Lexical and Syntax Analysis

Hui Chen

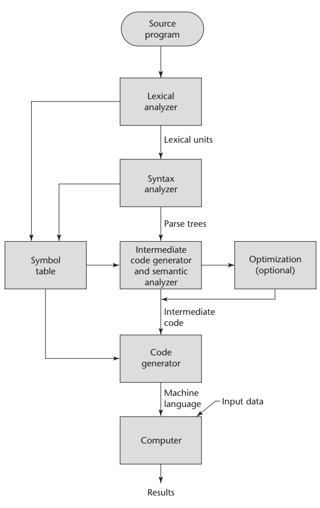
Computer Science
Virginia State University
Petersburg, Virginia

January 20, 2016

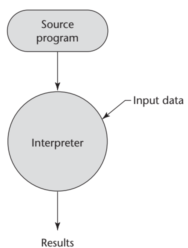
Acknowledgement

- Slides are prepared based on the textbook [Sebesta, 2012].

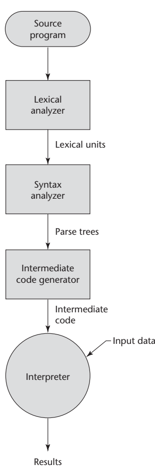
Language Implementation



(a) Compilation



(b) Pure Interpretation



(c) Hybrid Implementation

Syntax Analysis

- Consisting of two parts
  - Lexical analyzer (a finite automaton/finite state machine based on a regular grammar)
  - Syntax analyzer (a pushdown automaton based on a context-free grammar)

Lexical Analyzer

- Front-end for the parser
- Identifies *lexemes* and the tokens to which they belong
- Example: consider Java statement

```
index = 2 * count + 17;
```

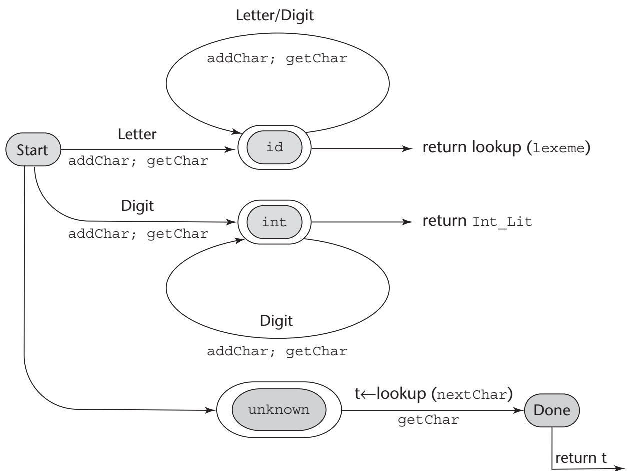
Lexeme	Token
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

Building Lexical Analyzer

- Directly implementing the state diagram of a finite automaton from scratch
  - Design a state diagram that describes the tokens
  - write a program that implements the state diagram
- Implementing the state diagram of a finite automaton using a table-driven approach
  - Design a state diagram that describes the tokens
  - Hand-construct a table-driven implementation of the state diagram
- Implementing a finite automaton using a table-driven approach with a software tool
  - Write a formal description of the tokens
  - Use a software tool that constructs a table-driven lexical analyzer from formal description of tokens

An Example of Lexical Analyzer

- State Diagram



An Example of Lexical Analyzer

- Implementation: In Github

Obtaining Program from Github and Run Example on Linux System

```
$ git clone https://github.com/huichen-cs/sebesta.git
$ cd sebesta/lexer
$ make lexer
$ make test
```

The Example of Lexical Analyzer in Lex

- Implementing a finite automaton using a table-driven approach with a software tool
  - Write a formal description of the tokens
  - Use a software tool that constructs a table-driven lexical analyzer from formal description of tokens
  - Example software tool: Lex (C, Java, Python ...)

Run Example on Linux System

```
$ cd sebesta/lexer/lex
$ make test
```

Syntax Analysis

- Syntax analysis is also called *parsing*.
- Top-down parsing
- Tottom-up parsing
- Complexity of parsing

Notation

- Terminal symbols: lowercase letters at the beginning of the alphabet $(a, b, \ldots)$
- Nonterminal symbols: uppercase letters at the beginning of the alphabet $(A, B, \ldots)$
- Terminals or nonterminals: uppercase letters at the end of the alphabet (W, X, Y, Z)
- Strings of terminals: lowercase letters at the end of the alphabet (w, x, y, z)
- Mixed strings (terminals and/or nonterminals): lowercase Greek letters $(\alpha, \beta, \ldots)$

Goal of Parsing

- Determine whether an input program is syntactically correct, produce a diagnostic message and recover.
- Produce a complete parse tree, or at least trace the structure of the complete parse tree, for syntactically correct input for translation.

Categories of Parser

- Top down
  - Produce the parse tree, beginning at the root
  - Order is that of a leftmost derivation
  - Traces or builds the parse tree in preorder
- Bottom up
  - Produce the parse tree, beginning at the leaves
  - Order is that of the reverse of a rightmost derivation
  - Useful parsers look only one token ahead in the input

Top-Down Parser

- 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, where x is a string of terminal symbols, α is a mixed string of terminals and nonterminals, and A is a nonterminal.
- The most common top-down parsing algorithms:
  - Recursive descent: a coded implementation
  - LL parsers: a table driven implementation

Top-Down Parser: Example

- Given $xA\alpha$ and A-rules,

$$A \rightarrow bB$$
$$A \rightarrow cBb$$
$$A \rightarrow a$$

which one of the three rules to choose to get the next sentential form, which could be xbB, $xcBb$, or xa.

Bottom-Up Parser

- Given a right sentential form, α, a mixed string of terminals and nonterminals, 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

Bottom-Up Parser: Example

- Consider the following grammar,

$$S \rightarrow aAc$$
$$A \rightarrow aA|b$$

and derivation:

$$S \Rightarrow aAc \Rightarrow aaAc \Rightarrow aabc$$

where S is a start nonterminal symbol; A is a nonterminal; a, b, and c are nonterminals.

- A bottom-up parser of this sentence, $aabc$, starts with the sentence and must find the handle (i.e., the correct RHS to reduce) in it.

Complexity of Parsing

- The time complexity of parsers that work for any unambiguous grammar are of $O(n^3)$ where n is the length of the input.
- Compilers use parsers that only work for a subset of all unambiguous grammars and do it in linear time, i.e., $O(n)$

Implementation of Parsers

- Top-down: Recursivee descent parsers
- Top-down: LL parsers
- Bottom-up: LR parsers

Recursive Descent Parsers

- 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 the extensions in EBNF minimizes the number of nonterminals

Recursive Descent Parsers: Example

- Consider the following EBNF description of simple arithmetic expressions:

$$
\begin{aligned}
<\text{expr}> &\rightarrow <\text{term}> \{(+|-)<\text{term}> \} \\
<\text{term}> &\rightarrow <\text{factor}> \{(*|/)<\text{factor}> \} \\
<\text{factor}> &\rightarrow \text{ id} \mid \text{int_constant} \mid (<\text{expr}>)
\end{aligned}
$$

Recursive Descent Parsers: Example

- Assume we have a lexical analyzer named `lex` that puts the next token code in `nextToken`
- *When a nonterminal has only one RHS*, the coding process:
  - 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 Parsers: Example

- For the first rule,

$$\langle\text{expr}\rangle \rightarrow \langle\text{term}\rangle \; \{(+|-)\langle\text{term}\rangle\}$$

```
/* expr
 * Parses strings in the language generated by the rule:
 * <expr> -> <term> {(+ | -) <term>}
 */
void expr() {
    printf("Enter <expr>");
    /* Parse the first term */
    term();
    /* As long as the next token is + or -, get
    the next token and parse the next term */
    while (nextToken == ADD_OP || nextToken == SUB_OP) {
        lex();
        term();
    }
    printf("Exit <expr>");
} /* End of function expr */
```

Recursive Descent Parsers: Example

- For the second rule,

$$<\text{term}> \rightarrow <\text{factor}> \{(*|/)<\text{factor}> \}$$

```
/* term
 * Parses strings in the language generated by the rule:
 * <term> -> <factor> {(* | /) <factor>)
 */
void term() {
    printf("Enter <term>");
    /* Parse the first factor */
    factor();
    /* As long as the next token is * or /, get the
    next token and parse the next factor */
    while (nextToken == MULT_OP || nextToken == DIV_OP) {
            lex();
            factor();
    }
    printf("Exit <term>");
} /* End of function term */
```

Recursive Descent Parsers: Example

- A nonterminal that has *more than one RHS*, it 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 Parsers: Example

- For the third rule,

$$<factor> \rightarrow \text{id} \mid \text{int_constant} \mid (<expr>)$$

```
void factor() {
    printf("Enter␣<factor>\n");
    /* Determine which RHS */
    if (nextToken == IDENT || nextToken == INT_LIT) {
        lex(); /* Get the next token */
    } else {
        /* If the RHS is (<expr>), call lex to pass over the
        left parenthesis, call expr, and check for the right
        parenthesis */
        if (nextToken == LEFT_PAREN) {
            lex(); expr();
            if (nextToken == RIGHT_PAREN) lex(); else error();
        } /* End of if (nextToken == ... */
        /* It was not an id, an integer literal, or a left parent
        else { error(); }
    } /* End of else */
    printf("Exit␣<factor>\n");;
} /* End of function factor */
```

Recursive Descent Parsers: Example

Run the Example Parser on Linux System

```
$ cd sebesta/parser
$ make test
```

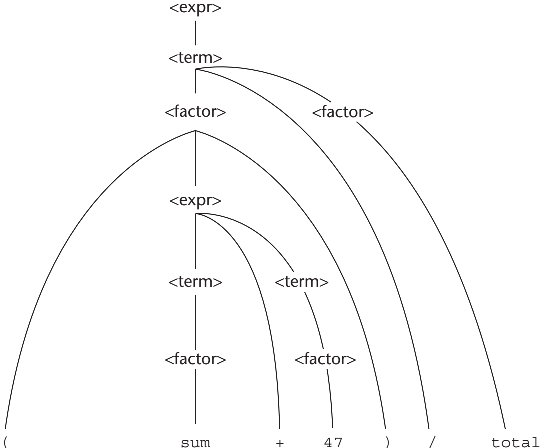
Recursive Descent Parsers: Example

- The parsing result

```
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 Parsers: Example

- The resulting parse tree



Recursive Descent Parsers: Example

- Grammatical description of the Java `if` statement

$$<\text{ifstmt}> \rightarrow \text{if } (<\text{boolexpr}>)<\text{statement}> [\text{ else } <\text{statement}>]$$

Recursive Descent Parsers: Example

```
/* Function ifstmt
Parses strings in the language generated by the rule:
<ifstmt> -> if (<boolexpr>) <statement> [else <statement>]
*/
void ifstmt() { /* Be sure the first token is 'if' */
  if (nextToken != IF_CODE) { error(); }
  else { lex(); /* Call lex to get to the next token */
    /* Check for the left parenthesis */
    if (nextToken != LEFT_PAREN) { error(); }
    else {
      boolexpr(); /* Call it to parse the Boolean expression */
      /* Check for the right parenthesis */
      if (nextToken != RIGHT_PAREN) { error(); }
      else { statement(); /* Call it to parse the then clause */
        /* If an else is next, parse the else clause */
        if (nextToken == ELSE_CODE) {
          lex(); /* Call lex to get over the else */
          statement();
        } /* end of if (nextToken == ELSE_CODE ... */
      } /* end of else of if (nextToken != RIGHT ... */
    } /* end of else of if (nextToken != LEFT ... */
  } /* end of else of if (nextToken != IF_CODE ... */
} /* end of ifstmt */
```

The Left Recursion Problem

- If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser
- Example: consider the following grammar and its recursive descent parser,

$$A \rightarrow A + B$$

```
void A() {
    A();
    lex();
    if (nexToken != ADD_OP) {
        error();
    } else {
        lex();
        B();
    }
}
```

Removing Direct Left Recursion: Example

- Consider the following grammar,

$$E \to E + T | T$$
$$T \to T * F | F$$
$$F \to (E) | id$$

- Which rules are *direct left recursions*?

Removing Direct Left Recursion: Example

- For the E-rule,

$$E \to E + T | T$$

Let $\alpha_1 = +T$ and $\beta_1 = T$, then, replace the E-rules with,

$$E \to \beta_1 E'$$
$$E' \to \alpha_1 E' | \epsilon$$

i.e.,

$$E \to TE'$$
$$E' \to +TE' | \epsilon$$

Removing Direct Left Recursion

- A grammar can be modified to remove direct left recursion

Removing Direct Left Recursion

For each nonterminal, A,

1. Group the A-rules as $A \rightarrow A\alpha_1, | \ldots | A\alpha_m | \beta_1 | \beta_2 | \ldots | \beta_n$ where none of the β's begins with A
2. Replace the original A-rules with

$$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_n A'$$
$$A \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | \epsilon$$

Removing Direct Left Recursion: Example

- For the T-rule,

$$T \rightarrow T * F | F$$

Let $\alpha_1 = *F$ and $\beta_1 = F$, then, replace the T-rules with,

$$T \rightarrow \beta_1 T'$$
$$T' \rightarrow \alpha_1 T' | \epsilon$$

i.e.,

$$T \rightarrow FT'$$
$$T' \rightarrow *FT' | \epsilon$$

Removing Direct Left Recursion: Example

- The complete replacement grammar without direct left recursion becomes,

$$E \rightarrow TE'$$
$$E' \rightarrow +TE'|\epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT'|\epsilon$$
$$F \rightarrow (E)|id$$

Indirect Left Recursion

- Indirect left recursion poses the same problem as direct left recursion
- Example:

$$A \rightarrow BaA$$
$$B \rightarrow Ab$$

```
void A() {
    B();
    lex();
    if (nextToken != TOKEN_CLASS_a) { error(); }
    else { lex(); A(); }
}
void B() {
    A();
    lex();
    if (nextToken != TOKEN_CLASS_b) { error(); }
}
```

- An algorithm to modify a given grammar to remove indirect left recursion is in [Aho et al., 2006].

Direct and Indirect Left Recursion

- The problem of left recursion is not confined to the recursive-descent approach to building topdown parsers.
- It is a problem for all top-down parsing algorithms
- When writing a grammar for a programming language, one can usually avoid including left recursion, both direct and indirect.

Pairwise Disjointness

- Top-down parsers can *not* always choose the correct RHS on the basis of the next token of input, using only the first token generated by the leftmost nonterminal in the current sentential form
- Pairwise disjointness test: whether the correct RHS on the basis of one token of lookahead can be determined, given a non–left recursive grammar

Pairwise Disjointness

FIRST Set

$$\text{FIRST}(\alpha) = \{a | \alpha \Rightarrow^* a\beta\} \text{ (If } \alpha \Rightarrow^* \epsilon, \epsilon \text{ is in FIRST}(\alpha))$$

where $\Rightarrow^*$ means 0 or more derivation steps.

The Pairwise Disjointness Test

For each nonterminal, A, in the grammar that has n RHS and $n > 1$, i.e., the grammar has n rules, $A \to \alpha_k$, $1 \leq k \leq n$. It must be true that $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \phi$, for any i and j where $i \neq j$, $1 \leq i \leq n$, $1 \leq j \leq n$.

- An algorithm to compute FIRST for any mixed string can be found in [Aho et al., 2006]

Pairwise Disjointness

- Informally, if a nonterminal A has more than one RHS, the first terminal symbol that can be generated in a derivation for each of them must be unique to that RHS.

Pairwise Disjointness

- Consider the following grammar rules:

$$A \rightarrow aB|bAb|Bb$$
$$B \rightarrow cB|d$$

- The FIRST sets for the RHSs of the A-rules are $\{a\}$, $\{b\}$, and $\{c, d\}$, which are clearly disjoint. Therefore, these rules pass the pairwise disjointness test

Pairwise Disjointness

- Consider the following grammar rules:

$$A \rightarrow aB|BAb$$
$$B \rightarrow aB|b$$

- The FIRST sets for the RHSs in the A-rules are $\{a\}$ and $\{a, b\}$, which are clearly not disjoint. So, these rules fail the pairwise disjointness test.

Left Factoring

- In *many cases*, a grammar that fails the pairwise disjointness test can be modified so that it will pass the test.
- Example: the following rules clearly do not pass the pairwise disjointness test,

$$<\text{variable}> \;\rightarrow\; \text{identifier} \mid \text{identifier}\ [<\text{expression}>\ ]$$

  Replace them with the equivalent rules,

$$<\text{variable}> \;\rightarrow\; \text{identifier}\ <\text{new}>$$
$$<\text{new}> \;\rightarrow \epsilon | [<\text{expression}>\ ]$$

  or

$$<\text{variable}> \;\rightarrow\; \text{identifier}\ [[<\text{expression}>\ ]]$$

  which would pass the pairwise disjointness test.
- A formal algorithm for left factoring is in [Aho et al., 2006].

Bottom-up Parsing

- To find the handle of any given right sentential form that can be generated by its associated grammar.

Examples of Grammar and Deriviation

- Consider the following grammar for arithmetic expressions,

$$E \rightarrow E + T | T$$
$$T \rightarrow T * F | F$$
$$F \rightarrow (E) | id$$

and the following rightmost derivation,

$$\begin{aligned} E &\Rightarrow \underline{E + T} \\ &\Rightarrow E + \underline{T * F} \\ &\Rightarrow E + T * \underline{id} \\ &\Rightarrow E + \underline{F} * id \\ &\Rightarrow E + \underline{id} * id \\ &\Rightarrow \underline{T} + id * id \\ &\Rightarrow \underline{F} + id * id \\ &\Rightarrow \underline{id} + id * id \end{aligned}$$

Examples of Grammar and Derivation

- Consider the following grammar for arithmetic expressions,

$$E \rightarrow E + T|T$$
$$T \rightarrow T * F|F$$
$$F \rightarrow (E)|id$$

  the right sentential form,

$$E + T * id$$

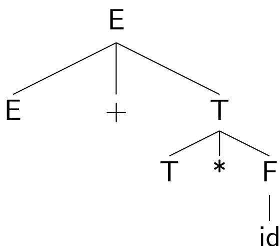
  what is the handle (or the correct RHS to reduce)?
- There are 3 RHS, $E + T$, T, and id
- Which one is the correct RHS to reduce?

Handle

- Definition: β is the *handle* of the right sentential form $\gamma = \alpha\beta w$ if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow^*_{rm} \alpha\beta w$.

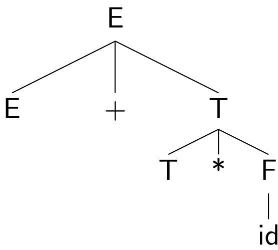
Phrase and Simple Phrase

- Definition: β is a *phrase* of the right sentential form γ if and only if $S \Rightarrow^* \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$
- Definition: β is a *simple phrase* of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$
- What a phrase is and what a simple phrase is relative to a parse tree?
- For example, consider the a parse tree for $E + T * id$



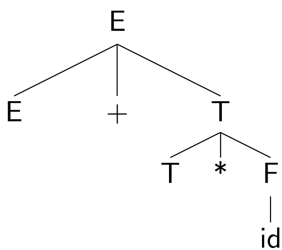
Phrase and Simple Phrase

- Consider the a parse tree for $E + T * id$
- 3 internal nodes, 3 subtrees, and 3 phrases
  - Leaves $E + T * id$ rooted at E
  - Leaves $T * id$ rooted at T
  - Leaves id rooted F
- Simple phrases are a subset of the phrases, in this example,
  - Leaves id rooted F



Intuition about Handles

- The handle of any rightmost sentential form is its leftmost simple phrase.
- With this intuition, consider again the a parse tree for $E + T * id$, what is the handle (or the correct RHS to reduce)?
  - There are 3 RHS, $E + T$, T, and id, which one is the correct RHS to reduce?
  - Since we know phrase id rooted F in the parse tree is a simple phrase, id is the handle.
  - $E + T * id$ should be reduced to $E + T * F$



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

Shift-Reduce Algorithms

- Bottom-up parsers are often called 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

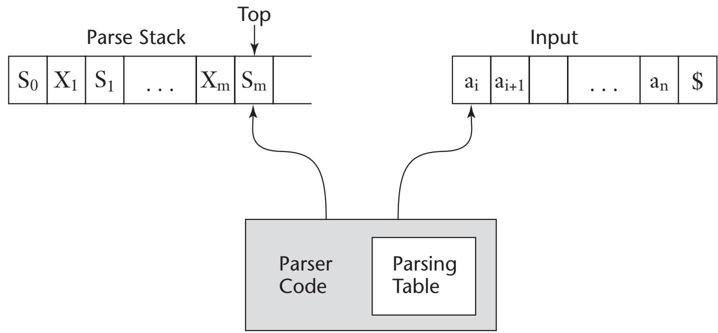
LR Parsers

- Many bottom-up parsing algorithms are variations of a process called LR
- Advantage
  - 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.
- Disadvantage
  - It is difficult to produce by hand the parsing table for a given grammar for a complete programming language.
- LR parsers are generally constructed with a tool.

Knuth's Insight

- The original LR algorithm was designed by Donald Knuth [Knuth, 1965]
- 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

The Structure of an LR Parser



- The LR parser configuration is a pair of strings (stack, input),

$$(S_0X_1S_1X_2S_2\ldots X_mS_m, a_ia_{i+1}\ldots a_n\$)$$

where the Ss are state symbols and the Xs are grammar symbols

LR Parsing Table

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

Parser Initialization and Actions

- Initial configuration: $(S_0, a_1 \ldots a_n\$)$
- Parser actions (determined by the ACTION table using the input as column index and the state on the stack as row index)
  - For a Shift, *push* the next symbol of input 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
    - Because for every grammar symbol on the stack there is a state symbol, the number of symbols removed from the stack is *twice* the number of symbols in the handle
  - 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: Example

Consider the following grammar for arithmetic expressions,

$$E \rightarrow E + T \quad (1)$$
$$E \rightarrow T \quad (2)$$
$$T \rightarrow T * F \quad (3)$$
$$T \rightarrow F \quad (4)$$
$$F \rightarrow (E) \quad (5)$$
$$F \rightarrow id \quad (6)$$

LR Parsing Table: Example

R for reduce and S for shift. $R4$ means reduce using rule 4; $S6$ means shift the next symbol of input onto the stack and push state $S6$ onto the stack.

	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			

LR Parsing: Example

Parsing string the string $id + id * id$

Stack	Input	Action
0	$id + id * id\$$	Shift 5
$0id5$	$+id * id\$$	Reduce 6 (use GOTO[0, F])
$0F3$	$+id * id\$$	Reduce 4 (use GOTO[0, T])
$0T2$	$+id * id\$$	Reduce 2 (use GOTO[0, E])
$0E1$	$+id * id\$$	Shift 6
$0E1 + 6$	$id * id\$$	Shift 5
$0E1 + 6id5$	$*id\$$	Reduce 6 (use GOTO[6, F])
$0E1 + 6F3$	$*id\$$	Reduce 4 (use GOTO[6, T])
$0E1 + 6T9$	$*id\$$	Shift 7
$0E1 + 6T9 * 7$	$id\$$	Shift 5
$0E1 + 6T9 * 7id5$	$\$$	Reduce 6 (use GOTO[7, F])
$0E1 + 6T9 * 7F10$	$\$$	Reduce 3 (use GOTO[6, T])
$0E1 + 6T9$	$\$$	Reduce 1 (use GOTO[0, E])
$0E1$	$\$$	Accept

Tool for Generating Parsing Table

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

Bottom-Up Parser Example using Lex and Yacc

Reimplemented previous example using a bttom-up parser with Lex and Yacc

Lex and Yacc Example

```
$
$ cd sebester/yyparser
$ make
$ ./yyparser < front.in
```

Bottom-Up Parser Example using Lex and Yacc

Result of Lex and Yacc Example

```
Graph 0:


     [/]
      |
   |------|
   |      |
  [+]   total
   |
  |--|
  |  |
 sum 47
```

Summary

- Syntax analysis is a common part of language implementation
- A lexical analyzer is a pattern matcher that isolates small-scale parts of a program
- A recursive-descent parser is an LL parser
- Parsing problem for bottom-up parsers: find the substring of current sentential form
- The LR family of shift-reduce parsers is the most common bottom-up parsing approach

References I

Aho, A., Lam, M., Sethi, R., and Ullman, J. (2006).
Compilers: Principles, techniques, and tools.
Addison-Wesley, 2nd edition.

Knuth, D. E. (1965).
On the translation of languages from left to right.
Information and Control, 8(6):607 – 639.

Sebesta, R. W. (2012).
Concepts of Programming Languages.
Pearson, 10th edition.