown automata"

- These are similar to finite automata, but they can keep track of state on a STACK.

# THE CHOMSKY HIERARCHY

| **Type** | **Language Class** (**Production**) | **Theoretical Machine** | **Tool** (**Example**) |
|---|---|---|---|
| 3 | Regular Langs ($R \rightarrow abcR$) | Finite Automaton (single state) | Lex (Scanner for C) |
| 2 | Context Free Languages ($S \rightarrow xSx$) | Deterministic Push Down Automaton (stack) | Yacc (Parser for C) |
| 1 | Context Sensitive Languages ($QR \rightarrow \alpha XY\beta$) $\lvert\alpha\rvert \leq \lvert\beta\rvert$ | Linear Bounded Automaton (tape proportional to input) | Computer (Fixed size mem) |
| 0 | Unrestricted Grammar ($aSTb \rightarrow xUVy$) | Turing Machine (infinite tape) | Computer (any program) |

# Composition of Languages

- A real parser is usually built as a composition of simpler languages

$$L = L_2 \circ L_1 \circ L_0$$

  where the output of $L_i$ is the input to $L_{i+1}$.

- E.g. The token classes of the scanner comprise the alphabet $\Sigma$ of the parser.

```
L0: ASCII -> { ID, '+', '*', '(', ')', COMMENT}

L1: { ID, '+', '*', '(', ')', COMMENT} ->
    { ID, '+', '*', '(', ')'}

L2: { ID, '+', '*', '(', ')'} -> Parse Tree
```

# Character Sets

- ASCII: American Standard Code for Information Interchange. (7 bit)
- (EBCDIC)
- Latin1: Extension to ASCII for accented characters, etc. (8 bit)
- Unicode:
  All scripts in use (Han, Armenian, Klingon, ...)
  17 planes of 16 bit.

# UTF-8

- A way to store Unicode data in an ASCII-compatible form

```
0x00000000 - 0x0000007F: 0xxxxxxx

0x00000080 - 0x000007FF: 110xxxxx 10xxxxxx

0x00000800 - 0x0000FFFF: 1110xxxx 10xxxxxx 10xxxxxx

0x00010000 - 0x001FFFFF: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

0x00200000 - 0x03FFFFFF: 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

0x04000000 - 0x7FFFFFFF: 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
```

- Examples (from Linux Man page):
  The Unicode character `0xa9 = 1010 1001` (©) is encoded in UTF-8 as:

  ```
  11000010 10101001 = 0xc2 0xa9
  ```

  The character `0x2260 = 0010 0010 0110 0000` (≠) is encoded as:

  ```
  11100010 10001001 10100000 = 0xe2 0x89 0xa0
  ```

# Operator Precedence Parsing

- Each operator has left- and right- precedence E.g.

$$_{100}+_{101} \qquad _{200}\times_{201} \qquad _{301}\uparrow_{300}$$

- Group subexpressions by binding highest numbers first.

$$A + B \times C \times D \uparrow E \uparrow F$$

$$A \;_{100}+_{101}\; B \;_{200}\times_{201}\; C \;_{200}\times_{201}\; D \;_{301}\uparrow_{300}\; E \;_{301}\uparrow_{300}\; F$$

$$A \;_{100}+_{101}\; B \;_{200}\times_{201}\; C \;_{200}\times_{201}\; D \;_{301}\uparrow_{300}\; (E \;_{301}\uparrow_{300}\; F)$$

$$A \;_{100}+_{101}\; B \;_{200}\times_{201}\; C \;_{200}\times_{201}\; (D \;_{301}\uparrow_{300}\; (E \;_{301}\uparrow_{300}\; F))$$

$$A \;_{100}+_{101}\; (B \;_{200}\times_{201}\; C) \;_{200}\times_{201}\; (D \;_{301}\uparrow_{300}\; (E \;_{301}\uparrow_{300}\; F))$$

$$A \;_{100}+_{101}\; ((B \;_{200}\times_{201}\; C) \;_{200}\times_{201}\; (D \;_{301}\uparrow_{300}\; (E \;_{301}\uparrow_{300}\; F)))$$

$$A + ((B \times C) \times (D \uparrow (E \uparrow F)))$$

- Works fine for expressions but not well for general CFL

# Lexical Structure of Scheme

- This is a simplified set of rules for Scheme's lexical structure:

```
     Token  →  Atmosphere * RealToken
Atmosphere  →  space | newline | ";" not-newline *
 RealToken  →  Identifier | Boolean | Number | Character | String
               | "(" | ")" | "#(" | "'" | "`" | "," | ",@" | "."

Identifier  →  Initial Subsequent * | "+" | "-" | "..."
   Initial  →  "a" | "b" | "c" | ... | "z"
               | "!" | "$" | "%" | "&" | "*" | "/" | ":"
               | "<" | "=" | ">" | "?" | "^" | "_" | "~"
Subsequent  →  Initial | Digit | "+" | "-" | "." | "@"

   Boolean  →  "#t" | "#f"
    Number  →  Sign Digit Digit *
      Sign  →  "+" | "-" | empty
     Digit  →  "0" | "1" | "2" | ... | "9"
 Character  →  "#\" any character | "#\space" | "#\newline"
    String  →  "StringChar * "
StringChar  →  not " or \ | \" | \\
```

- The complete lexical rules for Scheme replace the production for **Number** with a more complicated description which is itself about as big as the above.

# Lexical structure

- We will belabour the point to practice what we have learned:

  A grammar for Scheme's tokens is given by $G = (\Sigma, V, P, S)$, where
  - *A* is the input alphabet:

    ```
    space newline
    _ - , ; : !  ?  / . ` ^ ~ ' " ( ) @ $ * \ & # % +
    0 1 2 3 4 5 6 7 8 9 a b c d e f i l n o p s t w x z
    ```

  - *V* is the set of variables:

    | | | | |
    |---|---|---|---|
    | **Token** | **Atmosphere** | **RealToken** | **Identifier** |
    | **Initial** | **Subsequent** | **Boolean** | **Number** |
    | **Sign** | **Digit** | **Character** | **String** |
    | **StringChar** | | | |

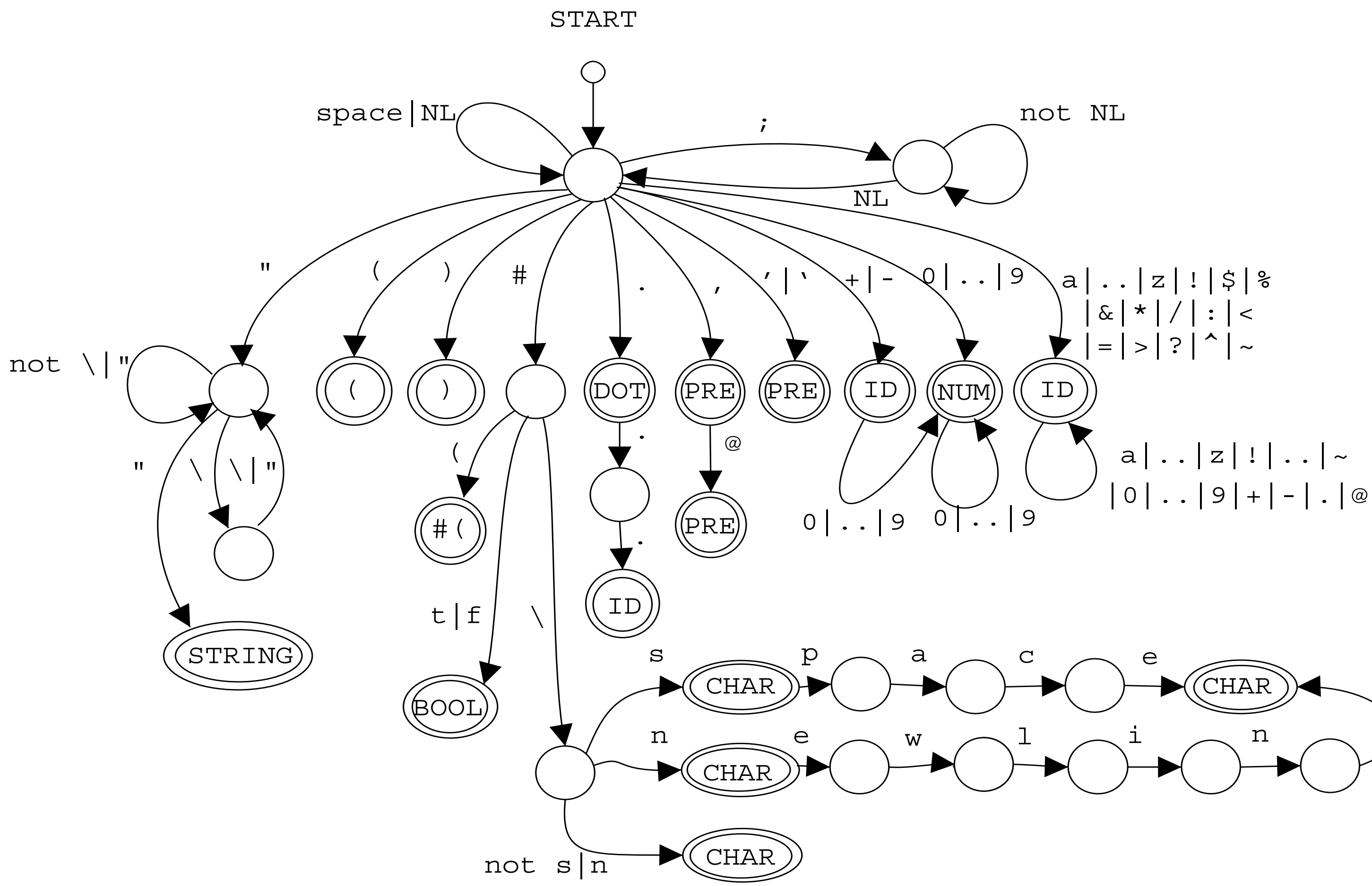
  - *P* is the set of productions given on the previous page.

    Note that $A \to B|C$ is a short-hand for two productions $A \to B$ and $A \to C$.

  - *S* is the "start symbol", in this case **Token**.

# Lexical structure

Tokens of this language are accepted by the following finite automaton:




A full Scheme scanner would have a large part for scanning numbers.

# Syntactic structure

The syntax of the external representation of programs can be expressed by the following grammar for S-expressions $(A_{SE}, V_{SE}, P_{SE}, S_{SE})$, with

- The input "alphabet", $A_{SE}$:

  ```
  BOOL   NUM   CHAR   STRING  ID   PRE "#("   "("   ")"    "."
  ```

  PRE can be any of `"'"`   `` "`" ``   `","`   `",@"`

- The set of variables, $V_{SE}$

  ```
  Datum  Simple  Compound  Vector  List  Abbrev
  ```

- The set of productions, $P_{SE}$

  ```
  Datum    -> Simple | Compound
  Simple   -> BOOL | NUM | CHAR | STRING | ID
  Compound -> List | Vector | Abbrev
  List     -> "(" Datum * ")" | "(" Datum* Datum "." Datum ")"
  Vector   -> "#(" Datum* ")"
  Abbrev   -> PRE Datum
  ```

- The start symbol, $S_{SE}$, is `Datum`.

## Syntactic structure – further notes

- The following abbreviations are equivalent to other S-expressions:

```
'X        (quote X)
`X        (quasiquote X)
,X        (unquote X)
,@ X      (unquote-splicing X)
```

- In practice, a more useful grammar classifies the following keywords individually:

```
quote   lambda  if         set!   begin   cond
and     or      case       let    let*    letrec
do      delay   quasiquote else   =>      define
unquote unquote-splicing
```

- The grammar then specifies more narrowly the form of the S-expressions using these forms.