Compiler Design

Lecture-8

Ambiguity

Topics Covered

Ambiguity

Famous Ambiguity: “Dangling Else”

Resolving Ambiguity

Ambiguity

- Some grammars are **ambiguous**
  - Multiple distinct parse trees for the same terminal string
- Structure of the parse tree captures much of the meaning of the program
  - ambiguity implies multiple possible meanings for the same program

Famous Ambiguity: “Dangling Else”

```
Stmt ::= ... |
     if ( Expr ) Stmt |
     if ( Expr ) Stmt else Stmt
```

if (e_1) if (e_2) s_1 else s_2 : if (e_1) if (e_2) s_1 else s_2

Resolving Ambiguity

- Option 1: add a meta-rule
  - For example "**`else`** associates with closest previous **`if`**"
    - works, keeps original grammar intact
    - ad hoc and informal

Resolving Ambiguity [continued]

Option 2: rewrite the grammar to resolve ambiguity explicitly

```
Stmt           ::= MatchedStmt | UnmatchedStmt
MatchedStmt    ::= ... |
           if ( Expr ) MatchedStmt else MatchedStmt
UnmatchedStmt ::= if ( Expr ) Stmt |
           if ( Expr ) MatchedStmt else UnmatchedStmt
```

- formal, no additional rules beyond syntax
- sometimes obscures original grammar

Resolving Ambiguity Example

```
Stmt           ::= MatchedStmt | UnmatchedStmt
MatchedStmt    ::= ... |
       if ( Expr ) MatchedStmt else MatchedStmt
UnmatchedStmt ::= if ( Expr ) Stmt |
        if ( Expr ) MatchedStmt else UnmatchedStmt
```

if (e_1) if (e_2) s_1 else s_2

Resolving Ambiguity [continued]

Option 3: redesign the language to remove the ambiguity

```
Stmt ::= ... |
         if Expr then Stmt end |
         if Expr then Stmt else Stmt
  end
```

- formal, clear, elegant
- allows sequence of Stmts in **`then`** and **`else`** branches, no { , } needed
- extra **`end`** required for every **`if`**

Another Famous Example

```
E  ::= E Op E | - E | ( E ) | id
Op ::= + | - | * | /
```

```
a  +  b  *  c  :  a  +  b  *  c
```

Resolving Ambiguity (Option 1)

Add some meta-rules, e.g. precedence and associativity rules

Example:

```
E ::= E Op E | - E | E ++
      | ( E ) | id
Op::=  + | - | * | / | %
      | ** | == | < | &&
      | ||
```

Operator	Preced	Assoc
Postfix ++	Highest	Left
Prefix -		Right
** (Exp)		Right
*, /, %		Left
+, -		Left
==, <		None
&&		Left
||	Lowest	Left

Removing Ambiguity (Option 2)

Option2: Modify the grammar to explicitly resolve the ambiguity

Strategy:

- create a nonterminal for each precedence level
- expr is lowest precedence nonterminal,
  each nonterminal can be rewritten with higher
  precedence operator, highest precedence operator includes atomic exprs
- at each precedence level, use:
  - left recursion for left-associative operators
  - right recursion for right-associative operators
  - no recursion for non-associative operators

Redone Example

```
E  ::= E0
E0 ::= E0 || E1 | E1                    left associative
E1 ::= E1 && E2 | E2                    left associative
E2 ::= E3 (== | <) E3                   non associative
E3 ::= E3 (+ | -) E4 | E4               left associative
E4 ::= E4 (* | / | %) E5 | E5           left associative
E5 ::= E6 ** E5 | E6                    right
  associative
E6 ::= - E6 | E7                        right
  associative
E7 ::= E7 ++ | E8                       left associative
E8 ::= id | ( E )
```