Syntax and Formal Languages

CS 4447 / 9545 – Stephen M. Watt
University of Western Ontario

© Copyright 2001-2008 Stephen M. Watt

Outline

- 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

Syntax 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’, ‘ψ’.
- Out: Sequence of tokens belonging to a fixed set of classes.
- E.g.

```
1234      →  INT
/* … */   →  COMMENT
hello     →  ID
if        →  IF
```

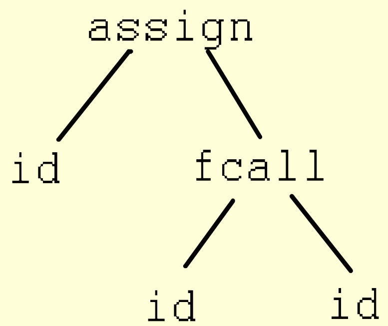
- The rules for the classes are language-specific and can usually be described by a “regular language.”

Syntactic Analysis

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

  `ID, ASSIGNOP, ID, LPREN, ID, RPREN`

  yields



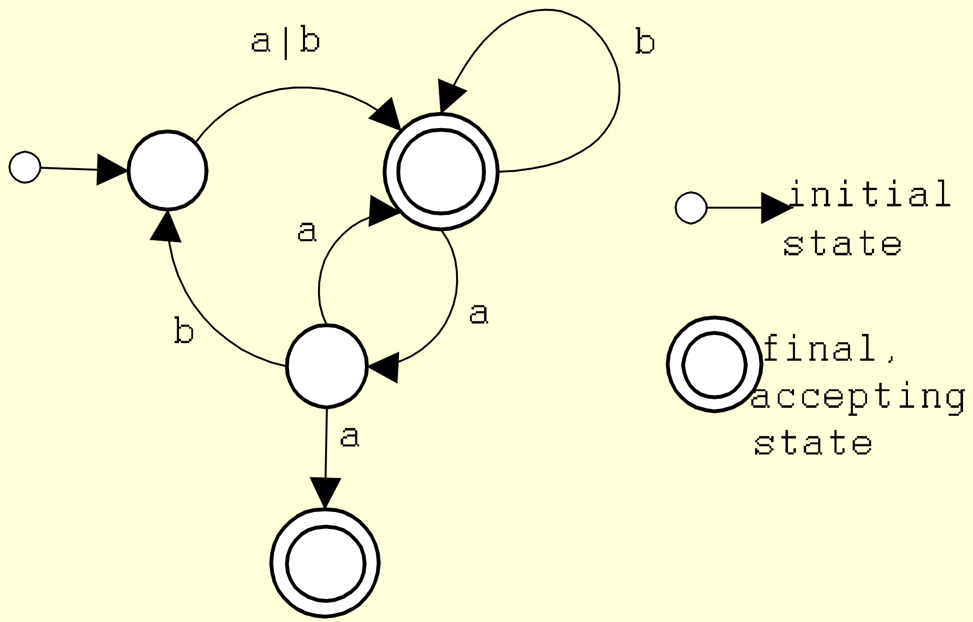
- The rules for making the trees are language-specific and can usually be described by a “context-free language.”

Regular Languages

- Described by regular expressions

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

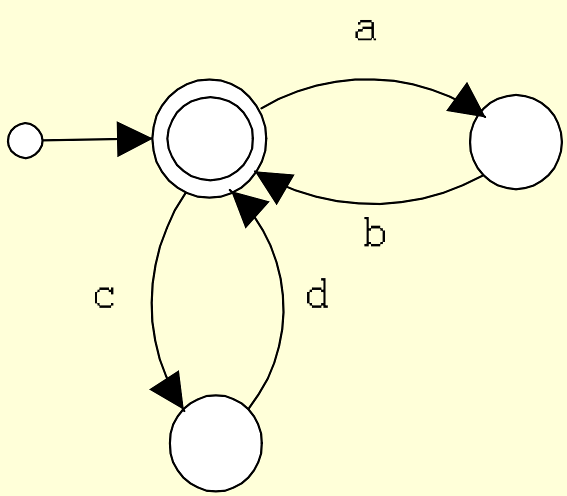
- Accepted by finite automata



A Regular Expression and Its Finite Automaton

- There is a correspondence between regular expressions and finite automata.

(ab|cd)*



Grammars for Regular Languages

- Σ, 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 $\in$ $(\Sigma \cup V)^*$.
- 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.

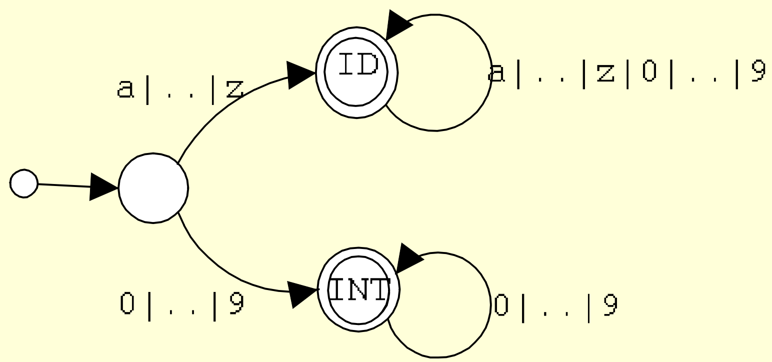
  I.e. rule for v[i] is a regular expression on $\Sigma \cup \{ v[j] \mid j < i \}$.

  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: Σ alphabet, V variables, $S \in V$ start, P productions
- Rules (productions) are of the form $v \rightarrow \alpha$,
  where $v \in V$, $\alpha \in (\Sigma \cup V)^*$.
- Arbitrary recursion allowed in the RHS.

- **Example**: Well-nested parentheses.

  Σ = { ‘(‘, ‘)’ }   V = {E}   S = E

  P = { E → nothing, E → ‘(‘ E ‘)’, E → E E }

- **Example**: Arithmetic expressions.
  Σ = {‘(‘, ‘)’, ‘+’, ‘-’, ‘*’, ID, INT},
  V = {Expr, Sum, Product, Factor},
  S = Expr,
  P = { Expr → Sum,

  Sum → Product ‘+’ Sum | Product ‘−’ Sum | Product

  Product → Factor ‘*’ Product | Factor

  Factor → ID | ‘(‘ Expr ‘)’   }

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 Languages ($R \rightarrow abcR$)	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 ($QR \rightarrow \alpha XY\beta$, $\|\alpha\| \leq \|\beta\|$)	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 Σ of the parser.

L[0] : ASCII → { ID, ‘+’, ‘*’, ‘(‘, ‘)’, COMMENT}
L[1] : { ID, ‘+’, ‘*’, ‘(‘, ‘)’, COMMENT} → { ID, ‘+’, ‘*’, ‘(‘, ‘)’}
L[2] : { 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 modern use (Han, Armenian, Klingon,...)
  17 planes of 16 bit characters.

UTF-8

- A way to store Unicode data in 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 character 0x00a9 = 0000 0000 1010 1001 (©) is encoded in UTF-8 as: 11000010 10101001 = 0xc2 0xa9

The character 0x2260 = 0010 0010 0110 0000 (≠) is encoded in UTF-8 as: 11100010 10001001 10100000 = 0xe2 0x89 0xa0