Lecture 16

Review

Grammar rewriting, language abstractions, ideas for final projects

Ras Bodik
Ali and Mangpo

Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2013

UC Berkeley

Today

Grammar disambiguation via rewriting

- if-then-else
- google calculator

Modular operators

- queues
- game trees

DSLs

- d3 joins
- ideas for final projects

Grammar rewriting

Why grammar rewriting

Scenario 1: your parser doesn't disambiguate

ie, %left, %right are not supported

Scenario 2: declarative disambiguation too weak

sometimes %left, %right can't help you

Example for scenario 2: 3/4/m/s in the google calc

- parses into ((3/4)/(m/s))
- That is, there is one symbol ('/') which serves two roles
- similar to how '-' is both a unary and binary operator

Grammar rewriting

Rewrite the grammar into a unambiguous grammar

new grammar describes the same language (set of strings)

but eliminates undesirable parse trees

Example: Rewrite the ambiguous

E → E + E | E * E | (E) | int

into

E → E + T | T E generates T+T+… +T

T → T * F | F T generates F*F*… *F

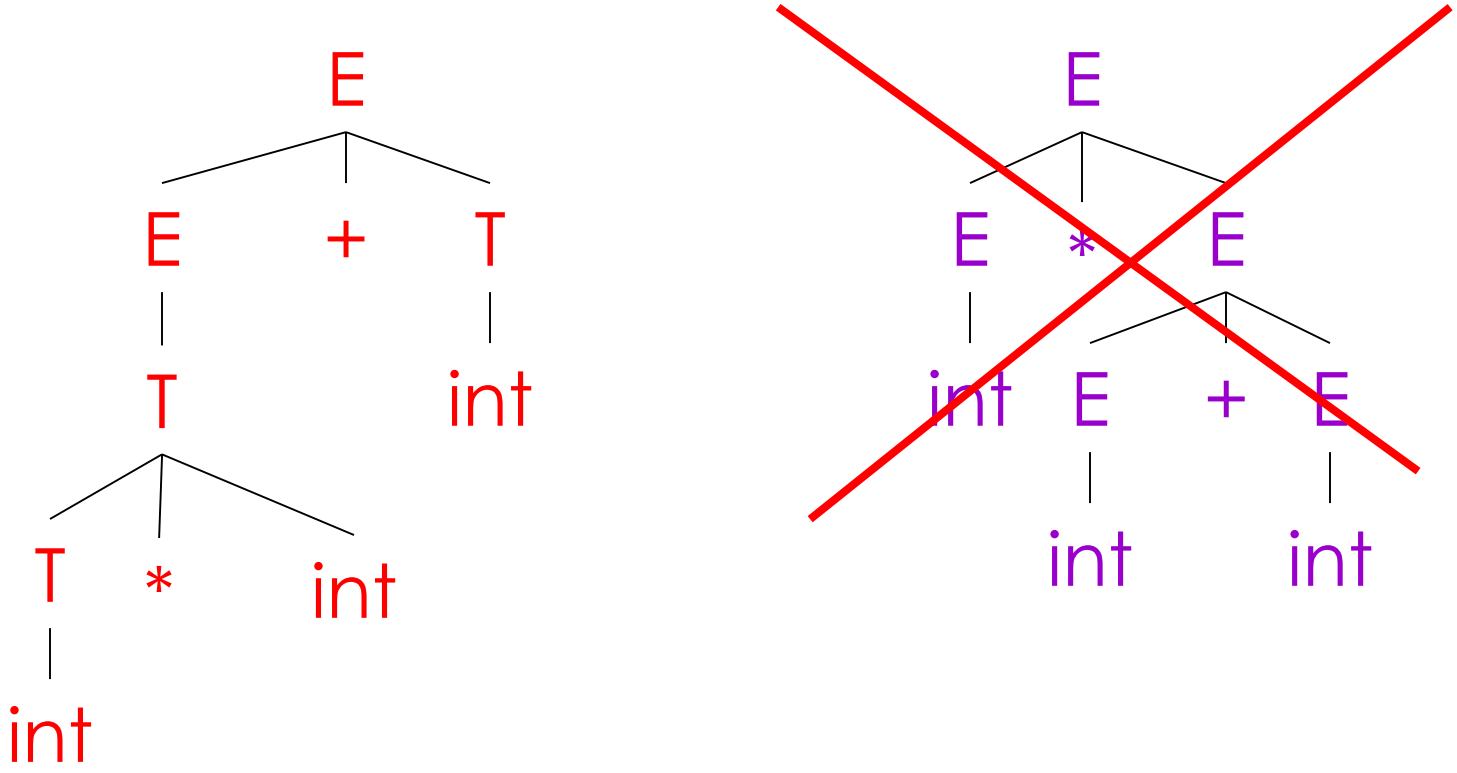
F → int | (E)

Draw a few parse trees and you will see that new grammar

- enforces precedence of * over + (* are lower in the tree)
- enforces left-associativity of + and *

Parse tree with the new grammar

The int * int + int has only one parse tree now



Note: these parse tress omit the F nonterminal to save space

Rewriting the grammar: what's the trick?

Trick 1: Fixing precedence (* computed before +)

$E \rightarrow E + E \mid E * E \mid id$

In the parse tree for id + id * id, we want id*id to be subtree of E+E.

How to do this by rewriting? Create a new nonterminal (T)

- make it derive id*id, …
- ensure T's trees are nested in E's of E+E

Your new grammar (associativity is still ambig):

$E \rightarrow E + E \mid T$

$T \rightarrow T * T \mid id$

Rewriting the grammar: what's the trick? (part 2)

Trick 2: Fixing associativity (+, *, associate to the left)

$E \rightarrow E + E \mid T$

$T \rightarrow T * T \mid id$

In the parse tree for $id +_1 id +_2 id$, we want the left id+id to be subtree of $E +_2 id$. Same for id*id*id.

Trick: use left recursion

- it will ensure that +, * associate to the left

New grammar (a simple change):

$E \rightarrow E + T \mid T$

$T \rightarrow T * id \mid id$

Summary

You can think of the rewrite in two alternative ways:

- Force the operators that must be evaluated first to be lower in the tree. Holds for both precedence and associativity.

- Make sure your grammar only generates only correct trees.

Ambiguity: The Dangling Else

Consider the ambiguous grammar

$S \rightarrow$ if E then S

| if E then S else S

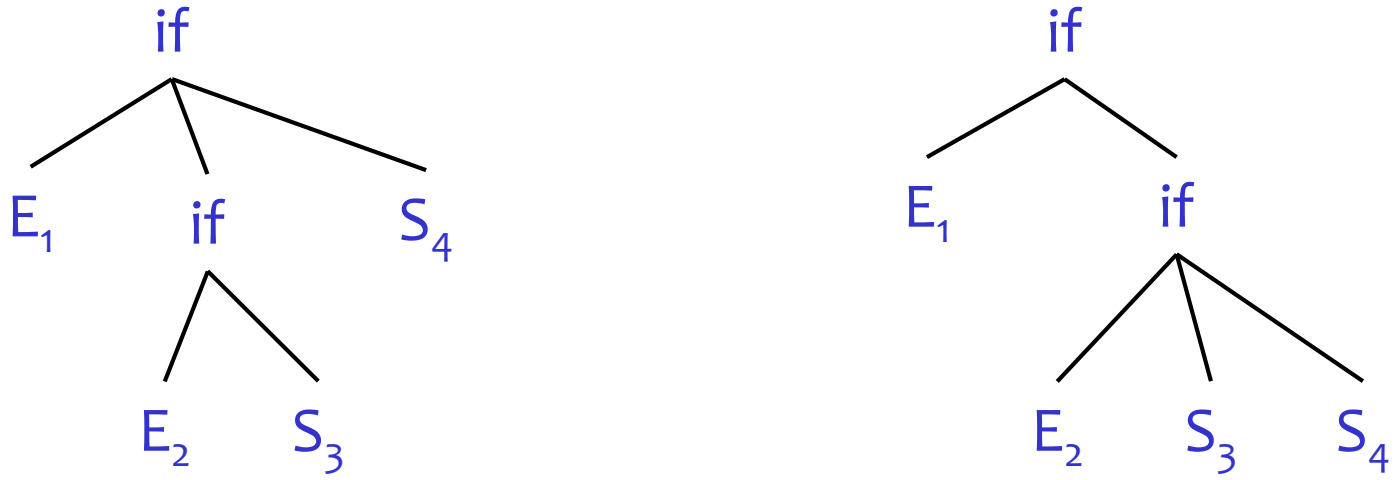
| OTHER

The Dangling Else: Example

The expression

$$\text{if } E_1 \text{ then if } E_2 \text{ then } S_3 \text{ else } S_4$$

has two parse trees



Typically we want the second form

The Dangling Else: A Fix

Usual rule: else matches the closest unmatched then

We *can* describe this in a grammar

Idea:

- distinguish matched and unmatched then's
- force matched then's into lower part of the tree

Example

New grammar describes the same set of strings

but forces *matched* ifs (those that have an else part) to the bottom of parse tree

Define two new non-terminals for IF:

- matched IF
- unmatched IF

Rewritten if-then-else grammar

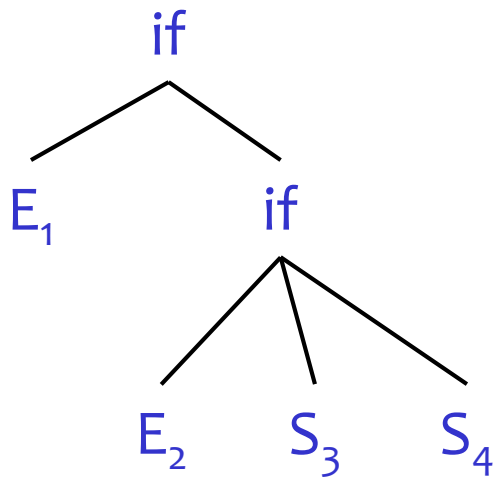
S → MIF /* all then are matched */
| UIF /* some then are unmatched */
MIF → if E then MIF else MIF
| OTHER
UIF → if E then S
| if E then MIF else UIF

Notes:

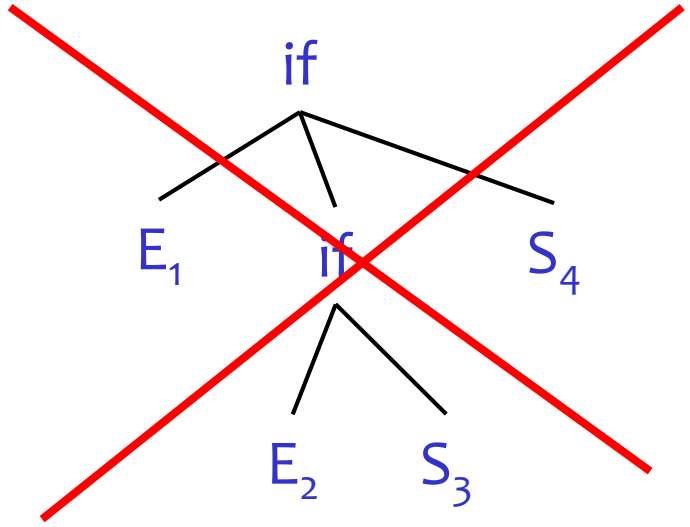
- notice that MIF does not refer to UIF,
- so all unmatched ifs (if-then) will be high in the tree

The Dangling Else: Example Revisited

- The expression if E_1 then if E_2 then S_3 else S_4



- A valid parse tree (for a UIF)



- Not valid because the then expression is not a MIF

Modular operators

Why design new abstractions

The only method for writing large software is through modularity – clear, composable abstractions

Composable:

- can snap them together with operators like Legos

Create a dataflow on streams

Process the values from merge(t1,t2)

We can apply operations :

```
for v in toUppercaseF(merge(tree1,tree2)) { process(v) }
```

How to create “filters” like `toUpperCaseF`?

A filter element of the pipeline

```
def filter(ant)
    def co = coroutine(function() {
        while (True) {
            --resume antecessor to obtain value
            def x=ant()
            -- yield transformed value
            yield(f(x))
    }    }
    lambda() { resume(co,0) }
}
consumer(filter1(filter2(producer())))
```

How to implement such pipelines

Producer-consumer patter: often a pipeline structure

- producer → filter → consumer
- All we need to say in code is

```
consumer(filter(producer()))
```

Producer-driven (push) or consumer-driven (pull)

- This decides who initiates resume(). In pull, the consumer resumes to producer who yields datum to consumer.

Each producer, consumer, filter is a coroutine

- Who initiates resume is the main coroutine.
- In for x in producer, the main coroutine is the for loop.

More details on queues

See assigned reading on Lua coroutines.

Large or infinite trees

Imagine working with a tree of a large or infinite size.

- the tree could describe a **file system**
  - inner nodes are directories, leaves are files
- or a **game tree**
  - each node is a board configuration
  - children are new configurations resulting from moving a piece

Programmers using such trees face two interesting challenges:

- usually, these trees are built lazily: i.e., children are created only when the client/user of the tree (eg, a traversal that prints a part of the tree) decides to visit the children
- programmers may want to prune such a tree, so that the traversal sees only a fragment of the tree, say, the top k levels.

Pruning operators

The DSL designer must design a pruning operator that …

- works on all trees
  - regardless of whether the tree is lazy or not
- produces a tree iterator, which could be passed to another operator
  - one pruning operator may prune depth, another may prune width of tree

Examples:

You might traverse the entire tree breadth-first with a preorder iterator:

```
for node in preorder(tree) { print(node) }
```

To prune the traversal to depth 5, you want a prune operator:

```
for node in preorder(prune(tree, 5)) { print(node) }
```

Ali prepared an example code with lazy game trees

http://www.cs.berkeley.edu/~bodik/cs164/sp13/lectures/game.lua

the pruning is used in function play_turn(), and is defined in function prune().

DSLs

Example of cs164 final projects

From cs164 debugging to education and data visualization

Build on cs164 artefacts:

- 164 grammar to generate tests
- extend cs164 “HTML” with better modularity
- add mapReduce to 164

List of sp12 final projects (1)

- Regular expressions for the common man!
- A language that teaches by allowing you to command virtual spaceships.
- A debugger for the 164 language.
- Adding rendering commands to the L3 language
- Autogenerating (useful) regression tests for the 164 language
- Erlang-style concurrency in 164
- Generating tests for cs164 and cs164-like languages
- scrapes webpages with the power of a thousand beautiful soups
- Sound synthesis language
- Query language for data visualizations
- Regex-like language for chess boards

List of sp12 final projects (2)

- Data Visualizer for aggregated data and extension to cs164 browser language
- Solves logic puzzles written in English.
- quick and easy way to keep large inventory
- Custom and composable widgets for HTML to eliminate boilerplate and enable fast prototyping
- simplifying Android programming
- algorithm visualization
- simple natural language programming
- Improve BASH script usability and features in Python
- Generalized locator for web elements
- Better scripting and environment management in bash
- Simplifying the RPC development process

List of sp12 final projects (3)

- a simple Python to C++ translator
- a simple presentation maker
- Adding MapReduce functionality to cs164
- Semantic version control
- High-level graph manipulation for the baller in all of us.
- A DSL for creating board games
- the declarative templating language for real-time apps
- interfacing with running binaries (x86)
- DSL for building location-based applications
- DSL for generating music
- An Oracle the parses webpages for you based on provide samples from the page.
- An Intermediate Language Representation for Android Application Execution Paths

Example problems solved by DSL abstractions

Let's look at d3 *data-joins*

The problem solved:

- how to explore several data sets, by animating a data visualization between these data sets
- a subproblem: mapping data to be visualized with visual data element, such as rectangles and circles

Reading:

- Three Little Circles
- Thinking with Joins
- http://bl.ocks.org/mbostock/3808218

Motivation

We want to visualize a list of data as a bar chart:

[5,10,13,19] → 

Must map each data point to a bar-like visual element:

eg, a CSS an SVG rectangle

This particular problem is easy. Solution in d3:

```
d3.select("body").selectAll("div")
  .data(dataset)
  .enter()
  .append("div")
  .attr("class", "bar")
 .style("height", function(d) {
    var barHeight = d * 5;
   return barHeight + "px";
 });
```

from: http://alignedleft.com/tutorials/d3/making-a-bar-chart/

But now consider changing the data set

On each click/tick, we want to modify the data:

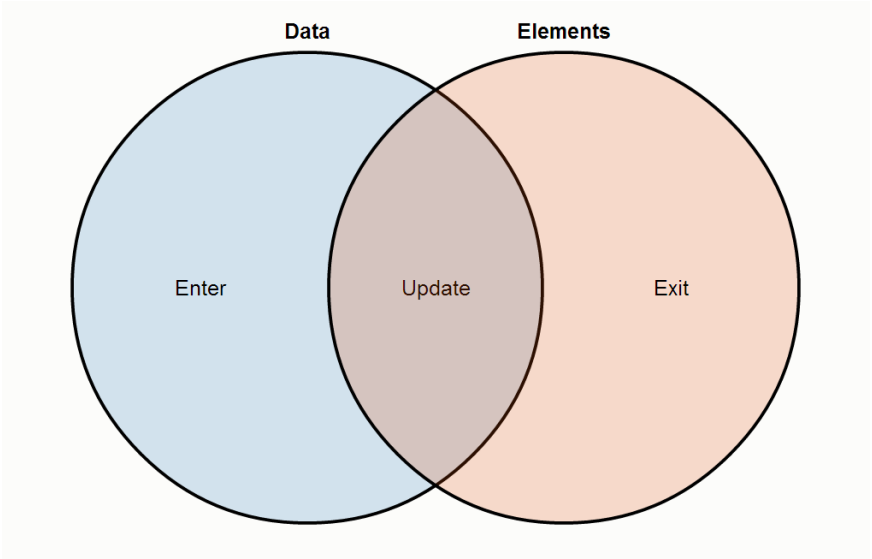
i) change values of element(s) in the data set
   - we need to visually animate (ie perform tweening) between new and old data value

ii) shrink or grow the data set
   - we need to remove or add new visual elements

data-join: d3 abstraction for this problem

We want to pair up data and elements.

We do it with tthree d3 “selections”:



http://bost.ocks.org/mike/join/

Tutorial on data-join selections

Three Little Circles: http://mbostock.github.com/d3/tutorial/circle.html

Beautifully explains the virtual selections (enter, update, exit), using the metaphor of the stage.