Recursive Descend

- Chapter 4, section 4.4
Recursive Descend

- Several uses
  - Parsing technique
  - Traversal of a given parse tree
    - for printing, code generation, etc.
- Basic idea: use a separate procedure for each non-terminal of the grammar
  - the body of the procedure “applies” some production for that non-terminal
- Start by calling the procedure for the starting non-terminal
Parser and Scanner Interactions

- The scanner maintains a “current” token
  - Initialized to the first token in the stream
- The parser calls `currentToken()` to get the first remaining token
  - `currentToken()` *does not change the token*
- The parser calls `nextToken()` to ask the scanner move to the next token
- Special token `END_OF_FILE` to represent the end of the input stream
Example: Simple Expressions

\[
\begin{align*}
\text{<expr>} & ::= \text{<term>} \mid \text{<term>} + \text{<expr>} \\
\text{<term>} & ::= \text{id} \mid \text{const} \mid (\text{<expr>})
\end{align*}
\]

Ignore error checking for now …

```cpp
procedure Expr() {
    Term();
    if (currentToken() == PLUS) {
        nextToken(); // consume the plus token
        Expr();
    }
}
```

We could rewrite the code to use a loop instead of recursion. How?
**Example: Simple Expressions**

\[
<expr> ::= <term> | <term> + <expr>
\]

\[
<term> ::= id | const | (<expr>)
\]

```plaintext
procedure Term() {
    if (currentToken() == ID)
        nextToken();
    else if (currentToken() == CONST)
        nextToken();
    else if (currentToken() == LPAREN) {
        nextToken(); // consume left parenthesis
        Expr(); // consume the expression
        nextToken(); // consume right parenthesis
    }
}
```
Error Checking During Parsing

- What checks of currentToken() do we need to make in Term()?
  - e.g. to catch “+a”, “(a+b” and “a+b)”
- May need to tweak the grammar to catch unexpected “leftover” tokens
  - We want to see END_OF_FILE token
How About the Parse Tree?

- Example: simple table representation
  - Each row corresponds to a parse tree node
  - Each row contains the non-terminal, the alternative, and info about children
    - For non-terminal children: the row number
    - For terminal children (tokens): the token
    - For ID - pointer to the symbol table
  - Some tokens are not used after parsing:
    - PROGRAM, BEGIN, END, INT, SEMICOL, INPUT, OUTPUT, IF, THEN, ELSE, ENDIF, WHILE, ENDWHILE, LPAREN, RPAREN, PLUS
$$xyz + ( 5 + abc )$$

<table>
<thead>
<tr>
<th>Row</th>
<th>NT</th>
<th>Altern</th>
<th>First</th>
<th>Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;expr&gt;</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>&lt;term&gt;</td>
<td>1</td>
<td>ID[1]</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>&lt;expr&gt;</td>
<td>1</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>&lt;term&gt;</td>
<td>3</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>&lt;expr&gt;</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>&lt;term&gt;</td>
<td>2</td>
<td>CONST[5]</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>&lt;expr&gt;</td>
<td>1</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>&lt;term&gt;</td>
<td>1</td>
<td>ID[2]</td>
<td>-</td>
</tr>
</tbody>
</table>

Symbol table:

<table>
<thead>
<tr>
<th>Index</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>xyz</td>
</tr>
<tr>
<td>2</td>
<td>abc</td>
</tr>
</tbody>
</table>

2nd alternative in the production for <expr>: i.e., <expr>::=<expr>+<term>
Implementing a Parser for Core

- One procedure per non-terminal
- Global table PT for the tree
- Global variable nextRow for the next available row in PT
  - initialized to 1
- Each procedure returns the row number of its node (needed for the parent)
Simple Example without Error Checking

<prog> ::= program <decl-seq> begin <stmt-seq> end

procedure Prog() returns integer {
    integer myRow = nextRow; nextRow++;
    PT[myRow,1] = "<prog>" // which non-terminal?
    PT[myRow,2] = 1 // which alternative?

    nextToken(); // consume the PROGRAM token

    integer declSeqRow = DeclSeq();
    PT[myRow,3] = declSeqRow; // the first child

    // more code for BEGIN, StmtSeq(), END

    return myRow;
}
Another Example

\[
\langle \text{if} \rangle ::= \text{if} \ \langle \text{cond} \rangle \ \text{then} \ \langle \text{stmt-seq} \rangle \ \text{endif} \ ; \\
| \text{if} \ \langle \text{cond} \rangle \ \text{then} \ \langle \text{stmt-seq} \rangle \ \text{else} \ \langle \text{stmt-seq} \rangle \ \text{endif} \ ;
\]

procedure \textbf{If}() returns integer {
    integer myRow = nextRow; nextRow++;
    PT[myRow,1] = "<if>" \ // which non-terminal?
    nextToken(); \ // consume the \textbf{IF} token
    integer condRow = \textbf{Cond}(); PT[myRow,3] = condRow;
    nextToken(); \ // consume the \textbf{THEN} token
    integer thenRow = \textbf{StmtSeq}(); PT[myRow,4] = thenRow;
    if (currentToken() == ELSE) {
        nextToken(); \ // consume \textbf{ELSE} token
        integer elseRow = \textbf{StmtSeq}(); PT[myRow,5] = elseRow;
        PT[myRow,2] = 2; \ // second alternative
    } else PT[myRow,2] = 1; \ // first alternative
    nextToken(); nextToken(); \ // \textbf{ENDIF} and \textbf{SEMICOL}
    return myRow; }

Which Alternative to Use?

- The key issue: must be able to decide which alternative to use, based on the current token

- For each alternative: what is the set FIRST of all terminals that can be at the very beginning of strings derived from that alternative?

- If the sets FIRST are disjoint, we can decide uniquely which alternative to use
Sets FIRST

\[ \text{<decl-seq> ::= <decl> | <decl><decl-seq>} \]
\[ \text{<decl> ::= int <id-list> ;} \]

FIRST is \{ \text{int} \} for both alternatives: not disjoint!!

1. Introduce a helper non-terminal \text{<rest>}

\[ \text{<decl-seq> ::= <decl> <rest>} \]
\[ \text{<rest> ::= empty string | <decl-seq>} \]

2. FIRST for the empty string is \{ begin \}, because of \text{<prog> ::= program <decl-seq> begin ...} \\

3. FIRST for \text{<decl-seq> is \{ int \}}
procedure DeclSeq() returns integer {
    ...
    integer declRow = Decl();
    integer restRow = Rest();
    ...
}

procedure Rest() returns integer {
    ...
    if (currentToken() == BEGIN) return myRow;
    if (currentToken() == INT) {
        ... DeclSeq(); return myRow; }
}
A Simplification

- After this, we can even remove the helper non-terminal

procedure DeclSeq() returns integer {
    ...
    integer declRow = Decl();
    ...
    if (currentToken() == BEGIN) ...
    if (currentToken() == INT) { ...
        DeclSeq(); ...
    }
    ...
    return myRow;
}
Core: a toy imperative language

```plaintext
<prog> ::= program <decl-seq> begin <stmt-seq> end
<decl-seq> ::= <decl> | <decl><decl-seq>
<stmt-seq> ::= <stmt> | <stmt><stmt-seq>
<decl> ::= int <id-list> ;   <id-list> ::= id | id , <id-list>
<stmt> ::= <assign> | <if> | <loop> | <in> | <out>
<assign> ::= id := <exp> ;
<in> ::= input <id-list> ;    <out> ::= output <id-list> ;
<if> ::= if <cond> then <stmt-seq> endif ;
  | if <cond> then <stmt-seq> else <stmt-seq> endif ;
```
Core: a toy imperative language

<loop> ::= while <cond> begin <stmt-seq> endwhile ;
<cond> ::= <comp> | ! <cond> | ( ( <cond> AND <cond> ) ) | ( <cond> OR <cond> )
<comp> ::= [ <operand> <comp-op> <operand> ]
<comp-op> ::= < | = | != | > | >= | <=
<exp> ::= <term> | <term> + <exp> | <term> - <exp>
<term> ::= <operand> | <operand> * <term>
<operand> ::= const | id | ( <exp> )
Another Grammar

- **id** and **const** are terminal symbols for the grammar of the language
  - **tokens** that are provided from the lexical analysis to the parser

- But they are non-terminals for the regular grammar in the lexical analysis
  - The terminals now are characters
    - `<id> ::= <letter> | <id><letter> | <id><digit>`
    - `<letter> ::= A | B | ... | Z | a | b | ... | z`
    - `<const> ::= <digit> | <const><digit>`
    - `<digit> ::= 0 | 1 | ... | 9`
Sets FIRST

Q1: `<id-list> ::= id | id , <id-list>`

What do we do here? What are sets FIRST?

Q2: `<stmt> ::= <assign> | <if> | <loop> | <in> | <out>`

What are sets FIRST here?

Q3: `<stmt-seq> ::= <stmt> | <stmt><stmt-seq>`

Q4: `<cond> ::= <comp> | ! <cond> | ( <cond> AND <cond> ) | ( <cond> OR <cond> )

<comp> ::= [ <operand> <comp-op> <operand> ]`

Q5: `<exp> ::= <term> | <term>+<exp> | <term>-<exp>

<term> ::= <operand> | <operand> * <term>`
How about this?

- We have
  - \(<exp> ::= <term> | <term> + <exp> | <term> - <exp>\)

- How about recursive descend for
  - \(<exp> ::= <term> | <exp> + <term> | <exp> - <term>\)

- There are systematic ways to deal with this problem, but we will not cover them in this class

- Recursive descent is “top-down”, but a more powerful and useful is “bottom-up” parsing (e.g., used in real compilers)
Recursive Descend Printing

- Given a parse tree, how can we print the underlying program?

```plaintext
<if> ::= if <cond> then <stmt-seq> endif ;
| if <cond> then <stmt-seq> else <stmt-seq> endif ;
procedure PrintIf(integer row) {
    print ("if ");
    PrintCond( PT[row,3] ); // the row for the first child
    print(" then ");
    PrintStmtSeq( PT[row,4] );
    if (PT[row,2] == 2) // the second alternative, with else
    { print(" else "); PrintStmtSeq( PT[row,5] ); } 
    print(" endif; ");
}
```
Recursive Descend Execution

Given a parse tree, how can we execute the underlying program?

\[
<\text{if}> ::= \text{if} \ <\text{cond}> \ \text{then} \ <\text{stmt-seq}> \ \text{endif} \ ; \\
| \ \text{if} \ <\text{cond}> \ \text{then} <\text{stmt-seq}> \ \text{else} <\text{stmt-seq}> \ \text{endif} \ ;
\]

procedure \text{ExecIf}(\text{integer} \ \text{row}) \{
    \text{boolean} \ x = \text{EvalCond}( \text{PT}[\text{row},3] ) ; \\
    \text{if} \ (x) \{ \text{ExecStmtSeq}( \text{PT}[\text{row},4] ) ; \text{return} ; \} \\
    \text{if} \ (\text{PT}[\text{row},2] == 2) \ // \text{the second alternative, with else} \\
        \{ \text{ExecStmtSeq}( \text{PT}[\text{row},5] ) ; \}
\}
Hmm, how about data abstraction?

- The low-level details of the parse tree representation are exposed to the parser, the printer, and the executor.
- What if we want to change this representation?
  - e.g., move to a representation based on singly-linked lists?
  - what if later we want to change from singly-linked to doubly-linked list?
- Key principle: hide the low-level details.
A ParseTree data type

- Hides the implementation details behind a “wall” of operations
  - Could be implemented, for example, as a C++ or Java class
  - Maintains a “cursor” to a current node

- What are the operations that should be available to the parser, the printer, and the executor?
  - moveCursorToRoot()
  - isCursorAtRoot()
  - moveCursorUp() - precondition: not at root
More Operations

- Traversing the children
  - `moveCursorToFirstChild()`, etc.

- Info about the node
  - `getNonterminal()`: returns some representation: e.g., an integer id or a string
  - `getAlternativeNumber()`: which alternative in the production was used?

- During parsing: creating parse tree nodes
  - Need to maintain a symbol table - either inside the PT, or as a separate data type
Example with Printing

procedure PrintIf(PT * tree) { // C++ pointer parameter
print("if ");
tree->moveCursorToFirstChild();
PrintCond(tree);
tree->moveCursorUp();
print(" then ");
tree->moveCursorToSecondChild();
PrintStmtSeq(tree);
tree->moveCursorUp();
if (tree->getAlternativeNumber() == 2) {
    print(" else ");
tree->moveCursorToThirdChild();
PrintStmtSeq(tree);
tree->moveCursorUp();
}
print(" endif; "); }

CSE 655
Another Possible Implementation

- The object-oriented way: put the data and the code together
  - The C++ solution in the next few slides has a lot of room for improvement
- A separate class for each non-terminal $X$
  - An instance of $X$ (i.e., an object of class $X$) represents a parse tree node
  - Fields inside the object are pointers to the children nodes
  - Methods $\text{parse}X()$, $\text{print}X()$, $\text{exec}X()$
Class Prog for Non-Terminal <prog>

class Prog {
private: DS* ds; SS* ss;
public:
    Prog() { ds = NULL; ss = NULL; }
    void parseProg() {
        // check and read token PROGRAM
        ds = new DS(); ds->parseDS();
        // check and read token BEGIN
        ss = new SS(); ss->parseSS();
        // check and read END and ENDOFFILE
    }
    void printProg() {
        cout << "program"; ds->printDS();
        cout << "begin"; ss->printSS(); cout << "end;";
    }
    void execProg() {
        ds->execDS(); ss->execSS();
    }
};
Class **SS** for Non-Terminal <stmt-seq>

class **SS** {
    private: Stmt* s; SS* ss;
    public:
        **SS**() { s = NULL; ss = NULL; }
        void **parseSS**() {
            s = new Stmt(); s->**parseStmt**();
            // if the current token is END, return.
            // otherwise, do error checking.
            ss = new SS(); ss->**parseSS**();
        }
        void **printSS**() {
            s->**printStmt**();
            if (ss == NULL) return; ss->**printSS**();
        }
        void **execSS**() {
            s->**execStmt**();
            if (ss == NULL) return; ss->**execSS**();
        }
};
Class Stmt for Non-Terminal <stmt>

class Stmt {
    private: int altNo; Assign* s1; IfThenElse* s2;
                Loop* s3; Input* s4; Output* s5;

    public:
    Stmt() { altNo = 0; s1 = NULL; ... }
    void parseStmt() {
        if (...) { // current token is ID
            altNo = 1; s1 = new Assign(); s1->parseAssign(); return;
        }
        if (...) ...
    }
    void printStmt() {
        if (altNo == 1) { s1->printAssign(); return; }
        ...
    }
    void execStmt() {
        if (altNo == 1) { s1->execAssign(); return; }
        ...
    }
};