NJIT

New Jersey's Science &
Technology University

THE EDGE IN KNOWLEDGE

SYNTAX ANALYSIS AND PARSING

Describing Syntax: Terminology

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`)
- A *token* is a category of lexemes (e.g., identifier)

Formal Definition of Languages

- **Recognizers**
  - A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
  - The syntax analysis part of a compiler is a recognizer

BNF and Context-Free Grammars

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s
  - Language generators, meant to describe the syntax of natural languages
  - Define a class of languages called context-free languages
- Backus-Naur Form (1959)
  - Invented by John Backus to describe the syntax of Algol 58
  - BNF is equivalent to context-free grammars

BNF Fundamentals

- In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called *nonterminal symbols*, or just *nonterminals*)

- *Terminals* are lexemes or tokens

- A rule has a left-hand side (LHS), which must be a single nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals

BNF Fundamentals (continued)

- Nonterminals can be shown as enclosed in angle brackets, tokens in bold
- Nonterminals as all lowercase and terminals as all uppercase works as well
  - Examples of BNF rules:

```
<ident_list> → identifier | identifier comma <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
ident_list ::= IDENTIFIER | IDENTIFIER COMMA ident_list
is_stmt ::= IF logic_expr THEN stmt
```

- Grammar: a finite non-empty set of rules

- The *start symbol* is a special element of the nonterminals of a grammar; it indicates where the recognition of the language begins

BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

Describing Lists

- Syntactic lists are described using recursion

```
<ident_list> → ident
             | ident, <ident_list>
```

- A *derivation* is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

An Example Grammar

```
<program> → <stmts>
 <stmts> → <stmt> | <stmt> ; <stmts>
 <stmt> → <var> = <expr>
 <var> → a | b | c | d
 <expr> → <term> + <term> | <term> - <term>
 <term> → <var> | const
```

An Example Derivation

For this sentence:

```
 a = b + const
```

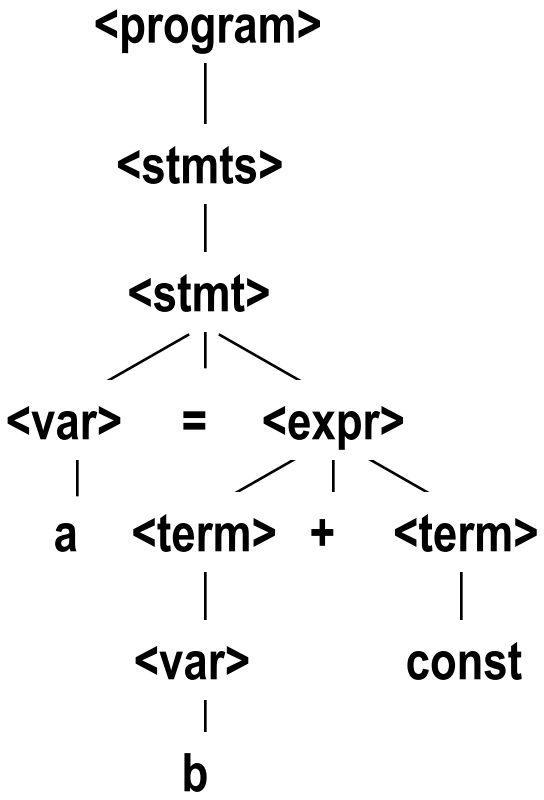
```
<program>   => <stmts>
            => <stmt>
            => <var> = <expr>
            => a = <expr>
            => a = <term> + <term>
            => a = <var> + <term>
            => a = b + <term>
            => a = b + const
```

Derivations

- Every string of symbols in a derivation is a *sentential form*
- A *sentence* is a sentential form that has only terminal symbols
- A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be either leftmost or rightmost

Parse Tree

- A hierarchical representation of a derivation



Ambiguity in Grammars

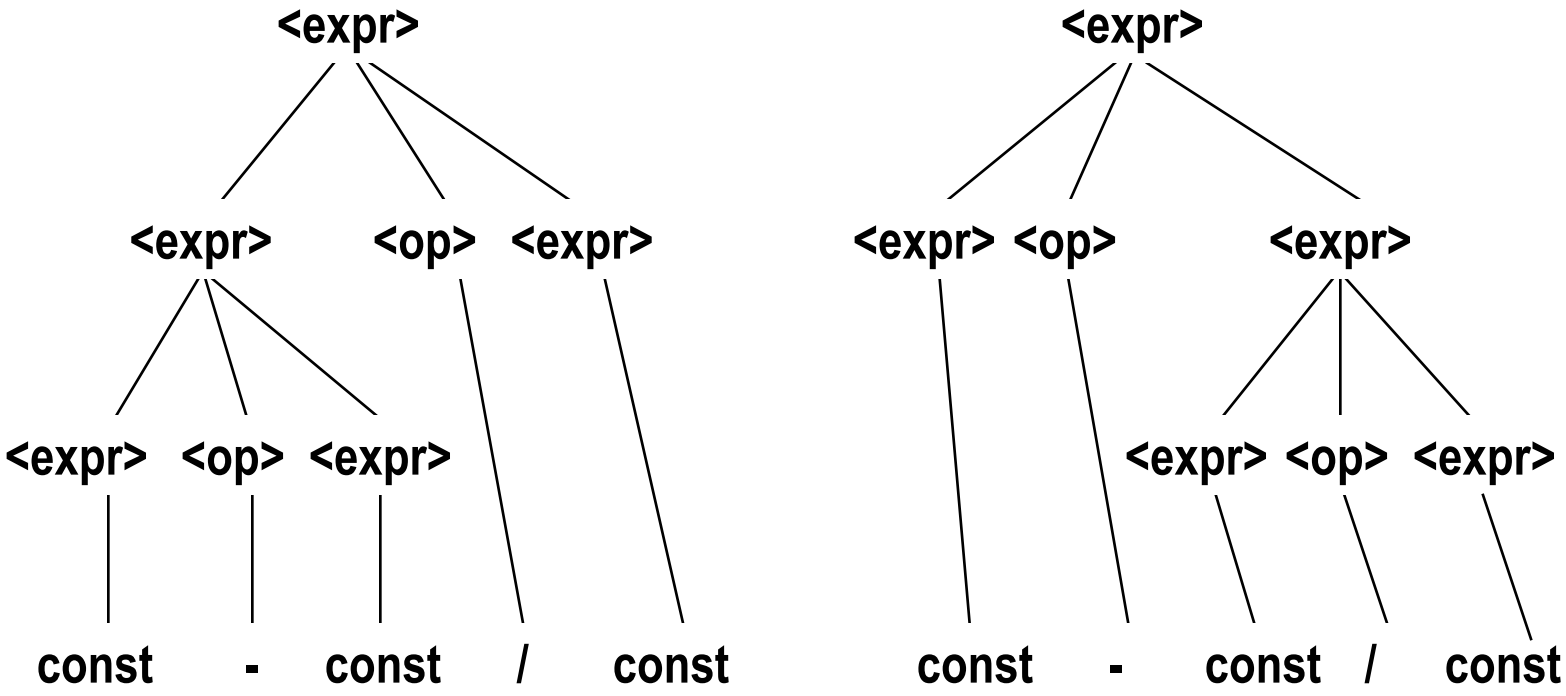
- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const

<op> → / | -

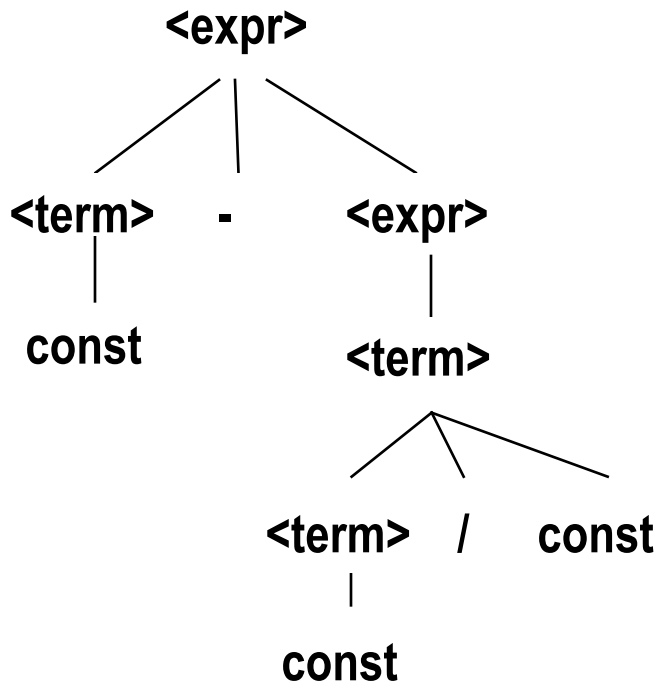
const - const / const



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we will not have ambiguity

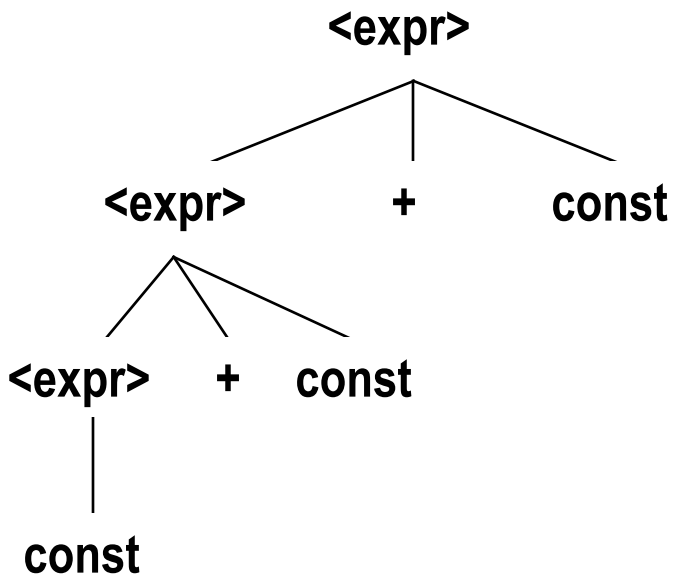
```
<expr> → <term> - <expr> | <term>
<term> → <term> / const | const
```



Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const   (ambiguous)
<expr> -> <expr> + const  |  const   (unambiguous)
```



Extended BNF

- Optional parts are placed in brackets [ ]

  `<proc_call> -> ident ([<expr_list>])`

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

  `<term> → <term> (+|-) const`

- Repetitions (0 or more) are placed inside braces { }

  `<ident> → letter {letter|digit}`

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

- EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```

Variations in EBNF

- Alternative RHSs are put on separate lines
- Use of a colon instead of `=>`
- Use of `opt` for optional parts
- Use of `oneof` for choices

The Parsing Problem

- Goals of the parser, given an input program:
  - Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly
  - Produce the parse tree, or at least a trace of the parse tree, for the program

The Parsing Problem (continued)

- Two categories of parsers
  - *Top down* - produce the parse tree, beginning at the root
    - Start from the top, work down and to the right
  - *Bottom up* - produce the parse tree, beginning at the leaves
    - Start from the bottom, work up and to the right
- Useful parsers look only one token ahead in the input

The Parsing Problem (continued)

- Top-down Parsers
  - Given a sentential form, $xA\alpha$, the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by A
- The most common top-down parsing algorithms:
  - Recursive descent - a coded implementation
  - LL parsers - table driven implementation

The Parsing Problem (continued)

- Bottom-up parsers
  - Given a right sentential form, α, determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation
  - The most common bottom-up parsing algorithms are in the LR family

The Parsing Problem (continued)

- The Complexity of Parsing
  - Parsers that work for any unambiguous grammar are complex and inefficient ($O(n^3)$, where n is the length of the input)
  - Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time (O(n), where n is the length of the input)

Recursive-Descent Parsing

- There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
- EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals

Recursive-Descent Parsing (continued)

- A grammar for simple expressions:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int_constant | ( <expr> )
```

Recursive-Descent Parsing (continued)

- Assume we have a lexical analyzer named `lex`, which puts the next token code in `nextToken`
- Note: no error cases are handled in these examples
- The coding process when there is only one RHS:
  - For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error
  - For each nonterminal symbol in the RHS, call its associated parsing subprogram

Recursive-Descent Parsing (continued)

```
/* Function expr
   Parses strings in the language
   generated by the rule:
   <expr> → <term> {(+ | -) <term>}
 */

void expr() {

/* Parse the first term */

  term();
/* As long as the next token is + or -, call
   lex to get the next token and parse the
   next term */

  while (nextToken == ADD_OP ||
         nextToken == SUB_OP){
    lex();
    term();
  }
}
```

Recursive-Descent Parsing (continued)

```
/* term
Parses strings in the language generated by the rule:
<term> -> <factor> {(* | /) <factor>)
*/
void term() {

/* Parse the first factor */
  factor();

/* As long as the next token is * or /,
   next token and parse the next factor */
  while (nextToken == MULT_OP || nextToken == DIV_OP) {
    lex();
    factor();
  }
} /* End of function term */
```

Recursive-Descent Parsing (continued)

- A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse
  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - The next token is compared with the first token that can be generated by each RHS until a match is found
  - If no match is found, it is a syntax error

Recursive-Descent Parsing (continued)

```
/* Function factor
   Parses strings in the language
   generated by the rule:
   <factor> -> id  |  (<expr>)  */

 void factor() {

 /* Determine which RHS */
   if (nextToken) == ID_CODE || nextToken == INT_CODE)

 /* For the RHS id, just call lex */
     lex();

/* If the RHS is (<expr>) - call lex to pass over the left parenthesis,
   call expr, and check for the right parenthesis */
   else if (nextToken == LP_CODE) {
     lex();
      expr();
     if (nextToken == RP_CODE)
        lex();
      else
        error();
    }  /* End of else if (nextToken == ...  */

   else error(); /* Neither RHS matches */
}
```

Recursive-Descent Parsing (continued)

- **Trace of the lexical and syntax analyzers on** `(sum + 47) / total`

```
Next token is: 25 Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11 Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21 Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26 Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24 Next lexeme is /
Exit <factor>
Next token is: 11 Next lexeme is total
Enter <factor>
Next token is: -1 Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

Recursive-Descent Parsing (continued)

- The LL Grammar Class
  - The Left Recursion Problem
    - If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser
      - A grammar can be modified to remove direct left recursion as follows:

For each nonterminal, A,

1. Group the A-rules as $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

   where none of the β's begins with A

2. Replace the original A-rules with

   $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

   $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

Recursive-Descent Parsing (continued)

- The other characteristic of grammars that disallows top-down parsing is the lack of pairwise disjointness
  - The inability to determine the correct RHS on the basis of one token of lookahead
  - Def: FIRST(α) = {a | $\alpha \Rightarrow^* a\beta$ }

    (If $\alpha \Rightarrow^* \varepsilon$, ε is in FIRST(α))

Recursive-Descent Parsing (continued)

- Pairwise Disjointness Test:
  - For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that

$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$$

- Example:

$$A \rightarrow a \mid bB \mid cAb$$

$$A \rightarrow a \mid aB$$

Recursive-Descent Parsing (continued)

- Left factoring can resolve the problem

  Replace

  ⟨variable⟩ → identifier | identifier [⟨expression⟩]

  with

  ⟨variable⟩ → identifier ⟨new⟩

  ⟨new⟩ → ε | [⟨expression⟩]

  or

  ⟨variable⟩ → identifier [[⟨expression⟩]]

  (the outer brackets are metasymbols of EBNF)

Bottom-up Parsing

- The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation

Bottom-up Parsing (continued)

- Handles:
  - Def: β is the *handle* of the right sentential form

    $\gamma = \alpha\beta w$ if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$

  - Def: β is a *phrase* of the right sentential form

    γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1\beta\alpha_2$

  - Def: β is a *simple phrase* of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1\beta\alpha_2$

Bottom-up Parsing (continued)

- Intuition about handles:
  - The handle of a right sentential form is its leftmost simple phrase
  - Given a parse tree, it is now easy to find the handle
  - Parsing can be thought of as handle pruning

Bottom-up Parsing (continued)

- Shift-Reduce Algorithms
  - Reduce is the action of replacing the handle on the top of the parse stack with its corresponding LHS
  - Shift is the action of moving the next token to the top of the parse stack

Bottom-up Parsing (continued)

- Advantages of LR parsers:
  - They will work for nearly all grammars that describe programming languages.
  - They work on a larger class of grammars than other bottom-up algorithms, but are as efficient as any other bottom-up parser.
  - They can detect syntax errors as soon as it is possible.
  - The LR class of grammars is a superset of the class parsable by LL parsers.

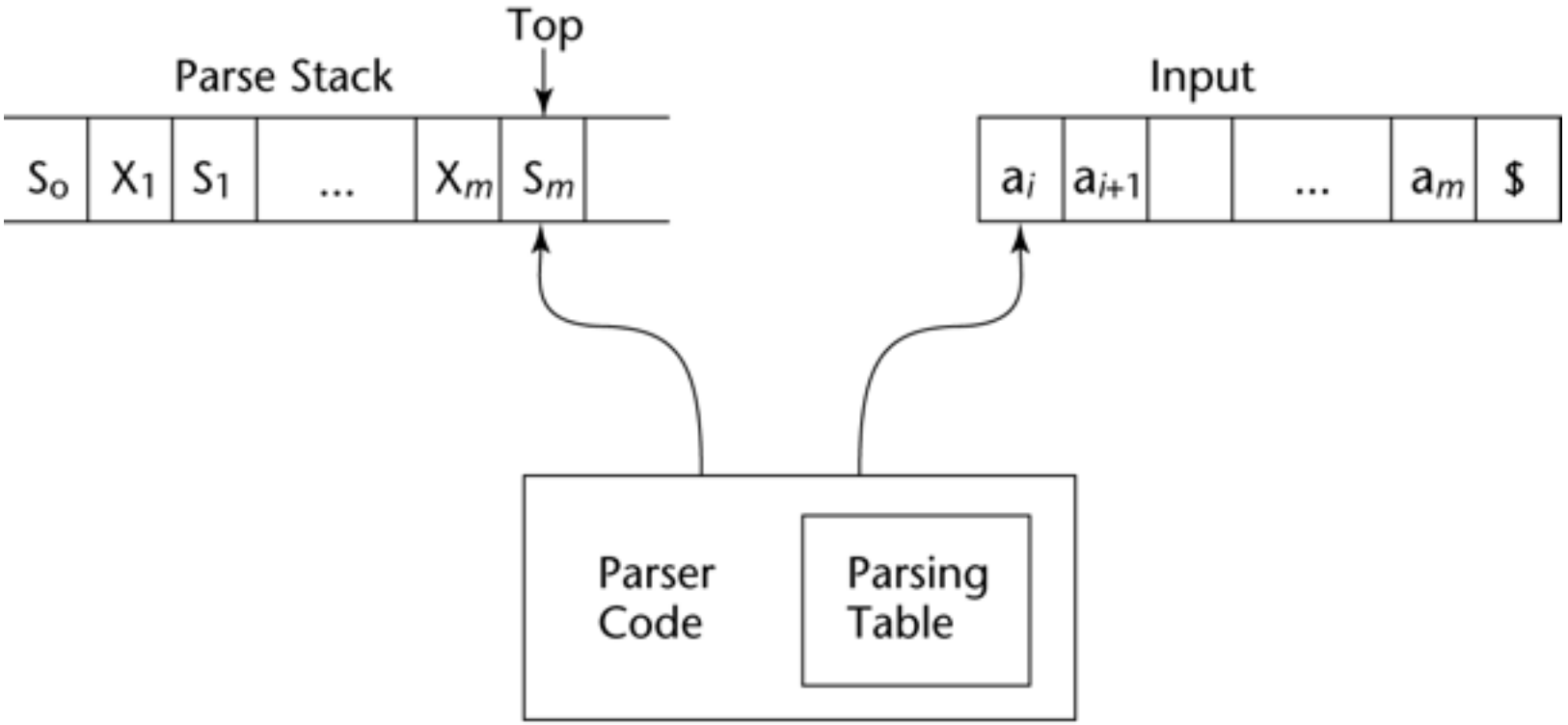
Bottom-up Parsing (continued)

- LR parsers must be constructed with a tool
- Knuth's insight: A bottom-up parser could use the entire history of the parse, up to the current point, to make parsing decisions
  - There are only a finite and relatively small number of different parse situations that could have occurred, so the history could be stored in a parser state, on the parse stack

Bottom-up Parsing (continued)

- LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table
  - The ACTION table specifies the action of the parser, given the parser state and the next token
    - Rows are state names; columns are terminals
  - The GOTO table specifies which state to put on top of the parse stack after a reduction action is done
    - Rows are state names; columns are nonterminals

Structure of An LR Parser



Bottom-up Parsing (continued)

- Initial configuration: (S_0, $a_1 \ldots a_n\$$)
- Parser actions:
  - For a Shift, the next symbol of input is pushed onto the stack, along with the state symbol that is part of the Shift specification in the Action table
  - For a Reduce, remove the handle from the stack, along with its state symbols. Push the LHS of the rule. Push the state symbol from the GOTO table, using the state symbol just below the new LHS in the stack and the LHS of the new rule as the row and column into the GOTO table

Bottom-up Parsing (continued)

- Parser actions (continued):
  - For an Accept, the parse is complete and no errors were found.
  - For an Error, the parser calls an error-handling routine.

LR Parsing Table

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Bottom-up Parsing (continued)

- A parser table can be generated from a given grammar with a tool, e.g., **`yacc`** or **`bison`**

NJIT

THE EDGE IN KNOWLEDGE