Declarative Computation Model
Defining practical programming languages (CTM 2.1)

Carlos Varela
RPI
September 26, 2014

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

• A computation model: describes a language and how the sentences (expressions, statements) of the language are executed by an abstract machine

• A set of programming techniques: to express solutions to the problems you want to solve

• A set of reasoning techniques: to reason about programs to increase the confidence that they behave correctly and to calculate their efficiency
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:
  \[
  \langle \text{nonterminal} \rangle ::= \langle \text{rule body} \rangle
  \]
Grammar rules

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

- \langle x \rangle : is a nonterminal \( x \)
- \langle x \rangle ::= Body : \langle x \rangle is defined by Body
- \langle x \rangle | \langle y \rangle : either \langle x \rangle or \langle y \rangle (choice)
- \langle x \rangle \langle y \rangle : the sequence \langle x \rangle followed by \langle y \rangle
- \{ \langle x \rangle \} : a sequence of zero or more occurrences of \langle x \rangle
- \{ \langle x \rangle \}^+ : a sequence of one or more occurrences of \langle x \rangle
- [ \langle x \rangle ] : zero or one occurrences of \langle x \rangle
- 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 \langle x \rangle | \langle y \rangle pick any of the alternatives
  - For each \langle x \rangle \langle y \rangle add the sequence \langle x \rangle followed by the sequence \langle y \rangle
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
Context-free and context-sensitive grammars

• It is easy to read and understand
• Defines a superset of the language

• Expresses restrictions imposed by the language (e.g. variable must be declared before use)
• Makes the grammar rules context sensitive

Context-free grammar (e.g. with EBNF) + Set of extra conditions
Examples

- \langle\text{statement}\rangle ::= \text{skip} \mid \langle\text{expression}\rangle \ ' = ' \langle\text{expression}\rangle \mid \ldots
- \langle\text{expression}\rangle ::= \langle\text{variable}\rangle \mid \langle\text{integer}\rangle \mid \ldots

- \langle\text{statement}\rangle ::= \text{if} \langle\text{expression}\rangle \text{then} \langle\text{statement}\rangle
  \begin{array}{l}
  \{ \text{elseif} \langle\text{expression}\rangle \text{then} \langle\text{statement}\rangle \} \\
  \mid \text{else} \langle\text{statement}\rangle \end{array}
  \text{end} \mid \ldots
Example: (Parse Trees)

- \textbf{if} \langle \text{expression} \rangle \textbf{then} \langle \text{statement}_1 \rangle \textbf{else} \langle \text{statement}_2 \rangle \textbf{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
Kernel Language Approach

- Provides useful abstractions for the programmer
- Can be extended with linguistic abstractions
- Is easy to understand and reason with
- Has a precise (formal) semantics
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

```plaintext
if N==1 then [1]
else
    local L in
    ...
end
end
```

```plaintext
if N==1 then [1]
else L in
    ...
end
```
Approaches to semantics

Programming Language

- Operational model
- Kernel Language
- Formal calculus
- Abstract machine

Aid the programmer in reasoning and understanding
Mathematical study of programming (languages)
\( \lambda \)-calculus, predicate calculus, \( \pi \)-calculus
Aid to the implementer
Efficient execution on a real machine
Exercises

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

36. 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:
     $$((\lambda x.x \; \lambda y.y) \; \lambda z.z)$$

37. The grammar

   $$<\text{exp}> ::= <\text{int}> \mid <\text{exp}> <\text{op}> <\text{exp}>$$
   $$<\text{op}> ::= + \mid *$$

   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.