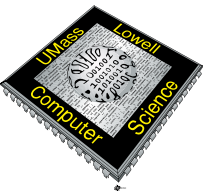
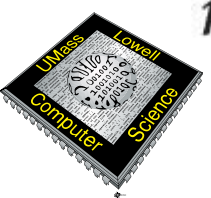
COMP 3080 – OPL Exception-Based Syntax Error Recovery (for Assignment 2)

Dr. Tom Wilkes

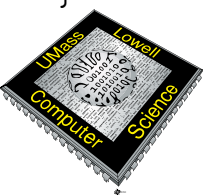
Spring 2019



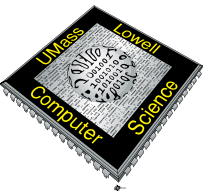
$$
\begin{aligned}
\textit{program} &\longrightarrow \textit{stmt\_list}\ \texttt{\$\$} \\
\textit{stmt\_list} &\longrightarrow \textit{stmt}\ \textit{stmt\_list} \mid \epsilon \\
\textit{stmt} &\longrightarrow \texttt{id :=}\ \textit{expr} \mid \texttt{read id} \mid \texttt{write}\ \textit{expr} \\
\textit{expr} &\longrightarrow \textit{term}\ \textit{term\_tail} \\
\textit{term\_tail} &\longrightarrow \textit{add\_op}\ \textit{term}\ \textit{term\_tail} \mid \epsilon \\
\textit{term} &\longrightarrow \textit{factor}\ \textit{factor\_tail} \\
\textit{factor\_tail} &\longrightarrow \textit{mult\_op}\ \textit{factor}\ \textit{factor\_tail} \mid \epsilon \\
\textit{factor} &\longrightarrow \texttt{(}\ \textit{expr}\ \texttt{)} \mid \texttt{id} \mid \texttt{number} \\
\textit{add\_op} &\longrightarrow \texttt{+} \mid \texttt{-} \\
\textit{mult\_op} &\longrightarrow \texttt{*} \mid \texttt{/}
\end{aligned}
$$



```
void factor () {
    switch (input_token) {
        case t_id :
            printf ("predict factor --> id\n");
            match (t_id);
            break;
        case t_literal:
            printf ("predict factor --> literal\n");
            match (t_literal);
            break;
        case t_lparen:
            printf ("predict factor --> lparen expr rparen\n");
            match (t_lparen);
            expr ();
            match (t_rparen);
            break;
        default: error ();
    }
}
```



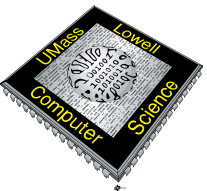
```
void error () {
    printf ("syntax error\n");
    exit (1);
}

void match (token expected) {
    if (input_token == expected) {
        printf ("matched %s", names[input_token]);
        if (input_token == t_id || input_token == t_literal)
            printf (": %s", token_image);
        printf ("\n");
        input_token = scan ();
    }
    else error ();
}
```



[From the description of Assignment 2:]

Implement exception-based syntax error recovery, as described in Section 2.3.5 on the textbook's companion site. **At the least, you should attach handlers to statements, relations, and expressions.** […]

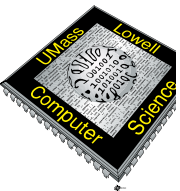
When `match` sees a token other than the one it expects, it could simply throw a `Syntax_Error` exception. The resulting algorithm would recover by deletion only: The exception handler will delete tokens until it finds something in either the FIRST set or the FOLLOW set of the nonterminal corresponding to the current recursive descent routine. **An attractive alternative is to mirror Wirth's recovery algorithm, and have `match` insert what it expects and continue (presumably after printing an error message).** You may implement either strategy.



Exception-Based Recovery in Recursive Descent

Pseudo-code for statement-level error recovery:

```
procedure statement()
    try
        . . .                                  -- code to parse a statement
    except when syntax_error
        loop
            if next_token ∈ FIRST(statement)
                statement()                    -- try again
                return
            elsif next_token ∈ FOLLOW(statement)
                return
            else get_next_token()
```



FIRST

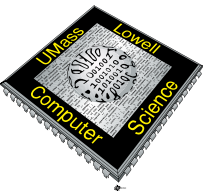
program {`id`, `read`, `write`, `$$`}
stmt_list {`id`, `read`, `write`}
stmt {`id`, `read`, `write`}
expr {`(`, `id`, `number`}
term_tail {`+`, `-`}
term {`(`, `id`, `number`}
factor_tail {`*`, `/`}
factor {`(`, `id`, `number`}
add_op {`+`, `-`}
mult_op {`*`, `/`}

FOLLOW

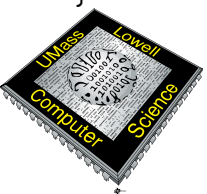
program ∅
stmt_list {`$$`}
stmt {`id`, `read`, `write`, `$$`}
expr {`)`, `id`, `read`, `write`, `$$`}
term_tail {`)`, `id`, `read`, `write`, `$$`}
term {`+`, `-`, `)`, `id`, `read`, `write`, `$$`}
factor_tail {`+`, `-`, `)`, `id`, `read`, `write`, `$$`}
factor {`+`, `-`, `*`, `/`, `)`, `id`, `read`, `write`, `$$`}
add_op {`(`, `id`, `number`}
mult_op {`(`, `id`, `number`}

PREDICT

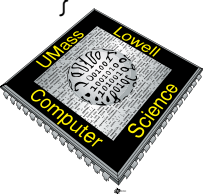
1. *program* ⟶ *stmt_list* `$$` {`id`, `read`, `write`, `$$`}
2. *stmt_list* ⟶ *stmt* *stmt_list* {`id`, `read`, `write`}
3. *stmt_list* ⟶ ϵ {`$$`}
4. *stmt* ⟶ `id :=` *expr* {`id`}
5. *stmt* ⟶ `read id` {`read`}
6. *stmt* ⟶ `write` *expr* {`write`}
7. *expr* ⟶ *term* *term_tail* {`(`, `id`, `number`}
8. *term_tail* ⟶ *add_op* *term* *term_tail* {`+`, `-`}
9. *term_tail* ⟶ ϵ {`)`, `id`, `read`, `write`, `$$`}
10. *term* ⟶ *factor* *factor_tail* {`(`, `id`, `number`}
11. *factor_tail* ⟶ *mult_op* *factor* *factor_tail* {`*`, `/`}
12. *factor_tail* ⟶ ϵ {`+`, `-`, `)`, `id`, `read`, `write`, `$$`}
13. *factor* ⟶ `(` *expr* `)` {`(`}
14. *factor* ⟶ `id` {`id`}
15. *factor* ⟶ `number` {`number`}
16. *add_op* ⟶ `+` {`+`}
17. *add_op* ⟶ `-` {`-`}
18. *mult_op* ⟶ `*` {`*`}
19. *mult_op* ⟶ `/` {`/`}



```
void error () {
    printf ("syntax error\n");
    return;
}

void match (token expected) throw (Syntax_Error) {
    if (input_token == expected) {
        printf ("matched %s", names[input_token]);
        if (input_token == t_id || input_token == t_literal)
            printf (": %s", token_image);
        printf ("\n");
        input_token = scan ();
    }
    else {
        error ();
        input_token = expected; // Implement Wirth's method
        throw Syntax_Error(/* parameters? */);
    }
}
```



```
void factor () {
  try {
     switch (input_token) {
          case t_id :
               printf ("predict factor --> id\n");
               match (t_id);
               break;
          case t_literal:
               printf ("predict factor --> literal\n");
               match (t_literal);
               break;
          case t_lparen:
               printf ("predict factor --> lparen expr rparen\n");
               match (t_lparen);
               expr ();
               match (t_rparen);
               break;
          default: error (); throw Syntax_Error( /* parameters? */ );
     }
  }
  catch (Syntax_Error /* parameters? */) {
    /* Implement recovery loop from pseudo-code – see next slide */
  }
}
```



```
catch (Syntax_Error /* parameters? */) {
  while (true) {
    switch (input_token)
    {
    /* FIRST (factor) */
    case t_lparen: case t_id: case t_literal:
      factor (); // try again
      return;

    /* FOLLOW (factor) */
    case t_add: case t_sub: case t_mul: case t_div:
    case t_rparen: case t_id: case t_read: case t_write:
    case t_eof:
      return;

    default:
      input_token = scan(); // get next token
  }
}
```