Principles of Programming Languages II

Wael Aboelsaadat
wael@cs.toronto.edu
http://www.cs.toronto.edu/~wael/324.html

Today

• Grammar
• Introduction to Functional Programming

Administrivia

• Office hour(s)
• Feedback

Grammar

Grammar: introduction

• Grammar:
  – A Grammar is a formalism that describes which sequence of terminals are meaningful in a PL. Mathematically, it is defined as a quadruple \( (N, T, P, S) \) where:
    • \( N \) is the set of symbols called Nonterminals
    • \( T \) is the set of symbols called Terminals
    • \( P \) is the set of productions
    • \( S \) subset of \( N \) is the nonterminal called the starting symbol
  – Example:
    \[ G = (N, T, P, S) \text{ where } N = \{S\}, T = \{a, b\}, P = \{S \rightarrow aS, S \rightarrow bS, S \rightarrow \} \]

• Production:
  – A production is a rule of the form \( X \rightarrow Y \) where \( X \) is a string of symbols (terminals or nonterminals) containing at least one nonterminal and \( Y \) is a string of symbols (terminals or nonterminals)

Grammar: context free

• A context free grammar (CFG) is a grammar in which \( |X| = 1 \), i.e. \( X \) is a single nonterminal
  – LHS: 1 nonterminal
  – RHS: a sequence of terminals and nonterminals
  – E.g.
    • \( S \rightarrow \) (CFG)
    • \( SA \rightarrow \) (nonCFG)

• CFG is sufficient to describe most of the constructs in programming languages

• Programming languages describable by CFG are recognizable by push down automata (analogous to FSA with a stack)
Grammar: backus Naur form

- Backus Naur Form (BNF) is a metalanguage for describing programming languages
- A BNF grammar is a context free grammar

Notation:
- Nonterminals are enclosed in angle brackets, i.e., "<" and ">
- Uses ::= instead of "à" in productions
- Productions having the same left hand side can be grouped together using the alteration symbol "|"
  * e.g. <S> ::= a <S> | b
- Lists are described using recursive rules
  * e.g. <ident.list> ::= identifier | identifier, <ident.list>

Grammar: extended BNF

- Notation:
  - (… |… |… ) Any one of the alterations
  - [... ] Optional part
  - (… )* or {… } or [... ]* repeat zero or more times
  - (… ) - or {… } - or [... ] - repeat one or more times
  - "x" or 'x' terminal symbol
  - Unquoted words non-terminal symbol

Example:
- Using the above notation
  <expression> ::= <expression> + <term> | <expression> - <term> | <term>

Grammar: BNF recursive rules

- Left Recursive BNF Grammar:
  - A BNF grammar rule is left recursive if its LHS appears at the left end of the RHS
  * e.g. <ident.list> ::= <ident.list> , identifier | identifier
- Right Recursive BNF Grammar:
  - A BNF grammar rule is right recursive if its LHS appears at the right end of the RHS
  * e.g. <ident.list> ::= identifier | identifier, <ident.list>

The order of recursion has implications on the order of evaluation and associativity.

Grammar: derivation types & parsers

- Leftmost Derivation:
  - In a leftmost derivation, the replaced nonterminal is always the leftmost nonterminal
- Rightmost Derivation:
  - In a rightmost derivation, the replaced nonterminal is always the rightmost nonterminal.

Typically, language parsers either do leftmost or rightmost derivation on the grammar
- LR parsers: left-to-right, leftmost derivation
- LL parsers: left-to-right, left most derivation

Grammar: definitions ...

- Sentence
  - A finite sequence of terminals constructed according to the rules of the grammar for that PL
- Sentential form
  - A finite sequence of terminals and non-terminals constructed according to the rules of the grammar for that PL
- Derivation
- Parse

Grammar: derivation

Given the grammar

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;letter&gt;</td>
<td>(a</td>
</tr>
<tr>
<td>&lt;digit&gt;</td>
<td>(0</td>
</tr>
<tr>
<td>&lt;identifier&gt;</td>
<td>&lt;letter&gt;</td>
</tr>
<tr>
<td>&lt;assign-stat&gt;</td>
<td>&lt;identifier&gt; = 0 [4]</td>
</tr>
</tbody>
</table>

Can we generate 'x2=0' from these rules?
- <assign-stat> => <identifier> = 0 (using 4)
- <identifier> = 0 => <digit> = 0 (using 4)
- <letter> => <digit> = 0 (using 4)
- x => <digit> = 0 (using 1)
- x => 2 => 0 (using 2)

Yes! This is a derivation of a sentence in the language described by the grammar above. Each sequence in this derivation is a sentential form. At each step, the rule indicated is used to substitute the rhs of the rule for the leftmost non-terminal in the sentential form.
Grammar: parsing

• Given the grammar

  \[
  \begin{align*}
  \text{<letter>} & ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z \quad [1] \\
  \text{<digit>} & ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \quad [2] \\
  \text{<identifier>} & ::= \text{<letter>} | \text{<identifier>} \text{<letter>} | \text{<identifier>} \text{<digit>} \quad [3] \\
  \text{<assign-stmt>} & ::= \text{<identifier>} = 0 \quad [4]
  \end{align*}
  \]

• Can we recognize \( x^2 = 0 \) from these rules?

  \[
  x^2 = 0 \rightarrow \text{<letter>} 2 = 0 \quad (\text{using 1})
  \]

  \[
  \rightarrow \text{<identifier>} 2 = 0 \quad (\text{using 3a})
  \]

  \[
  \rightarrow \text{<identifier>} \text{<digit>} = 0 \quad (\text{using 2})
  \]

  \[
  \rightarrow \text{<identifier>} = 0 \quad (\text{using 3c})
  \]

  \[
  \rightarrow \text{<assign-stmt>} \quad (\text{using 4})
  \]

• This is a parse of the sentence \( x^2 = 0 \)

Grammar: building parse trees

\[
\begin{align*}
\text{<assign-stmt>} & \rightarrow \text{<identifier>} \text{<digit>} \quad \text{(using 1)} \\
& \rightarrow \text{<identifier>} = 0 \\
& \rightarrow \text{<assign-stmt>}
\end{align*}
\]

A parse tree describes the hierarchical syntactic structure of the sentence based on a given language.

• In a parse tree
  – Each internal node is a non-terminal, its children are the rhs of a rule for that non-terminal.
  – All leaves are terminals

Grammar: grammars are not unique

\[
\begin{align*}
\text{<letter>} & ::= a | b | c | | | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z \\
\text{<digit>} & ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \\
\text{id} & ::= \text{<letter>} | \text{id} \text{<letterordigit>} \\
\text{assign-stmt} & ::= \text{id} = 0 \\
\text{letterordigit} & ::= \text{<letter>} | \text{<digit>}
\end{align*}
\]

• This grammar generates the same language (i.e. set of trees whose frontiers/leafs are the same), but has different parse trees than the previous grammar. Many grammars can correspond to 1 PL, but only 1 PL should correspond to any context-free language.

Grammar: ambiguity

• Ambiguity
  – If there are 2 different derivations (or alternatively 2 parse trees) for the same sentence then the grammar is ambiguous.

• There is no algorithm which can examine an arbitrary context-free grammar and tell if is is ambiguous or not
  – This is undecidable

• There is no algorithm which can examine two arbitrary context-free grammars and tell if they generate the same language
  – This is undecidable

Grammar: inherently ambiguous

• Sometimes we can remove an ambiguity from a grammar by restructuring the productions but it is not always possible.

• An inherently ambiguous language does not posses an unambiguous grammar.

Consider \( L = \{ a^i b^j c^k \mid i = j \text{ or } j = k \text{ for } i, j, k > 1 \} \) generated by the grammar:

\[
\begin{align*}
\text{S} & ::= \text{L} \text{ D} \\
\text{L} & ::= \text{a} \text{ L} | \text{b} \\
\text{D} & ::= \text{c} \text{ D} | \text{d} \\
\text{A} & ::= \text{a} \\
\end{align*}
\]

Problem is \( L \) contains a non-context free language \( a^n b^n c^n \text{ where } n \geq 1 \).

Grammar: sources of ambiguity

• Operator Problem: associativity and precedence
  – E.g.
  – Procedure of multiplication/subtraction/addition/…
  – Solution:
  – Change the grammar to reflect operator precedence
  – \( X \cdot Y \cdot Z \text{ means } (X \cdot Y) \cdot Z \)

• Obscure recursion
  – E.g.
  – \( \exp \rightarrow \exp \exp \)
  – \( A \rightarrow AB \)
  – Solution:
  – ???

• Substructure Problem: extent of a substructure
  – E.g.
  – Dangling else
  – Solution:
  – Coming slides!
Grammar: is this ambiguous?

Grammar: is it really a problem?

Yes, because the sentence \( A = B - C - A \) has two different parse trees.
The grammar does not force “normal” left-to-right evaluation of addition and subtraction.

Yes, because the sentence \( A = B - C - A \) has two different parse trees.
The grammar does not force “normal” left-to-right evaluation of addition and subtraction.

The operation of addition is associative in mathematics. Hence \( A + B + C \) can be performed as either \((A + B) + C\) or \(A + (B + C)\).

The multiply operation is also associative.

Therefore one might say the previous ambiguous grammar would be satisfactory for addition and multiplication.
But would it?

Grammar: substructure problem, how to solve ambiguity?

• Use block structure to enclose if statement (e.g. Algol60)
  - E.g. if \( s = 0 \) then begin \( d = 0 \); \( x = 1 \) end else \( w = 2 \).

• Use statement begin/end markers (e.g. Algol68)
  - E.g. if \( x \neq 0 \) then \( fi \) else \( fi \).

• Change the if statement grammar to disallow parse tree 2; that is, always associate an else with the closest if (e.g. Pascal)

• Grammar:
  \[<\text{assign}> :: = <\text{identifier}> <\text{expression}> \]
  \[<\text{identifier}> :: = \text{ABC} \]
  \[<\text{expression}> :: = <\text{expression}> + <\text{term}> \]
  \[<\text{term}> :: = <\text{term}> * <\text{factor}> \]
  \[<\text{factor}> :: = ( <\text{expression}> ) | <\text{identifier}> \]

Grammar: If-then-else

• Grammar:
  \[<\text{if stmt}> :: = \text{if} \ <\text{logic expression}> \text{then} \ <\text{stmt}> \]

  \[<\text{stmt}> :: = <\text{if stmt}> | . . . \]

• Consider:
  if \( \text{logic-expression} \) then
  if \( \text{logic-expression} \) then
  statement 1
  else
  statement 2

• Grammar:
  \[<\text{if stmt}> :: = \text{if} \ <\text{logic expression}> \text{then} \ <\text{stmt}> \]

  \[<\text{stmt}> :: = <\text{if stmt}> | . . . \]

• Example:
  if \( x = 0 \) then
  if \( y = 0 \) then \( z := 1 \)
  else \( w := 2 \)

• Use block structure to enclose if statement (e.g. Algol60)
  - E.g. if \( x = 0 \) then begin \( y = 0 \); \( z = 1 \) end else \( w = 2 \).

• Use statement begin/end markers (e.g. Algol68)
  - E.g. if \( x = 0 \) then \( fi \) else \( fi \).

Grammar: If-then-else

• Grammar:
  \[<\text{if stmt}> :: = \text{if} \ <\text{logic expression}> \text{then} \ <\text{stmt}> \]

  \[<\text{stmt}> :: = <\text{if stmt}> | . . . \]

• Consider:
  if \( \text{logic-expression} \) then
  if \( \text{logic-expression} \) then
  statement 1
  else
  statement 2

• Use block structure to enclose if statement (e.g. Algol60)
  - E.g. if \( x = 0 \) then begin \( y = 0 \); \( z = 1 \) end else \( w = 2 \).

• Use statement begin/end markers (e.g. Algol68)
  - E.g. if \( x = 0 \) then \( fi \) else \( fi \).

• Change the if statement grammar to disallow parse tree 2; that is, always associate an else with the closest if (e.g. Pascal)
Grammar: If-then-else, Pascal solution

```
Start ::= Stmt
Stmt ::= Stmt1 | Stmt2
Stmt1 ::= IF Logical THEN Stmt1 ELSE Stmt2 | Stmt1
Stmt2 ::= IF Logical THEN Stmt2 IF Logical THEN Stmt2 | Stmt2
Logical ::= id | id=Exp | Logical OR Logical
Logical ::= id | id=id
Logical ::= id | id=id
```

Note: only if statements with IF THEN ELSE are allowed, else you have to choose one of an IF THEN ELSE statement. `if x=0 THEN f = 0 ELSE f = x^2`.

---

Functional Programming Languages

FPL

FPL: why?

- “Can programming be liberated from the Von Neumann style?”
  John Backus

- Problems with Imperative programming languages:
  - Von Neumann bottleneck (i.e. fetching words across bus)
  - Assignment
  - Side effects
  - State-based transformation

- FPL alternative:
  - Goal? mimic mathematical functions to the greatest extent possible
  - How? Use λ calculus for the computation
  - The program is a mathematical function

FPL based solutions:

- Hardware: Symbolics Machine, TI Explorer ...
- Software: Lisp, Scheme, ML, Haskell, Miranda ...

FPL: mathematical vs. imperative functions

- Recall: how do imperative functions work?
  - Specify a sequence of operations on values in memory to produce a value
  - Evaluation is controlled by sequencing and iteration

- Why are mathematical functions different?
  - The value is defined and not produced
  - Evaluation order is controlled by recursion and conditional expressions

- Example:
  - Write a procedure to implement the following function: `f(x) = x * x / 3`
    - Imperative:
      ```
      procedure float foo(var int x)
      begin
      int product;
      float quotient;
      product := x * x;
      quotient := product / 3;
      return quotient;
      end;
      ```

    - Functional: ??

FPL: desiderata

A program consists of:
- Function definitions
- Function calls
- There is no other structure.

Control flow:
- Recursion and function application is the only way to achieve repetition

No assignment
- Values are bound to values only through parameter association

No side effects
- A function may not change its parameters
- A function cannot do input or output

No variables
- No variable declarations
- No explicit typing

FPL: desiderata cont’d

- Implicit memory management:
  - no new or free
  - Program unaware of underlying memory structure

- Referential transparency:
  - Execution of a function always produces the same result when given the same parameters
  - Implication: all variables in a function body must be local to that function; why?

- Functions can be:
  - Passed as an argument
  - Returned from a function
  - Represented by a data structure and that data structure can then be evaluated

- Is this possible?
FPL: \( \lambda \) calculus

- **What is a Calculus?**
  - A method of analysis or calculation using a special symbolic notation
    - E.g.
      - differential calculus and integral calculus (mathematical calculus/calculus)
      - Bi-calculus
    - Defined by Alonzo Church, a logician, in 1930s as a computational theory of recursive functions
  - \( \lambda \) calculus is equivalent in computational power to Turing machines

- **Recall: what’s a Turing machine?**
  - Turing machines are abstract machines that emphasize computation as a series of state transitions driven by symbols on an input tape (which leads naturally to an imperative style of programming based on assignment)

- **How is \( \lambda \) calculus different?**
  - \( \lambda \) calculus emphasizes typed expressions and functions (which naturally leads to a functional style of programming)
  - No state transitions.

FPL: \( \lambda \) calculus & functional forms

- A higher-order function (functional form) is one that:
  - Takes functions as parameters
  - Yields a function as a result
  - E.g.:
    - \( f(x) = x \cdot 2 \), \( g(x) = 3 \cdot x \)
      - Then, \( h(x) = f(g(x)) \) or \( h(x) = (3 \cdot x) + 2 \)
    - \( h(x) \) is called a higher-order function

- **Types of Functional Forms:**
  - Construction form:
    - E.g. \( g(x) = x \cdot x \), \( h(x) = 2 \cdot x \), \( i(x) = x / 2 \)
      - \( [g,h,i](4) = (16,8,2) \)
  - Apply-to-all form:
    - E.g. \( h(x) = x \cdot x \)
      - \( y(h,(2,3,4)) = (4,9,16) \)

FPL: \( \lambda \) calculus turing complete?

- Can we represent the class of Turing computable functions?
  - Yes, we can represent:
    - Booleans and conditional functions
    - Numerals and arithmetic functions
    - Data structures, such as ordered pairs, lists, etc…
    - Recursion

- Doing so however is syntactically tedious:
  - Actual programming have a lot of syntactic sugar (gives the programmer an alternative way of coding that is more succinct or more like some familiar notation)
  - \( \lambda \) calculus is more suitable as an abstract model of a programming language rather than a practical programming language

Both Turing machines and \( \lambda \) calculus are idealized, mathematical models of computation

Scheme: introduction

- **History:**
  - Scheme is based on lambda calculus; that is the meaning of all syntactic programming constructs in the language are defined in terms of mathematical functions
  - A Scheme program consists of function definitions and calls. There is no other structure.
  - Automatic garbage collection.

- A variable assumes the type of the value that is bound to them at run-time. So, the type of a variable changes dynamically during execution

Scheme: basic data types
Scheme: expressions

• An expression in Scheme has the form \((E_1 E_2 E_3 \ldots E_n)\)
  - \(E_1\) evaluates to an operator
  - \(E_2\) through \(E_n\) are evaluated as operands

• Examples:
  - \((+ a b c)\) \((a + b + c)\)
  - \((+ 1 (* 2 3) 4 5)\) \((1 + (2 * 3) + 4 + 5)\)
  - \((- 6 3 / (10 2) 2 (* 2 3))\) \(16\)
  - \((- (5 3) (2 1 3 5) 14)\) \(17\)
  - \((\text{max} (+ 2 3) (\text{abs} -4) \text{remainder} 12 5))\) \(16\)

• Postfix vs. Infix:
  - Scheme expressions use prefix notation while imperative languages use infix notation, which is better?

Scheme: evaluating expressions

• `eval FORM`
  - Evaluate `FORM` to compute/fetch value of an expression
  - Form: an expression to be evaluated
  - Rules:
    - A number evaluates to itself
    - A variable evaluates to its value
    - A quoted symbol evaluates to the symbol itself
    - A string evaluates to itself
    - A single quoted list evaluates to a simple list of symbols
    - An unquoted list evaluates to a function call

• `define`
  - E.g.: \((\text{define} \text{sum} \text{'} (+ 1 3 4 5))\)
  - \((\text{sum})\)

Scheme: expression evaluation order

• Scheme follows a depth-first applicative evaluation order
  - Example: \((\text{if} (\text{if} a (b c) e) 8 (f 2 3 2))\)

Scheme: lists

• A list is denoted by a collection of items enclosed in parentheses

• The empty list is denoted \(\text{}\)

• Improper list: a list that does not end with an empty list.

• Lists can be nested
  - E.g. \((a \text{ (b c)} \text{ (d b)})\)

• Note: Lists should be quoted when fed to the interpreter, otherwise the interpreter will try to apply the first item in the list to the other items
  - E.g.: \((\text{quote} (2 4 6 8)\)
  - error: procedure application: expected procedure, given: 2.

Scheme: lists cont’d

• Constructing Lists:
  - `(cons arg1 arg2)`
    - The second argument to `cons` must be a list
      - E.g. \((\text{cons} \text{'} \text{peanut butter and jelly})\)
        - `(peanut butter and jelly)`
  - `(append arg1 arg2)`
    - Returns the list formed by joining the elements of `a` and `b` together.
      - Precondition: `arg1` and `arg2` must be lists
      - E.g. \((\text{list} \text{arg1 arg2 ... argn})\)
**Scheme: lists cont’d**

- **Internal implementation**
  - Linked list storage management used
  - **Head**: first member of the list.
  - **Tail**: everything else other than the head

- **Useful Operations**:
  - `(reverse list)`; reverse the order of the elements in list
  - `(member element list)`; return tail of the list
  - `(car list)`; return head of the list
  - `(cdr list)`; return tail of the list
  - `(cadr list)`; return tail of the list
  - `(cadar list)`; return tail of the list
  - `(caar list)`; return head of the list
  - `(cddr list)`; return tail of the list
  - `(caadr list)`; return tail of the list
  - `(cadadr list)`; return tail of the list

- **Example**:
  - `=> (define lista '(1 2 5 67 3 2 5 88))`
  - `=> (define fruits '(apple pear orange banana))`
  - `=> (define colors '(red blue green yellow orange))`
  - `=> (define prices '((banana 0.98) (orange 0.33) (lemon 0.20)))`
  - `=> (car lista)`
  - `1`
  - `=> (car colors)`
  - `red`
  - `=> (cdr lista)`
  - `(2 5 67 3 2 5 88)`
  - `=> (cdr colors)`
  - `(blue green yellow orange)`
  - `=> (cadr colors)`
  - `blue`
  - `=> (cadr fruits)`
  - `pear`
  - `=> (caddr fruits)`
  - `orange`
  - `=> (cddr fruits)`
  - `(banana)`
  - `=> (car prices)`
  - `(banana 0.98)`
  - `=> (caar prices)`
  - `banana`
  - `=> (cadar prices)`
  - `0.98`

**Scheme: MIT interpreter**

```
<Scheme: MIT interpreter>
```

**Scheme: read-evaluate-print Cycle**

- **Read** input from user:
  - A function definition or abstraction
  - A function evaluation
- **Evaluate** input:
  - Store function definition (e1 e2 e3 … ek)
  - Evaluate e1 to obtain a function
  - Evaluate e2, … , ek to values
  - Execute function body using values from previous step as formal parameter values
  - Return value of function
- **Print** return value