Recursive Descent Parser

Expression Grammar

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4	| - *term expr2*
5	| ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| ε
10	*factor* → number
11	| identifier
12	| (*expr*)

- Give the left-most derivation and draw the parse tree for the following string:
- **number + (number + identifier) * number**
- **ϵ here is the empty string, i.e.**
  - **If you apply expr2 → ϵ in** number + number expr2
  - **you simply get** number + number

Recursive Descent

#	Production rule
1	goal → expr
2	expr → term expr2
3	expr2 → + term expr2
4	| - term expr2
5	| ε
6	term → factor term2
7	term2 → * factor term2
8	| / factor term2
9	| ε
10	factor → number
11	| identifier
12	| (expr)

- This produces a parser with six *mutually recursive* routines:
  - *Goal*
  - *Expr*
  - *Expr2*
  - *Term*
  - *Term2*
  - *Factor*
- The term *descent* refers to the direction in which the parse tree is built.

Example Code

- Goal symbol:

```
main()
  /* Match goal ––> expr */
  tok = nextToken();
  if (expr() && tok == EOF)
    then proceed to next step;
    else return false;
```

- Top-level expression

```
expr()
  /* Match expr ––> term expr2 */
  if (term() && expr2());
    return true;
  else return false;
```

Example Code

- Match expr2

```
expr2()
  /* Match expr2 ⟶ + term expr2 */
  /* Match expr2 ⟶ - term expr2 */

  if (tok == '+' or tok == '-')
    tok = nextToken();
    if (term())
      then return expr2();
      else return false;

  /* Match expr2 --> empty */
  return true;
```

Example Code

```
factor()
  /* Match factor --> ( expr ) */
  if (tok == '(')
    tok = nextToken();
    if (expr() && tok == ')')
      return true;
    else
      syntax error: expecting )
      return false

  /* Match factor --> num */
  if (tok is a num)
    return true

  /* Match factor --> id */
  if (tok is an id)
    return true;
```

Top-Down Parsing

- So far:
  - Gives us a yes or no answer
  - We want to build the parse tree
  - How?

- Add actions to matching routines
  - Create a node for each production
  - How do we assemble the tree?

Building a Parse Tree

- Notice:
  - Recursive calls match the shape of the tree

```
main
  expr
    term
      factor
    expr2
      term
```

- Idea: use a stack
  - Each routine:
    - Pops off the children it needs
    - Creates its own node
    - Pushes that node back on the stack

Building a Parse Tree

- With stack operations

```
expr()
  /* Match expr --> term expr2 */
  if (term() && expr2())
    expr2_node = pop();
    term_node = pop();
    expr_node = new exprNode(term_node,
                             expr2_node)
    push(expr_node);
    return true;
  else return false;
```

Practice Assignment#2

- Write a recursive descent parser for the expression grammar.
- **Input**: any string
  - If there is any black space remove it
  - All valid **numbers** will be single digits and all **identifiers** will be single characters.
- **Output**: is either *Success and the list of productions that will generate the string or Error.*
  - **Success** if the string can be generated by the grammar
    - **Each production required to generate the string should be printed in a separate line**
  - **Error** if there is a syntax error.
    - **Produce appropriate error messages** (for example : “operator missing”, “unexpected (”, “illegal character”)

Submission

- Please submit your codeany programming language you like (C, C++, JAVA etc.) + a file containing sample input and output of your program.
  - **Put these two in a zip file and name it pa2_roll_name.zip**
  - **Put you name and id as comments in the source code file**
  - **Upload the file in piazza as a private note to instructor.**

Good Luck