CSE 401 – Compilers

Lecture 6: LR Parsing (part I)

Michael Ringenburg

Winter 2013

Reminders/ Announcements

- Homework 1 is due TODAY, 11:59pm
- No class or office hours on Monday (MLK day)

Agenda

- Finish discussing the "if-else" ambiguity
- Start our first parsing algorithm: LR Parsing

Reminder: "if-else" ambiguity

- Grammar for conditional statements

  stmt ::= if (*cond*) *stmt*

  | if (*cond*) *stmt* else *stmt*

- This is ambiguous
  - Consider

```
if (a) if (b) s1 else s2
```

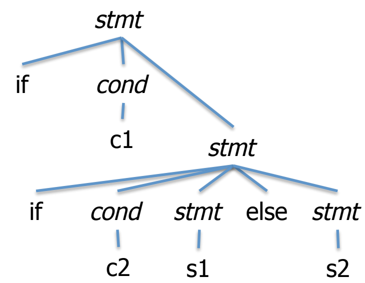
Derive if(c1) if(c2) s1 else s2

stmt

if (*cond*) *stmt*

if (c1) *stmt*

if (c1) if (*cond*) *stmt* else *stmt*
…
if (c1) if (c2) s1 else s2



stmt ::= if (*cond*) *stmt*
 | if (*cond*) *stmt* else *stmt*

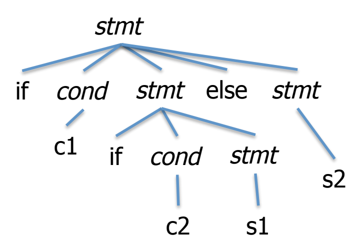
Derive if(c1) if(c2) s1 else s2

stmt

if (*cond*) *stmt* else *stmt*

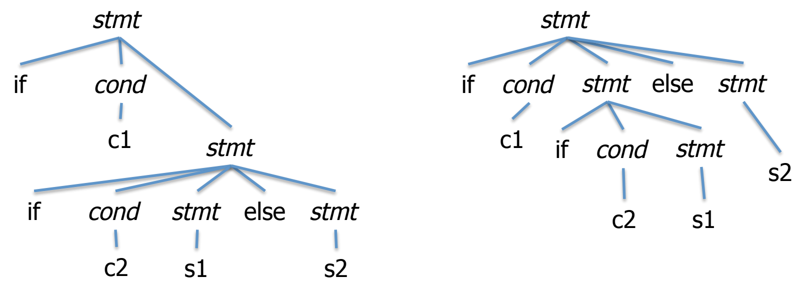
if (c1) *stmt* else *stmt*

if (c1) if (*cond*) *stmt* else *stmt*
…
if (c1) if (c2) s1 else s2



stmt ::= if (*cond*) *stmt*
 | if (*cond*) *stmt* else *stmt*

Compare Parse Trees



stmt ::= if (*cond*) *stmt*
| if (*cond*) *stmt* else *stmt*

Solving "if" Ambiguity

- Fix the grammar to separate if statements with else clause and if statements with no else
  - Done in Java reference grammar
  - Adds lots of non-terminals
- or, Change the language
  - But it'd better be ok to do this
- or, Use some ad-hoc rule in the parser
  - "else matches closest unpaired if"

Resolving Ambiguity with Grammar

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

– Prevents if-without-else as then clause of if-then-else, forcing else to match closest if. But, can still generate exact same language (try it!)
– formal, no additional rules beyond syntax

Check: if (*c1*) if (*c2*) *stmt* else *stmt*

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

Resolving Ambiguity with Grammar (2)

- If you can (re-)design the language, can avoid the problem entirely, e.g., create an **end** to match closest **if**

  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
    (But maybe this is a good idea anyway? These ambiguities can lead to programmer bugs …)

Parser Tools and Operators

- Most parser tools can cope with ambiguous grammars
  - Makes life simpler if you're careful
- Typically one can specify operator precedence & associativity
  - Allows simpler, ambiguous grammar with fewer nonterminals as basis for generated parser, without creating problems

Parser Tools and Ambiguous Grammars

- Possible rules for resolving other problems
  - Earlier productions in the grammar preferred to later ones
  - Longest match used if there is a choice
- Parser tools normally allow for this
  - But be sure that what the tool does is really what you want

Agenda

- Finish discussing the "if-else" ambiguity
- **Start our first parsing algorithm: LR Parsing**

Parsing Algorithms

- The two primary style of parsing are LL and LR parsing
- LL Parsing (Left-to-right scan, Leftmost derivation)
  - Top down – start with grammar start symbol, work your way down until you get to terminals.
  - Generates a leftmost derivation (*the* leftmost derivation assuming unambiguous grammar)
  - The "traditional" starting point for teaching parsing.
- We'll start with LR since you need it for your projects (and it's the most commonly used).

LR(1) Parsing

- We'll focus specifically on LR(1) parsers
  - Left to right scan, Rightmost derivation (reverse rightmost), 1 symbol lookahead
    - Lookahead: how far past current symbol we can look to determine which rule to apply.
  - Almost all practical programming languages have an LR(1) grammar
  - LALR(1), SLR(1), etc. – subsets of LR(1) with lower memory requirements, slightly less power
    - LALR(1) can mostly parse most real languages, and is used by YACC/Bison/CUP/etc.

Bottom-Up Parsing

- Basic Idea: Read tokens left to right, push (*shift*) onto a stack.
- Whenever the top of the stack matches the right hand side of a production, *reduce* it to the appropriate non-terminal and add that non-terminal to the parse tree.
- The upper edge of this partial parse tree is known as the *frontier.*
- Process called *shift-reduce* parsing.

Bottom-Up Parsing

- Basic Idea: Read tokens left to right, push (*shift*) onto a stack.
- **Whenever** the top of the stack matches the right hand side of a production, *reduce* it to the appropriate non-terminal and add that non-terminal to the parse tree. **←Slight Lie**
- The upper edge of this partial parse tree is known as the *frontier.*
- Process called *shift-reduce* parsing.

Example: Parse a b b c d e (bottom up)

S ::= a*AB* e
A ::= *A*bc | b
B ::= d

Details

- The bottom-up parser reconstructs a reverse rightmost derivation
- Given the rightmost derivation
  $S \Rightarrow_{rm} \beta_1 \Rightarrow_{rm} \beta_2 \Rightarrow_{rm} \ldots \Rightarrow_{rm} \beta_{n-2} \Rightarrow_{rm} \beta_{n-1} \Rightarrow_{rm} \beta_n = w$
  the parser will first discover $\beta_{n-1} \Rightarrow_{rm} \beta_n$, then $\beta_{n-2} \Rightarrow_{rm} \beta_{n-1}$, etc.
- Parsing terminates when
  - β_1 reduced to *S* (start symbol, success), or
  - No match can be found (syntax error)

How Does this Work?

- Key: given what we've already seen and the next input symbol (the lookahead), decide what to do.
- Choices:
  - Perform a reduction
  - Look ahead further (shift another symbol onto the stack)
- Can reduce *A*=>β if both of these hold:
  - *A*=>β is a valid production
  - *A*=>β is a step in the rightmost derivation (e.g., don't use the A=>b reduction for the second 'b' in our example).
- That's why we call it a *shift-reduce parser*

Difficulties

- Tricky parts:
  - How do we do this efficiently?
    - Prefer O(sourceLength + derivationLength). Can't really do better than O(input + output)!
    - Naïve approach (examine full stack at every step) is O((sourceLength + derivationLength)*sourceLength), since stack is potentially as long as program
  - How do we know whether A=>β is a step in the rightmost derivation (second condition for reducing)?
- Preview: Generate DFAs encoded by tables …

Sentential Forms

- If S =>* α, the string α is called a *sentential form* of the of the grammar
- In the derivation
  $S => \beta_1 => \beta_2 => ... => \beta_{n-2} => \beta_{n-1} => \beta_n = w$
  each of the β_i are sentential forms
- A sentential form in a rightmost derivation is called a right-sentential form (similarly for leftmost and left-sentential)
  - I.e., α is a right-sentential form of the grammar if $S =>_{rm}^* \alpha$

Handles

- A substring of the tree frontier (the highest level that we've built) that matches the right side of a production, *and is used in the rightmost derivation of the current string*.
  - Even if $A ::= \beta$ is a production, β is a handle only if it matches the frontier at a point where $A ::= \beta$ was used in the current derivation
  - β may appear in other places in the frontier without being a handle for $A ::= \beta$
- Bottom-up parsing is all about finding these handles

Handles (cont.)

- Formally, a *handle* of a right-sentential form γ_i is a production *A* ::= β and a position in γ_i where β may be replaced by *A* to produce the previous right-sentential form γ_{i-1} in the rightmost derivation of the current string that is being parsed

Handle Examples

- In the derivation

  S => a*AB*e => a*A*de => a*A*bcde => abbcde

  – abbcde is a right sentential form whose handle is *A*::=b at position 2

  – a*A*bcde is a right sentential form whose handle is *A*::=*A*bc at position 4

    - A::=b at position 3 is **not** a handle

- (Note: some books take the left of the match as the position)

Implementing Shift-Reduce Parsers

- Key Data structures
  - A stack holding the frontier of the tree
  - A string with the remaining input
  - Something that encodes the rules that tell us what action to take given the state of the stack and lookahead
    - This is typically a table that encodes a finite automata

Shift-Reduce Parser Actions

- What are these actions that we may take?
  - *Reduce* – if the top of the stack is the right side of a handle *A*::=β, pop the right side β and push the left side *A*
  - *Shift* – push the next input symbol onto the stack
  - *Accept* – announce success
  - *Error* – syntax error discovered

Shift-Reduce Example

Stack	Input	Action
$	abbcde$	*shift*
$a	bbcde$	*shift*
$ab	bcde$	*reduce A=>b*
$aA	bcde$	*shift*
$aAb	cde$	*shift*
$aAbc	de$	*reduce A=>Abc*
$aA	de$	*shift*
$aAd	e$	*reduce B=>d*
$aAB	e$	*shift*
$aABe	$	*reduce S=>aABe*
$S	$	*accept*

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

How Do We Decide which action to take?

- Def. *Viable prefix* – a prefix of a right-sentential form that can appear on the stack of the shift-reduce parser
  - Equivalent: a prefix of a right-sentential form that does not continue past the rightmost handle of that sentential form
  - Fact: the set of viable prefixes of a CFG is a regular language.
- Idea: Construct a DFA to recognize viable prefixes given the stack and remaining input
  - Recall, any regular language is recognizable by a DFA
  - Perform reductions when we recognize them

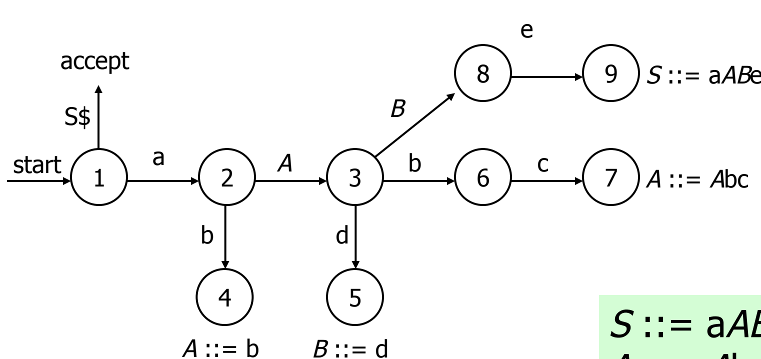
Viable Prefixes for our Example Grammar

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Viable Prefix	Handle/Action
S	*Accept*
aABe	S ::= aABe
aAd	B ::= d
aAbc	A ::= Abc
Ab	A ::= b
Plus prefixes of above…	*Shift…*

- The listed prefixes are those that extend all the way to the end of a handle – these correspond to reduction actions. Their prefixes are also viable prefixes.
- Why not aAbcbc? Extends past the handle (Abc).

DFA for viable prefixes of our example grammar

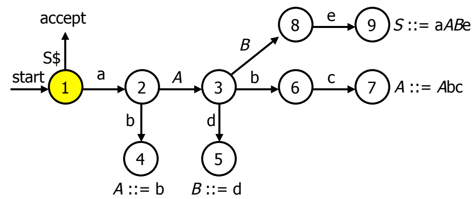


S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Trace

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

Stack	Input
$	abbcde$



Observations

- Way too much backtracking (start down a path, end up having to shift and restart)
  - We want the parser to run in time proportional to the length of the input
- Where the heck did this DFA come from anyway?
  - From the underlying grammar – in this simple case we were able to intuitively see all of the viable prefixes. But how do we find them in general?
  - We'll defer construction details for now