Syntax and Semantics of Programming Languages

CSIT 313

Fall 2013

J.W. Benham

Preliminaries

- Need for precise specifications
  - Initial evaluators
  - Implementors
  - Users
- **Syntax** describes the *form* of a language's expressions, statements, and program units
- **Semantics** describes the *meaning* of those expressions, statements, and program units
- Semantics should follow syntax closely

Describing Syntax (1): Lexemes and Tokens

- A **language** is a set of strings from some alphabet.
  - The strings in a language are called **sentences** of the language
- To simplify the description, the lowest-level syntactic units, called **lexemes** are often described separately, given by a lexical specification
- A **token** is a category of lexemes that have a similar structure and function within the language.

Describing Syntax (2): Backus-Naur Form and Context-Free Grammars

- Language recognizers and language generators
- Chomsky’s language classification
  - Regular languages
  - Context-free languages
- Backus-Naur Form
- Terminals, non-terminals, and productions

Example: A Simple Grammar

<assign> → <var> = <expr>

<var> → x | y | z

<expr> → <var> + <var> | <var> * <var> | <var>

- **Derivations** and **sentential forms**
- Example: derivation of x = y * z

<assign> => <var> = <expr>
=> x = <expr>
=> x = <var> * <var>
=> x = y * <var>
=> x = y * z

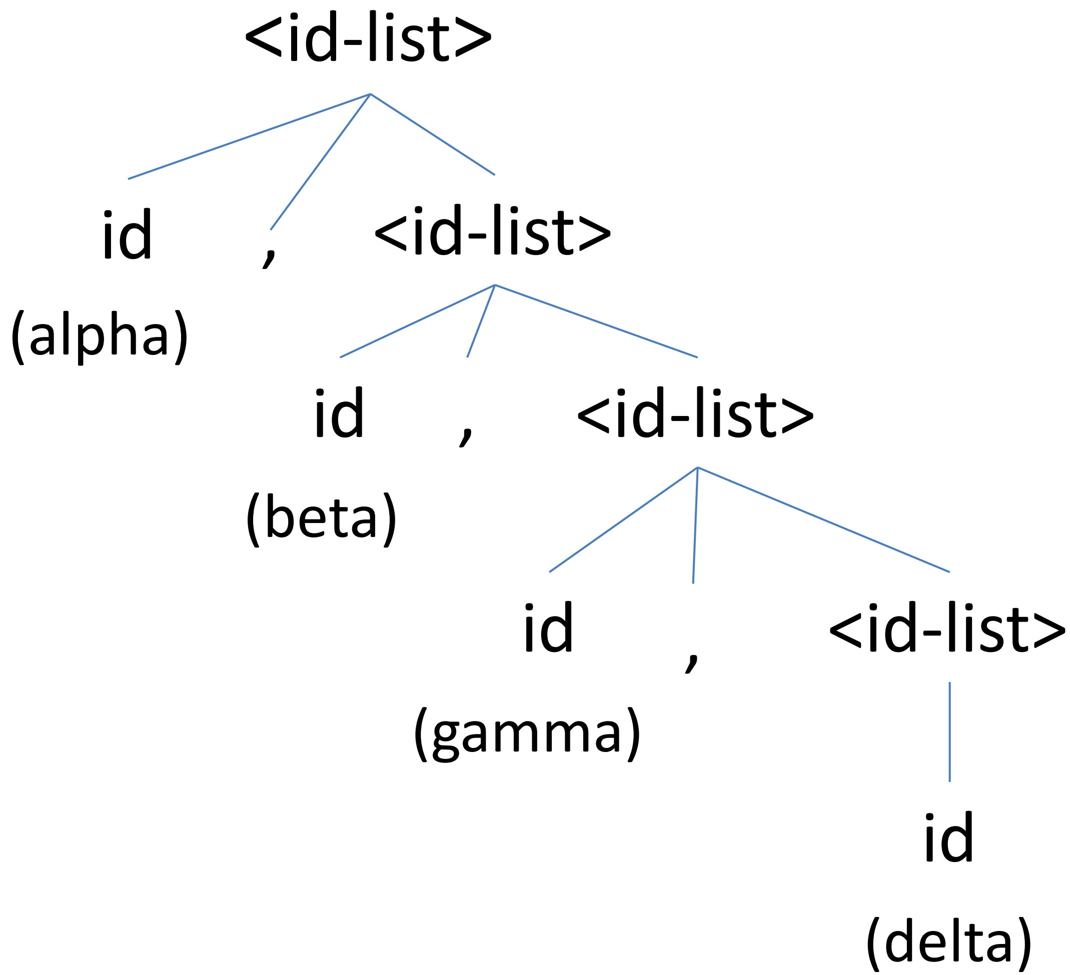
Recursive Productions

- Example: identifier lists
- First attempt: <id-list> → ident | ident,…,ident
  - What does the ellipsis (…) mean?
  - Difficult, if not impossible, to write a derivation
- Second attempt (using recursion)
  - <id-list> → id | id , <id-list>
  - **Class exercise:** Give leftmost derivation for the list "alpha, beta, gamma, delta" (lexemes) or "ident, ident, ident, ident" (tokens)

Parse Trees

- Starting non-terminal at root
- Internal nodes are non-terminals and leaves are terminals
- Children of any internal node are from right-hand-side of some production for its non-terminal

Parse Tree Example



Ambiguous Grammars

- An **ambiguous grammar** produces more that one parse tree for the same sentence
- Example:
  <assign> → id = <expr>
  <expr> → <expr> + <expr> | <expr> * <expr> |
  (<expr>) | id | int-literal
- Parse trees for **alpha = alpha + 12 * beta** on next slides

Disambiguating the Grammar

<assign> → id = <expr>
<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → (<expr>) | id | int-literal

Notes:

1. We had to add extra non-terminals
2. Not all languages have unambiguous grammars
3. **<u>Classroom exercise</u>**: Use the same assignment statement as above to show that this grammar is unambiguous (at least for that statement)

The Dangling Else Problem

<stmt> → <assign> | <if-stmt>

<assign> → id = <expr>

<expr> → … (as on previous slide)

<If-stmt> → **if** (<logical-expr>) <stmt> |
if (<logical-expr>) <stmt> **else** <stmt>

<logical-expr> → id relop id

(lexemes for token relop: ==, <, <=, >, >=, <>)

Example: if (a < b) if (b < c) d = c-a else d = a – b

<u>**Classroom exercise**</u>: Show this grammar is ambiguous

An Unambiguous Grammar for if-else

<stmt> → <matched> | <unmatched>

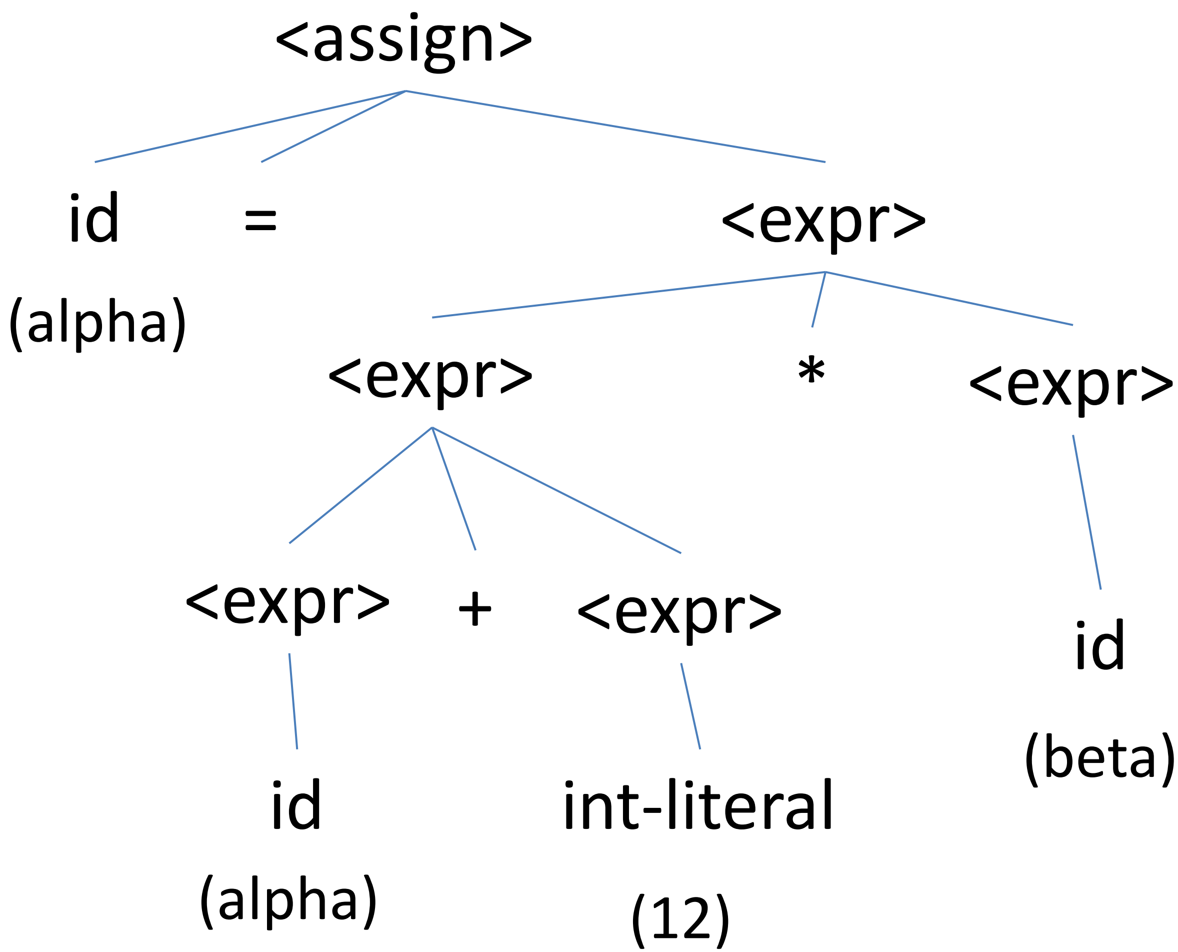
<matched> → <assign> |
    if (<logical-expr>) <matched> **else** <matched>

<unmatched> → **if** (<logical-expr>) <stmt> |

    if (<logical-expr>) <matched> **else** <unmatched>

Classroom exercise: Use the same if-else statement to show this grammar is unambiguous (at least for that statement)

First Parse Tree



Second Parse Tree

- **Classroom exercise**: Produce a different parse tree for this sentence.
- Why are ambiguous grammars a problem?
- Which of these two parse trees is "correct"? **Why?**

Semantics: Attribute Grammars

- **Static semantics**
  - Type constraints and compatibility rules
  - Requiring variables to be declared before they are used
- **Attribute grammars** associate one or more **attributes** with symbols of the grammar
  - Functions to compute some attribute values from others
  - Synthesized attributes
  - Inherited attributes
  - Intrinsic attributes

Attribute Grammars (1)

- Associate a set A(X) of attributes with each grammar symbol X – the union of disjoint subsets S(X) and I(x)
- For attributes in S(X) and productions $X_0 \rightarrow X_1\ X_2\ X_3\ \ldots\ X_n$, there is a *semantic function* $S(X_0) = f(A(X_1),\ldots, A(X_n))$ that defines the synthesized attributes of a parse-tree node in terms of the attribute values of its children.

Attribute Grammars (2)

- For attributes in I(X), we have semantic functions of the form
  $I(X_j) = f(A(X_0),A(X_1),\ldots, A(X_n))$ (with $1 \leq j \leq n$).
  - To avoid circularity, these functions are often restricted to functions
    $I(X_j) = f(A(X_0),A(X_1),\ldots, A(X_{j-1}))$ so value depends of parent and left siblings
- A **predicate function** is a boolean expression on the attribute set $\{A(X_0),A(X_1),\ldots, A(X_n)\}$
  - For program to be valid, every predicate function must be true. A false predicate indicates a violation of syntax of static semantic rules of language.

Simple Attribute Grammar Example: Type Constraints for Assignment

- Syntax: <assign> → var = <expr>
  Semantic: <expr>.expected = var.actual
- Syntax: <expr> → <expr>[2] + <expr>[3]
  Semantic: if (<expr>[2].actual == int
  AND <expr>[3].actual == int)
  THEN <expr>.actual ← int
  ELSE <expr>.actual ← float
  Predicate: <expr>.actual == <expr>.expected
- Syntax: <expr> → var
  Semantic: <expr>.actual
  ← type-lookup(var.string)

Example (continued)

4. Syntax: <expr> → int-literal
   Semantic: <expr>.actual ← int
5. Syntax: <expr> → float-literal
   Semantic: <expr>.actual ← float

Example: Construct the **decorated parse tree** for the assignment alpha = beta + 3.14159, where alpha and beta are declared to be integers. Does this assignment conform to the type rules for the language?

Decorated Parse Tree