

Welcome to CS560 Programming Languages



Dr. Amie Souter

Announcements

- Assignment 1
 - due next week in class
- Term Project - First Part
 - due Feb 3rd (in class)
- Change in email address:
 - yogi @ plan.cs.drexel.edu
- Any Questions??

Class Tonight

- How do we specify tokens?
- How do we specify grammars?
- Lambda Calculus
- Functional vs Imperative Programming

Specification of Programming Languages

- PLs require precise definitions (i.e. no ambiguity)
 - Language *form* (Syntax)
 - Language *meaning* (Semantics)
- Consequently, PLs are specified using formal notation:
 - Formal syntax
 - Tokens
 - Grammar
 - Formal semantics

Syntax and Semantics of Programs

- Syntax
 - The symbols used to write a program
- Semantics
 - The actions that occur when a program is executed
- Programming language implementation
 - Syntax → Semantics
 - Transform program syntax into machine instructions that can be executed to cause the correct sequence of actions to occur

Regular Expressions

- A regular expression (RE) is:
 - A single character
 - The empty string, ϵ
 - The concatenation of two regular expressions
 - *Notation:* $RE_1 RE_2$ (i.e. RE_1 followed by RE_2)
 - The union of two regular expressions
 - *Notation:* $RE_1 | RE_2$
 - The closure of a regular expression
 - *Notation:* RE^*
 - * is known as the *Kleene star*
 - * represents the concatenation of 0 or more strings

Token Definition Example

- Numeric literals in Pascal
 - Definition of the token *unsigned_number*
 $digit \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
 $unsigned_integer \rightarrow digit\ digit^*$
 $unsigned_number \rightarrow$
 $unsigned_integer ((. unsigned_integer) | \epsilon)$
 $((e | + | - | \epsilon) unsigned_integer) / \epsilon$

- Recursion is not allowed!

Exercise

- Describe the language denoted by the following regular expressions:
 1. $a(a|b)^*a$
 2. $a^*ba^*ba^*ba^*$
 3. $(aa|bb)^*b$
- Write regular expressions for the following languages:
 - the set of identifiers that can contain either upper or lower case letters, digits, or underscores, but can only start with a capital letter and end with a capital letter

Syntax Analysis

- **Syntax:**
 - Webster's definition: *1 a : the way in which linguistic elements (as words) are put together to form constituents (as phrases or clauses)*
- **The syntax of a programming language**
 - Describes its form
 - *i.e. Organization of tokens (elements)*
 - Formal notation
 - Context Free Grammars (CFGs)

Context Free Grammars

- **CFGs**
 - Add recursion to regular expressions
 - Nested constructions
 - Notation
 - $expression \rightarrow identifier \mid number \mid - expression$
 - $\mid (expression)$
 - $\mid expression operator expression$
 - $operator \rightarrow + \mid - \mid * \mid /$
 - Terminal symbols
 - Non-terminal symbols
 - Production rule (i.e. substitution rule)

Backus-Naur Form

- **Equivalent to CFGs in power**
 - CFG
 - $expression \rightarrow identifier \mid number \mid - expression$
 - $\mid (expression)$
 - $\mid expression operator expression$
 - $operator \rightarrow + \mid - \mid * \mid /$
 - BNF
 - $(expression) \rightarrow (identifier) \mid (number) \mid - (expression)$
 - $\mid ((expression))$
 - $\mid (expression)(operator)(expression)$
 - $(operator) \rightarrow + \mid - \mid * \mid /$



Derivations

- **A derivation shows how to generate a syntactically valid string**
 - Given a CFG
 - Example:

$expression \rightarrow identifier \mid number \mid - expression$
 $\mid (expression)$
 $\mid expression operator expression$
 $operator \rightarrow + \mid - \mid * \mid /$

- Derivation of

$slope * x + intercept$

Derivation Example

- Derivation of $\text{slope} * x + \text{intercept}$

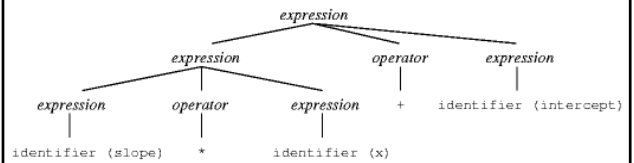
$\text{expression} \Rightarrow \text{expression operator expression}$
 $\Rightarrow \text{expression operator intercept}$
 $\Rightarrow \text{expression} + \text{intercept}$
 $\Rightarrow \text{expression operator expression} + \text{intercept}$
 $\Rightarrow \text{expression operator } x + \text{intercept}$
 $\Rightarrow \text{expression} * x + \text{intercept}$
 $\Rightarrow \text{slope} * x + \text{intercept}$

$\text{expression} \Rightarrow * \text{slope} * x + \text{intercept}$

- Identifiers were not derived for simplicity

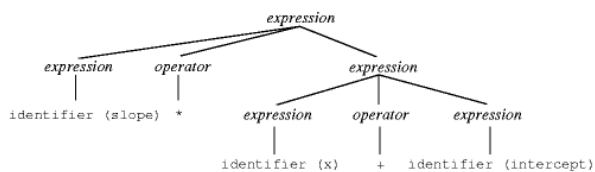
Parse Trees

- A parse tree is graphical representation of a derivation
- Example



Ambiguous Grammars

- Alternative parse tree
 - same expression
 - same grammar



- This grammar is ambiguous

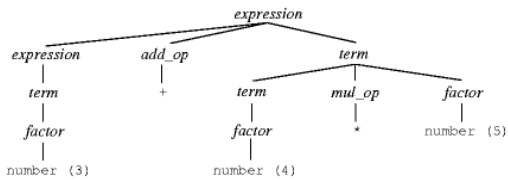
Designing unambiguous grammars

- Specify more grammatical structure
 - In our example, left associativity and operator precedence
 - $10 - 4 - 3$ means $(10 - 4) - 3$
 - $3 + 4 * 5$ means $3 + (4 * 5)$

- (1) $\text{expression} \rightarrow \text{term} \mid \text{expression add_op term}$
- (2) $\text{term} \rightarrow \text{factor} \mid \text{term mult_op factor}$
- (3) $\text{factor} \rightarrow \text{identifier} \mid \text{number} \mid - \text{factor} \mid (\text{expression})$
- (4) $\text{add_op} \rightarrow + \mid -$
- (5) $\text{mult_op} \rightarrow * \mid /$

Example

- Parse tree for $3 + 4 * 5$



- Exercise: parse tree for

$- 10 / 5 * 8 - 4 - 5$

Examples of Real-life Ambiguity

- Consider the grammar:
 - statement ::= conditional
 - conditional ::= if (expression) statement
 - conditional ::= if (expression) statement
else statement
- and try to parse:
 - if (i>1) if (i>2) j=1 else j=2

Exercise

sentence ::= noun verb object
 noun ::= cat | dog
 verb ::= eats | likes | abhors
 object ::= noun | mouse | vacuum

Compute parse tree and derivation showing
 "dog likes cats" is a sentence.

Add the production rules:

sentence ::= noun action

action ::= likes mouse

Is the resulting grammar ambiguous?

Theoretical Foundations

- Many foundational systems
 - Computability Theory
 - Program Logics
 - Lambda Calculus
 - Denotational Semantics
 - Operational Semantics
 - Type Theory
- Consider two of these methods
 - Lambda calculus (syntax, operational semantics)
 - Denotational semantics

Lambda Calculus

- Formal system with three parts
 - Notation for function expressions
 - Proof system for equations
 - Calculation rules called *reduction*
- Additional topics in lambda calculus
 - Mathematical semantics (=model theory)
 - Type systems

We will look at syntax, equations and reduction

There is more detail in book than we will cover in class

History

- Original intention
 - Formal theory of substitution
- More successful for computable functions
 - Substitution \rightarrow symbolic computation
 - Church/Turing thesis
- Influenced design of Lisp, ML, other languages
- Important part of CS history and theory



Why study this now?

- Basic syntactic notions
 - Free and bound variables
 - Functions
 - Declarations
- Calculation rule
 - Symbolic evaluation useful for discussing programs
 - Used in optimization (in-lining), macro expansion
 - Illustrates some ideas about scope of binding

Expressions and Functions

- Expressions
 $x + y$ $x + 2*y + z$
- Functions
 $\lambda x. (x+y)$ $\lambda z. (x + 2*y + z)$
- Application
 $(\lambda x. (x+y)) 3$ $= 3 + y$
 $(\lambda z. (x + 2*y + z)) 5$ $= x + 2*y + 5$

Parsing: $\lambda x. f (f x) = \lambda x. (f (f (x)))$

Declarations as "Syntactic Sugar"

```
function f(x)
  return x+2
end;
f(5);
```

$(\lambda f. f(5))$ $(\lambda x. x+2)$
 block body declared function

$\text{let } x = e_1 \text{ in } e_2 = (\lambda x. e_2) e_1$

Higher-Order Functions

◆ Given function f , return function $f \circ f$

$\lambda f. \lambda x. f (f x)$

◆ How does this work?

$(\lambda f. \lambda x. f (f x)) (\lambda y. y+1)$
 $= \lambda x. (\lambda y. y+1) ((\lambda y. y+1) x)$
 $= \lambda x. (\lambda y. y+1) (x+1)$
 $= \lambda x. (x+1)+1$

Same result if step 2 is altered.

Same procedure, Lisp syntax

- Given function f , return function $f \circ f$
 $(\text{lambda } (f) (\text{lambda } (x) (f (f x))))$
- How does this work?
 $((\text{lambda } (f) (\text{lambda } (x) (f (f x)))) (\text{lambda } (y) (+ y 1)))$
 $= (\text{lambda } (x) ((\text{lambda } (y) (+ y 1)) ((\text{lambda } (y) (+ y 1)) x))))$
 $= (\text{lambda } (x) ((\text{lambda } (y) (+ y 1)) (+ x 1))))$
 $= (\text{lambda } (x) (+ (+ x 1) 1))$

CS560 Winter 2004 Drexel University

1/20/2005 32

Free and Bound Variables

- ◆ Bound variable is "placeholder"
 - Variable x is bound in $\lambda x. (x+y)$
 - Function $\lambda x. (x+y)$ is same function as $\lambda z. (z+y)$
- ◆ Compare
 $\int x+y dx = \int z+y dz \quad \forall x P(x) = \forall z P(z)$
- ◆ Name of free (=unbound) variable does matter
 - Variable y is free in $\lambda x. (x+y)$
 - Function $\lambda x. (x+y)$ is *not* same as $\lambda x. (x+z)$
- ◆ Occurrences
 - y is free and bound in $\lambda x. ((\lambda y. y+2) x) + y$

Reduction

- Basic computation rule is β -reduction

$$(\lambda x. e_1) e_2 \rightarrow [e_2/x]e_1$$

where substitution involves renaming as needed
(next slide)

- **Reduction:**
 - Apply basic computation rule to any subexpression
 - Repeat
- **Confluence:**
 - Final result (if there is one) is uniquely determined

Rename Bound Variables

- ◆ Function application

$$(\lambda f. \lambda x. f (f x)) (\lambda y. y+x)$$

apply twice add x to argument

- ◆ Substitute "blindly"

$$\lambda x. [(\lambda y. y+x) ((\lambda y. y+x) x)] = \lambda x. x+x+x$$

- ◆ Rename bound variables

$$(\lambda f. \lambda z. f (f z)) (\lambda y. y+x)$$

$$= \lambda z. [(\lambda y. y+x) ((\lambda y. y+x) z)] = \lambda z. z+x+x$$

Easy rule: always rename variables to be distinct

Functional vs Imperative Programs

Assumptions:

Functional languages support higher-order functions, garbage collection, no assignment.

Imperative languages support assignment, but not higher-order functions or GC.

1. Are there inherent reasons why imperative languages are superior to functional ones for the majority of programming tasks?
2. Which is easier to implement on a machine with limited disk and memory?
3. Which require bigger executables when compiled?
4. What consequences might these facts had in the early days of computing?
5. Are these concerns still valid?

QUESTIONS?