CS 314 Principles of Programming Languages

Lecture 10: Syntax Directed Translation

Prof. Zheng Zhang

Rutgers University

October 5, 2018

Class Information

- Homework 3 is being graded.
- Homework 4 will be released by the end of today.
- Project 1 will be released after hw4 is due (Tuesday 10/9/2018).

Review: Recursive Descent Parsing

Recursive descent parser for LL(1)

- Each **non-terminal** has an associated parsing procedure that can recognize any sequence of tokens generated by that **non-terminal**
- There is a main routine to initialize all globals (e.g:the ***token*** variable in previous code example) and call the start symbol. On return, check whether ***token*** == EOF, and whether errors occurred.
- Within a parsing procedure, both **non-terminals** and **terminals** are matched:
  - ➡ Non-terminal A: call procedure for A
  - ➡ Token t: compare t with current the first of the remaining tokens; If matched, **consume input**, otherwise, ERROR
- Parsing procedure may contain code that performs some useful "computations" (*syntax directed translation)*

Example

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

```
void expr( ) {
   switch token {
      case +:    token := next_token( );
                 expr( ) ;
                 expr( );
                 break;
      case 0..9:
                 digit( ); break;
      …
   } // End switch case
} //End expr()

void digit( ):  // return value of constant
   switch token {
      case 1: token := next_token( ); break;
      case 2: token := next_token( ); break;
      …
   } // End switch case
}// End digit( )
```

Rule 1:
< expr > ::= + < expr > < expr >

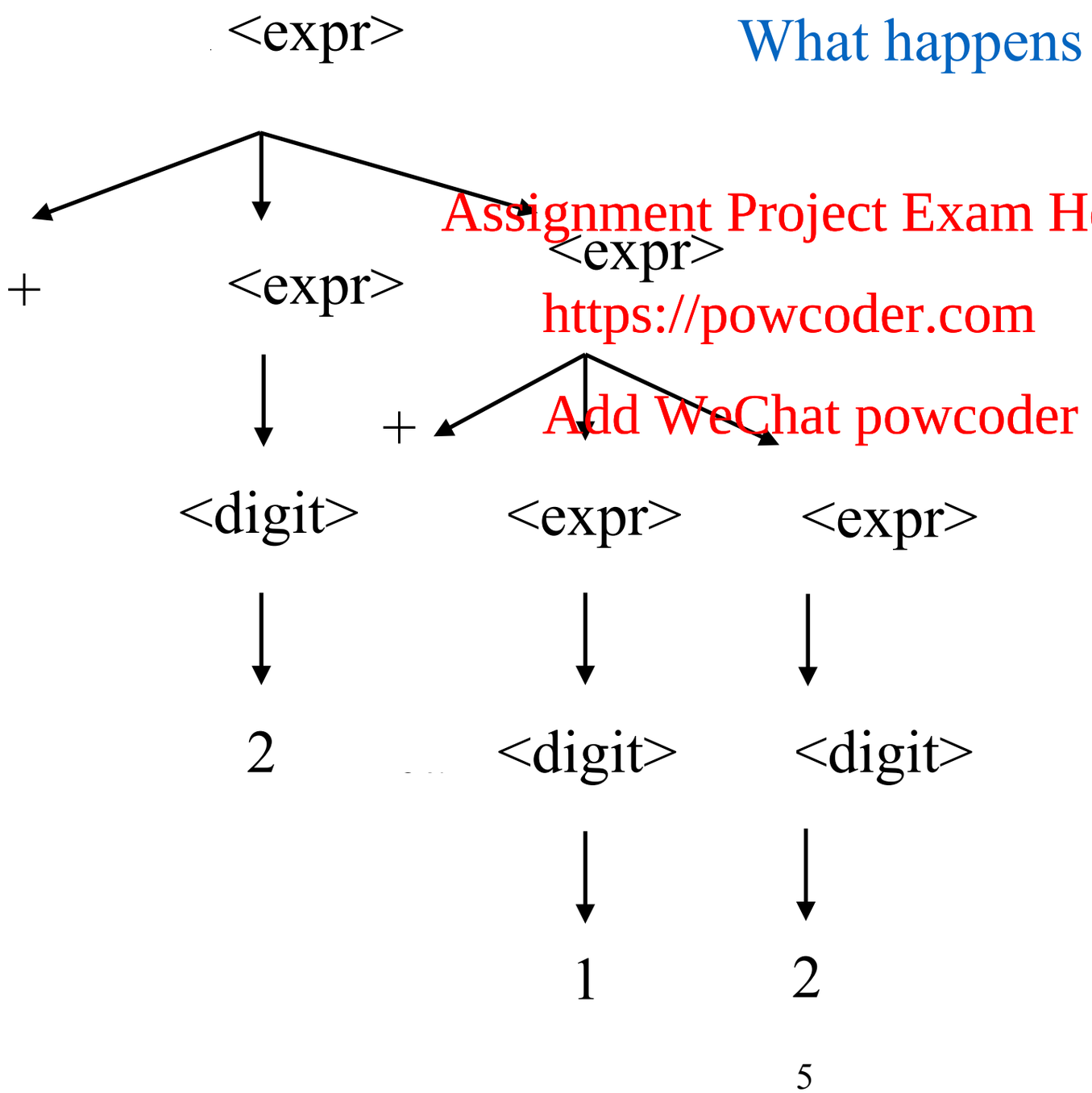
Rule 2:
< expr > ::= <digit>

Rule 3
< digit > ::= 0 | 1 | 2 | 3 | … | 9

Example: the Original Parser

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	rule 1	rule 2	error
< digit >	error	rule 3	error



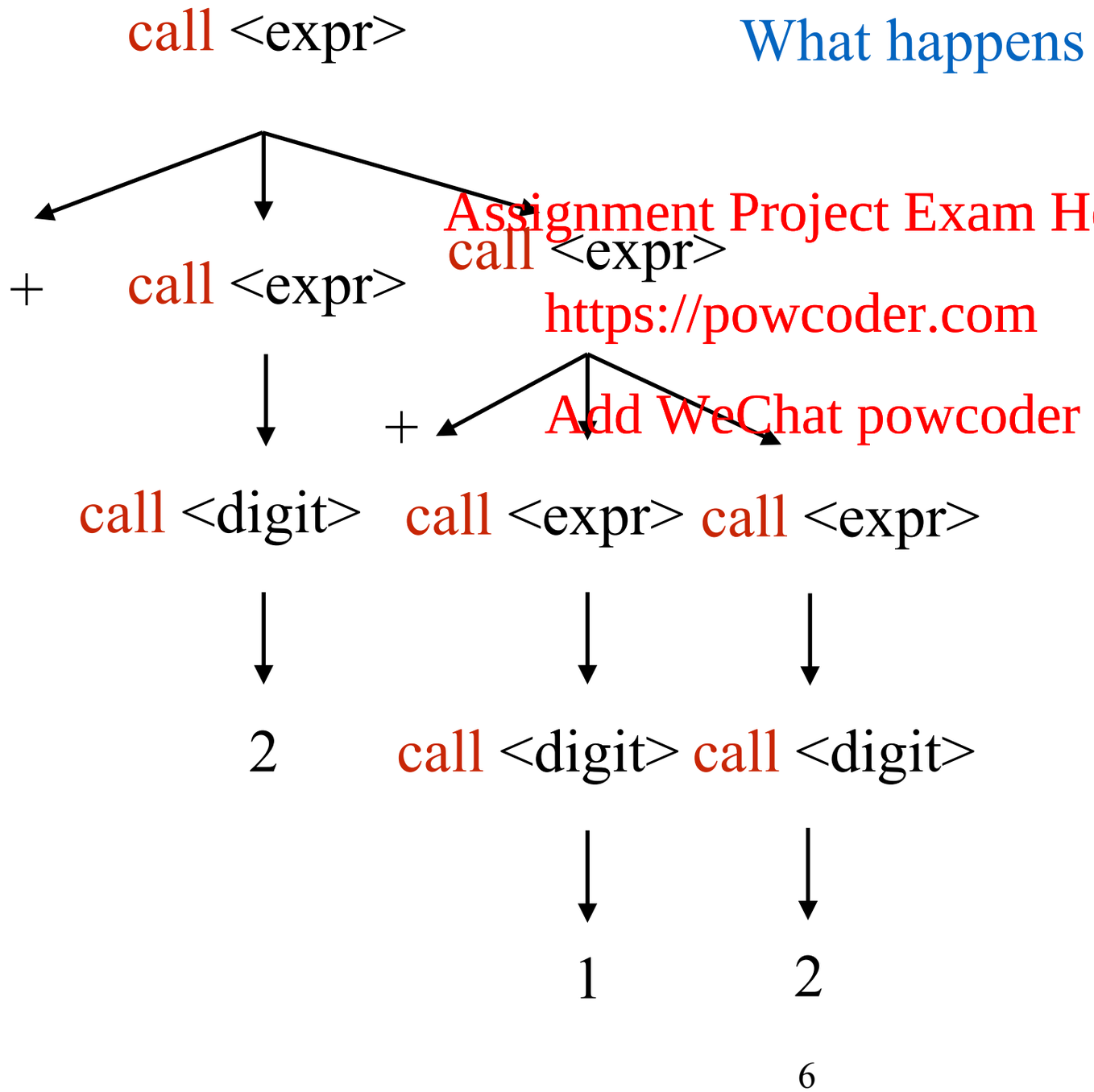
What happens when you parse expression
" + **2** + **1 2**"

Example: the Original Parser

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	rule 1	rule 2	error
< digit >	error	rule 3	error

What happens when you parse expression
" + **2** + **1 2**"

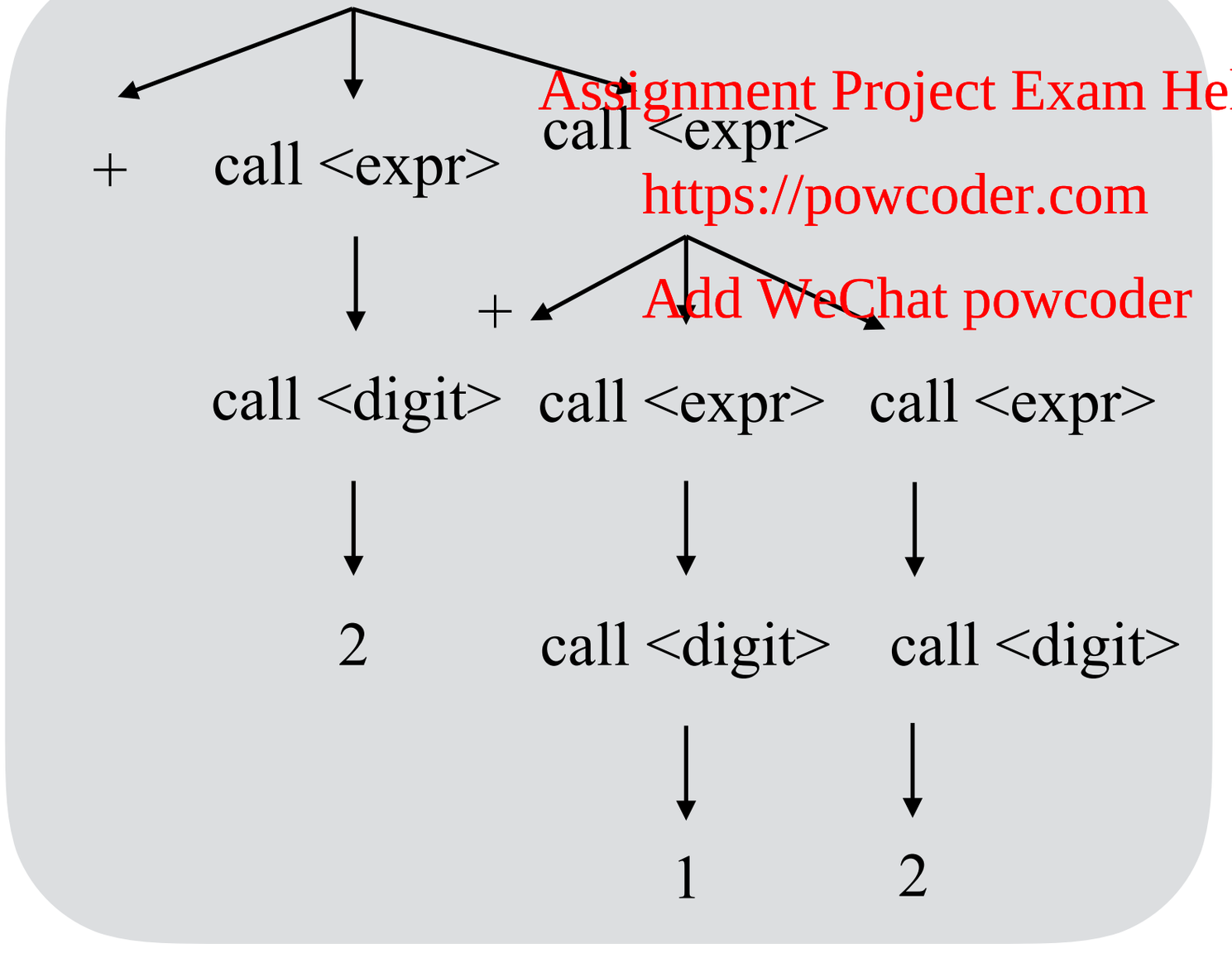


Example: the Original Parser

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	rule 1	rule 2	error
< digit >	error	rule 3	error

call <expr>

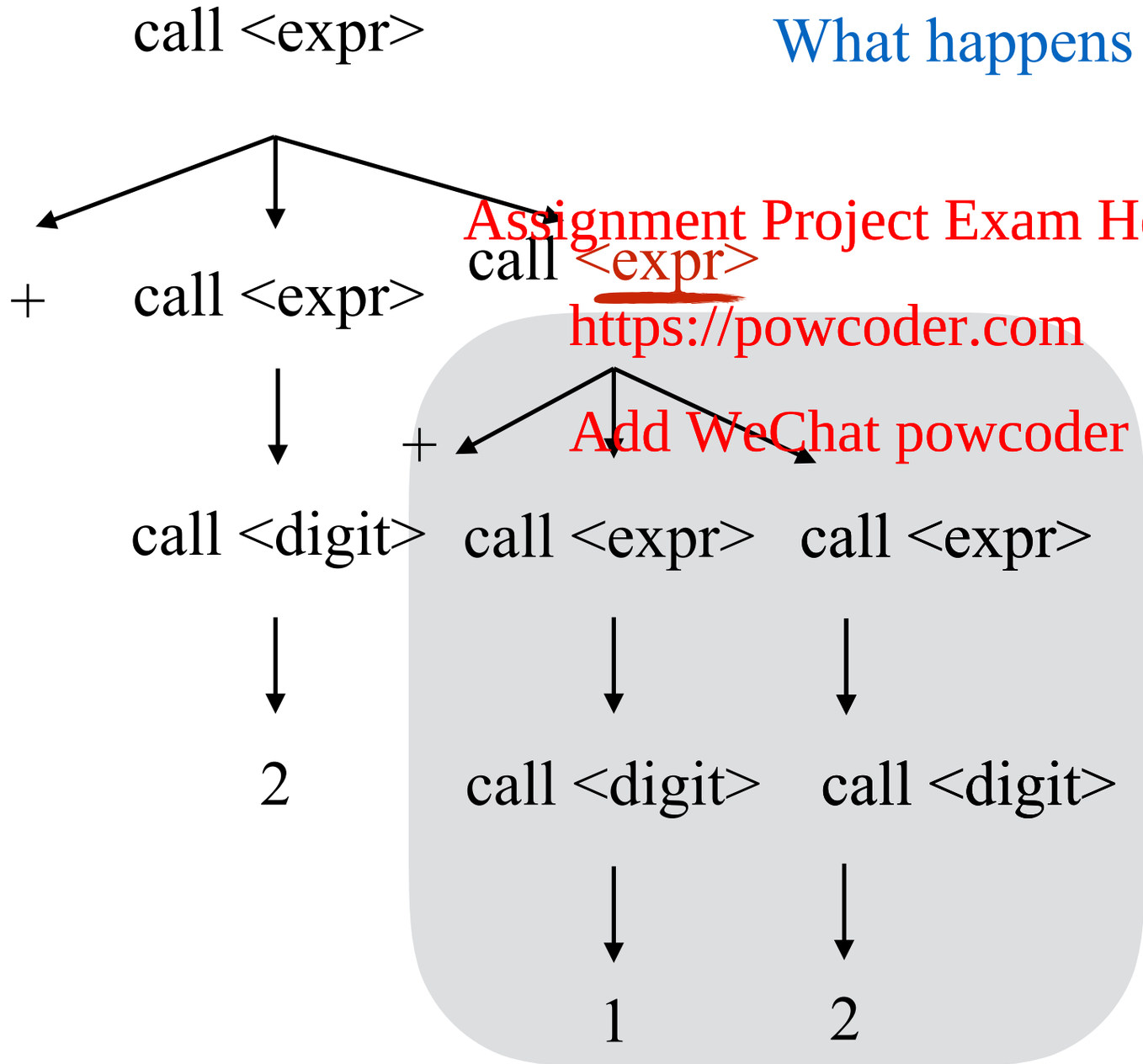


What happens when you parse expression
“ + **2** + **1** **2**”

Example: the Original Parser

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	rule 1	rule 2	error
< digit >	error	rule 3	error

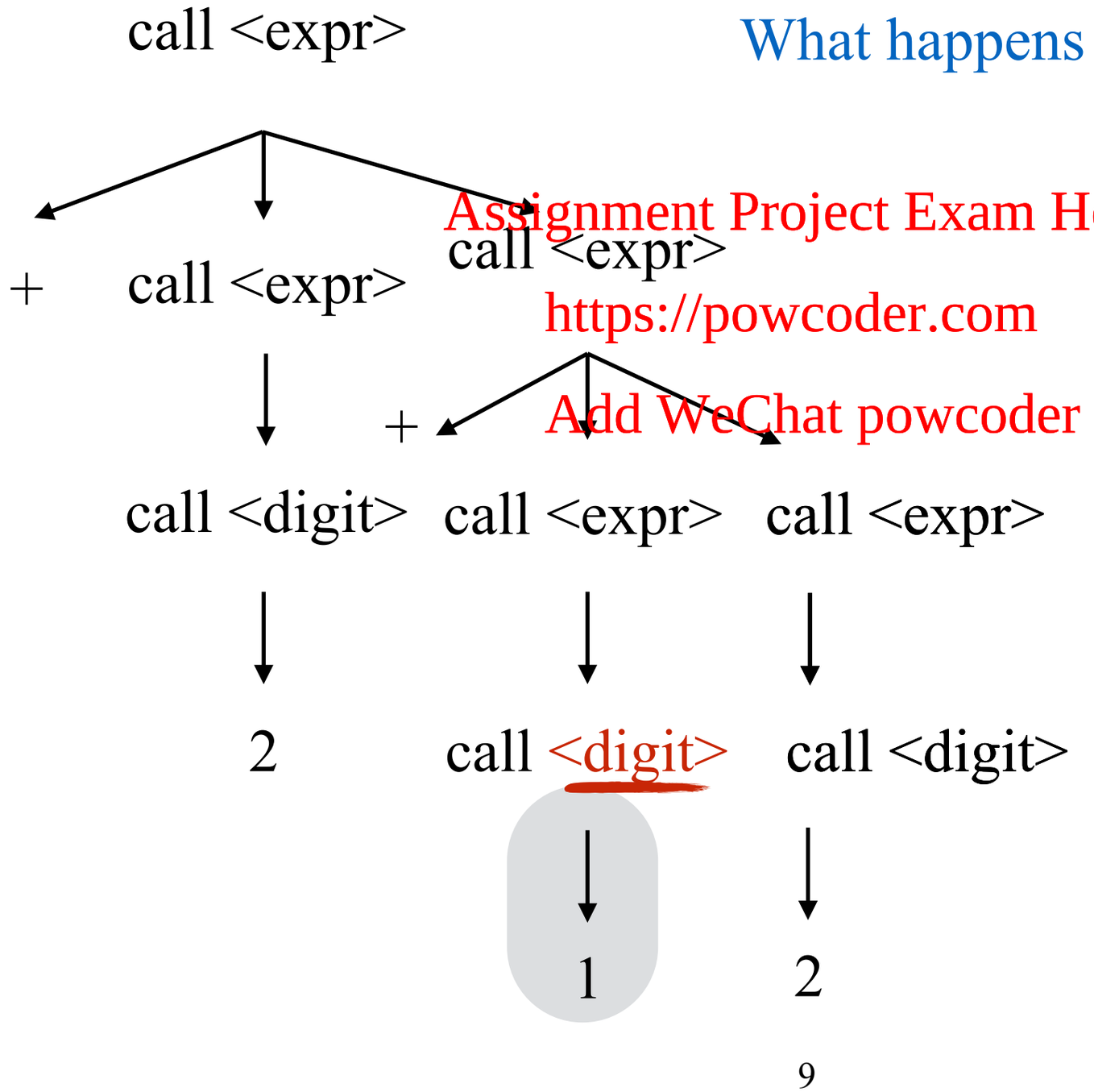


What happens when you parse expression
“ + **2** + **1** **2**”

Example: the Original Parser

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	rule 1	rule 2	error
< digit >	error	rule 3	error



What happens when you parse expression
" + **2** + **1 2**"

Review: Recursive Descent Parsing

Recursive descent parser for LL(1)

- Each **non-terminal** has an associated parsing procedure that can recognize any sequence of tokens generated by that **non-terminal**
- There is a main routine to initialize all globals (e.g:the ***token*** variable in previous code example) and call the start symbol. On return, check whether ***token*** == EOF, and whether errors occurred.
- Within a parsing procedure, both **non-terminals** and **terminals** are matched:
  - ➡ Non-terminal A: call procedure for A
  - ➡ Token t: compare t with current the first of the remaining tokens; If matched, **consume input**, otherwise, ERROR
- Parsing procedure may contain code that performs some useful "computations" (*syntax directed translation)*

Syntax Directed Translation

Examples:

- Interpreter
- Code generator
- Type checker
- Performance estimator

Use hand-written recursive descent LL(1) parser

Example: Interpreter

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

Original

```
void expr( ) {

  switch token {
    case +:   token := next_token( );
              expr( ) ;
              expr( );
              break;
    case 0..9:
              digit( ); break;
     …
  } // End switch case
} //End expr()

void digit( ):  // return value of constant
  switch token {
    case 1: token := next_token( ); break;
    case 2: token := next_token( ); break;
     …
  } // End switch case
}// End digit( )
```

Example: Interpreter

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

Interpreter

```
int expr( ) {
   int val1, val2; // two values
   switch token {
      case +:    token := next_token( );
                 val1 =  expr( );
                 val2 = expr( );
                 return val1 + val2;
      case 0..9:
                 return digit( );
      …
   } // End switch case
} //End expr()

int digit( ):  // return value of constant
   switch token {
      case 1: token := next_token( ); return 1;
      case 2: token := next_token( ); return 2;
      …
   } // End switch case
}// End digit( )
```

Original

```
void expr( ) {

   switch token {
      case +:    token := next_token( );
                 expr( ) ;
                 expr( );
                 break;
      case 0..9:
                 digit( ); break;
      …
   } // End switch case
} //End expr()

void digit( ):  // return value of constant
   switch token {
      case 1: token := next_token( ); break;
      case 2: token := next_token( ); break;
      …
   } // End switch case
}// End digit( )
```

Example: Interpreter

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

Interpreter

```
int expr( ) {
   int val1, val2; // two values
   switch token {
      case +:    token := next_token( );
                 val1 =  expr( );
                 val2 = expr( );
                 return val1 + val2;
      case 0..9:
                 return digit( );
      …
   } // End switch case
} //End expr()

int digit( ):  // return value of constant
   switch token {
      case 1: token := next_token( ); return 1;
      case 2: token := next_token( ); return 2;
      …
   } // End switch case
}// End digit( )
```

Each <expr> that used rule 1 returns the sum of its <expr>s.

Each <expr> that used rule 2 returns the <digit>'s value.

Each <digit> returns its value.

Example: Interpreter

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

Interpreter

```
int expr( ) {
   int val1, val2; // two values
   switch token {
      case +:    token := next_token( );
                 val1 =  expr( );
                 val2 = expr( );
                 return val1 + val2;
      case 0..9:
                 return digit( );
      …
   } // End switch case
} //End expr()

int digit( ):  // return value of constant
   switch token {
      case 1: token := next_token( ); return 1;
      case 2: token := next_token( ); return 2;
      …
   } // End switch case
}// End digit( )
```


What happens when parsing expression
“ + 2 + 1 2”

The parsing produces
?

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error

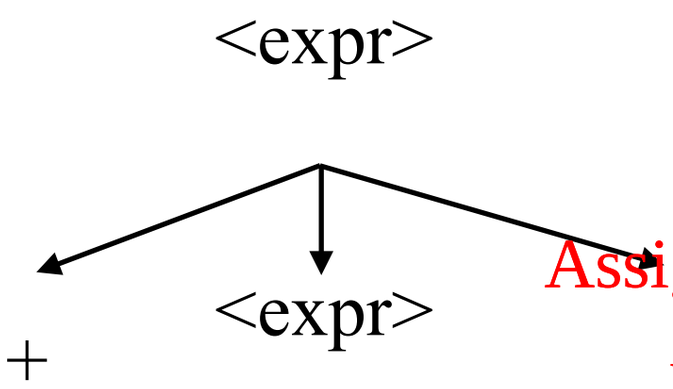
<expr>

What happens when you parse expression
“ + **2** + **1 2**”

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error

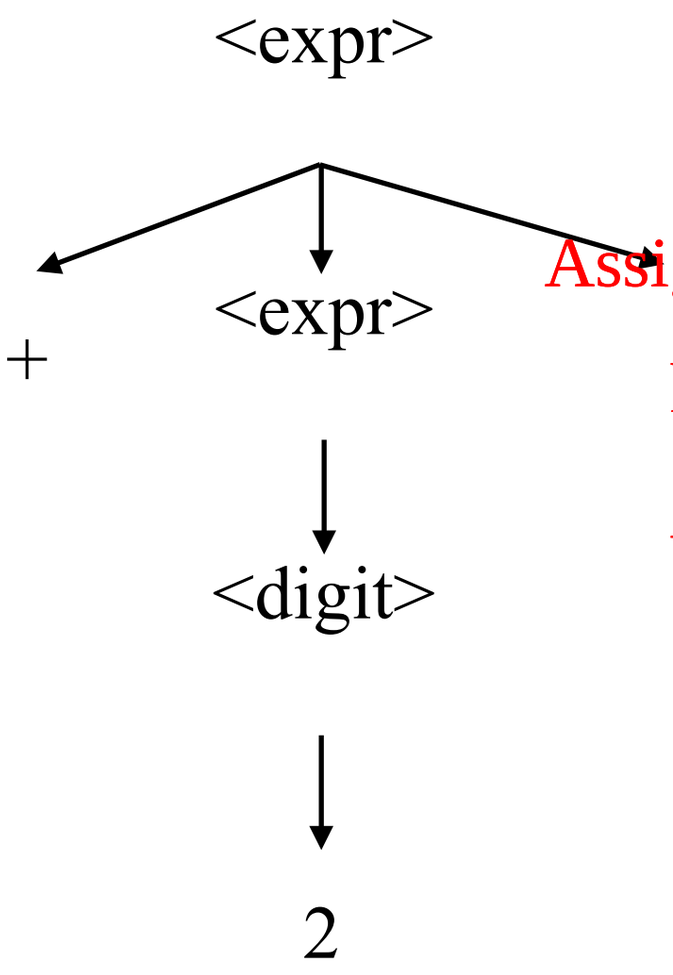


What happens when you parse expression
" + **2** + **1 2**"

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error

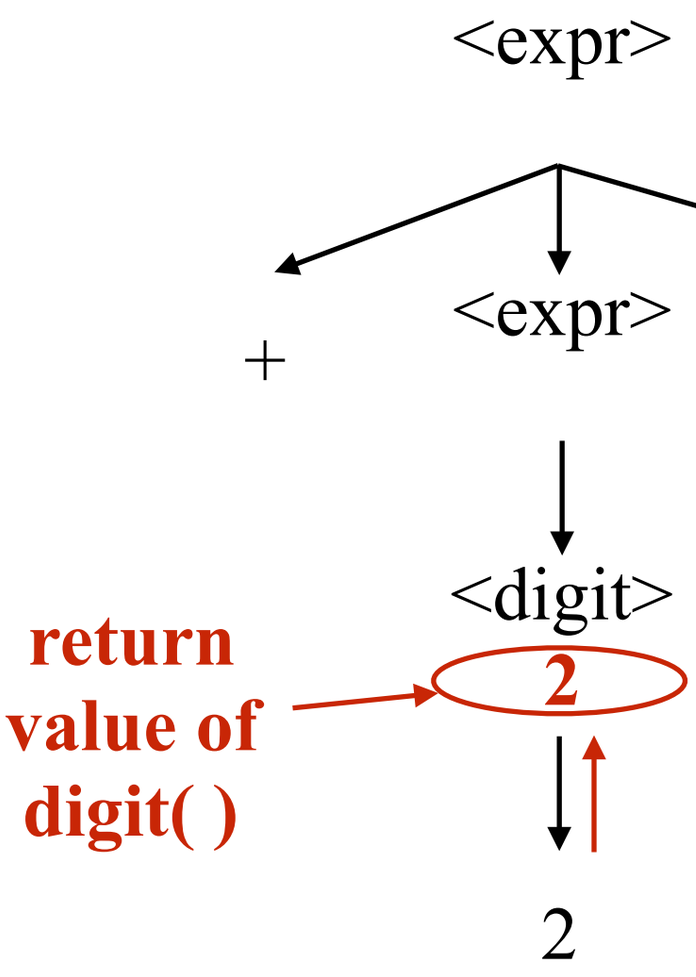


What happens when you parse expression
" + **2** + **1 2**"

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



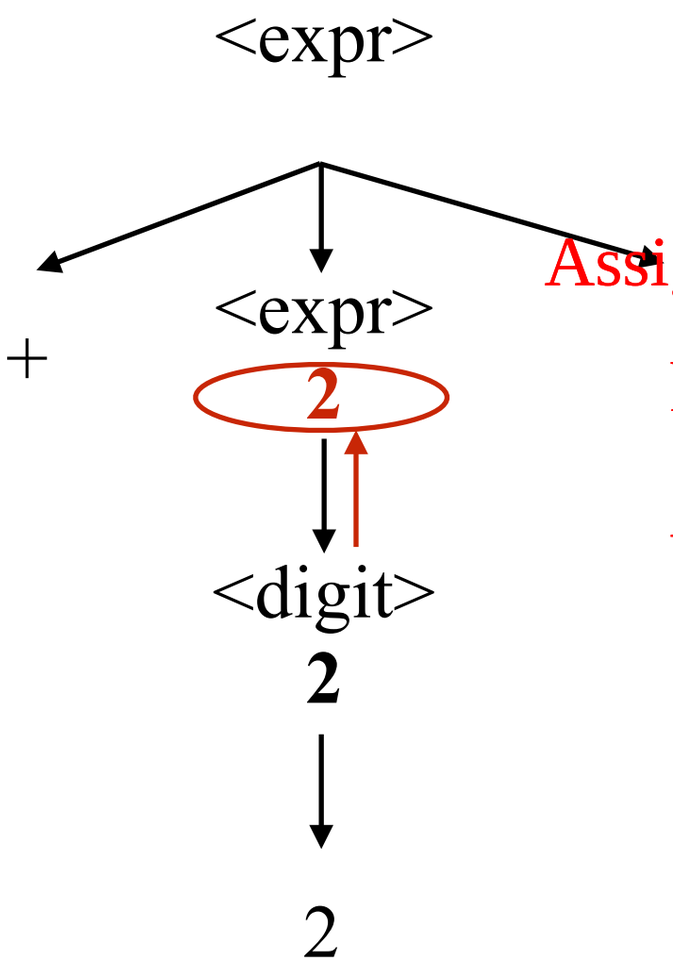
What happens when you parse expression
" + **2** + **1 2**"

The <digit> returns its value.

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



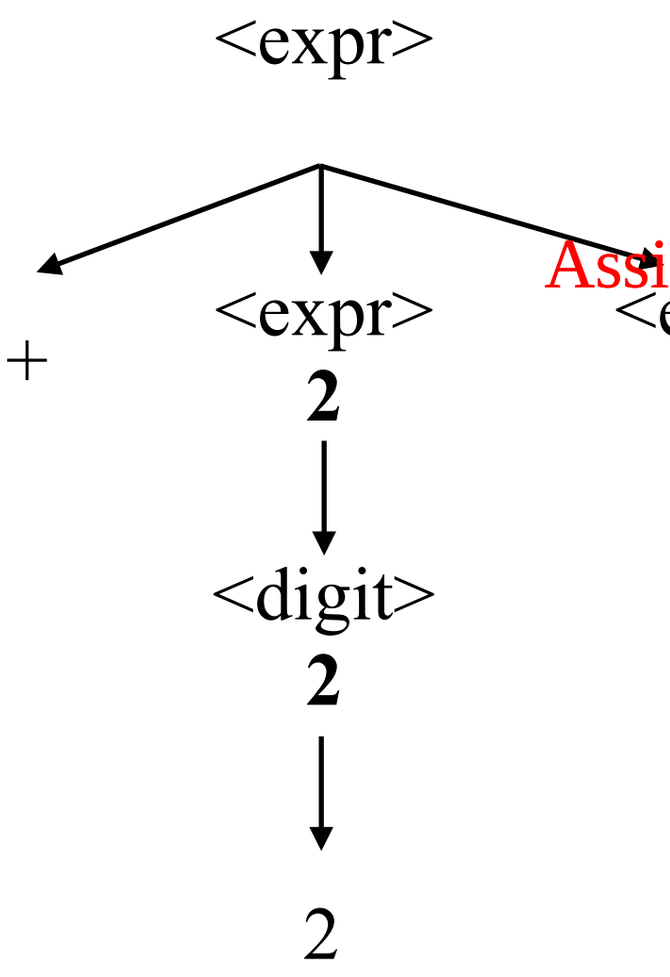
What happens when you parse expression
" + **2** + **1** **2**"

The <expr> that used rule 2 returns the <digit>'s value.

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error

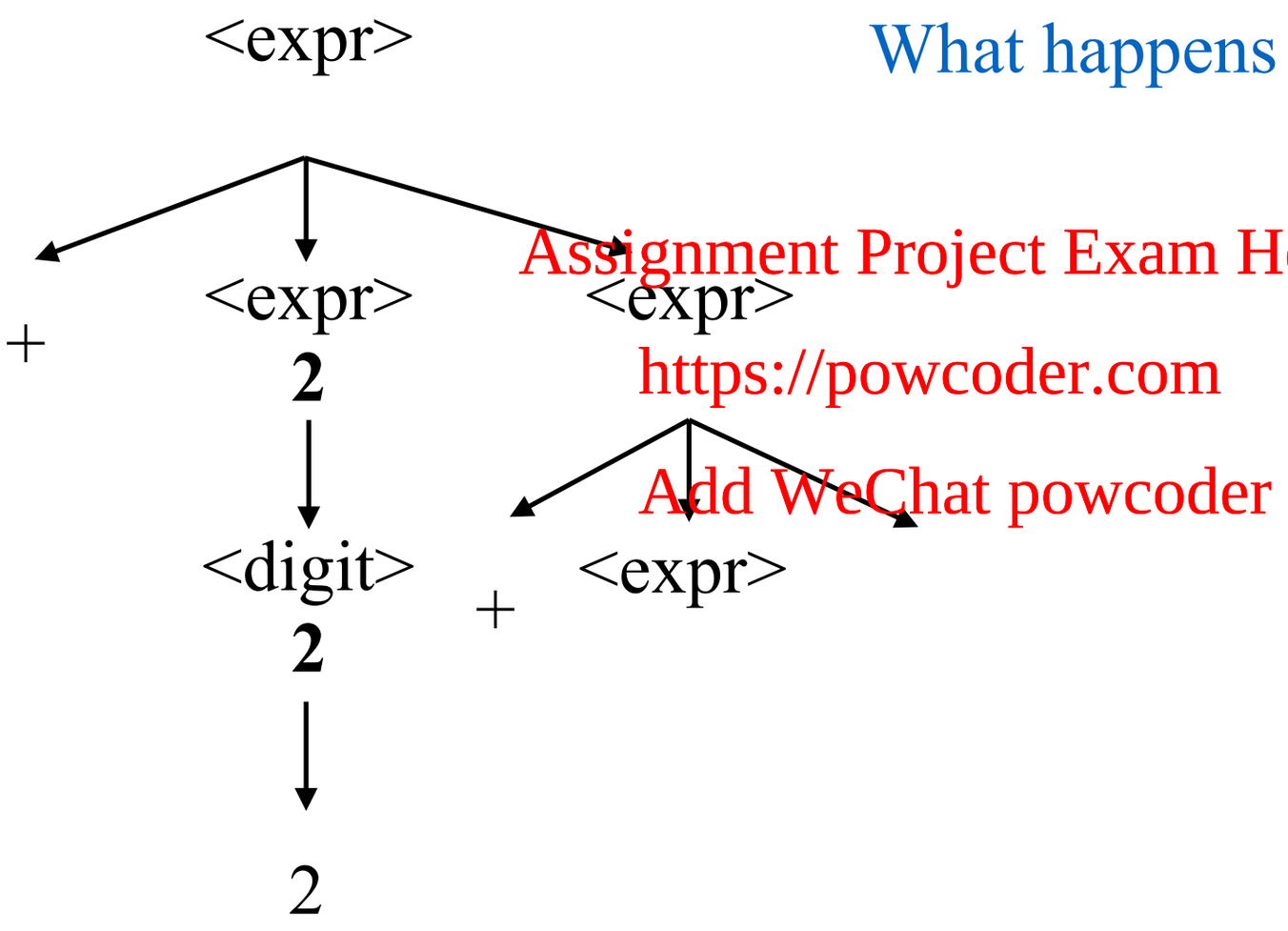


What happens when you parse expression
“ + **2** + **1** **2**”

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error

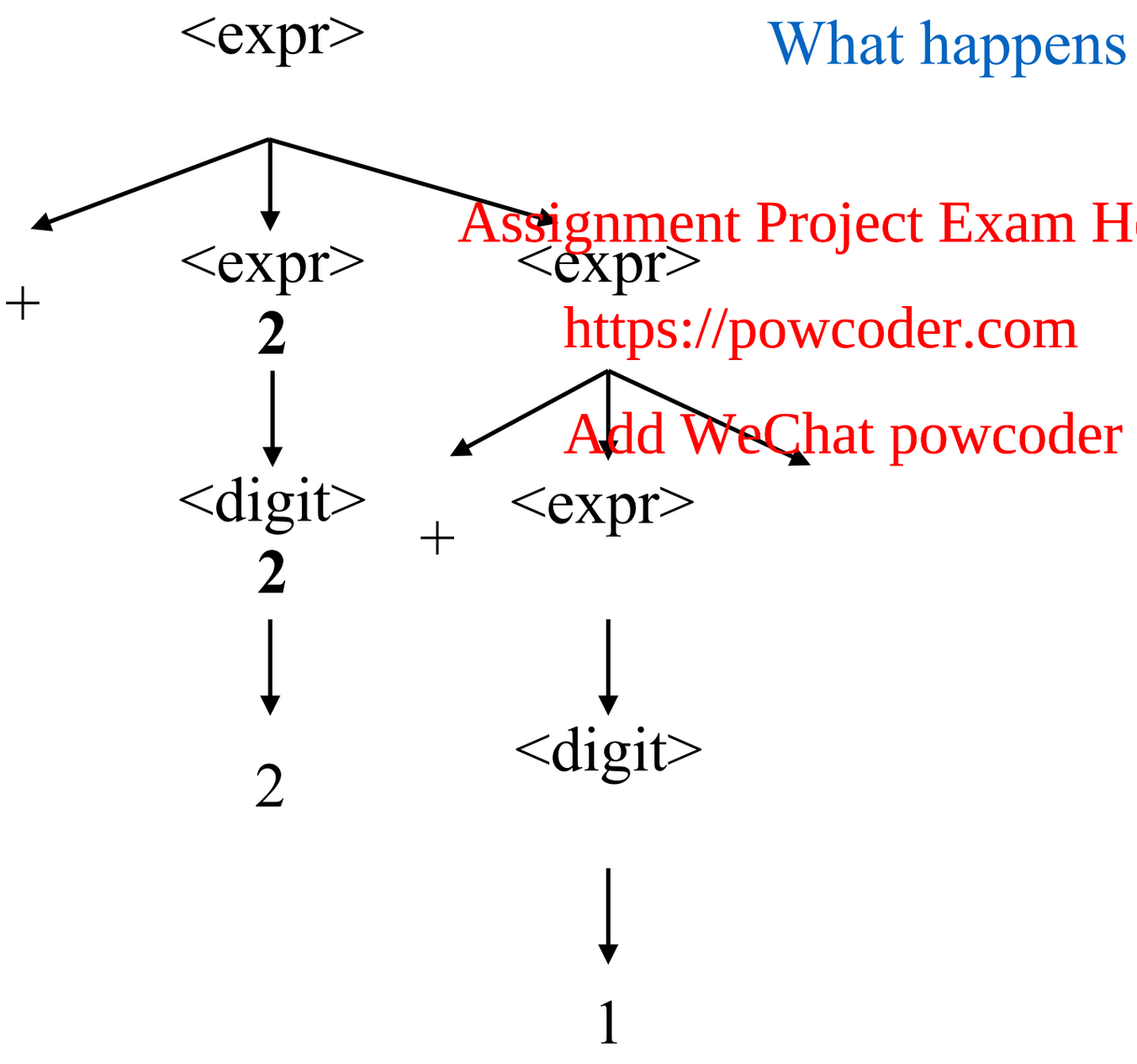


What happens when you parse expression
" + **2** + **1** **2**"

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error

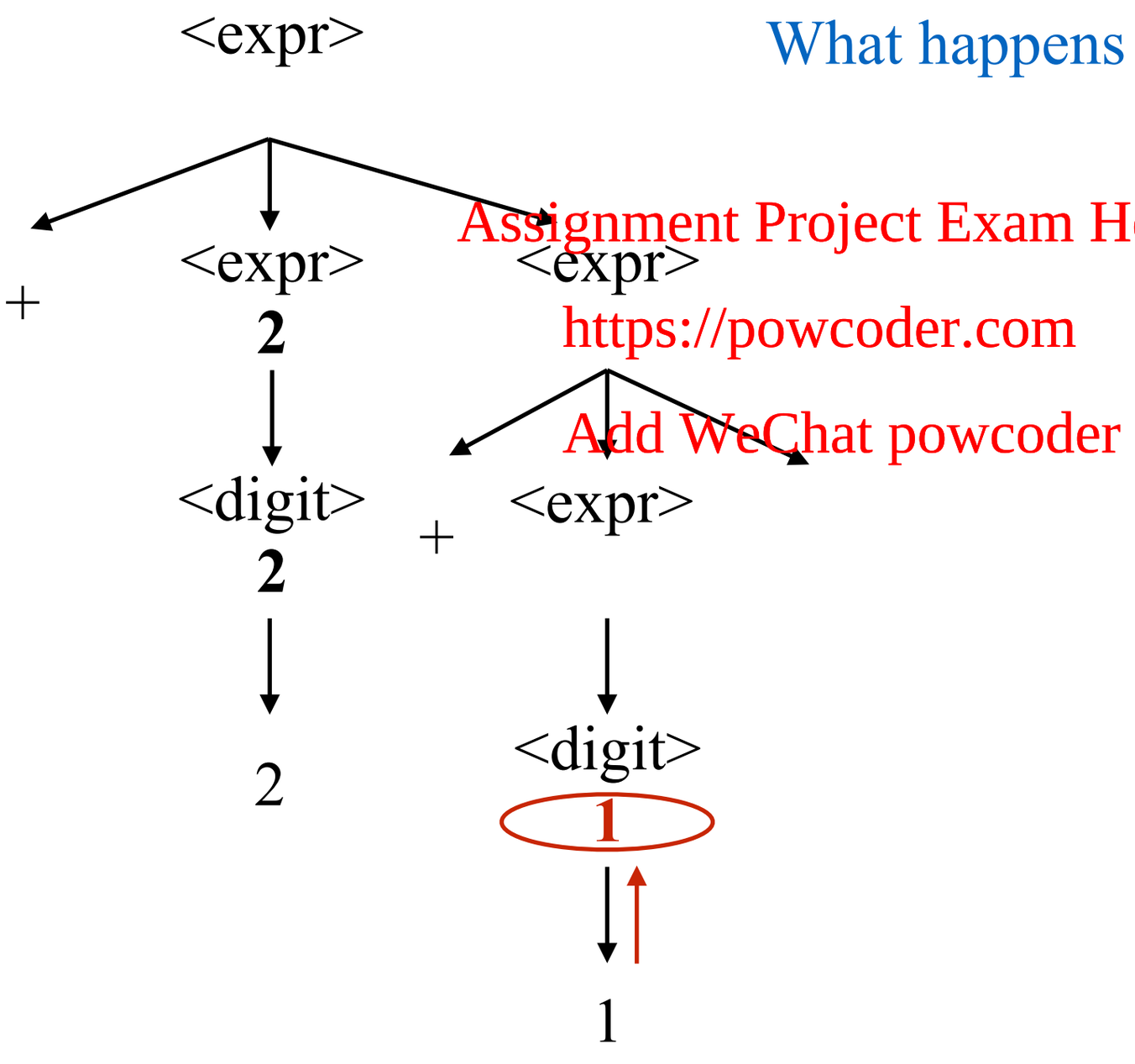


What happens when you parse expression
" + **2** + **1** **2**"

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error

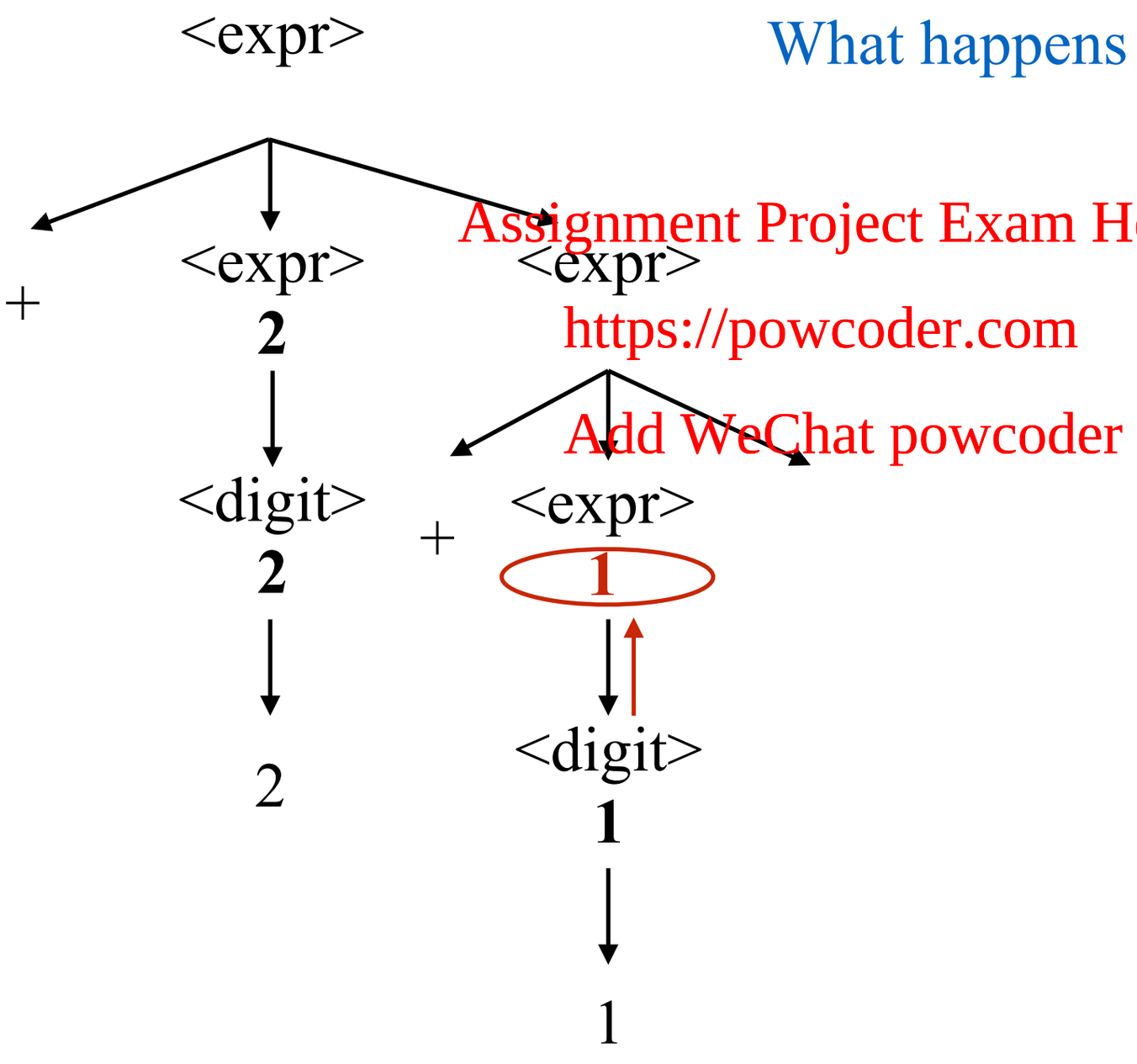


What happens when you parse expression
" + **2** + **1** **2**"

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error

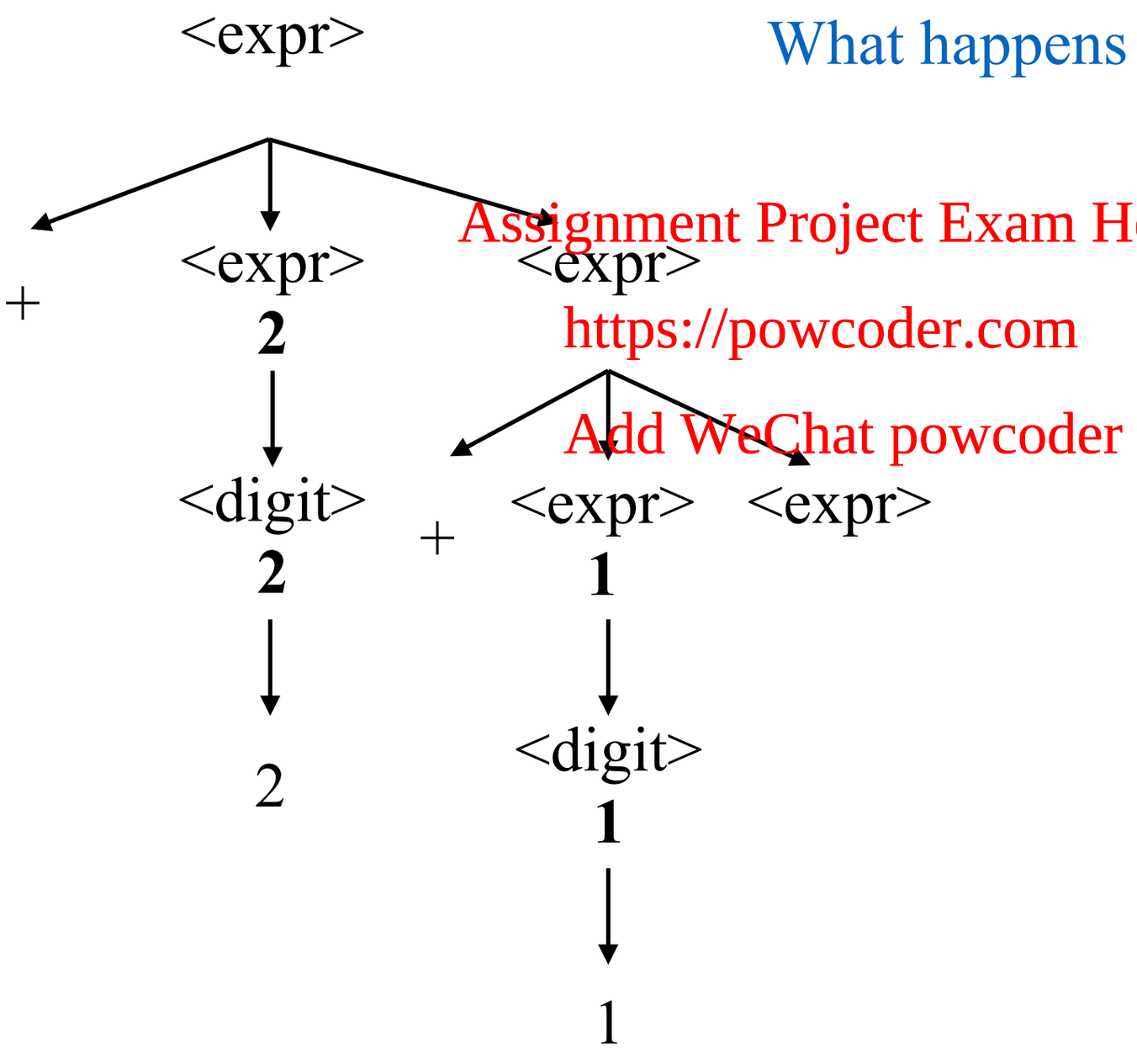


What happens when you parse expression
" + **2** + **1** **2**"

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error

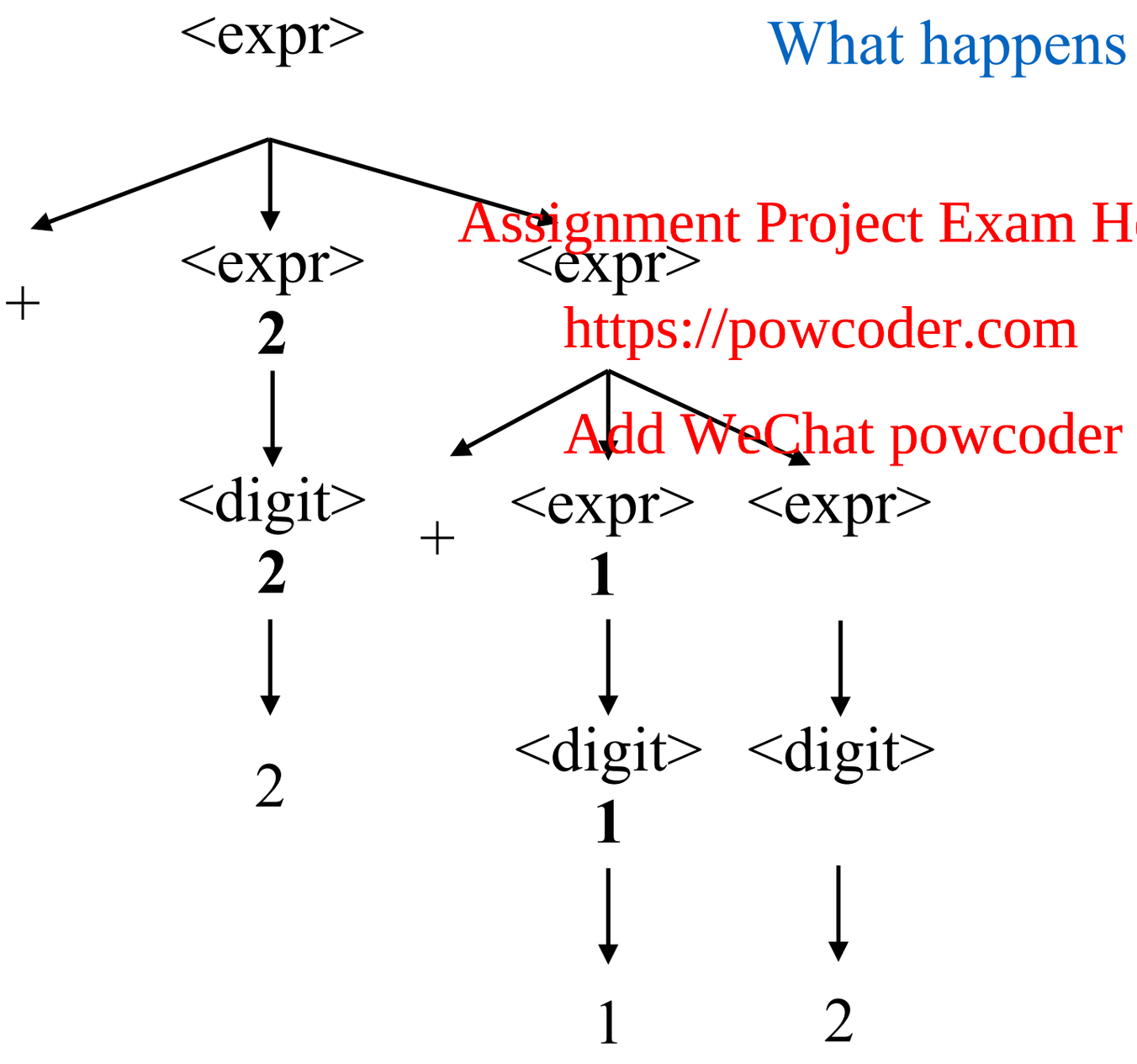


What happens when you parse expression
" + **2** + **1** **2**"

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error

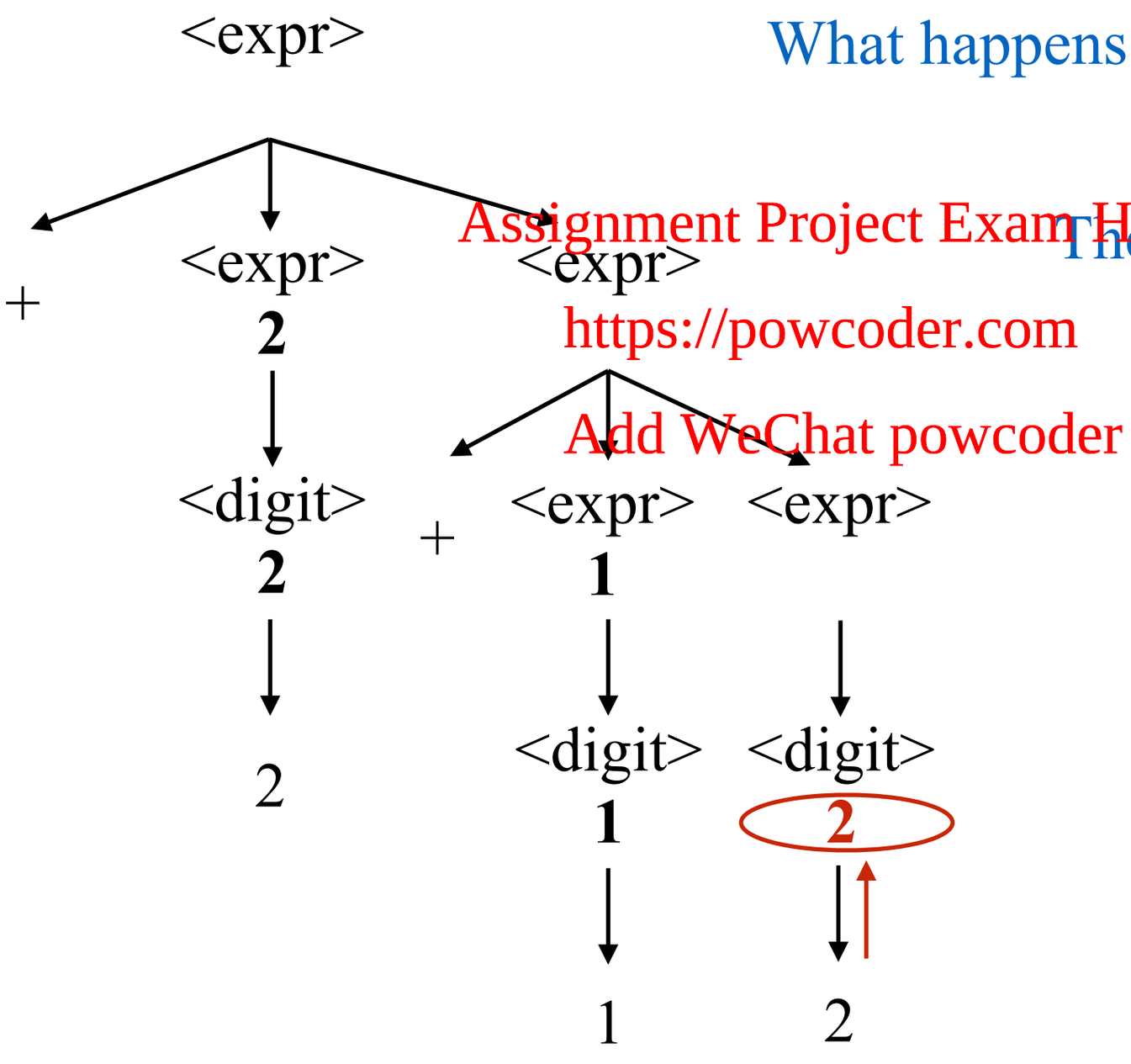


What happens when you parse expression
“ + **2** + **1** **2**”

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



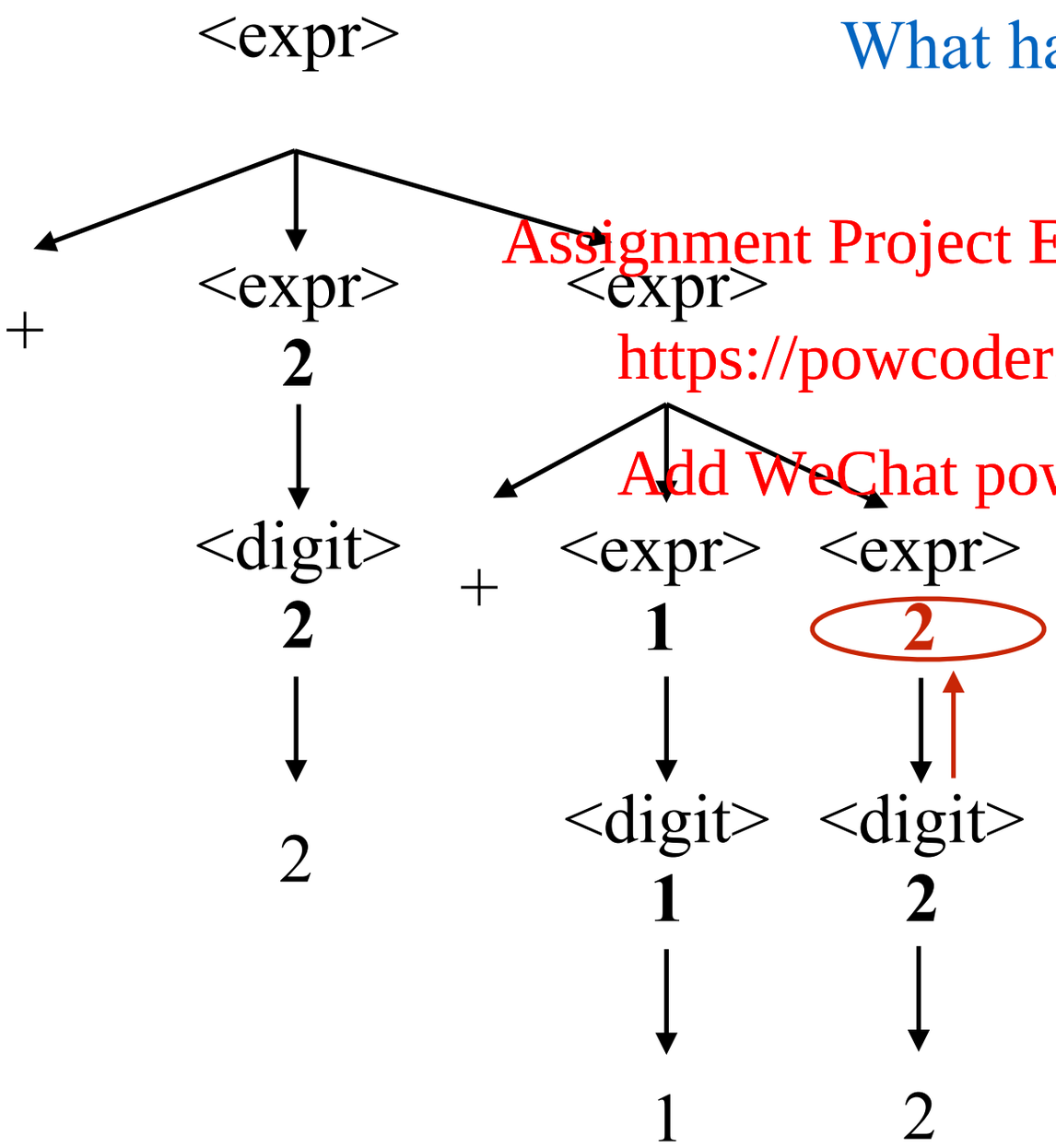
What happens when you parse expression
" + **2** + **1** **2**"

The <digit> returns its value.

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



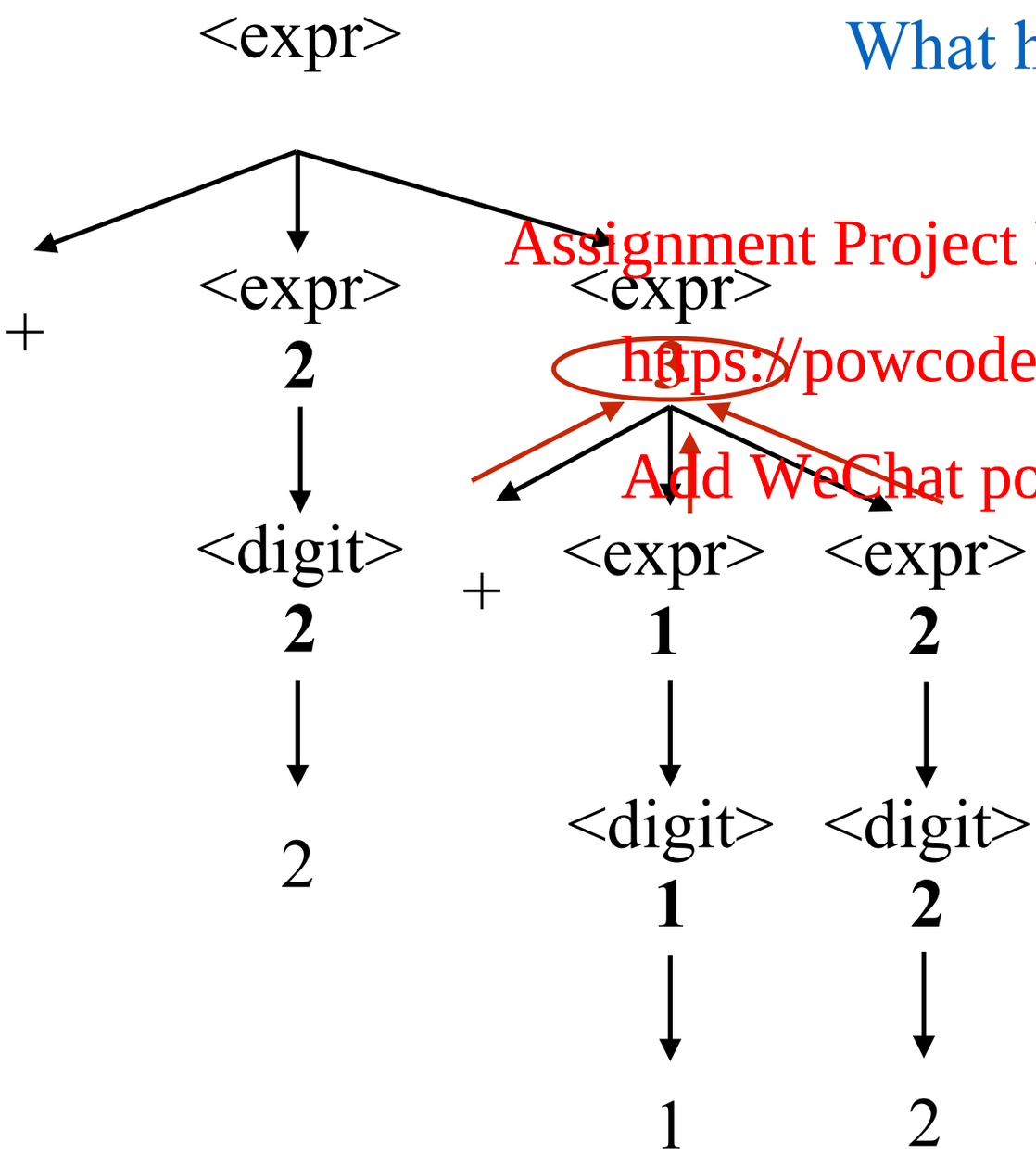
What happens when you parse expression
" + **2** + **1** **2**"

The <expr> that used rule 2 returns the <digit>'s value.

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



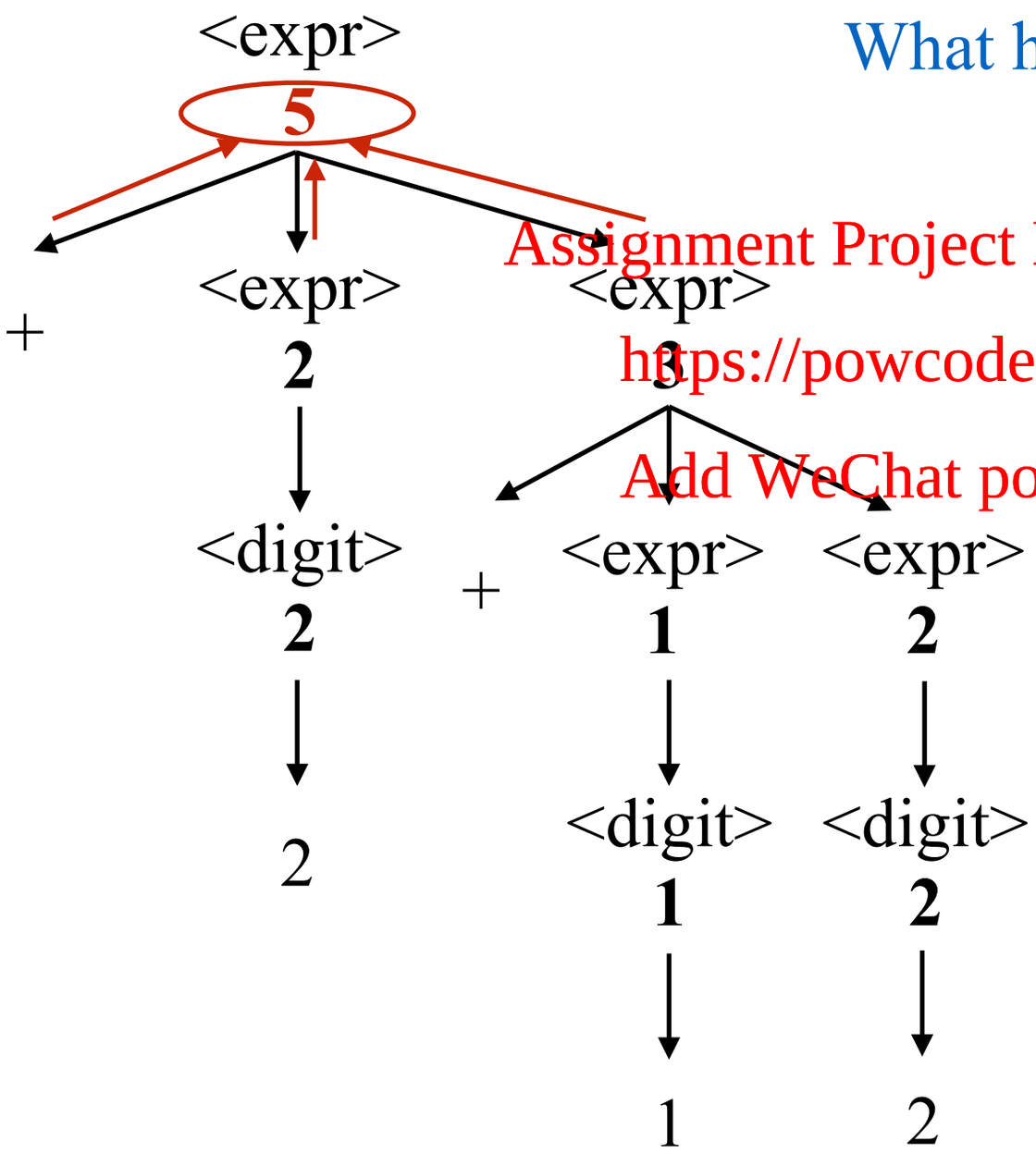
What happens when you parse expression
" + **2** + **1** **2**"

The <expr> that used rule 1 returns the sum of its <expr>s.

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



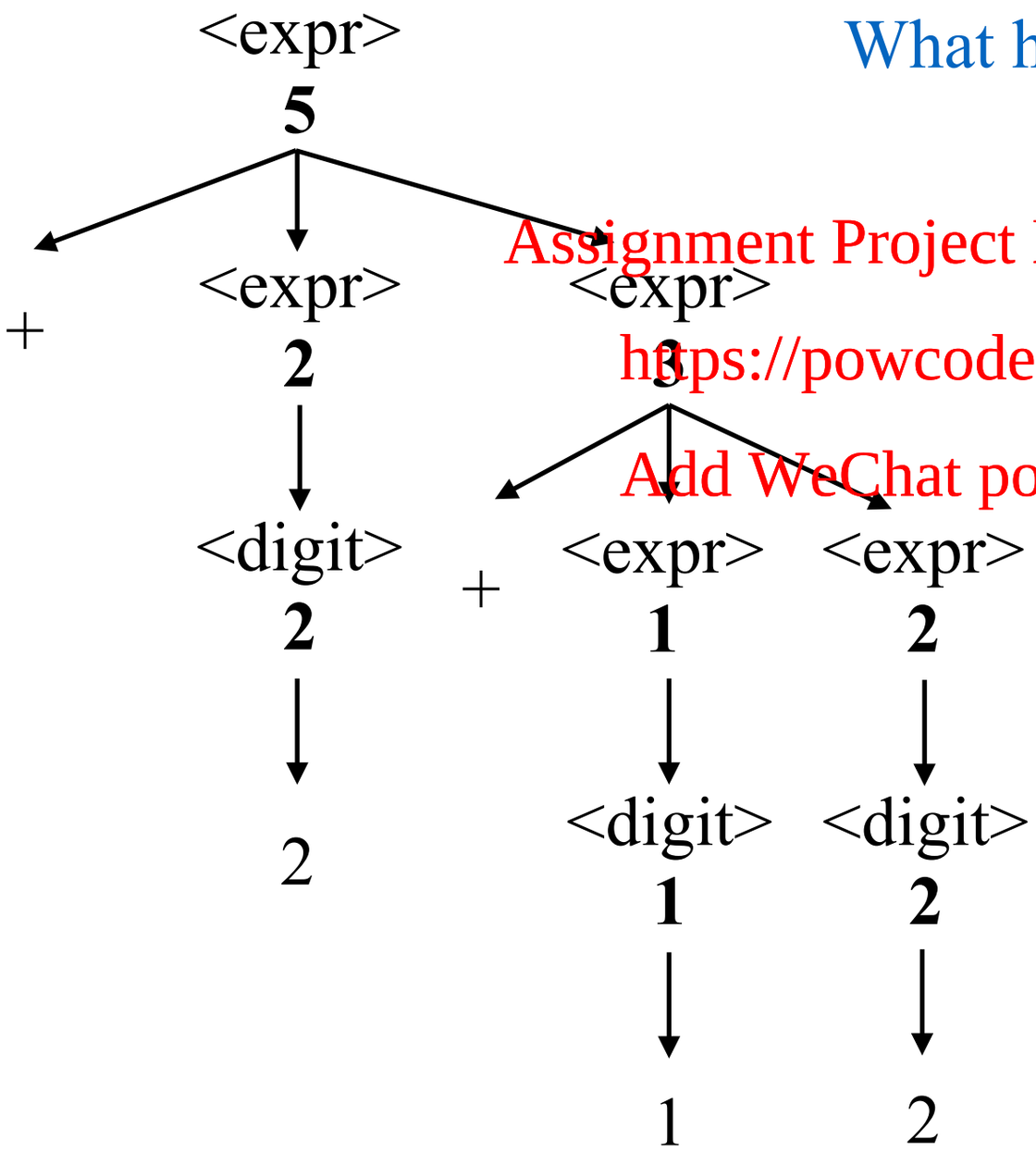
What happens when you parse expression
" + **2** + **1** **2**"

The <expr> that used rule 1 returns the sum of its <expr>s.

Example: Interpreter (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



What happens when you parse expression
" + **2** + **1** **2**"

Each <digit> returns its value.

Each <expr> that used rule 2 returns the <digit>'s value.

Each <expr> that used rule 1 returns the sum of its <expr>s.

The parsing returns:
5

Example: Type Checker

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

```
string expr( ) { // returns type expression
  string  type1, type2; // other type expressions
  switch token {
      case +:
        token := next_token( );
        type1 = expr( );
        type2 = expr( );
        if (type1 == "int" && type2 == "int"){
          return "int";
        } else{
          return "error";
        }
      case 0..9:
        return digit ( );
        …
  }
}
```

```
string digit( ) {  // returns type expression
  switch token {
    case 1:    token := next_token( );
               return "int";
    case 2:    token := next_token( );
               return "int";
    …
  }
}
```

Example: Type Checker (cont.)

What happens when you parse subprogram
“ + **2** + **1 2**” ?

The parsing produces:

“int”

Example: Code Generator (cont.)

What happens when you parse subprogram
“ + **2** + **1 2**” ?

The parsing produces:


```
LOADI  2 => r2
LOADI  1 => r4
LOADI  2 => r5
ADD  r4, r5 => r3
ADD  r2, r3 => r1
```

Example: Simple Code Generator

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

Code Generator

```
int expr( ) {
   int target_reg; // target register
   int reg1, reg2; // source registers
   switch token {
      case +:
         token := next_token( );
         target_reg = next_register( );
         reg1 =  expr( ) ;
         reg2 = expr( );
         print_inst(ADD, reg1, reg2, target_reg);
         return target_reg;
      case 0..9:
         return digit( );
      …
   } // End switch case
} //End expr()
```

```
int digit( ):  // return value of constant
   int target_reg;  // target register
   switch token {
      case 1:
         token := next_token( );
         target_reg = next_register( );
         print_inst(LOADI, 1, target_reg);
         return target_reg;
      case 2:
         ….
   } // End switch case
}// End digit( )
```


First call to next_register() will return 1

Example: Simple Code Generator

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

Code Generator

```
int expr( ) {
  int target_reg; // target register
  int reg1, reg2; // source registers
  switch token {
    case +:
      token := next_token( );
      target_reg = next_register( );
      reg1 =  expr( ) ;
      reg2 = expr( );
      print_inst(ADD, reg1, reg2, target_reg);
      return target_reg;
    case 0..9:
      return digit( );
    …
  } // End switch case
} //End expr()
```

```
int digit( ):  // return value of constant
  int target_reg;  // target register
  switch token {
    case 1:
      token := next_token( );
      target_reg = next_register( );
      print_inst(LOADI, 1, target_reg);
      return target_reg;
    case 2:
      ….
  } // End switch case
}// End digit( )
```

"ADD r<reg1>, r<reg2> => r<target_reg>"

Example: Simple Code Generator

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

Code Generator

```
int expr( ) {
  int target_reg; // target register
  int reg1, reg2; // source registers
  switch token {
    case +:
      token := next_token( );
      target_reg = next_register( );
      reg1 =  expr( ) ;
      reg2 = expr( );
      print_inst(ADD, reg1, reg2, target_reg);
      return target_reg;
    case 0..9:
      return digit( );
    …
  } // End switch case
} //End expr()
```

```
int digit( ):  // return value of constant
  int target_reg;  // target register
  switch token {
    case 1:
      token := next_token( );
      target_reg = next_register( );
      print_inst(LOADI, 1, target_reg);
      return target_reg;
    case 2:
      ….
  } // End switch case
}// End digit( )
```

“ LOADI 1 => r<target_reg>”

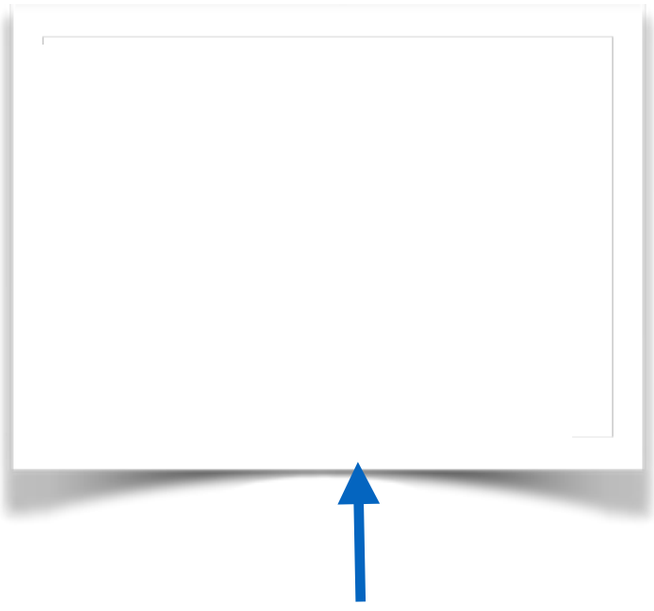
Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error

<expr>
r1

Register r1 is selected as target register.
We recurse into child <expr>.



generated code
(in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error

<expr>
r1

+ <expr>

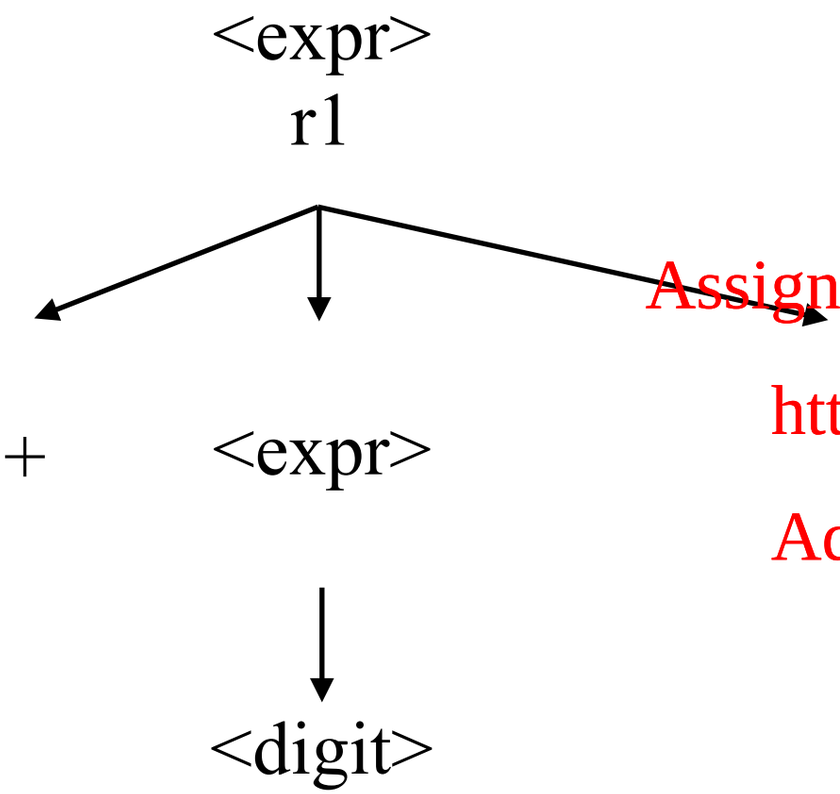
We call the <digit> function.

generated code
(in progress)

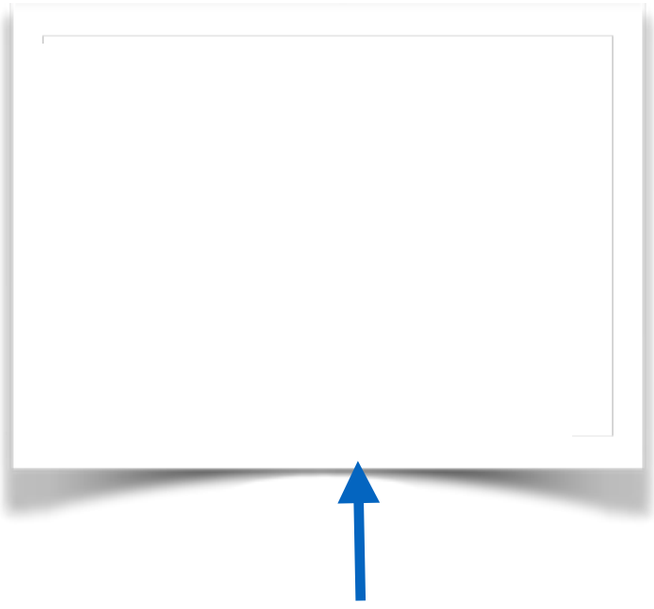
Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



Register r2 is selected as target register.
We generate code to load digit into r2.
We return r2.

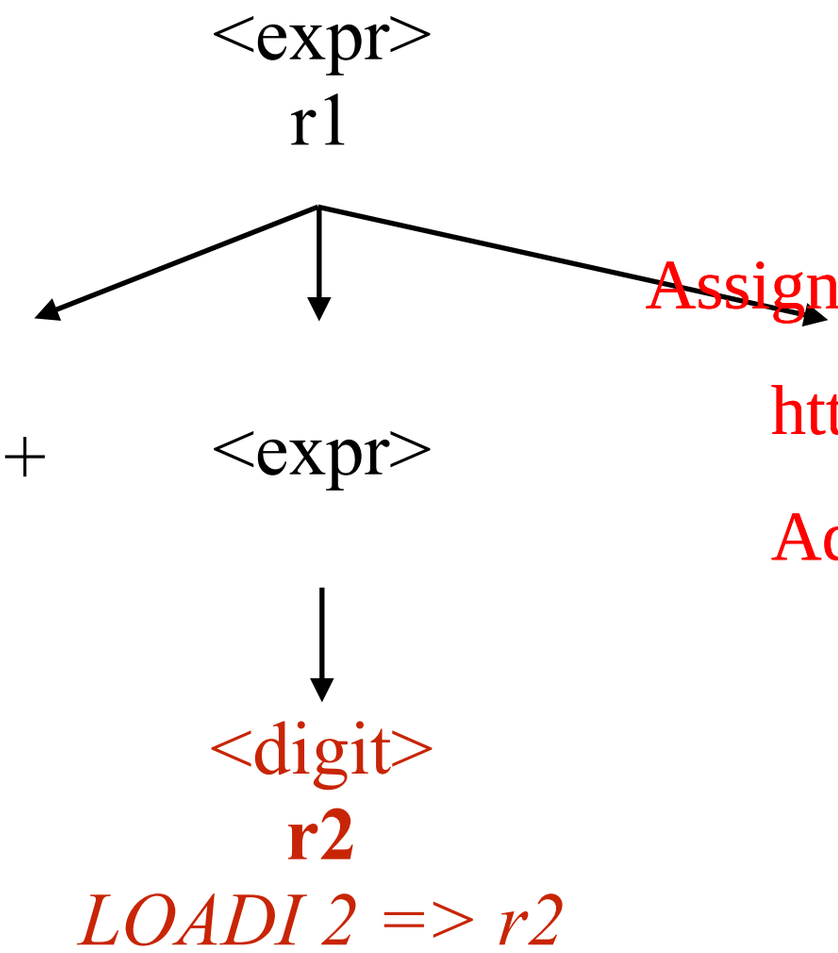


generated code
(in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



Register r2 is selected as target register.
We generate code to load digit into r2.
We return r2.

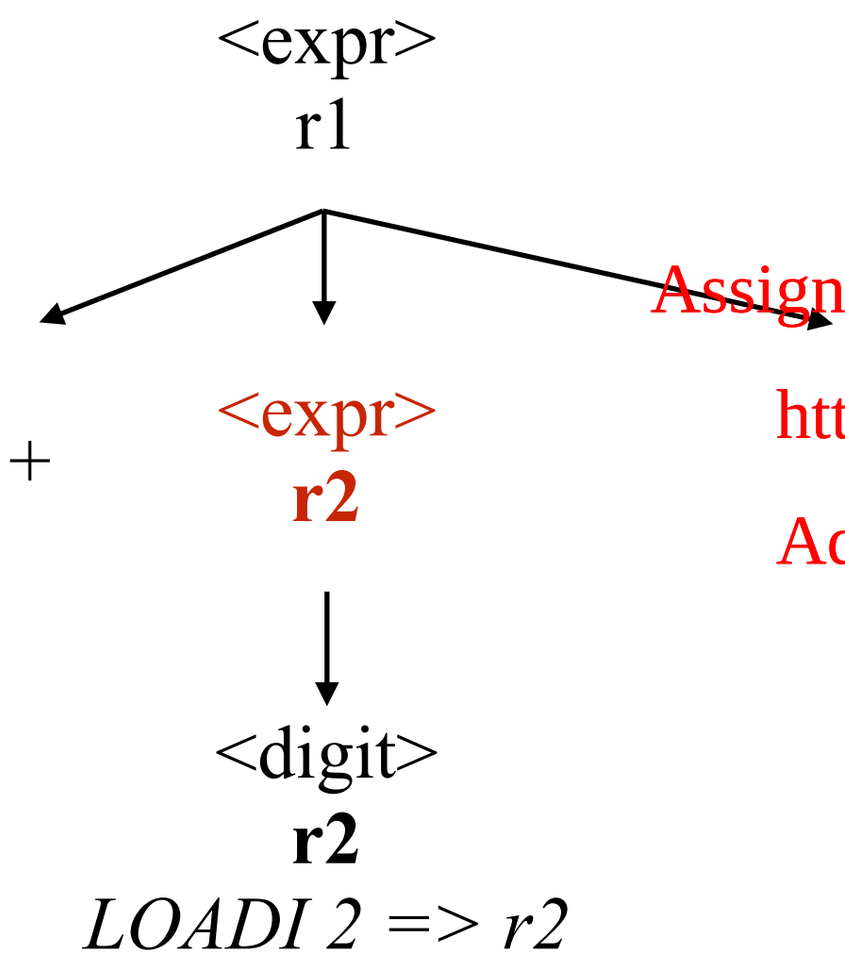
LOADI 2 => r2

generated code
(in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



We return register r2 from child <digit>.

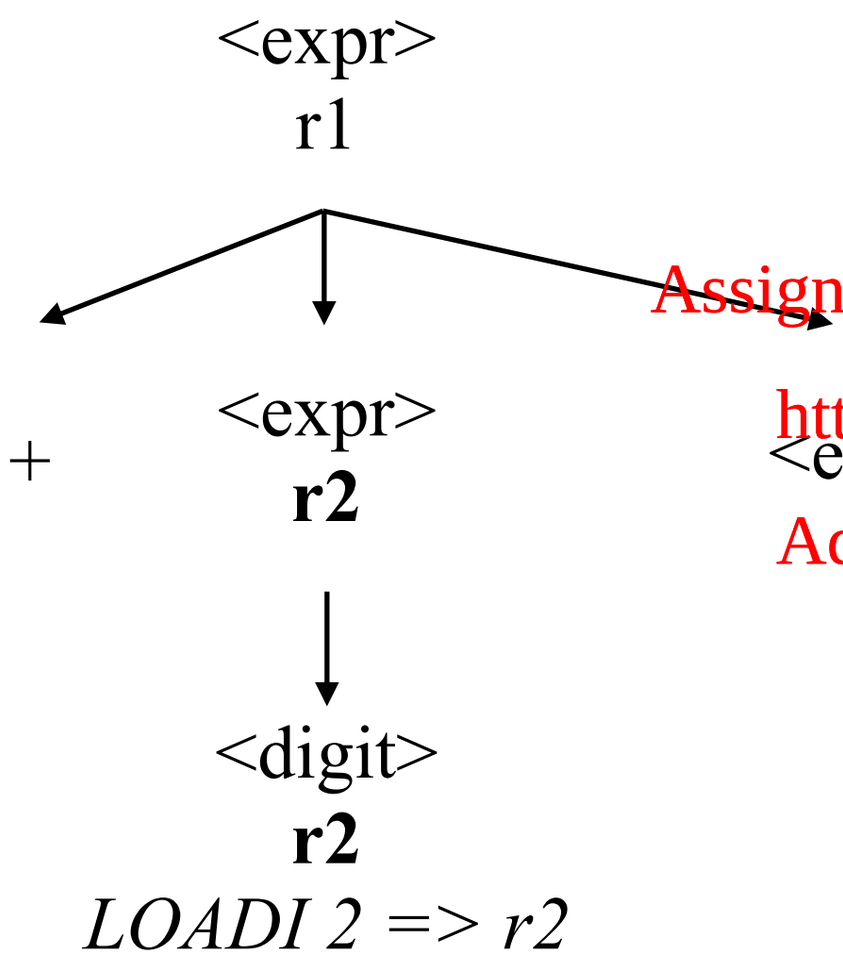

```
LOADI  2 => r2
```

generated code
(in progress)

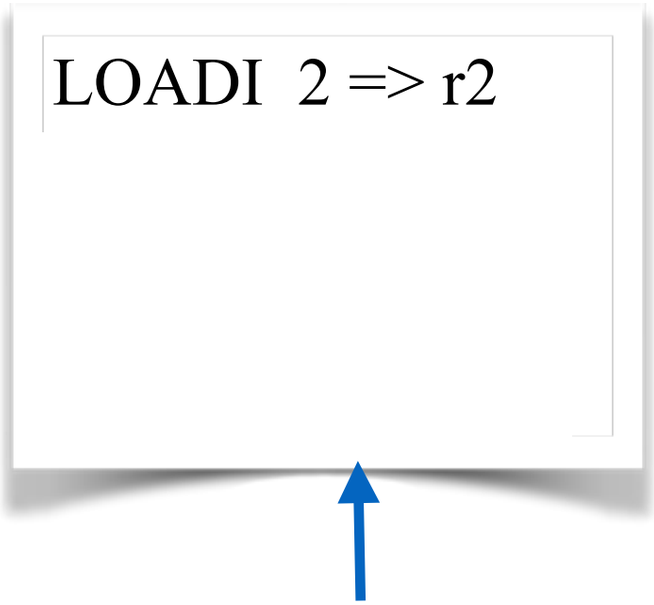
Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



We recurse into next <expr>.

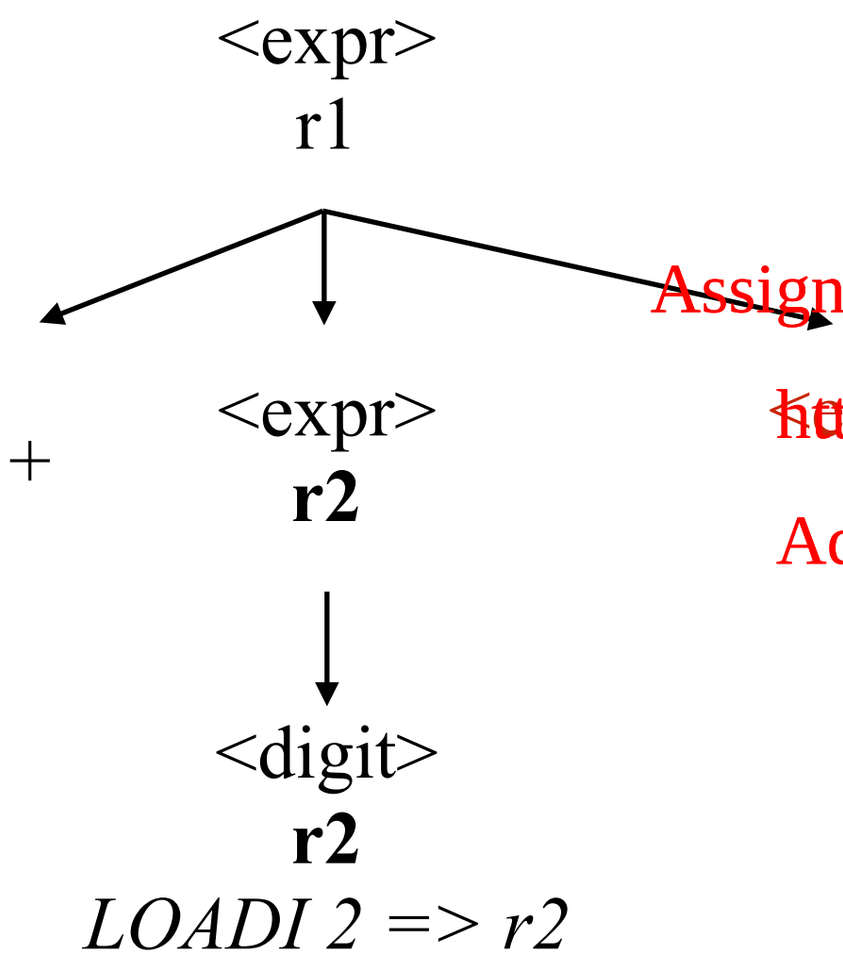


generated code
(in progress)

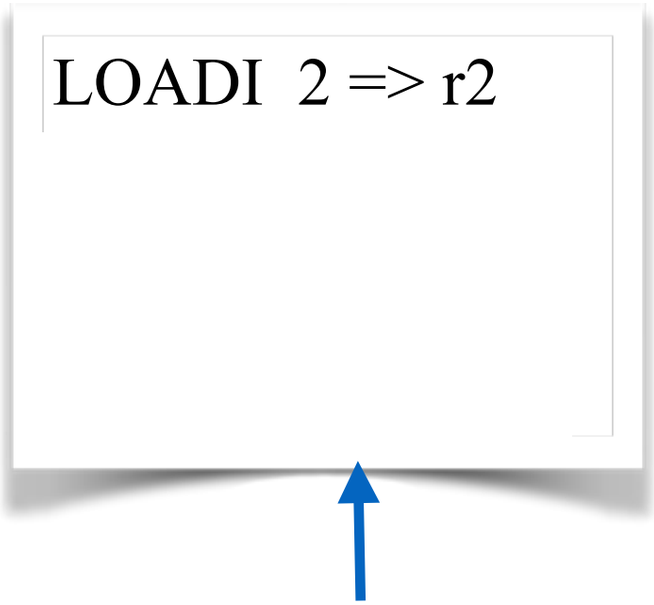
Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



We select r3 as target register.
We recurse into <expr>.

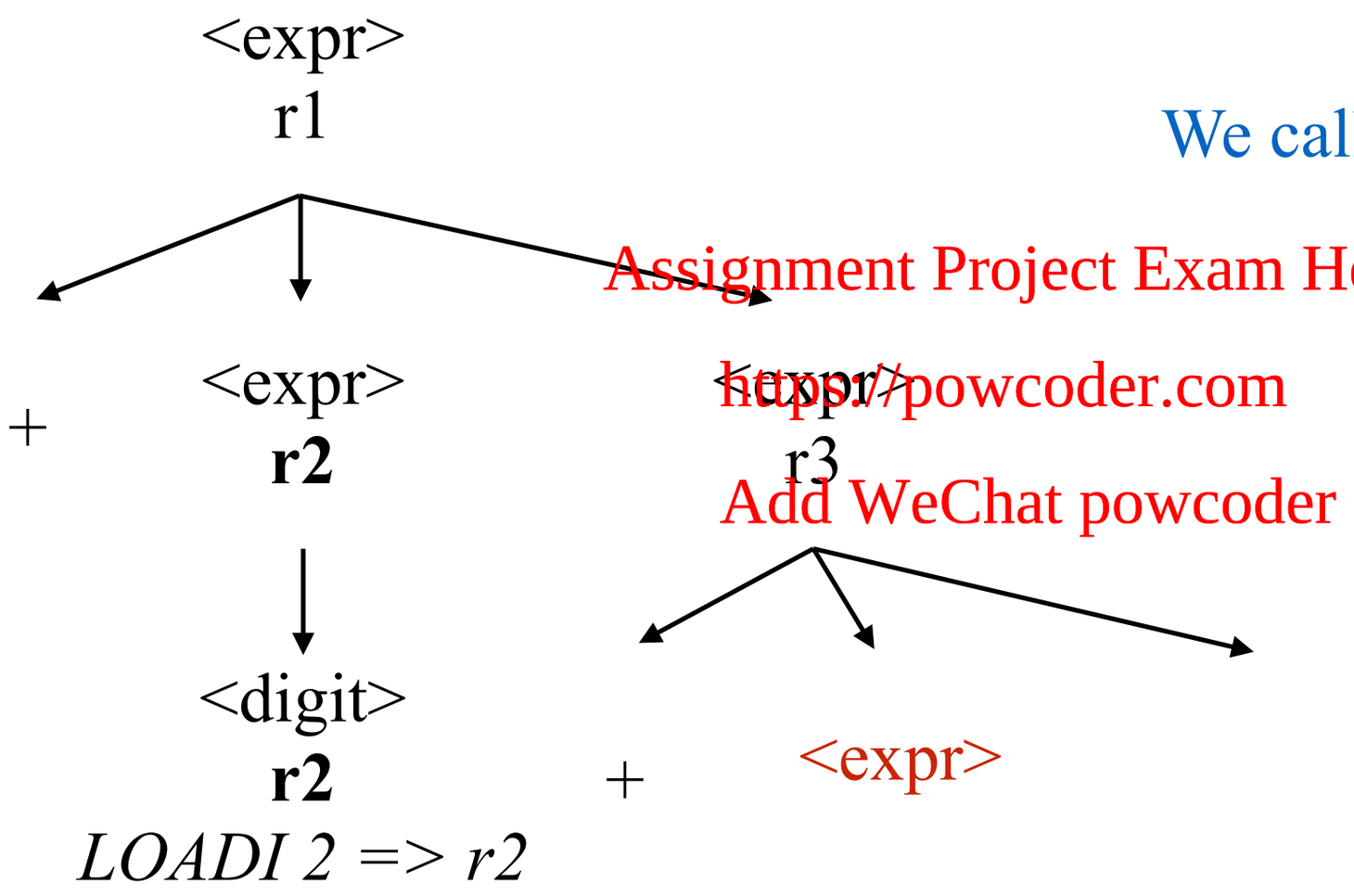


generated code
(in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



We call the <digit> function.

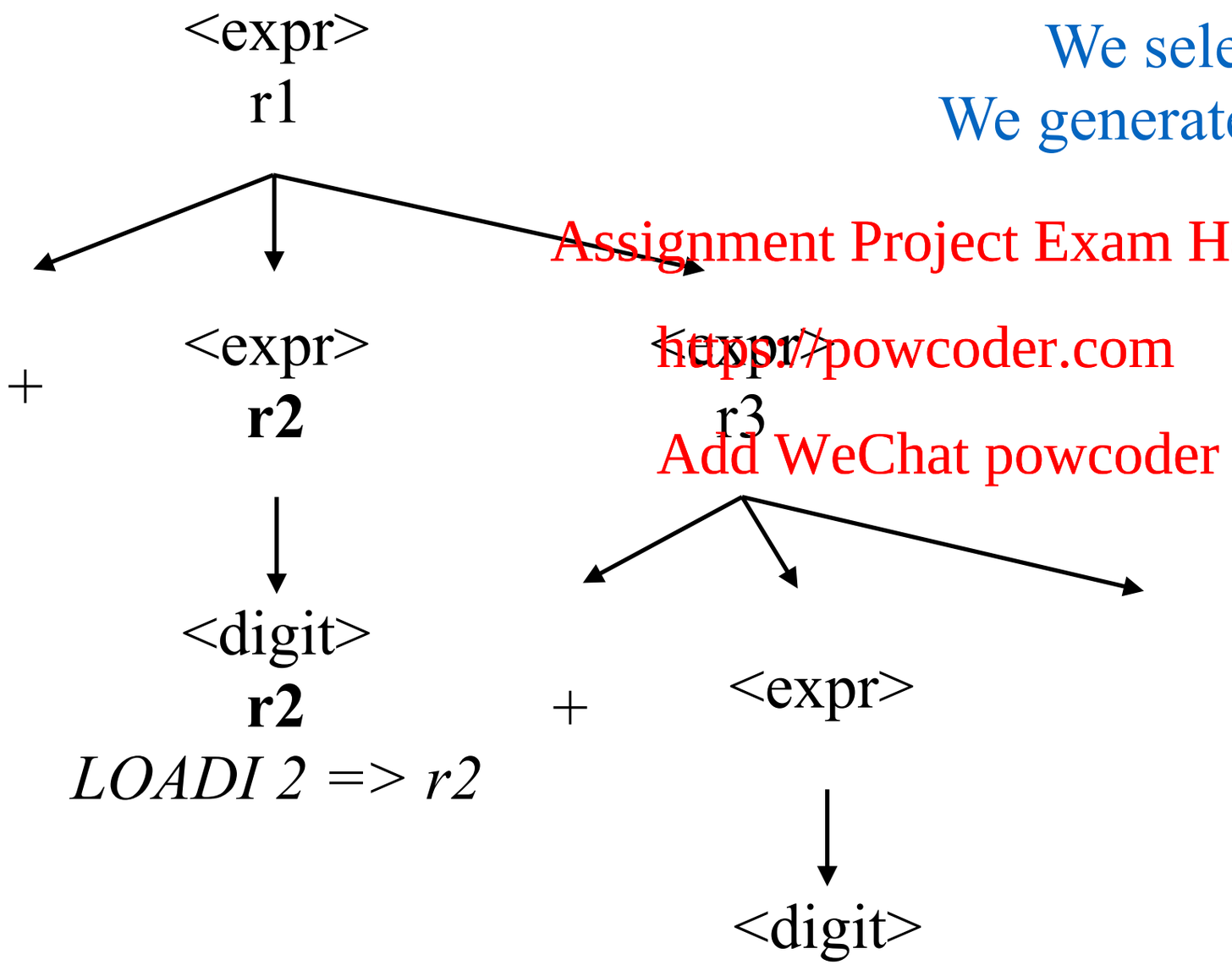

```
LOADI  2 => r2
```

generated code
(in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



We select r4 as target register.
We generate code to load digit into r4.
We return r4.

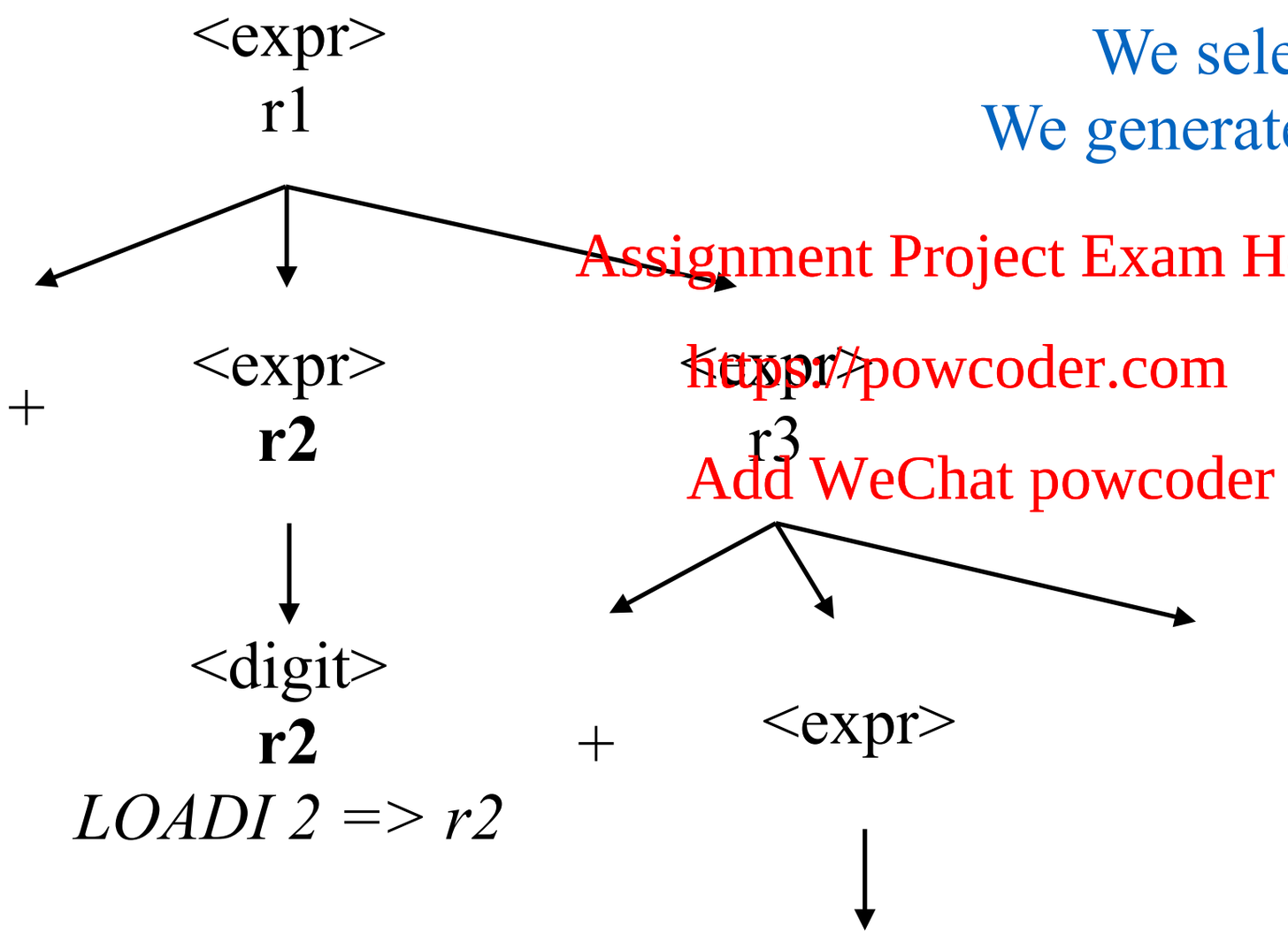
```
LOADI  2 => r2
```

generated code
(in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



We select r4 as target register.
We generate code to load digit into r4.
We return r4.

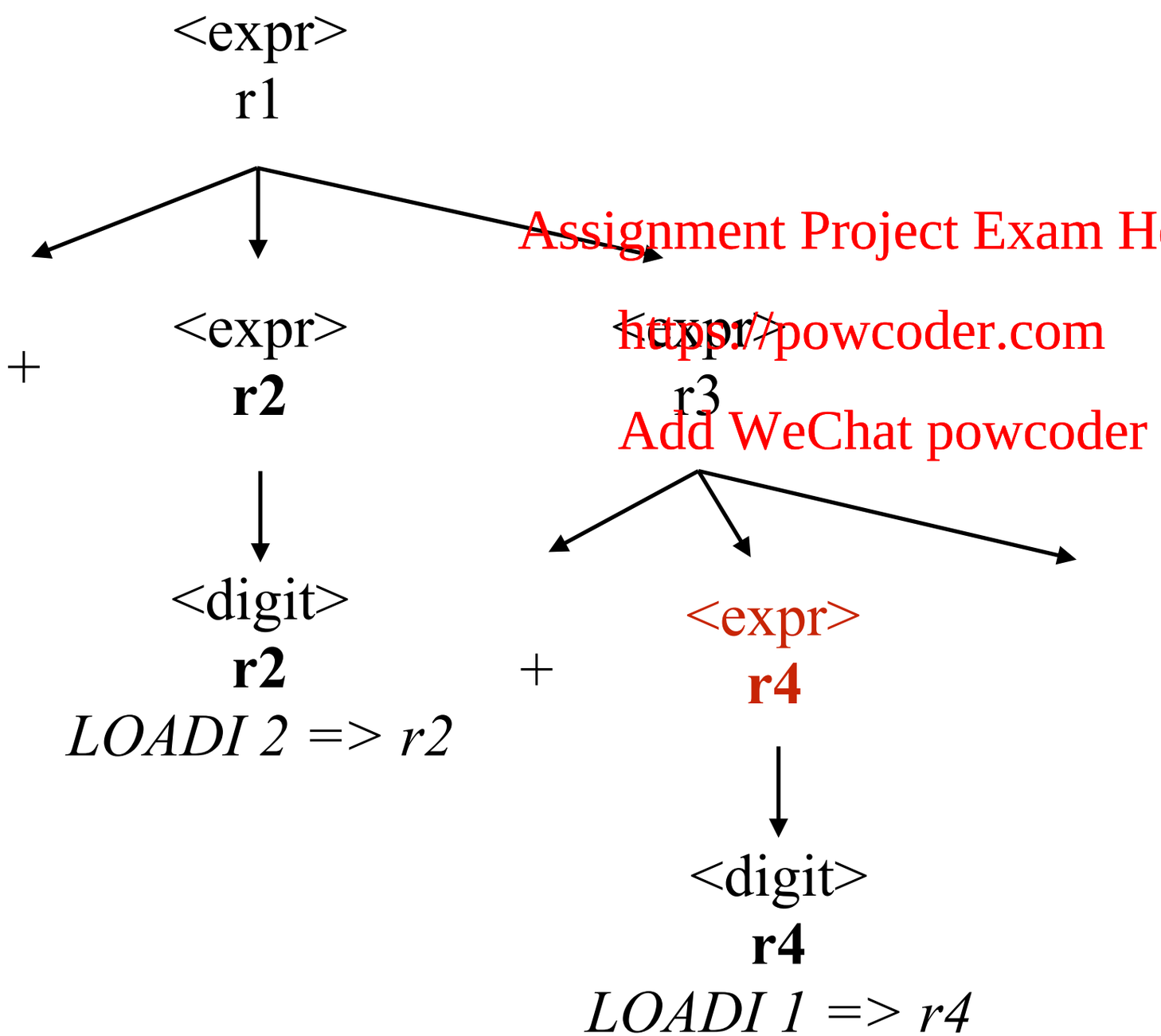
```
LOADI  2 => r2
LOADI  1 => r4
```

generated code
(in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



We return r4.

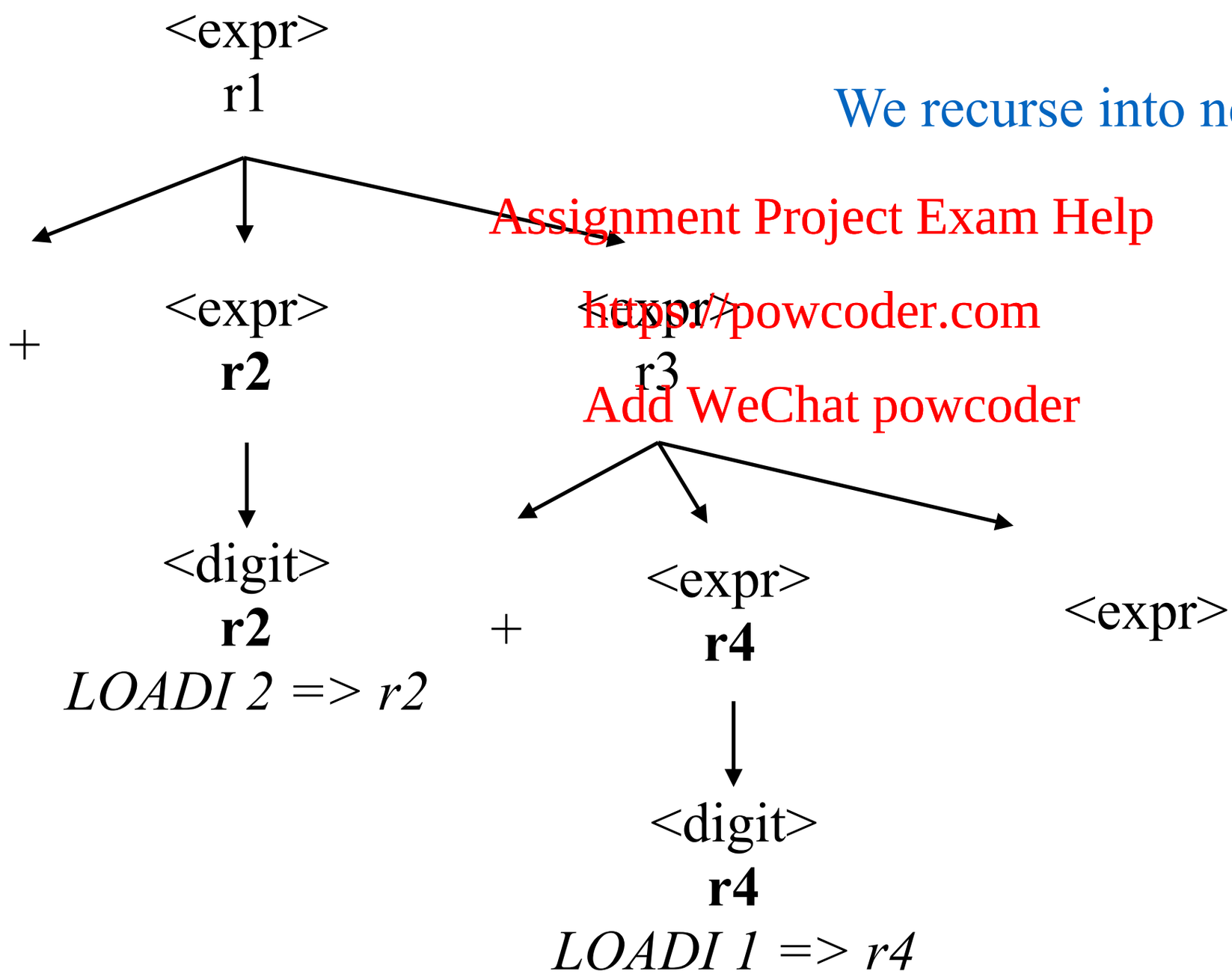
```
LOADI  2 => r2
LOADI  1 => r4
```

generated code
(in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



We recurse into next <expr> function.

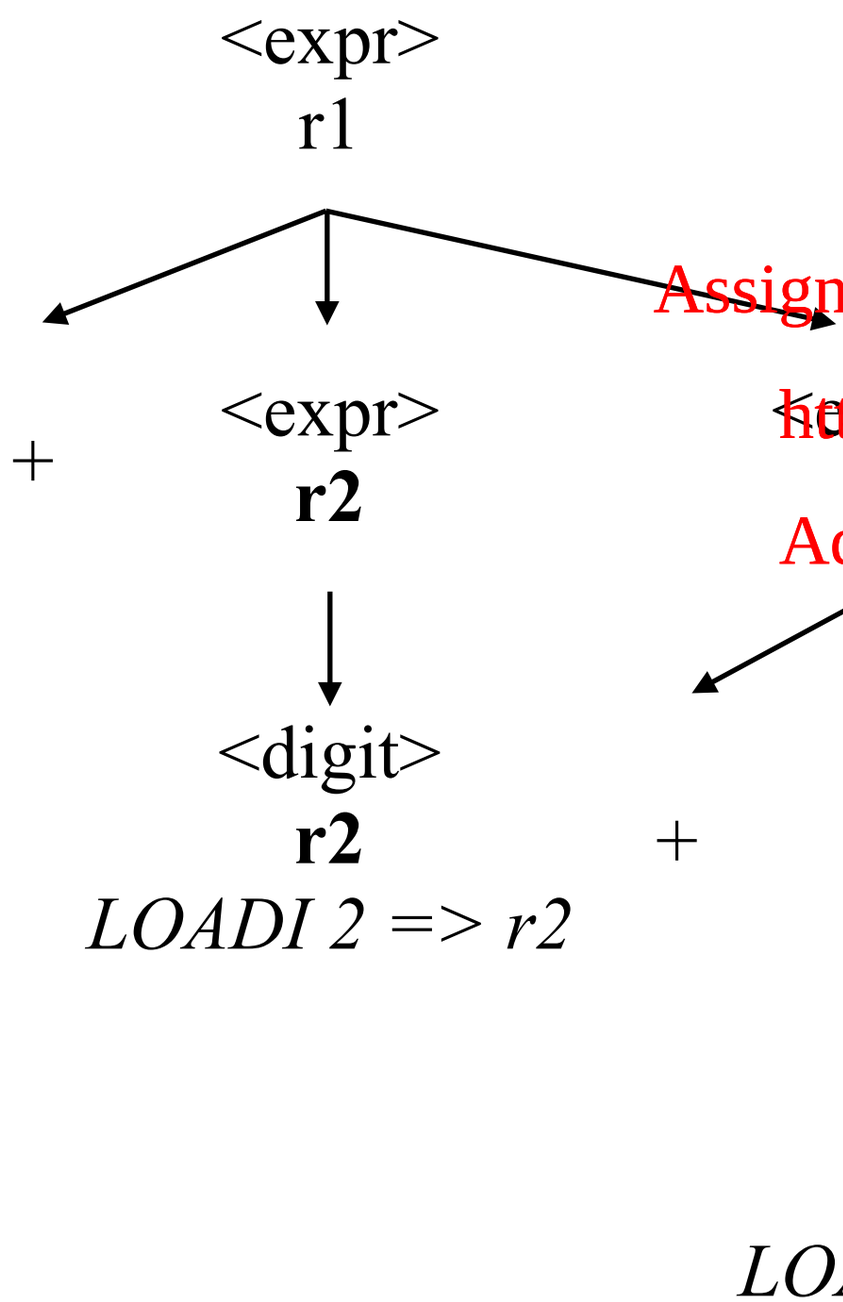

```
LOADI  2 => r2
LOADI  1 => r4
```

generated code
(in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



We call <digit> function.

<expr>

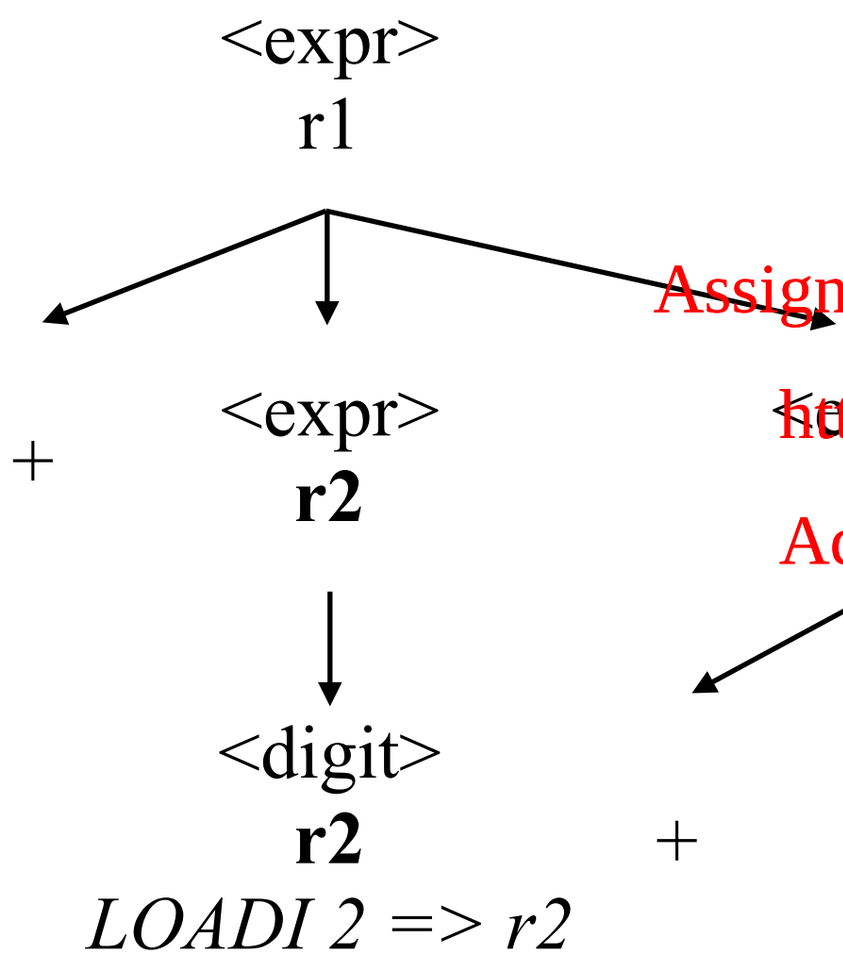
```
LOADI  2 => r2
LOADI  1 => r4
```

generated code
(in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



We select r5 as target register.
We generate code to load digit into r5.
We return r5.

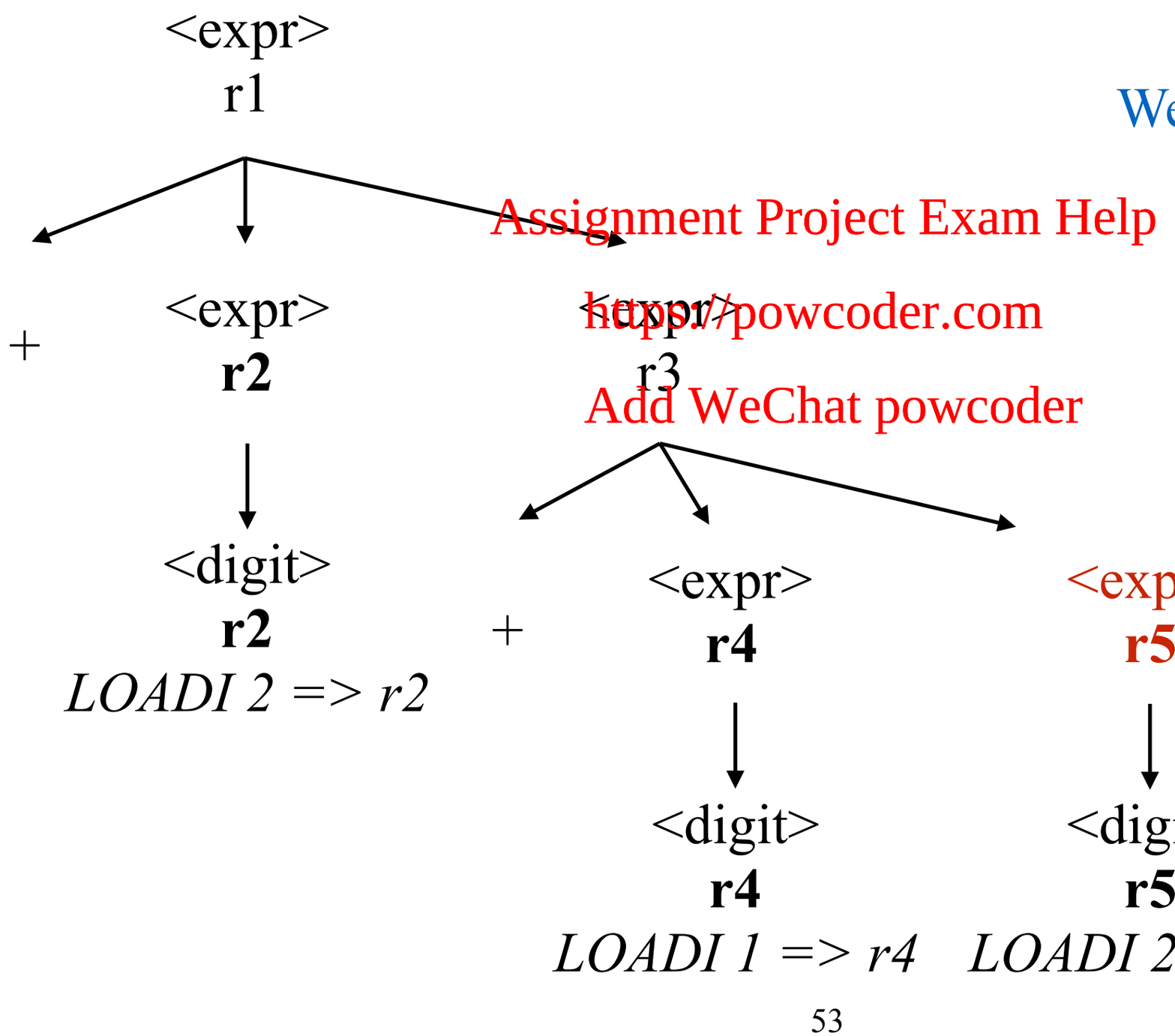
```
LOADI  2 => r2
LOADI  1 => r4
LOADI  2 => r5
```

generated code
(in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



We return r5.

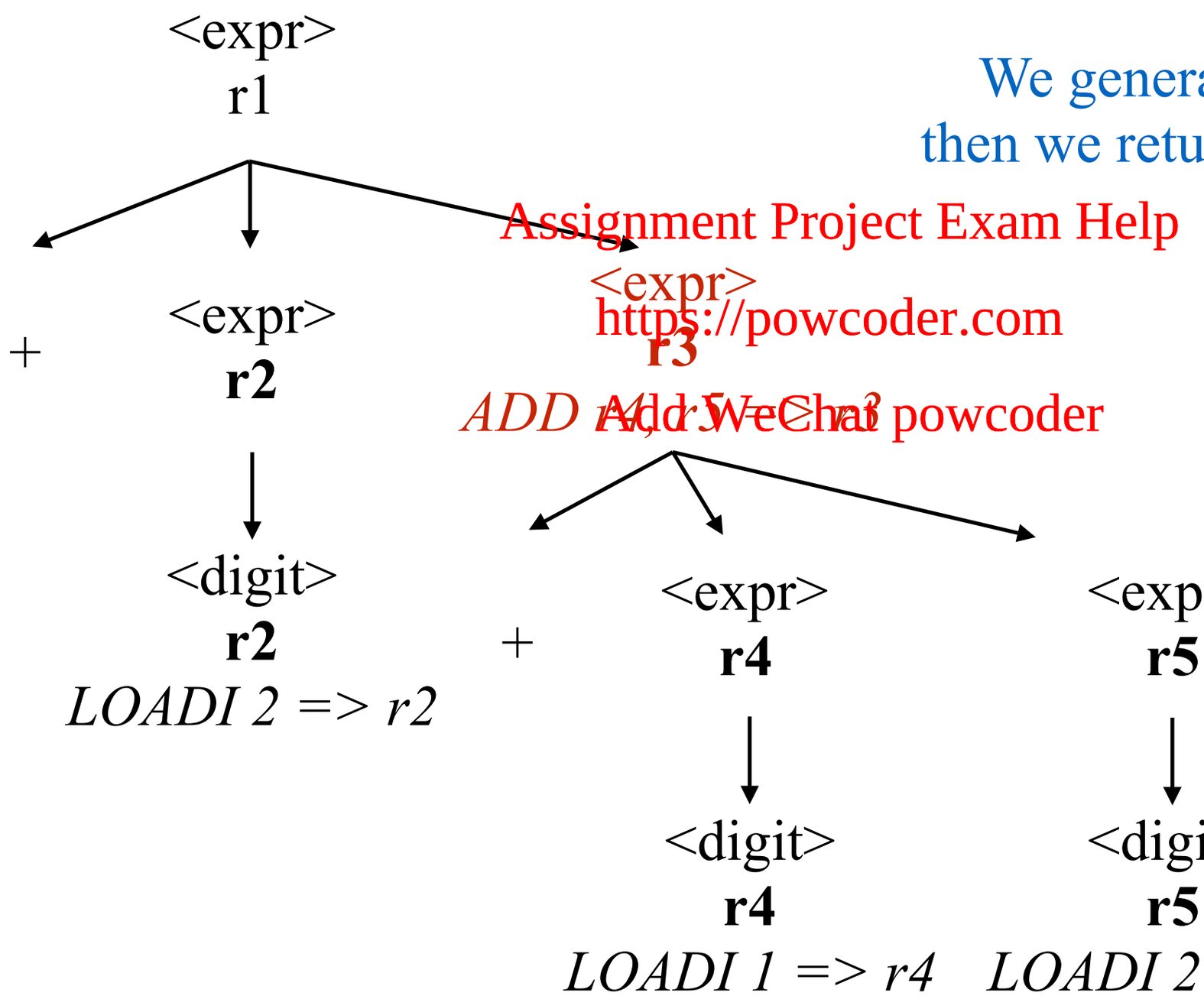
```
LOADI  2 => r2
LOADI  1 => r4
LOADI  2 => r5
```

generated code (in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



We generate addition code,
then we return target register r3.

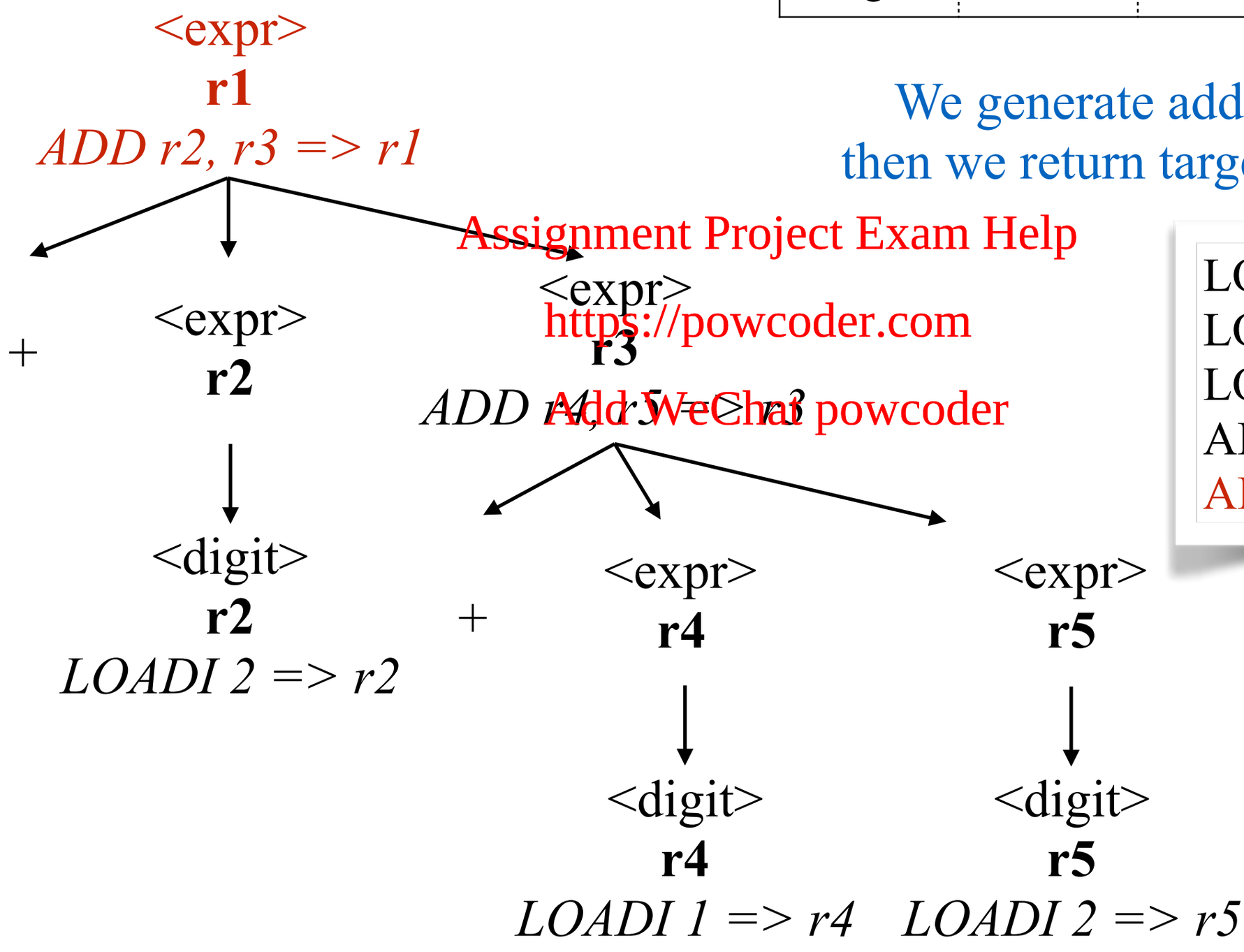
```
LOADI  2 => r2
LOADI  1 => r4
LOADI  2 => r5
ADD  r4, r5 => r3
```

generated code
(in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



We generate addition code,
then we return target register r1.

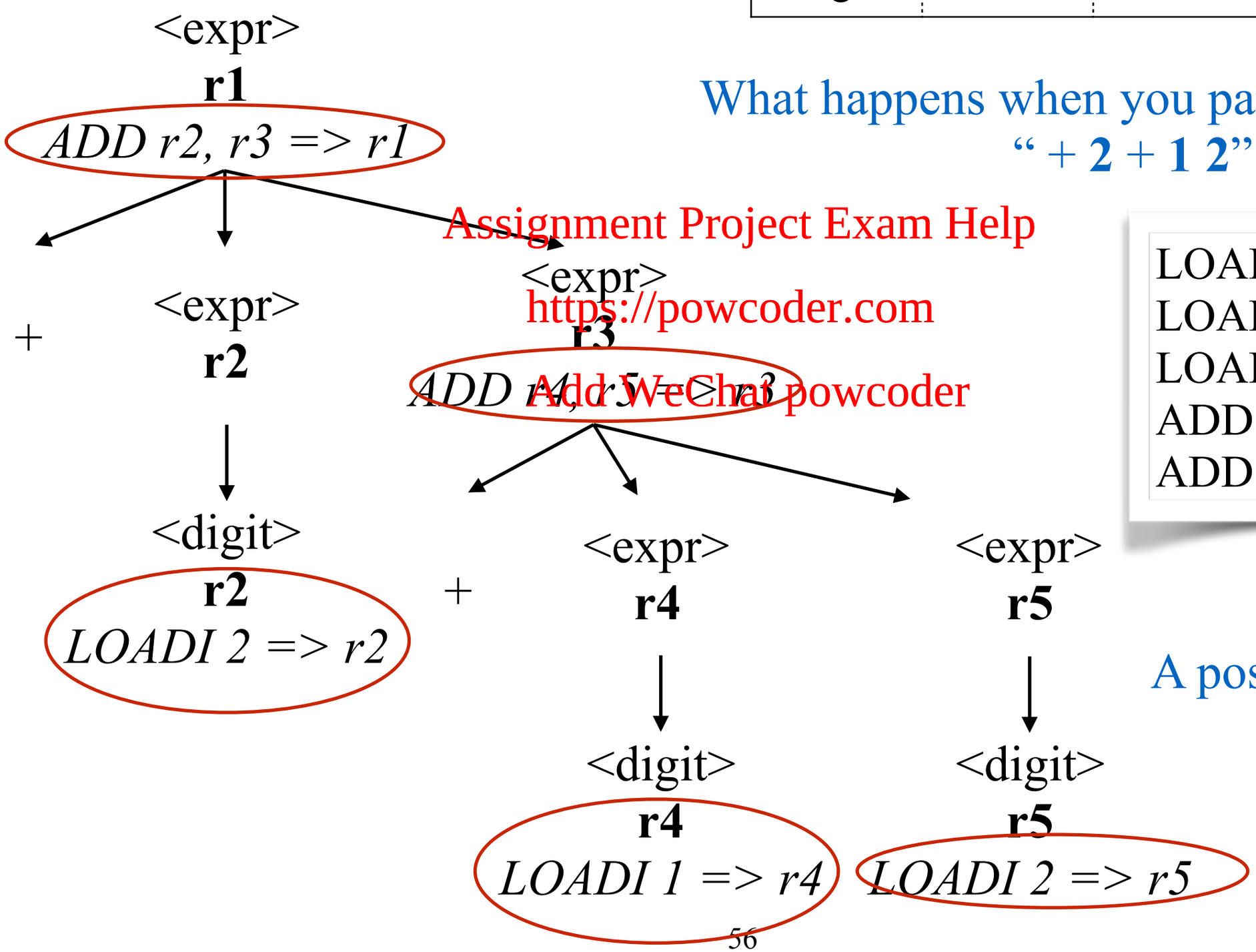
```
LOADI  2 => r2
LOADI  1 => r4
LOADI  2 => r5
ADD  r4, r5 => r3
ADD  r2, r3 => r1
```

generated code
(in progress)

Example: Code Generator (parse tree)

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	r_1	r_2	error
< digit >	error	r_3	error



What happens when you parse subprogram
" + **2** + **1** **2**"

```
LOADI  2 => r2
LOADI  1 => r4
LOADI  2 => r5
ADD  r4, r5 => r3
ADD  r2, r3 => r1
```

A postorder traversal

Example: Basic Performance Predictor

1: < expr > ::= + < expr > < expr > |
2: < digit >
3: < digit > ::= 0 | 1 | 2 | 3 | … | 9

	+	0…9	other
< expr >	Rule 1	Rule 2	
< digit >		Rule 3	

```
int expr( ) {  // returns cycles
  int cyc1, cyc2; // subexpression cycles
  switch token {
    case +:
      token := next_token ( )
      cyc1 = expr( );
      cyc2 = expr( );
      // ADD takes 2 cycles
      return cyc1 + cyc2 + 2;
    case 0..9:
      return digit ( );
    …
  }
}
```

```
int digit( ) {  // returns cycles
  switch token {
    case 1:
      token := next_token ( );
      return 1; // LOADI takes 1 cycle
    case 2:
      token := next_token ( );
      return 1; // LOADI takes 1 cycle
      …
  }
}
```

Example: Basic Performance Predictor (cont.)

What happens when you parse subprogram
“ + **2** + **1 2**” ?

The parsing produces:

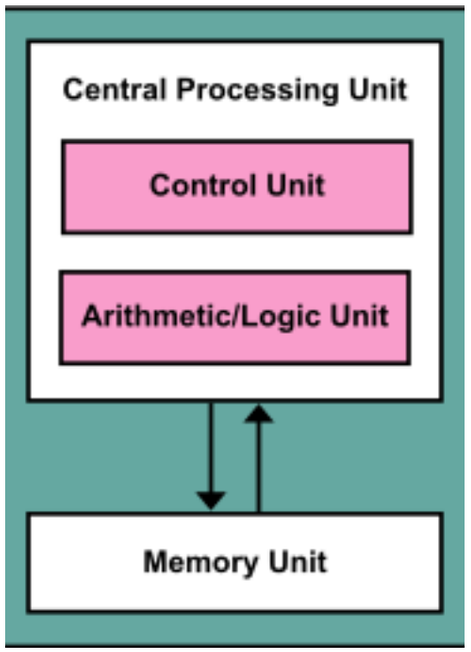
ADD takes 2 cycles

LOADI takes 1 cycle

Imperative Programming Language

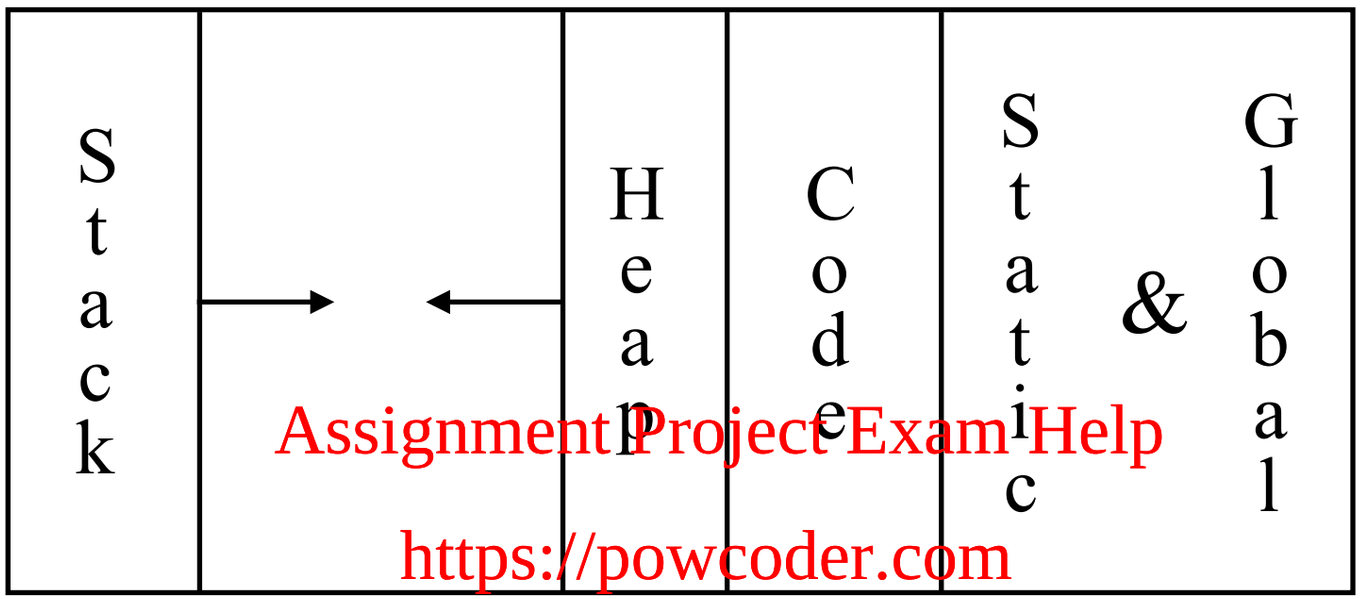
Imperative:
Sequence of state-changing actions.

- Manipulate an abstract machine with:
  1. Variables naming memory locations
  2. Arithmetic/logical operations
  3. Reference, evaluate, assign operations
  4. Explicit control flow statements
- Key operations: Assignment and control flow
- Fits the Von Neumann architecture



Run-time Storage Organization

Typical memory layout



Most Language runtime layout the address space in a similar way

- Pieces (stack, heap, code & globals) may move, but all will be there
- Stack and heap grow toward each other
- Arrays live on one of the stacks, in the global area, or in the heap

Stack vs Heap

Stack:

- Procedure activations, statically allocated local variables, parameter values
- Lifetime same as subroutine in which variables are declared
- Stack frame is pushed with each invocation of a subroutine, and popped after subroutine exit

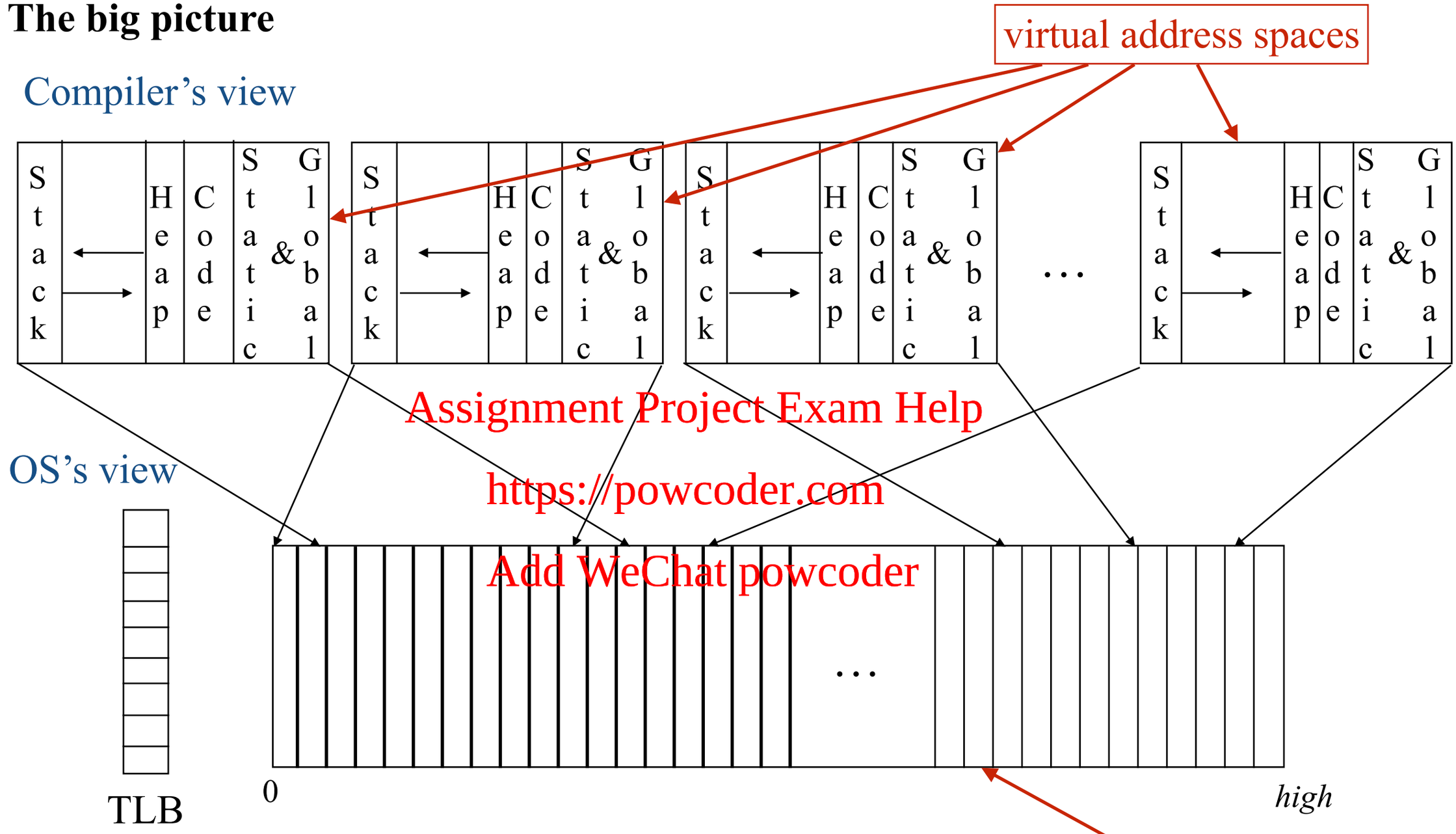
Heap:

- Dynamically allocated data structure, whose size may not be known in advance

- Lifetime extends beyond subroutine in which they are created
- Must be explicitly freed or garbage collected

How Does Address Space Mapping Work?

The big picture



TLB is an address cache used by the **OS** to speed virtual-to-physical address translation. A process may have >1 level **TLB**

C: An Imperative Programming Language

Expressions: include procedure and function calls and assignments, and thus can have side-effects

Control Structures:

- if statements, with or without else clauses
- loops, with break and continue exits

  while (< expr >) < stmt >

  do < stmt > while (< expr >)

  for (< expr >; < expr >; < expr >) < stmt >
- switch statements
- goto with labelled branch targets

Basic Comparisons (Incomplete)

C	JAVA
Basic types: int, double, char, boolean	Primitive types: int, double, char, boolean
Pointer (to a value)	**Reference (to an object)**
Aggregates: array, **struct**	Aggregates: array, **object (class)**
Control Flow:if-else, switch, while, break, continue, for, return, **goto**	Control Flow:if-else, switch, while, break, continue, for, return
Logic Operators: ||, &&, !	Logic Operators: ||, &&, !
Logical Comparisons: ==, !=	Logical Comparisons: ==, !=
Numeric Comparisons: <, >, <=, >=	Numeric Comparisons: <, >, <=, >=
string as **char* array**	**String** as an object

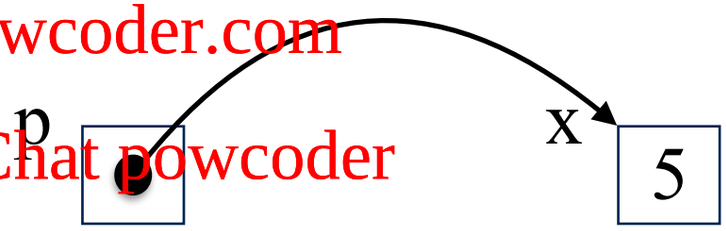
Pointers in C

Pointer: Variable whose R-value (content) is the L-value (address) of a variable

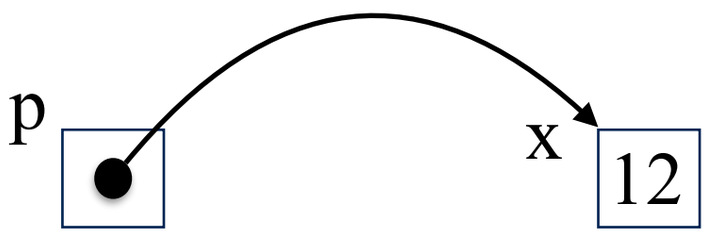
- "address-of" operator &
- dereference ("content-of") operator *

int *p, x;

p = &x;
*p = 5;



x = 12;



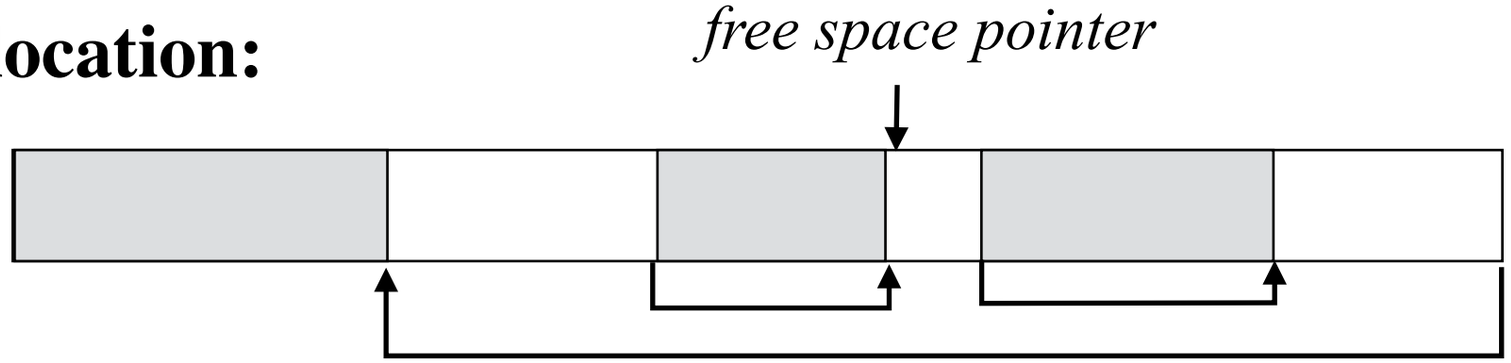
Example: Maintaining Free Lists

- A**llocate**: continuous block of memory; remove space from free list (here: singly-linked list).

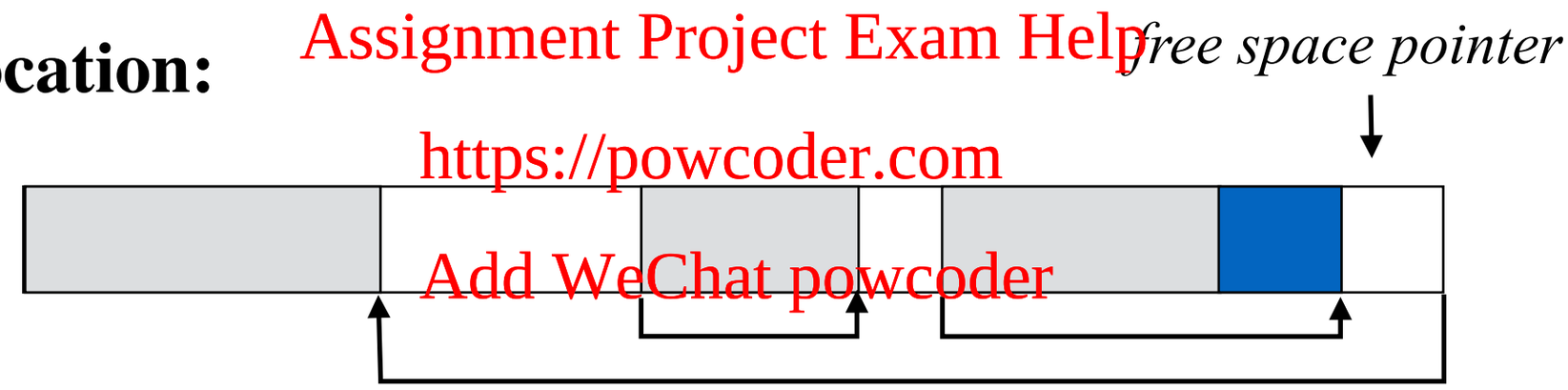
- F**ree**: return to free list after coalescing with adjacent free storage (if possible); may initiate compaction.

Example: Maintaining Free Lists

Before **Allocation:**



After **Allocation:**

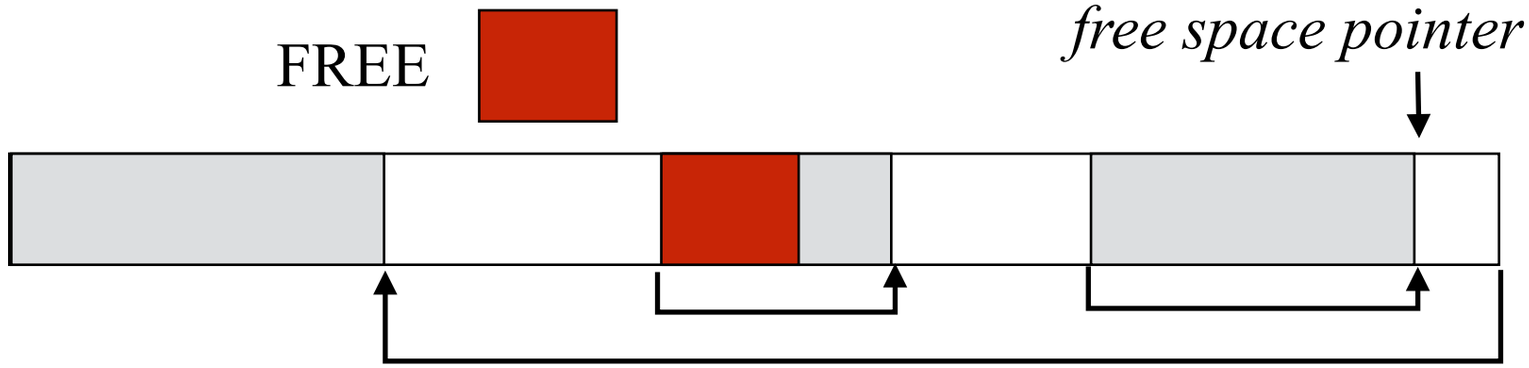


ALLOCATED

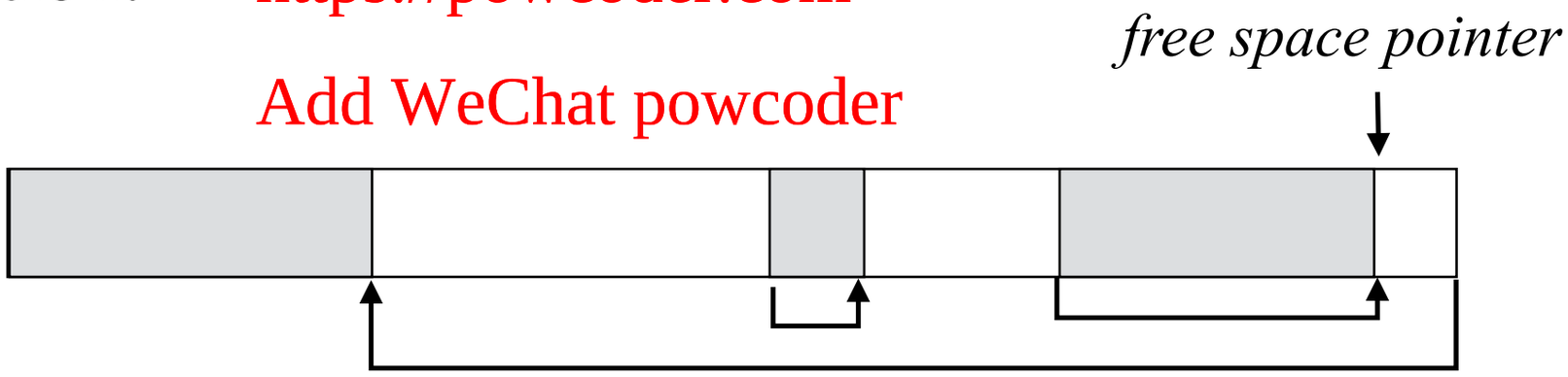
NEWLY ALLOCATED

Example: Maintaining Free Lists

Before De-allocation:



After **De-allocation:**



Potential Issues with Explicit Control of Heaps

- Dangling references
  - Storage pointed to is freed, but pointer is not set to NULL
  - Able to access storage whose values are not meaningful
- Garbage
  - Objects in heap that cannot be accessed by the program any more
  - Example

```
int *x, *y;
x = (int*) malloc (sizeof (int));
y = (int*) malloc (sizeof (int));
x = y;
```


- Memory leaks
  - Failure to release (reclaim) memory storage build up overtime

Potential Issues with Explicit Control of Heaps

- Dangling references
  - Storage pointed to is freed, but pointer is not set to NULL
  - Able to access storage whose values are not meaningful
- Garbage
  - Objects in heap that cannot be accessed by the program any more
  - Example

```
int *x, *y;
x = (int*) malloc (sizeof (int));
y = (int*) malloc (sizeof (int));
x = y;
```


- Memory leaks
  - Failure to release (reclaim) memory storage build up overtime

Potential Issues with Explicit Control of Heaps

- Dangling references
  - Storage pointed to is freed, but pointer is not set to NULL
  - Able to access storage whose values are not meaningful
- Garbage
  - Objects in heap that cannot be accessed by the program any more
  - Example

```
int *x, *y;
x = (int*) malloc (sizeof (int));
y = (int*) malloc (sizeof (int));
x = y;
```


- Memory leaks
  - Failure to release (reclaim) memory storage build up overtime

Example: Singly-Linked List

```
#include <stdio.h>
#include <stdlib.h>
/* TYPE DEFINITION */
typedef struct cell listcell;

struct cell
{       int num;
        listcell *next;
};

/* GLOBAL VARIABLES */
listcell  *head, *new_cell, *current_cell;
```

Example: Singly-Linked List

Let's deallocate,i.e., free all list elements

```
#include <list.h>
/* GLOBAL VARIABLES */
listcell *head, *new_cell, *current_cell;
int main(void){
        /* CREATE FIRST LIST ELEMENT */
        ...

        /* CREATE NINE MORE ELEMENTS */
        ...

        /* DEALLOCATE LIST */
        for(current_cell = head;
            current_cell != null;
            current_cell = current_cell -> next){
                free(current_cell);
        }
        ...
}
```

Does this work?

Example: Singly-Linked List

Let's deallocate,i.e., free all list elements

```
#include <list.h>
/* GLOBAL VARIABLES */
listcell *head, *new_cell, *current_cell;
int main(void){
        /* CREATE FIRST LIST ELEMENT */
        ...

        /* CREATE NINE MORE ELEMENTS */
        ...

        /* DEALLOCATE LIST */
        for(current_cell = head;
             current_cell != null;
             current_cell = current_cell -> next){
                free(current_cell);
        }
        ...
}
```

What went wrong?

Uninitialized variables and “dangerous” casting

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
        int *a;

        *a = 12;
        printf("%x,%x: %d\n", &a, a, *a);
        a = (int *)12;
        printf("%d \n", *a);
}
```



```
> a.out
effff60c effff68c: 12
segmentation fault (core dumped)
```

Note: Segmentation faults result in the generation of a core file which can be rather large. Don’t forget to delete it.

What went wrong?

That's better!

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
        int *a = NULL; /* good practice */

        a = (int *)malloc(sizeof(int));

        *a = 12;
        printf("%x,%x: %d\n", &a, a, *a);
}
```

```
> a.out
effff60c 20900: 12
```

What went wrong?

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
        int i;

        char* string = "Hello, how are you today.";
        printf("\n%s\n", string);

        for(i = 0; string[i] != '.'; i++){
                if(string[i] = ' ')
                        for(; string[i] = ' '; i++);
                printf("%c", string[i]);
        }
        printf(".\n");
}
```



```
> a.out
Hello, how are you today.
Segmentation fault (core dumped)
```

What went wrong?

“ = ” is not the same as “ == ”

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
        int i;

        char* string = "Hello, how are you today.";
        printf("\n%s\n", string);

        for(i = 0; string[i] != '.'; i++){
                if(string[i] == ' '){
                        for(; string[i] == ' '; i++);
                printf("%c", string[i]);
        }
        printf(".\n");
}
```

```
> a.out
Hello, how are you today.
Hello,howareyoutoday.
```

What went wrong?

“ Aliasing ” and freeing memory

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
        int *a = NULL;
        int *b = NULL;
        int *c = NULL;

        a = (int *)malloc(sizeof(int));
        b = a;
        *a = 12;
        printf("%x %x: %d\n", &a, a, *a);
        printf("%x %x: %d\n", &b, b, *b);
        free(a);
        printf("%x %x: %d\n", &b, b, *b);

        c = (int *)malloc(sizeof(int));
        *c = 10;
        printf("%x %x: %d\n", &c, c, *c);
        printf("%x %x: %d\n", &b, b, *b);
}
```

```
> a.out
effff60c 209d0: 12
effff608 209d0: 12
effff608 209d0: 12
effff604 209d0: 10
effff604 209d0: 10
```

What went wrong?

Use a subroutine to create an object

```
#include <stdio.h>
#include <stdlib.h>

/* TYPE DEFINITION */
typedef struct cell listcell;
struct cell
{
        int num;
        listcell *next;
};
listcell *head = NULL;
listcell *create_listcell(){
        listcell new;
        new.num = -1;
        new.next = NULL;
        return &new;
}
int main(void){
        head = create_listcell();
        printf("head -> num = %d\n", head -> num);
}
```

What went wrong?

Use a subroutine to create an object (cont.)

```
> gcc stack.c
  stack.c: In function “create_listcell”:
  stack.c:17: warning: function returns address of local variable

> ./a.out

head —> num = -1
```

What went wrong?

Use a subroutine to create an object: malloc

```
#include <stdio.h>
#include <stdlib.h>
/* TYPE DEFINITION */
typedef struct cell listcell;
struct cell
{
        int num;
        listcell *next;
};
listcell *head = NULL;
listcell *create_listcell(){
        listcell *new;
        new = (listcell *)malloc(sizeof(listcell));
        new -> num = -1;
        new -> next = NULL;
        return new;
}
int main(void){
        head = create_listcell();
        printf("head -> num = %d\n", head -> num);
}
```

What went wrong?

Use a subroutine to create an object: malloc (cont.)

```
> gcc heap.c
> ./a.out

head —> num = -1
```

Pointers and Arrays in C

Pointers and arrays are similar in C:

- Array name is a pointer to a[0]:

```
int a[10];
int *pa;
pa = &a[0];
```

pa and a have the same semantics

- Pointer arithmetic is array indexing

pa + 1 and a + 1 point to a[1]

- Exception: an array name is a constant pointer

```
a++ is ILLEGAL
a = pa is ILLEGAL (pa = a is LEGAL!)
```

Next Lecture

Things to do:

- Start programming in C.
- Read Scott, Chapter 8.1 - 8.2; ALSU 7.1 - 7.3.
- Next time:
  - Procedure abstractions; Runtime stack; Scoping