CS 445: Error Recovery

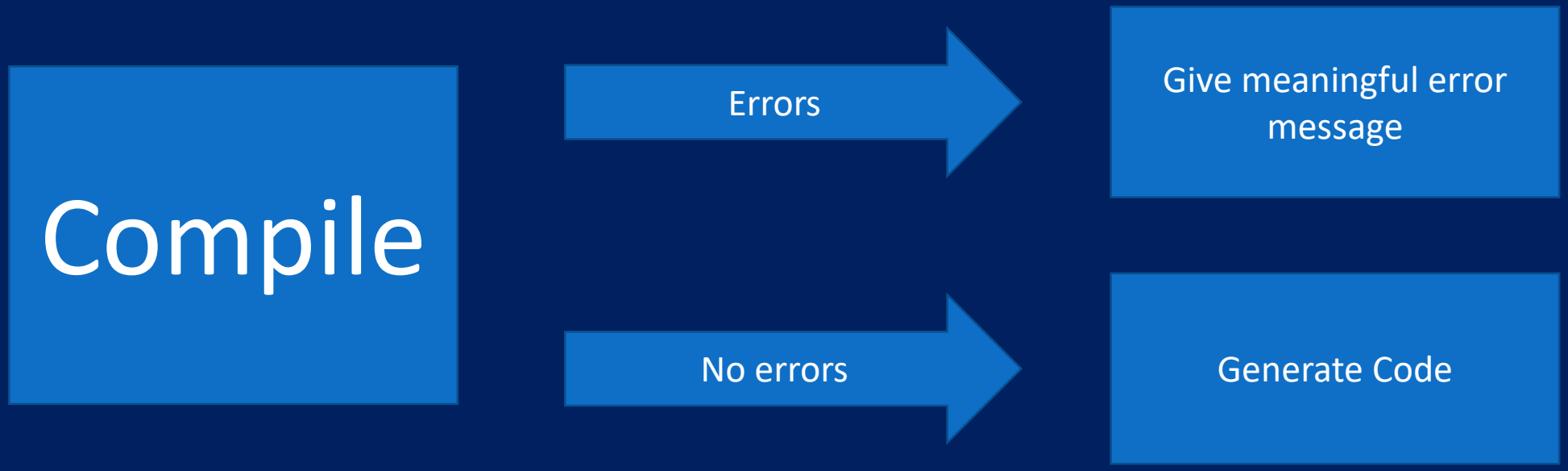
yyerror.cpp - You will need to update this

yyerror.h

You will also need to update semantics.cpp parser.y and (possibly) parser.l

Note:

- With every other assignment, you must complete the last one before you can move on to the next.
- You can do the last 2 in tandem.

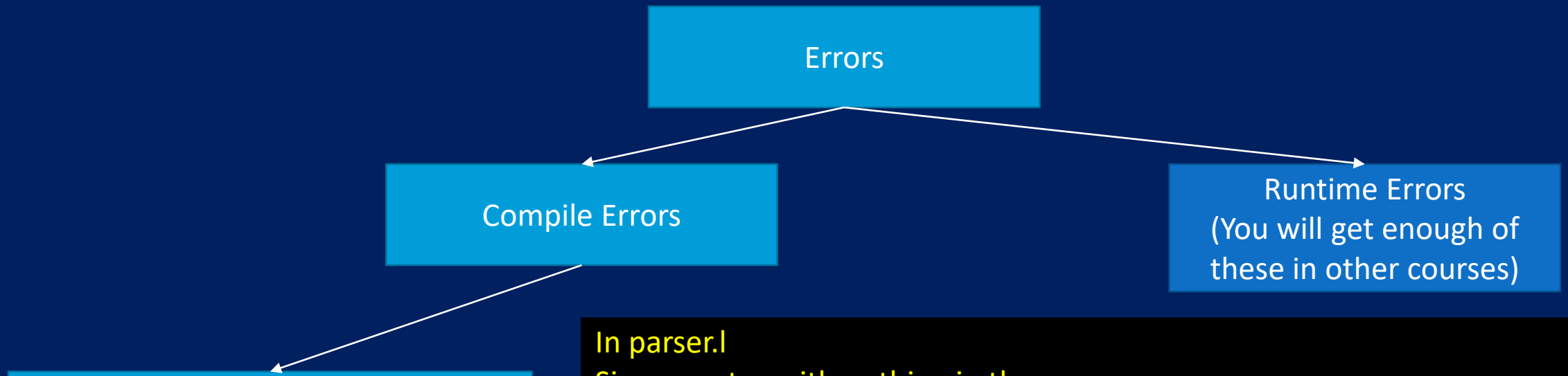


Error Recovery

- What should happen when your compiler finds an error in the user's input?
  - stop immediately and signal an error
  - record the error but try to continue
- In the first case, the user must recompile from scratch after possibly a trivial fix
- In the second case, the user might be overwhelmed by a whole series of error messages, all caused by essentially the same problem
- We will talk about how to do error recovery in a principled way

Error Recovery

- Error recovery:
  - process of adjusting input stream so that the parser can continue after unexpected input
- Possible adjustments:
  - delete tokens
  - insert tokens
  - substitute tokens
- Classes of recovery:
  - local recovery: adjust input at the point where error was detected (and also possibly immediately after)
  - global recovery: adjust input before point where error was detected.



In parser.l

Singe quotes with nothing in them:

```
{ printf("TOKEN ERROR(%d): empty character ''.  Characters ignored.\n", line);
numErrors++; }
```

Singe quotes with too much in them:

```
{ printf("TOKEN ERROR(%d): muli-char character %s.  Characters ignored.\n", line,
yytext); numErrors++;  }
```

Everything else: (.)

```
{ printf("TOKEN ERROR(%d): invalid or misplaced input character: \'%c\'.
Character Ignored.\n", line, yytext[0]); numErrors++;}
```

Errors

Compile Errors

Runtime Errors
(You will get enough of these in other courses)

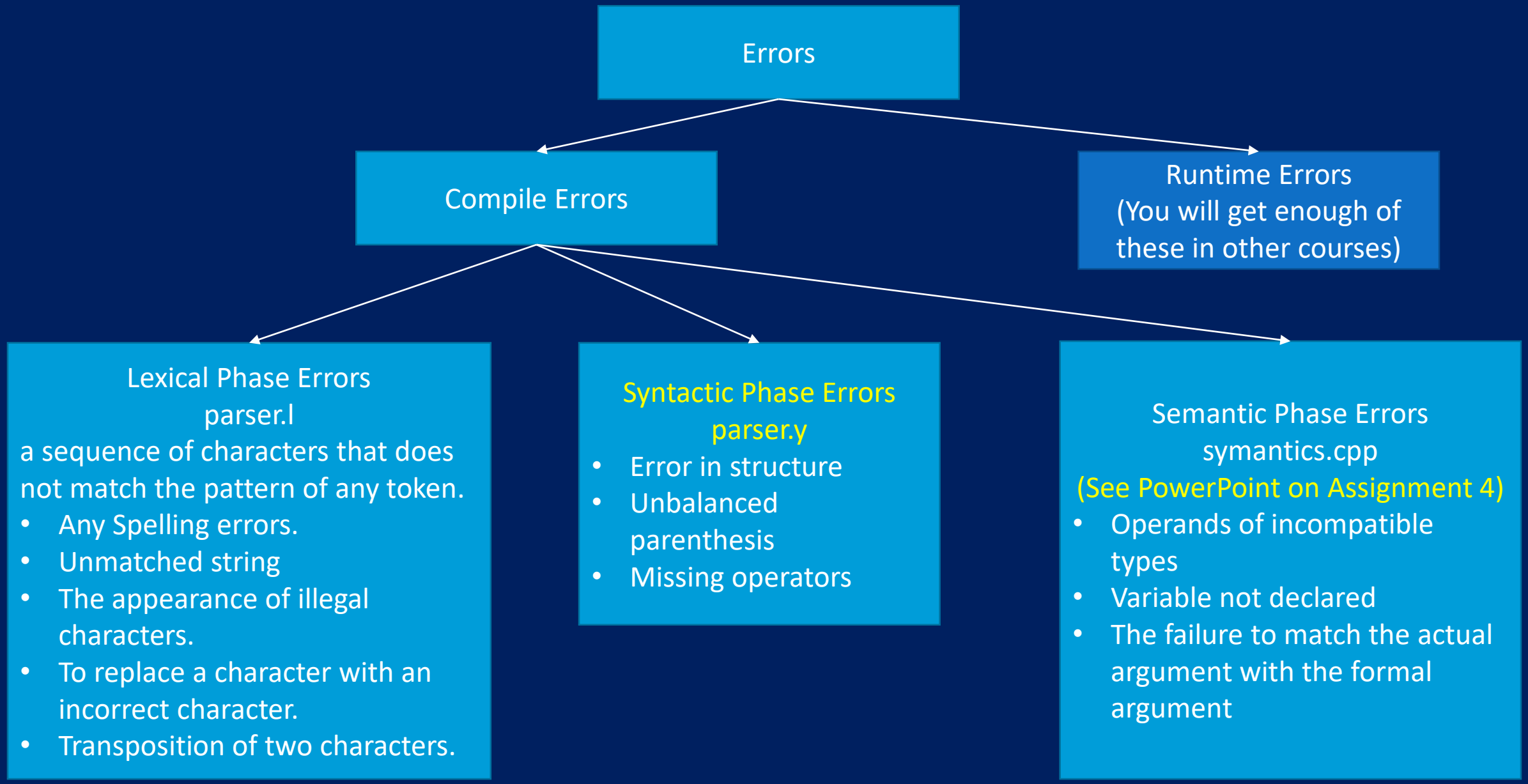
Lexical Phase Errors
parser.l
a sequence of characters that does not match the pattern of any token.

- Any Spelling errors.
- Unmatched string
- The appearance of illegal characters.
- To replace a character with an incorrect character.
- Transposition of two characters.

```
int x;

int x; // x is already defined

y = 73; //y is undefined

x = true; // incompatible type

Etc.
```

Semantic Phase Errors
symantics.cpp
(See PowerPoint on Assignment 4)

- Operands of incompatible types
- Variable not declared
- The failure to match the actual argument with the formal argument


Errors
Compile Errors
Runtime Errors
(You will get enough of these in other courses)
Lexical Phase Errors
parser.l
a sequence of characters that does not match the pattern of any token.
• Any Spelling errors.
• Unmatched string
• The appearance of illegal characters.
• To replace a character with an incorrect character.
• Transposition of two characters.
Syntactic Phase Errors
parser.y
• Error in structure
• Unbalanced parenthesis
• Missing operators
Semantic Phase Errors
symantics.cpp
(See PowerPoint on Assignment 4)
• Operands of incompatible types
• Variable not declared
• The failure to match the actual argument with the formal argument

Parsing Errors

- Error recovery is possible in both top-down and bottom-up parsers
- general strategy for both bottom-up and top-down:
- look for a synchronizing token
- Yacc/bison are LALR (look-ahead left recursive) parsers
- LR parsing is more powerful than LL parsing, given the same look ahead
  - LR parsers build bottom-up toward a goal, shifting tokens onto a stack and combining ("reducing") them according to rules.
  - to construct an LR parser, it is necessary to compute an LR parser table
  - the LR parser table represents a finite automaton that walks over the parser stack

synchronizing token

- Add error rules to grammar to find a synchronizing token in a bottom-up parser.

```
returnStmt   : RETURN ';'              { $$ = newStmtNode(ReturnK, $1); }
             | RETURN exp ';‘           { $$ = newStmtNode(ReturnK, $1, $2); yyerrok; }
             | RETURN error ';‘         { $$ = NULL; yyerrok; /*printf("ERR221\n");*/ }
```

- in general, follow error with a synchronizing token. Recovery steps:
  - Pop stack (if necessary) until a state is reached in which the
  - action for the error token is shift
  - Shift the error token
  - Discard input symbols (if necessary) until a state is reached that has
  - a non-error action
  - Resume normal parsing

- Consider this grammar:

```
%token YY ZZ
%%
slist : slist stmt ';' { printf("slist stmt\n"); }
      | stmt ';'       { printf("stmt\n"); }
      | error ';'      {$$ = NULL; printDebug("ERR12);  yyerrok; }
      ;

stmt  : ZZ stmt
      | ZZ
      ;
%%
```

In your program, yyerrok is optional (you decide when), but always $$ = NULL;

In my program I call a printDebug so I can see if this error ahs been called

- When yyparse() discovers ungrammatical input, it calls yyerror(). It also sets a flag saying that it is now in an error state. yyparse() stays in this error state until it sees three consecutive tokens that make sense (that is, are not part of the error).

In this case ; indicates the old error is complete. Anything after the ; is a new error.

The Bison language includes the reserved word error, which may be included in the grammar rules.

In effect, yyerrok says, ‘‘The old error is finished. If something else goes wrong, it is to be regarded as a new error.’’.

- Consider this grammar:

```
%token YY ZZ
%%
slist : slist stmt ';' { printf("slist stmt\n"); }
      | stmt ';'       { printf("stmt\n"); }
      | error ';'      { printf("ERROR!!!\n");  yyerrok; }
      ;

stmt  : ZZ stmt
      | ZZ
      ;
%%
```

This grammar produces this state machine

```
Rules:
    0 $accept: slist $end
    1 slist: slist stmt ';'
    2      | stmt ';'
    3      | error ';'
    4 stmt: ZZ stmt
    5     | ZZ
```

State Machine

```
State 0
    0 $accept: . slist $end
    error  shift, and go to state 1
    ZZ     shift, and go to state 2
    slist  go to state 3
    stmt   go to state 4

State 1
    3 slist: error . ';'
    ';'  shift, and go to state 5

State 2
    4 stmt: ZZ . stmt
    5     | ZZ .
    ZZ  shift, and go to state 2
    $default  reduce using rule 5 (stmt)
    stmt  go to state 6

State 3
    0 $accept: slist . $end
    1 slist: slist . stmt ';'
    $end  shift, and go to state 7
    ZZ    shift, and go to state 2
    stmt  go to state 8

State 4
    2 slist: stmt . ';'
    ';'  shift, and go to state 9

State 5
    3 slist: error ';' .
    $default  reduce using rule 3 (slist)

State 6

    4 stmt: ZZ stmt .
    $default  reduce using rule 4 (stmt)

State 7

    0 $accept: slist $end .
    $default  accept

State 8
    1 slist: slist stmt . ';'
    ';'  shift, and go to state 10


State 9
    2 slist: stmt ';' .
    $default  reduce using rule 2 (slist)

State 10
    1 slist: slist stmt ';' .
    $default  reduce using rule 1 (slist)
```

Jump to new State

New state to handle error

New state to handle error

Note that invalid token will do a reduce

```
Rules:
    0 $accept: slist $end
    1 slist: slist stmt ';'
    2      | stmt ';'
    3      | error ';'
    4 stmt: ZZ stmt
    5     | ZZ
```

- Consider the input zz zz yy zz zz ;
  - which has an error in the middle.
- We expect Bison to:
  - shift and reduce the initial zz's and then arrive and the bad token yy.
  - Then put on an error token until we get to the ;.
- It effectively does that. Note: each version of Bison seems to generate different debug output but the actions are the same.