Syntactic Modeling Tools

The Compilation Process

```
                  Source program
                        |
      <--------   Lexical analyzer
      |                 |
      <--------   Syntax analyzer
      |                 |
Symbol table ---> Intermediate code, semantic analyzer  --> Optimization
      |                 |                                         |
      -------->   Machine language code generator  <---------------
                        |
                  Hardware computation
```

Lexical and Syntactic Analysis

Programs are *strings*.

Lexical analysis scans the program string for valid character sequences. *Syntactic analysis* parses the program string for valid word sequences.

Structural rules for both character and words strings are defined by a *context-free language*. Formal specification techniques include BNF, diagrammatic BNF, finite automata, production rules, and syntax graphs.

Formal Languages

alphabet	a finite set of symbols	`{i,j,k,+,*,0,1,2,...}`
string	a finite ordered list of symbols	
language	all possible strings using a given alphabet	
grammar	a subset of all possible strings constrained by a set of composition rules	

Classes of Languages

These categories form a hierarchy, in that regular languages are fully contained in context-free languages, etc.

Context sensitive languages have rules

`A --> B` such that `|A| =< |B|`

That is, the result `B` is never smaller that the input `B`

Context free languages have rules

`A --> B` such that `A` is a single non-terminal string

That is, the input A never branches.

Regular languages have rules

`A --> B` such that `B` is a terminal or a terminal and a non-terminal

That is, the output `B` either ends the rule, or triggers another terminating rule.

Context-free Languages

These three operations define a *regular language*:

JOIN	concatenate tokens and strings
OR	choose between alternatives for one location
LOOP	Kleene closure, Kleene star *

The Kleene star, `s*`, is a notation for repeating a given string zero or more times. The Kleene plus, `r+`, means repeat the given string one or more times.

RECURSION	handle nested structures

Expressed as production rules, recursion allows

`A --> B` such that `B` can contain reference to `A`

For termination, at least one rule associated with `A` must not contain recursive reference to `A`.

Regular languages are defined by placing a constraint on context-free languages, that of not permitting recursion. Recursion is necessary to construct nested and hierarchical forms; regular languages permit only construction of flat strings and lists.

Backus-Naur Form

One way to specify structural rules is using BNF, Backus-Naur Form. BNF is a collection of transformation, or pattern-matching, rules to apply to a given expression. BNF defines a regular language. Valid token strings can be specified by

Base cases:	empty string	`E`
	single character	`u`
Compound cases:	concatenation	`r*s`
	disjunction	`r|s`
	repetition	`r*`

Parentheses are used to disambiguate forms.

A diagrammatic version of regular language specifications uses

JOIN	`A --> B`	arrow from `A` to `B`
OR	`A -->` ` \-->`	branching paths
LOOP	`A -->` `\<--/`	Kleene star

Example: PASCAL numerical strings

```
digit       -->   0|1|2|3|4|5|6|7|8|9

integer     -->   digit digit*

number      -->   integer ((. integer)| E)
```

Language classes and grammars were developed by Noam Chomsky. Computing languages became connected to grammars because Backus' BNF formalism was equivalent to Chomsky's. Thus, a context-free language can be defined as a BNF form with recursion.

Examples of Grammars

Balanced Parentheses

alphabet	`{(,)}`	
strings	`{(,),S}`	

grammar1	`S --> empty`
	`S --> S S`
	`S --> (S)`	

grammar2	`S  --> S1*` `S1 --> (S)`	Star allows zero occurrences

Simple Algebra

alphabet	`{,0,1,2,...,a,b,c,...,+,*,(,)}`
strings	`{+,*,(...), id, Term, Factor, Expression}`

grammar	`Expression --> Expression + Term | Term`
	`Term --> Term * Factor | Factor`
	`Factor --> (Expression) | id`

Terms, Factors, and Expressions are defined formally by the rules of the grammar. Intuitively, an Expression is any valid linear algebraic form. A Term is two forms multiplied together. A Factor is an id or any Expression (separated by parentheses for grouping).

Simple Arithmetic Parsing Example

```
                  (3 * (4 + 5)) * (2 + 7)

(3 * (4 + 5)) * (2 + 7)             expression1
                                                --> term1
(3 * (4 + 5)) * (2 + 7)             term1
                                                --> term2 * factor1
(3 * (4 + 5))                       term2
                                                --> factor2
(3 * (4 + 5))                       factor2
                                                --> (expression2)
 3 * (4 + 5)                        expression2
                                                --> term3
 3 * (4 + 5)                        term3
                                                --> term4 * factor3
 3                                  term4
                                                --> factor4
 3                                  factor4
                                                --> id1
     (4 + 5)                        factor3
                                                --> (expression3)
      4 + 5                         expression3
                                                --> expression4 + term5
      4                             expression4
                                                --> term6
      4                             term6
                                                --> factor5
      4                             factor5
                                                --> id2
          5                         term5
                                                --> factor6
          5                         factor6
                                                --> id3
               (2 + 7)              factor1
                                                --> (expression5)
                2 + 7               expression5
                                                --> expression6 + term7
                2                   expression6
                                                --> term8
                2                   term8
                                                --> factor7
                2                   factor7
                                                --> id4
                    7               term7
                                                --> factor8
                    7               factor      --> id5
```