COMP 200: Elements of Computer Science
Fall 2004
Lecture 34: November 19, 2004

More Expressive Ways of Describing Syntax

Regular Expressions, Again

Last class we introduced regular expressions as a way of denoting the languages that a deterministic finite automaton can recognize. To review, regular expressions (REs) are defined by the following rules:

1. If *s* is a symbol in the alphabet, we say that *s* is an RE that denotes that symbol.
2. If *s* & *t* are REs, then
   a. *s* followed by *t* is an RE, denoted *st*
   b. *s* or *t* is an RE, denoted *s* | *t*
   c. zero or more concatenated copies of *s* is an RE, denoted s^* and called the closure of *s* or the Kleene closure of *s*
3. We sometimes define the positive closure of *s* as ss^* denoted as s^+

We use the term concatenation

Regular expressions are equivalent in their expressive power to the finite automata that we saw last class. They can recognize any finite list of symbols in time proportional to the number of symbols in the input string. They cannot count, as in palindromes or matching parentheses.

Grammars

To express a broader class of languages, we need to introduce a more complex notation for syntax. Among Computer Scientists, the most common formalism used to express syntax is a grammar — a set of symbols and productions (or rewrite rules) that allow us to derive any word in the language. One ancient (in Computer Science terms) notation for writing down a grammar is called Backus-Naur form, after John Backus (leader of the team that created Fortran) and Petere Naur (leader of the team that created Algol). Both used a similar notation to describe the set of acceptable programs in these early programming languages. The result was widespread use of BNF.

Symbols in a grammar fall into two categories — those that can appear in a sentence in the language described by the grammar and those that are simply syntactic variables used to give structure to the grammar and to allow the grammar to express more than one sentence. The former symbols (the ones that appear in sentences) are called ***terminal symbols***. The latter symbols are called, conversely, ***nonterminal symbols***. Because BNF was invented in the days of typewriters (*i.e.*, before the rich typography of laser printers), the notation for distinguishing between terminal and nonterminal symbols was somewhat crude — terminals are underlined and nonterminals are written between <brackets>.

Productions, or rewrite rules, define the syntactic relationship between symbols. The rule

<fee> ::= fie foe fum

states that the syntactic variable <fee> derives (::=) the string of terminal symbols fie foe fum. A grammar is just a collection of such rules, along with an initial symbol that tells us where to start rewriting. (The rewriting process is simpler than the one we used to define the meaning of Scheme programs, because rewriting a grammar rule involves no substitution of one name — or parameter — for another symbol.)

To make this concrete, lets consider a complete grammar, ***SN***.

Symbols in *SN* = { <SN>, baa }

Rules in SN = { <SN> ::= baa <SN> ; <SN> ::= baa }

To simplify writing the rules, we can use the symbol "|" to mean also derives, which allows us to write the rules for *SN* as

{ <SN> ::= baa <SN> | baa }

Using this grammar, and starting with <SN>, we can derive a number of important sentences. (<SN> is the obvious start symbol, since it is the only nonterminal in the grammar. In a more complex grammar, we would need to designate the start symbol.)

— derive one, two, and three syllable Sheep Noise

Grammars, of course, have more dignified uses. We might, in specifying a programming language, want to specify its syntax. For Scheme, expressions might be defined as

<SExpr> ::= (<list of names>)

<list of names> ::= <list of names> name
| name

Using the rewrite rules, we can derive any Scheme expression, except ( ). A <list of names> must contain a single name. If we wanted zero-length lists, we could add an alternate rule to the definition of <SExpr> that derived ( ).

Programming languages are specified with a grammar, usually written in BNF. The BNF for a language does not constitute the complete specification. For example, in our definition of Scheme, the first name in the list must be a function (either built into Scheme or user-supplied) and the number of arguments that the function requires must match the cardinality of the list, minus one. This requirement is an extra-syntactic rule — that is, it is hard to write or to enforce in a grammatical way.

To enforce it in the grammar, we would need to have a different terminal symbol for function names than for the names of other objects. This requirement, in turn, would tremendously complicate the rules of Scheme or raise the algorithmic complexity of checking the syntax of a Scheme program. Instead, software systems use other mechanisms to enforce such restrictions.

As long as we restrict the form of the rules in our grammar so that they have only one nonterminal on the right hand side, the grammars are called *context-free grammars*. (With more than one nonterminal on the right hand side, the grammar can specify strange non-local properties, such as only use the name <fee> as a function name if it has already appeared as a function name. The extra power makes the process of recognizing — or parsing — a sentence in the language much more complex.)

Sentences in a context-free grammars can be recognized in time proportional to the number of steps in the rewriting sequence that produced the sentence — that is, the number of symbols in the derivation. **Note:** *recognizing a sentence is the inverse of deriving one*. The algorithms are arcane, but can be automated so that you provide a BNF for the grammar and the tool gives you back a program that builds a derivation tree for an arbitrary sentence in the language and reports errors if the sentence is not in the language.

Power of Context-free Grammars

A context-free grammar can describe languages that a regular expression cannot. For example, a CFG can handle palindromes and parentheses.

Formalism

Regular expressions are equivalent to deterministic finite automata, which are equivalent to deterministic Turing machines with one-way motion on the tape (can only move left).

A context-free grammar is equivalent to a deterministic Turing machine with a stack rather than a tape. (Parenthesis recognition is easy. Push open parentheses. Every time it encounters a close parenthesis, it pops the stack. At the end, if the stack is empty, the parentheses all matched. Popping an empty stack diagnoses too many close parentheses. A non-empty stack at termination diagnoses too many open parentheses.)

We have a hierarchy — REs <= CFGs <= CSGs ??<= ??Turing machiness