Compiling Techniques

Lecture 6: Ambiguous Grammars and Bottom-Up Parsing

Christophe Dubach

28 September 2018

Ambiguity definition

- If a grammar has more than one leftmost (or rightmost) derivation for a single sentential form, the grammar is ambiguous
- This is a problem when interpreting an input program or when building an internal representation

Ambiguous Grammar: example 1

```
Expr ::= Expr Op Expr | num | id
Op   ::= + | *
```

This grammar has multiple leftmost derivations for x + 2 * y

One possible derivation

```
Expr
Expr Op Expr
id(x) Op Expr
id(x) + Expr
id(x) + Expr Op Expr
id(x) + num(2) Op Expr
id(x) + num(2) * Expr
id(x) + num(2) * id(y)
```

x + (2 * y)

Another possible derivation

```
Expr
Expr Op Expr
Expr Op Expr Op Expr
id(x) Op Expr Op Expr
id(x) + Expr Op Expr
id(x) + num(2) Op Expr
id(x) + num(2) * Expr
id(x) + num(2) * id(y)
```

(x + 2) * y

Ambiguous grammar: example 2

```
Stmt ::= if Expr then Stmt
       | if Expr then Stmt else Stmt
       | OtherStmt
```

input

if E1 then if E2 then S1 else S2

One possible interpretation

```
if E1 then
  if E2 then
    S1
else
  S2
```

Another possible interpretation

```
if E1 then
  if E2 then
    S1
  else
    S2
```

Removing Ambiguity

- Must rewrite the grammar to avoid generating the problem
- Match each else to innermost unmatched if (common sense)

Unambiguous grammar

```
Stmt      ::= if Expr then Stmt
            | if Expr then WithElse else Stmt
            | OtherStmt
WithElse  ::= if Expr then WithElse else WithElse
            | OtherStmt
```

- Intuition: the WithElse restricts what can appear in the then part
- With this grammar, the example has only one derivation

```
Stmt      ::= if Expr then Stmt
            | if Expr then WithElse else Stmt
            | OtherStmt
WithElse  ::= if Expr then WithElse else WithElse
            | OtherStmt
```

Derivation for: if E1 then if E2 then S1 else S2

```
Stmt
if Expr then Stmt
if E1 then Stmt
if E1 then if Expr then WithElse else Stmt
if E1 then if E2 then WithElse else Stmt
if E1 then if E2 then S1 else Stmt
if E1 then if E2 then S1 else S2
```

This binds the else controlling S2 to the inner if.

Exercise:

Remove the ambiguity for the following grammar:

```
Expr ::= Expr Op Expr | num | id
Op   ::= '+' | '*'
```

Deeper ambiguity

- Ambiguity usually refers to confusion in the CFG (Context Free Grammar)
- Consider the following case: `a = f(17)`
  In Algol-like languages, `f` could be either a function of an array
- In such case, context is required
  - Need to track declarations
  - Really a type issue, not context-free syntax
  - Requires en extra-grammatical solution
  - Must handle these with a different mechanism

Step outside the grammar rather than making it more complex. This will be treated during semantic analysis.

Ambiguity Final Words

Ambiguity arises from two distinct sources:

- Confusion in the context-free syntax (*e.g.* if then else)
- Confusion that requires context to be resolved (*e.g.* array vs function)

Resolving ambiguity:

- To remove context-free ambiguity, rewrite the grammar
- To handle context-sensitive ambiguity delay the detection of such problem (semantic analysis phase)
  - For instance, it is legal during syntactic analysis to have: `void i; i=4;`

Bottom-Up Parser

A bottom-up parser builds a derivation by working from the input sentence back to the start symbol.

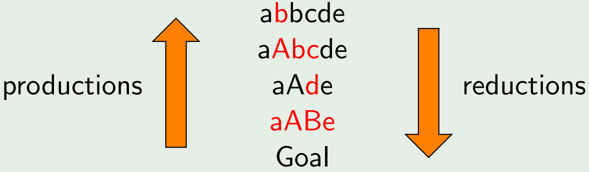
- $S \rightarrow \gamma_0 \rightarrow \gamma_1 \rightarrow \cdots \rightarrow \gamma_{n-1} \rightarrow \gamma_n$
- To reduce γ_i to γ_{i-1}, match some **rhs** β against γ_i then replace β with its corresponding **lhs**, A, assuming $A \rightarrow \beta$

Example: CFG

Goal ::= a A B e
A ::= A b c
A ::= b
B ::= d

Input: abbcde

Bottom-Up Parsing



Note that the production follows a rightmost derivation.

Leftmost vs Rightmost derivation

Example: CFG

Goal ::= a A B e
A ::= A b c | b
B ::= d

Leftmost derivation

Goal
aABe
aAbcBe
abbcBe
abbcde

Rightmost derivation

Goal
aABe
aAde
aAbcde
abbcde

LL parsers

LR parsers

Shift-reduce parser

- It consists of a stack and the input
- It uses four actions:
  1. **shift**: next symbol is shifted onto the stack
  2. **reduce**: pop the symbols $Y_n, \ldots, Y_1$ from the stack that form the right member of a production $X ::= Y_n, \ldots, Y_1$
  3. **accept**: stop parsing and report success
  4. **error**: error reporting routine

How does the parser know when to shift or when to reduce?

Similarly to the top-down parser, can back-track if wrong decision made or try to look ahead.
Can build a DFA to decide when we should shift or reduce.

Shift-reduce parser

Example: CFG

Goal ::= a A B e
A ::= A b c | b
B ::= d

Operation: shift shift reduce shift shift reduce shift reduce shift reduce

Input

abbcde bbcde bcde bcde cde de de e e

Stack

a ab aA aAb aAbc aA aAd aAB aABe Goal

Choice here: shift or reduce?

Can lookahead one symbol to make decision.
(Knowing what to do is not explain here, need to analyse the grammar, see EaC§3.5)

Top-Down vs Bottom-Up Parsing

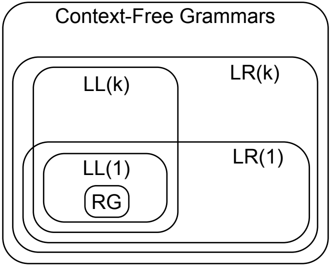
Top-Down

+ Easy to write by hand
+ Easy to integrate with compiler
- Recursion might lead to performance problems (table encoding possible)

Bottom-Up

+ Very efficient
+ Handles left/right recursion
+ Supports a larger classes of grammars
- Requires generation tools
- Rigid integration to compiler

Last words



Language ≠ Grammar

- There is more than one grammar that can used to define a language
- These grammars might be of different “complexity” (LL(1), LL(k), LR(k))
- ⇒ Language complexity ≠ grammar complexity

Next lecture

- Parse tree and abstract syntax tree