

6.035

Fall 2006

Introduction to Shift-Reduce Parsing

Saman Amarasinghe

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Last week...

- Lexical and Syntactic Levels of Structure
 - Lexical – regular expressions and automata
 - Syntactic – grammars
- Generation versus Recognition Approaches
 - Generation more convenient for specification
 - Recognition required in implementation

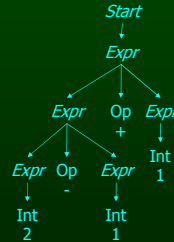
Ambiguity in Grammar

- Grammar is ambiguous if there are multiple derivations (therefore multiple parse trees) for a single string
- Derivation and parse tree usually reflect semantics of the program
- Ambiguity in grammar often reflects ambiguity in semantics of language (which is considered undesirable)

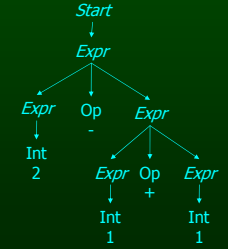
Ambiguity Example

Two parse trees for 2-1+1

Tree corresponding to $\langle 2-1 \rangle + 1$



Tree corresponding to $2 - \langle 1+1 \rangle$



Eliminating Ambiguity

Solution: hack the grammar

Original Grammar
 $Start \rightarrow Expr$
 $Expr \rightarrow Expr Op Expr$
 $Expr \rightarrow Int$
 $Expr \rightarrow Open Expr Close$

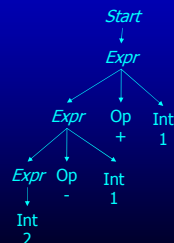
Hacked Grammar
 $Start \rightarrow Expr$
 $Expr \rightarrow Expr Op Int$
 $Expr \rightarrow Int$
 $Expr \rightarrow Open Expr Close$

Conceptually, makes all operators associate to left

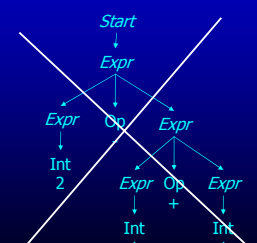
Parse Trees for Hacked Grammar

Only one parse tree for 2-1+1!

Valid parse tree

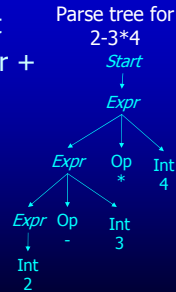


No longer valid parse tree



Precedence Violations

- All operators associate to left
- Violates precedence of * over +
 - 2-3*4 associates like <2-3>*4



Simon Arnsperg-Crotty

7

6.035 ©MIT Fall 2005

Hacking Around Precedence

Original Grammar	Hacked Grammar
Op = + - * /	AddOp = + -
Int = [0-9] [0-9]*	MulOp = * /
Open = <	Int = [0-9] [0-9]*
Close = >	Open = <
	Close = >
<i>Start</i> → <i>Expr</i>	<i>Start</i> → <i>Expr</i>
<i>Expr</i> → <i>Expr</i> Op <i>Int</i>	<i>Expr</i> → <i>Expr</i> AddOp <i>Term</i>
<i>Expr</i> → <i>Int</i>	<i>Expr</i> → <i>Term</i>
<i>Expr</i> → Open <i>Expr</i> Close	<i>Term</i> → <i>Term</i> MulOp <i>Num</i>
	<i>Term</i> → <i>Num</i>
	<i>Num</i> → <i>Int</i>
	<i>Num</i> → Open <i>Expr</i> Close

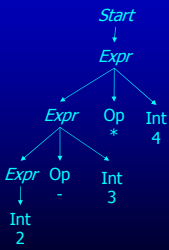
Simon Arnsperg-Crotty

8

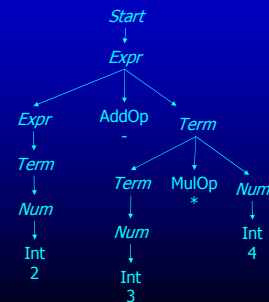
6.035 ©MIT Fall 2005

Parse Tree Changes

Old parse tree for 2-3*4



New parse tree for 2-3*4



Simon Arnsperg-Crotty

9

6.035 ©MIT Fall 2005

General Idea

- Group Operators into Precedence Levels
 - * and / are at top level, bind strongest
 - + and - are at next level, bind next strongest
- Nonterminal for each Precedence Level
 - Term* is nonterminal for * and /
 - Expr* is nonterminal for + and -
- Can make operators left or right associative within each level
- Generalizes for arbitrary levels of precedence

Simon Arnsperg-Crotty

10

6.035 ©MIT Fall 2005

What is different?

Hacked Grammar I

```

AddOp = +|-
MulOp = *|/
Int = [0-9] [0-9]*
Open = <
Close = >
Start → Expr
Expr → Expr AddOp Term
Expr → Term
Term → Term MulOp Num
Term → Num
Num → Int
Num → Open Expr Close
    
```

Hacked Grammar II

```

AddOp = +|-
MulOp = *|/
Int = [0-9] [0-9]*
Open = <
Close = >
Start → Expr
Expr → Expr AddOp Term
Expr → Term
Expr → Open Expr Close
Term → Term MulOp Num
Term → Num
Num → Int
    
```

Simon Arnsperg-Crotty

11

6.035 ©MIT Fall 2005

Handling If Then Else

```

Start → Stat
Stat → if Expr then Stat else Stat
Stat → if Expr then Stat
Stat → ...
    
```

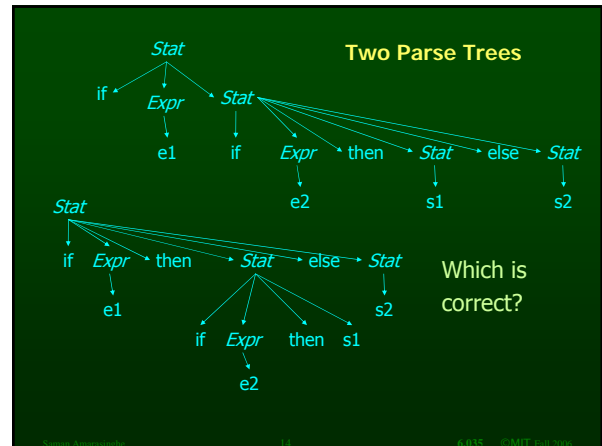
Simon Arnsperg-Crotty

12

6.035 ©MIT Fall 2005

Parse Trees

- Consider Statement
if e_1 then if e_2 then s_1 else s_2



Alternative Readings

- Parse Tree Number 1
 - if e_1
 - if e_2
 - s_1
 - else
 - s_2
- Grammar is ambiguous
- Parse Tree Number 2
 - if e_1
 - if e_2
 - s_1
 - else
 - s_2

Hacked Grammar

Hacked Grammar

- Basic Idea: control carefully where an if without an else can occur
 - Either at top level of statement
 - Or as very last in a sequence of if then else if then ... statements

Parser

- Converts program into a parse tree
- Can be written by hand
- Or produced automatically by parser generator
 - Accepts a grammar as input
 - Produces a parser as output
- Practical problem
 - Parse tree for hacked grammar is complicated
 - Would like to start with more intuitive parse tree

Solution

- Abstract versus Concrete Syntax
 - Abstract syntax corresponds to “intuitive” way of thinking of structure of program
 - Omits details like superfluous keywords that are there to make the language unambiguous
 - Abstract syntax may be ambiguous
 - Concrete Syntax corresponds to full grammar used to parse the language
- Parsers are often written to produce abstract syntax trees.

Abstract Syntax Trees

- Start with intuitive but ambiguous grammar
- Hack grammar to make it unambiguous
 - Concrete parse trees
 - Less intuitive
- Convert concrete parse trees to abstract syntax trees
 - Correspond to intuitive grammar for language
 - Simpler for program to manipulate

Hacked Unambiguous Grammar

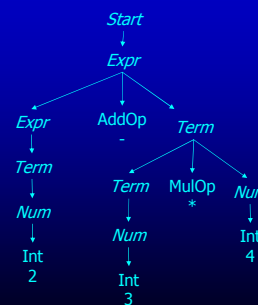
AddOp = +|-
 MulOp = */
 Int = [0-9] [0-9]*
 Open = <
 Close = >
 Start → Expr
 Expr → Expr AddOp Term
 Expr → Term
 Term → Term MulOp Num
 Term → Num
 Num → Int
 Num → Open Expr Close

Example

Intuitive but Ambiguous Grammar

Op = *|/|+|-
 Int = [0-9] [0-9]*
 Start → Expr
 Expr → Expr Op Expr
 Expr → Int

Concrete parse tree for <2-3>*4

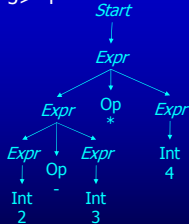


Abstract syntax tree for <2-3>*4

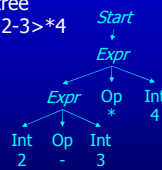


- Uses intuitive grammar
- Eliminates superfluous terminals
 - Open
 - Close

Abstract parse tree for <2-3>*4



Further simplified abstract syntax tree for <2-3>*4



Grammar and Automata Correspondence

Grammar	Automaton
Regular Grammar	Finite-State Automaton
Context-Free Grammar	Push-Down Automaton
Context-Sensitive Grammar	Turing Machine

CFG Versus PDA

- CFGs and PDAs are of equivalent power
- Grammar Implementation Mechanism:
 - Translate CFG to PDA, then use PDA to parse input string
 - Foundation for bottom-up parser generators

Simon Arnsperg-Crotty 75 6.035 ©MIT Fall 2005

Orientation

- Specify Syntax Using Context-Free Grammar
 - Nonterminals $Expr \rightarrow Expr Op Expr$
 - Terminals $Expr \rightarrow (Expr)$
 - Productions $Expr \rightarrow - Expr$
 - $Expr \rightarrow num$
 - $Op \rightarrow +$
 - $Op \rightarrow -$
 - $Op \rightarrow *$
- Given a grammar, Parser Generator produces a parser
 - Starts with input string
 - Produces parse tree

Simon Arnsperg-Crotty 76 6.035 ©MIT Fall 2005

Parser Generation

- How generated parser works
- How parser generator produces parser
- Central mechanism
 - Pushdown automaton, which implements
 - Shift-reduce parser

Simon Arnsperg-Crotty 77 6.035 ©MIT Fall 2005

Pushdown Automata

- Consists of
 - Pushdown stack (can have terminals and nonterminals)
 - Finite state automaton control
- Can do one of three actions (based on state and input):
 - Shift:
 - Shift current input symbol from input onto stack
 - Reduce:
 - If symbols on top of stack match right hand side of some grammar production $NT \rightarrow \beta$
 - Pop symbols (β) off of the stack
 - Push left hand side nonterminal (NT) onto stack
 - Accept the input string

Simon Arnsperg-Crotty 78 6.035 ©MIT Fall 2005

Shift-Reduce Parser Example

Stack

$Expr \rightarrow Expr Op Expr$
 $Expr \rightarrow (Expr)$
 $Expr \rightarrow - Expr$
 $Expr \rightarrow num$
 $Op \rightarrow +$
 $Op \rightarrow -$
 $Op \rightarrow *$

Input String

num	*	(num	+	num)
-----	---	---	-----	---	-----	---

Simon Arnsperg-Crotty 79 6.035 ©MIT Fall 2005

Shift-Reduce Parser Example

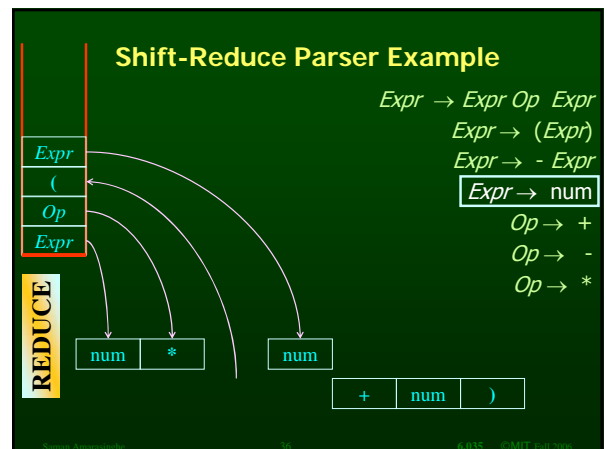
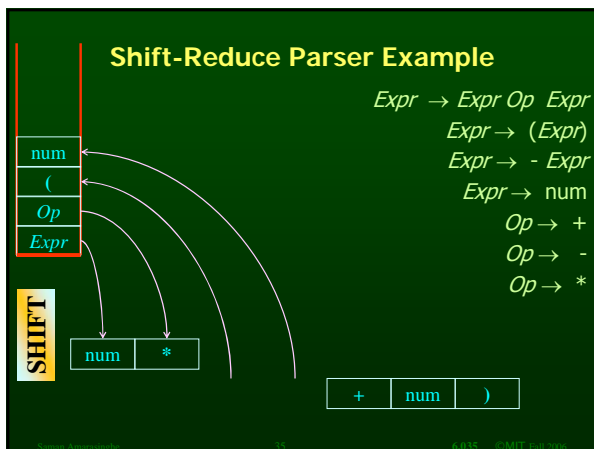
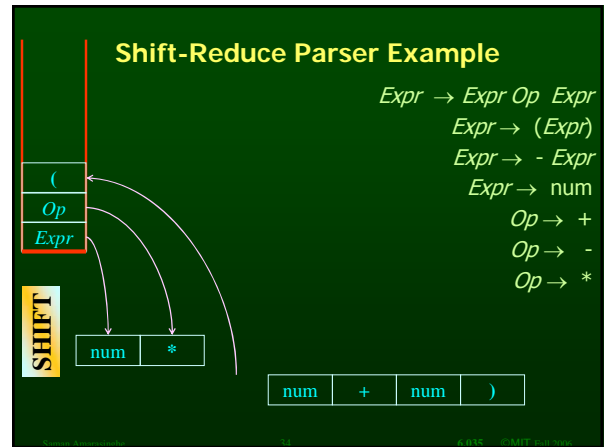
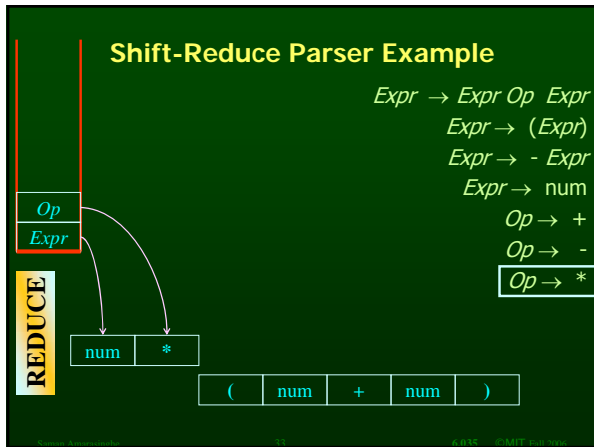
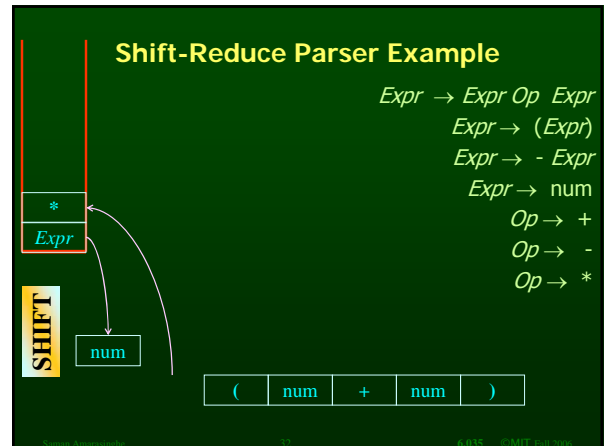
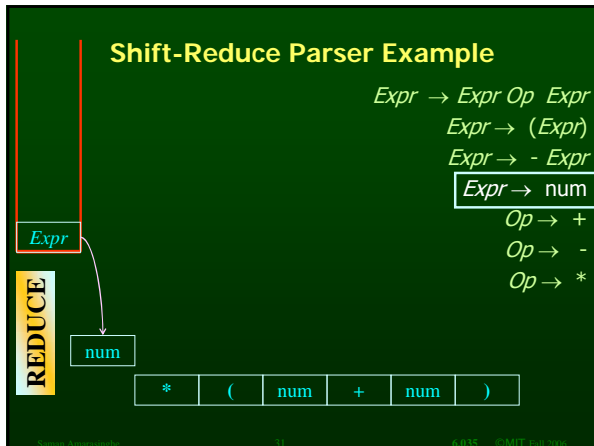
Stack

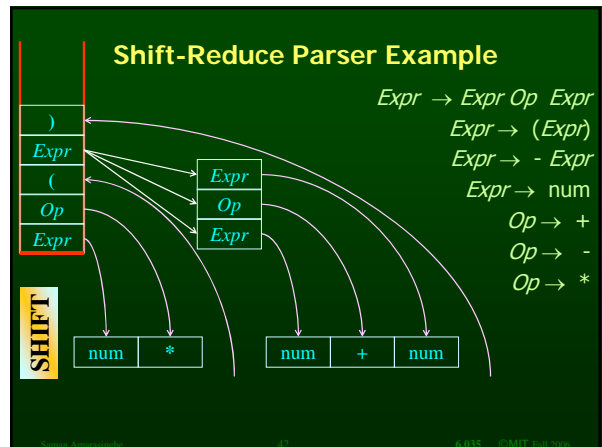
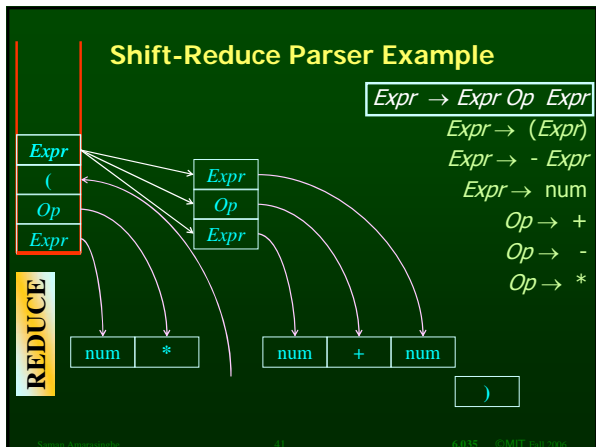
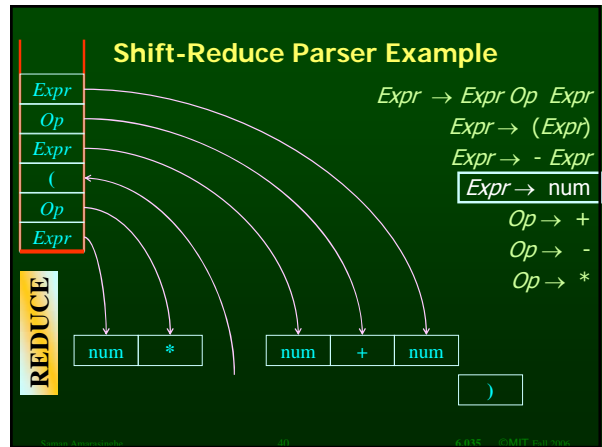
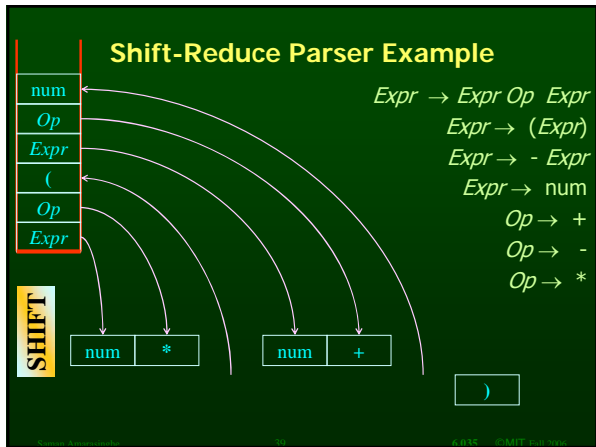
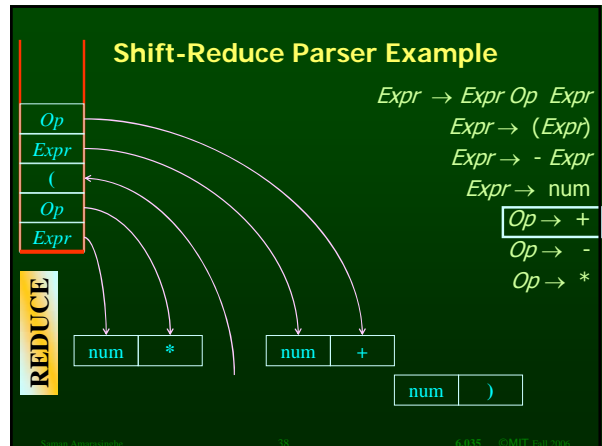
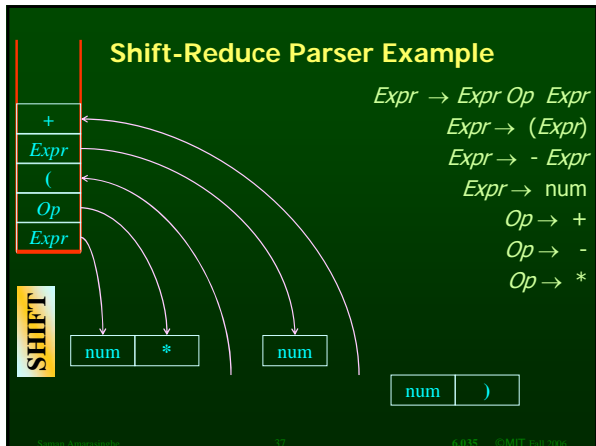
SHIFT

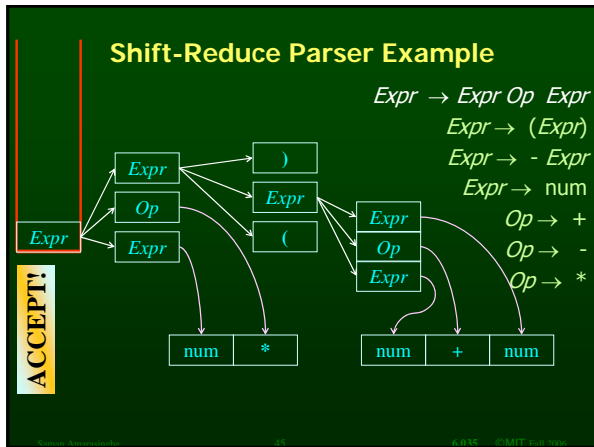
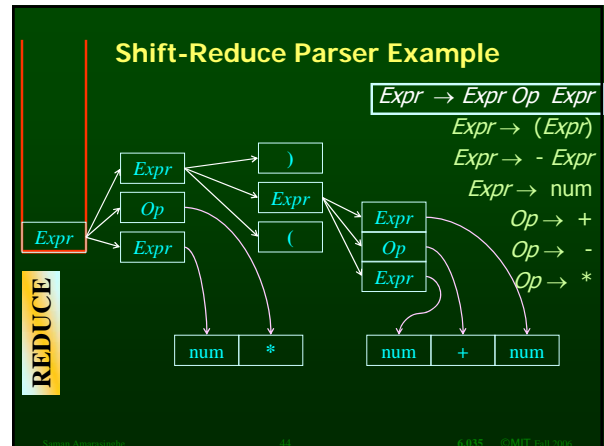
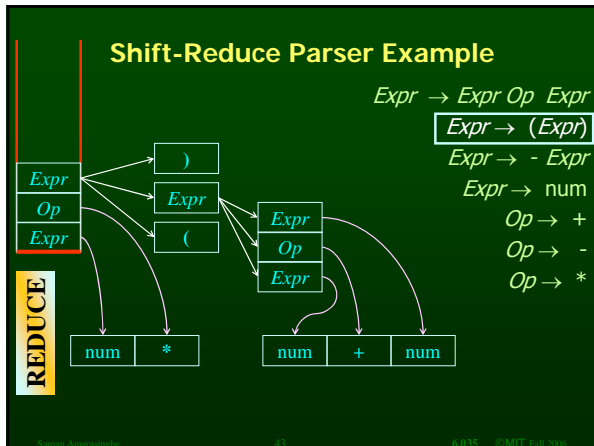
$Expr \rightarrow Expr Op Expr$
 $Expr \rightarrow (Expr)$
 $Expr \rightarrow - Expr$
 $Expr \rightarrow num$
 $Op \rightarrow +$
 $Op \rightarrow -$
 $Op \rightarrow *$

*	(num	+	num)
---	---	-----	---	-----	---

Simon Arnsperg-Crotty 80 6.035 ©MIT Fall 2005







- ### Basic Idea
- Goal: reconstruct parse tree for input string
 - Read input from left to right
 - Build tree in a bottom-up fashion
 - Use stack to hold pending sequences of terminals and nonterminals

- ### Potential Conflicts
- Reduce/Reduce Conflict
 - Top of the stack may match RHS of multiple productions
 - Which production to use in the reduction?
 - Shift/Reduce Conflict
 - Stack may match RHS of production
 - But that may not be the right match
 - May need to shift an input and later find a different reduction

- ### Conflicts
- | | |
|---------------------------------|---------------------------------|
| • Original Grammar | • New Grammar |
| $Expr \rightarrow Expr Op Expr$ | $Expr \rightarrow Expr Op Expr$ |
| $Expr \rightarrow (Expr)$ | $Expr \rightarrow Expr - Expr$ |
| $Expr \rightarrow - Expr$ | $Expr \rightarrow (Expr)$ |
| $Expr \rightarrow num$ | $Expr \rightarrow Expr -$ |
| $Op \rightarrow +$ | $Expr \rightarrow num$ |
| $Op \rightarrow -$ | $Op \rightarrow +$ |
| $Op \rightarrow *$ | $Op \rightarrow -$ |
| | $Op \rightarrow *$ |

Constructing a Parser

- We will construct version with no lookahead
- Key Decisions
 - Shift or Reduce
 - Which Production to Reduce
- Basic Idea
 - Build a DFA to control shift and reduce actions
 - In effect, convert grammar to pushdown automaton
 - Encode finite state control in parse table

Samy Abraham 49 6.035 ©MIT Fall 2005

Parser State

- Input Token Sequence (\$ for end of input)
- Current State from Finite State Automaton
- Two Stacks
 - State Stack (implements finite state automaton)
 - Symbol Stack (terminals from input and nonterminals from reductions)

Samy Abraham 50 6.035 ©MIT Fall 2005

Integrating Finite State Control

- Actions
 - Push Symbols and States Onto Stacks
 - Reduce According to a Given Production
 - Accept
- Selected action is a function of
 - Current input symbol
 - Current state of finite state control
- Each action specifies next state
- Implement control using parse table

Samy Abraham 51 6.035 ©MIT Fall 2005

Parse Tables

State	ACTION			
	()	\$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Implements finite state control
- At each step, look up
 - Table[top of state stack] [input symbol]
- Then carry out the action

Samy Abraham 52 6.035 ©MIT Fall 2005

Parse Table Example

State	ACTION			
	()	\$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack Symbol Stack Input Grammar

(())
s0 X $S \rightarrow X\$$ (1)
 $X \rightarrow (X)$ (2)
 $X \rightarrow ()$ (3)

Samy Abraham 53 6.035 ©MIT Fall 2005

Parser Tables

State	ACTION			
	()	\$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Shift to s7
 - Push input token into the symbol stack
 - Push s7 into state stack
 - Advance to next input symbol

Samy Abraham 54 6.035 ©MIT Fall 2005

Parser Tables

State	ACTION			Goto
	()	\$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Reduce (n)
 - Pop both stacks as many times as the number of symbols on the RHS of rule n
 - Push LHS of rule n into symbol stack

Source: Aravamudan, 6.035 ©MIT Fall 2004

Parser Tables

State	ACTION			Goto
	()	\$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Reduce (n) (continued)
 - Look up Table[top of the state stack][top of symbol stack]
 - Push that state (in goto part of table) onto state stack

Source: Aravamudan, 6.035 ©MIT Fall 2004

Parser Tables

State	ACTION			Goto
	()	\$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Accept
 - Stop parsing and report success

Source: Aravamudan, 6.035 ©MIT Fall 2004

Key Concepts

- Pushdown automaton for parsing
 - Stack, Finite state control
 - Parse actions: shift, reduce, accept
- Parse table for controlling parser actions
 - Indexed by parser state and input symbol
 - Entries specify action and next state
 - Use state stack to help control
- Parse tree construction
 - Reads input from left to right
 - Bottom-up construction of parse tree

Source: Aravamudan, 6.035 ©MIT Fall 2004

Grammar Example 1

- What is the context free grammar for?
 - $(digit^+ \cdot digit^*) \mid (digit^* \cdot digit^+)$
 - Where
 - digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
 - dot = .

Source: Aravamudan, 6.035 ©MIT Fall 2004

Grammar Example II

- What is the context free grammar for a string with x open wedges followed by $2*x$ close wedges for any $x \geq 0$

Source: Aravamudan, 6.035 ©MIT Fall 2004

Grammar Example III

- Is this grammar ambiguous?

$E \rightarrow \text{int}$
 $E \rightarrow E B E$
 $B \rightarrow +$
 $B \rightarrow *$

Grammar Example III

- What is the parse tree for?

$1+2+3$

Grammar Example III

- What is the parse tree for?

$1+2*3$

Grammar Example IV

- Hack the grammar to enforce precedence