Recursive Descent

Chapter 2: Section 2.3

Recursive Descent

- Several uses
  - Parsing technique
    - Call the scanner to obtain tokens, build a parse tree
  - 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
  - Calling currentToken() does **not** change the token
- The parser calls **nextToken()** to ask the scanner to move to the next token
- Special pseudo-token end-of-file **EOF** to represent the end of the input stream

Example: Simple Expressions (1/2)

<expr> ::= <term> | <term> **+** <expr>

<term> ::= **id** | **const** | **(**<expr>**)**

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

Ignore error checking for now …

Could rewrite to use a loop instead of recursion. How?

Example: Simple Expressions (2/2)

<expr> ::= <term> | <term> **+** <expr>

<term> ::= **id** | **const** | **(**<expr>**)**

```
procedure Term() {
    if (currentToken() == ID) nextToken();
    else if (currentToken() == CONST) nextToken();
    else if (currentToken() == LPAREN) {
        nextToken(); // consume left parenthesis
        Expr();
        nextToken(); // consume right parenthesis
    }}
```

Error Checking

- What checks of currentToken() do we need to make in Term()?
  - E.g., to catch **“+a”** and **“(a+b”**
- Unexpected leftover tokens: tweak the grammar
  - E.g., to catch **“a+b)”**
  - <start> ::= <expr> **eof**
  - Inside the code for Expr(), the current token should be either **PLUS** or **EOF**

Writing the Parser

- For each non-terminal N: a parsing procedure **N**()
- In the procedure: look at the current token and decide which alternative to apply
- For each symbol X in the alternative:
  - If X is a terminal: match it (e.g., via helper func **match**)
    - Check X == currentToken()
    - Consume it by calling nextToken()
  - If X is a non-terminal, call parsing procedure **X**()
- If S is the starting non-terminal, the parsing is done by a call **S**() followed by a call **match**(**EOF**)

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
    - E.g., for **Core**: PROGRAM, BEGIN, END, INT, SEMICOL, INPUT, OUTPUT, IF, THEN, ELSE, ENDIF, WHILE, ENDWHILE, LPAREN, RPAREN, PLUS

xyz + (5 + abc)

Row	NT	Altern	First	Second
1	<expr>	②	2	3
2	<term>	1	ID[1]	-
3	<expr>	1	4	-
4	<term>	3	5	-
5	<expr>	2	6	7
6	<term>	2	CONST[5]	-
7	<expr>	1	8	-
8	<term>	1	ID[2]	-

Index	Symbol
1	xyz
2	abc

Symbol 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 setting up the table row corresponding to the parent node

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?

  match(PROGRAM); // check and consume the PROGRAM token

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

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

  return myRow;
}
```

Another Example

<if> ::= **if** <cond> **then** <stmt-seq> **endif ;**
        | **if** <cond> **then** <stmt-seq> **else** <stmt-seq> **endif ;**

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

Which Alternative to Use?

- The key issue: must be able to decide which alternative to use, based on the current token
  - Predictive parsing: predict correctly (without backtracking) what we need to do, by looking at a few tokens ahead
  - In our case: look at just one token (the current one)
- 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

<decl-seq> ::= <decl> | <decl><decl-seq>

<decl> ::= **int** <id-list> **;**

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

1. Introduce a helper non-terminal <rest>

<decl-seq> ::= <decl> <decl-rest>

<decl-rest> ::= empty string | <decl-seq>

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

3. FIRST for <decl-seq> is { **int** }

Parser Code

```
procedure DeclSeq() returns integer {
   …
   integer declRow = Decl();
   integer restRow = DeclRest();
   … }
procedure DeclRest() returns integer {
   …
   if (currentToken() == BEGIN) return myRow;
   if (currentToken() == INT)
      { … DeclSeq(); … return myRow; }
}
```

Simplified Parser Code

Now we can remove the helper non-terminal

```
procedure DeclSeq() returns integer {
   …
   integer declRow = Decl();
   …
   if (currentToken() == BEGIN) return myRow;
   if (currentToken() == INT)
      { … DeclSeq(); … return myRow; }
}
```

Can we replace the recursion with a loop?

Core: A Toy Imperative Language (1/2)

<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 :=** <expr> **;**

<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 (2/2)

<loop> ::= **while** <cond> **begin** <stmt-seq> **endwhile ;**

<cond> ::= <cmpr> | **!** <cond> | **(** <cond> **AND** <cond> **)**

| **(** <cond> **OR** <cond> **)**

<cmpr> ::= **[** <expr> <cmpr-op> <expr> **]**

<cmpr-op> ::= **<** | **=** | **!=** | **>** | **>=** | **<=**

<expr> ::= <term> | <term> **+** <expr> | <term> **–** <expr>

<term> ::= <factor> | <factor> * <term>

<factor> ::= **const** | **id** | **–** <factor> | **(** <expr> **)**

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> ::= <cmpr> | **!** <cond> |
(<cond> **AND** <cond> **)** | **(** <cond> **OR** <cond> **)**
<cmpr> ::= **[** <expr> <cmpr-op> <expr> **]**

Q5: <expr> ::= <term>|<term> **+** <expr>|<term> **–** <expr>
<term> ::= <factor> | <factor> * <term>
<factor> ::= **const** | **id** | **–** <factor> | **(** <expr> **)**

More General Parsing

- We have

  <expr> ::= <term>|<term> **+** <expr>|<term> **–** <expr>

- How about

  <expr> ::= <term>|<expr> **+** <term>|<expr> **–** <term>

- Left-recursive grammar: possible $A \Rightarrow \ldots \Rightarrow A\alpha$
  - Not suitable for predictive recursive-descent parsing
- General parsing: top-down vs. bottom-up
  - We considered an example of top-down parsing for LL(1) grammars
  - In real compilers: bottom-up parsing for LR(k) grammars (more powerful, discussed in CSE 5343)

Recursive Descent Printing

- Given a parse tree, print the underlying program

<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 Descent Execution

- Given a parse tree, execute the underlying program

<if> ::= **if** <cond> **then** <stmt-seq> **endif ;**
  | **if** <cond> **then** <stmt-seq> **else** <stmt-seq> **endif ;**

```
procedure ExecIf(integer row) {
    boolean x = EvalCond( PT[row,3] );
    if (x) { ExecStmtSeq( PT[row,4] ); return; }
    if (PT[row,2] == 2) // the second alternative, with else
              { ExecStmtSeq( PT[row,5] ); }
}
```

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

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 the 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
  - moveCursorToChild(int x), where x is child number
- 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 ParseTree type, or as a separate data type

Example with Printing

```
procedure PrintIf(PT* tree) { // C++ pointer parameter
  print ("if ");
  tree->moveCursorToChild(1);
  PrintCond(tree);
  tree->moveCursorUp();
  print(" then ");
  tree->moveCursorToChild(2);
  PrintStmtSeq(tree);
  tree->moveCursorUp();
  if (tree->getAlternativeNumber() == 2) { // second alternative, with else
    print(" else ");
    tree->moveCursorToChild(3);
    PrintStmtSeq(tree);
    tree->moveCursorUp();
  }
  print(" endif;"); }
```

Another Possible Implementation

- The object-oriented way: put the data and the code together
  - The C++ solution in the next few slides is just a sketch; 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 **parse**(), **print**(), **exec**()

Class Prog for Non-Terminal <prog>

```
class Prog {
   private: DeclSeq* decl_seq; StmtSeq* stmt_seq;
   public:
      Prog() { decl_seq = NULL; stmt_seq = NULL; }
      void parse() {
            scanner->match(PROGRAM);
            decl_seq = new DeclSeq(); decl_seq->parse();
            scanner->match(BEGIN);
            stmt_seq = new StmtSeq(); stmt_seq->parse();
            scanner->match(END); scanner->match(EOF);
     }
     void print() {
            cout << "program "; decl_seq->print();
            cout << " begin "; stmt_seq->print(); cout << " end;";
     }
     void exec() {
            decl_seq->exec(); stmt_seq->exec();
     } };
```

Class StmtSeq for Non-Terminal <stmt-seq>

```
class StmtSeq {
  private: Stmt* stmt; StmtSeq* stmt_seq;
  public:
     StmtSeq() { stmt = NULL; stmt_seq = NULL; }
     void parse() {
            stmt = new Stmt(); stmt->parse();
            if (scanner->currentToken() == END) return;
            // Same for ELSE, ENDIF, ENDWHILE
            stmt_seq = new StmtSeq(); stmt_seq->parse();
      }
      void print() {
           stmt->print();
           if (stmt_seq != NULL) stmt_seq->print();
      }
      void exec() {
            stmt->exec();
            if (stmt_seq != NULL) stmt_seq->exec();
      } };
```

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 = s2 = s3 = s4 = s5 = NULL; }
     void parse() {
          if (scanner->currentToken() == ID) {
          altNo = 1; s1 = new Assign(); s1->parse();  return;}
          if (scanner->currentToken() == …) …
     }
     void print() {
          if (altNo == 1) { s1->print(); return; }
          …
     }
     void exec() {
          if (altNo == 1) { s1->exec(); return; }
          …
     } };
```