Declarative Computation Model

Defining practical programming languages (VRH2.1)

Carlos Varela
RPI
October 1, 2007

Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

Declarative Programming Model

• Guarantees that the computations are evaluating functions on (partial) data structures
• The core of functional programming (LISP, Scheme, ML, Haskell)
• The core of logic programming (Prolog, Mercury)
• Stateless programming vs. stateful (imperative) programming
• We will see how declarative programming underlies concurrent and object-oriented programming (Erlang, C++, Java, SALSA)

Defining a programming language

• Syntax (grammar)
• Semantics (meaning)

Language syntax

• Defines what are the legal programs, i.e. programs that can be executed by a machine (interpreter)
• Syntax is defined by grammar rules
• A grammar defines how to make ‘sentences’ out of ‘words’
• For programming languages: sentences are called statements (commands, expressions)
• For programming languages: words are called tokens
• Grammar rules are used to describe both tokens and statements

Language syntax (2)

• A statement is a sequence of tokens
• A token is a sequence of characters
• A program that recognizes a sequence of characters and produces a sequence of tokens is called a lexical analyzer
• A program that recognizes a sequence of tokens and produces a statement representation is called a parser
• Normally statements are represented as (parse) trees
Extended Backus-Naur Form

- EBNF (Extended Backus-Naur Form) is a common notation to define grammars for programming languages
- Terminal symbols and non-terminal symbols
- Terminal symbol is a token
- Nonterminal symbol is a sequence of tokens, and is represented by a grammar rule
  \( (\text{nonterminal}) := (\text{rule body}) \)

Grammar rules

- \((\text{digit}) := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \)
- \((\text{digit}) \) is defined to represent one of the ten tokens 0, 1, …, 9
- The symbol ‘\(|\)’ is read as ‘or’
- Another reading is that \((\text{digit}) \) describes the set of tokens \\{0,1,…,9\}\)
- Grammar rules may refer to other nonterminals
  - \((\text{integer}) := (\text{digit}) (\text{digit}) \)
  - \((\text{integer}) \) is defined as the sequence of a \((\text{digit}) \) followed by zero or more \((\text{digit}) \)’s

Context-free and context-sensitive grammars

- Grammar rules can be used either to verify that a statement is legal, or to generate all possible statements
- The set of all possible statements generated from a grammar and one nonterminal symbol is called a (formal) language
- EBNF notation defines a class of grammars called context-free grammars
- Expansion of a nonterminal is always the same regardless of where it is used
- For practical languages, a context-free grammar is not enough, usually a condition on the context is sometimes added

Examples

- \((\text{statement}) ::= \text{if} (\text{expression}) \text{ then} (\text{statement}) \{ \text{else} (\text{statement}) \text{ end} \} \)
- \((\text{expression}) ::= (\text{variable}) | (\text{integer}) \)

How to read grammar rules

- \((\phi) \) is a nonterminal
- \((\phi) \to \text{Body} \) : \((\phi) \) is defined by \text{Body}
- \((\phi) \to (\phi) \) : either \((\phi) \) or \((\phi) \) (choice)
- \((\phi) \to \) : the sequence \((\phi) \) followed by \((\phi) \)
- \{(\phi) \} : a sequence of zero or more occurrences of \((\phi) \)
- \{(\phi) \} : a sequence of one or more occurrences of \((\phi) \)
- \{0\} : zero or one occurrences of \((\phi) \)
- Read the grammar rule from left to right to give the following sequence:
  - Each terminal symbol is added to the sequence
  - Each nonterminal is replaced by its definition
  - For each \((\phi) \to (\phi) \) pick any of the alternatives
  - For each \((\phi) \to \) add the sequence \((\phi) \) followed by the sequence \((\phi) \)
Example: (Parse Trees)

- if (expression) then (statement) else (statement) end

Language Semantics

- Semantics defines what a program does when it executes
- Semantics should be simple and yet allows reasoning about programs (correctness, execution time, and memory use)
- How can this be achieved for a practical language that is used to build complex systems (millions of lines of code)?
- The kernel language approach

Kernel Language Approach

- Define a very simple language (kernel language)
- Define the computation model of the kernel language
- By defining how the constructs (statements) of the language manipulate (create and transform) the data structures (the entities) of the language
- Define a mapping scheme (translation) of the full programming language into the kernel language
- Two kinds of translations: linguistic abstractions and syntactic sugar

Linguistic abstractions vs. syntactic sugar

- Linguistic abstractions, provide higher level concepts that the programmer can use to model and reason about programs (systems)
- Examples: functions (fun), iterations (for), classes and objects (class), mailboxes (receive)
- The functions (calls) are translated to procedures (calls)
- The translation answers questions about the function call: \{(F1 \{F2 X\}, \{F3 X\}\}

Linguistic abstractions vs. syntactic sugar

- Linguistic abstractions, provide higher level concepts that the programmer can use to model and reason about programs (systems)
- Syntactic sugar are short cuts and conveniences to improve readability

if \(N=1\) then \([1]\)
else local \(L\) in ...
end

if \(N=1\) then \([1]\)
else local \(L\) in ...
end
Approaches to semantics

Programming Language

Kernel Language

Formal calculus

Abstract machine

Guides the programmer in reasoning and understanding

Mathematical study of programming (languages)

Aid to the implementer

Efficient execution on a real machine

Exercises

39. Write a valid EBNF grammar for lists of non-negative integers in Oz.

40. Write a valid EBNF grammar for the $\lambda$-calculus.
   - Which are terminal and which are non-terminal symbols?
   - Draw the parse tree for the expression:
     
     $$((\lambda x. (\lambda y. y)) \, \lambda x. x)$$

41. *The grammar

   $$\langle \text{exp} \rangle ::= \langle \text{int} \rangle \mid \langle \text{exp} \rangle \, \langle \text{op} \rangle \, \langle \text{exp} \rangle$$

   $$\langle \text{op} \rangle ::= + \mid *$$

   is ambiguous (e.g., it can produce two parse trees for the expression $2*3+4$). Rewrite the grammar so that it accepts the same language unambiguously.