CS 211 : Tues 01/16 (lecture 04)

Prof. Hummel
(he/him)

- **<u>Topics</u>: structs, unions, arrays, linked-lists, projects 01 and 02**

January 2024

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

www.a-printable-calendar.com

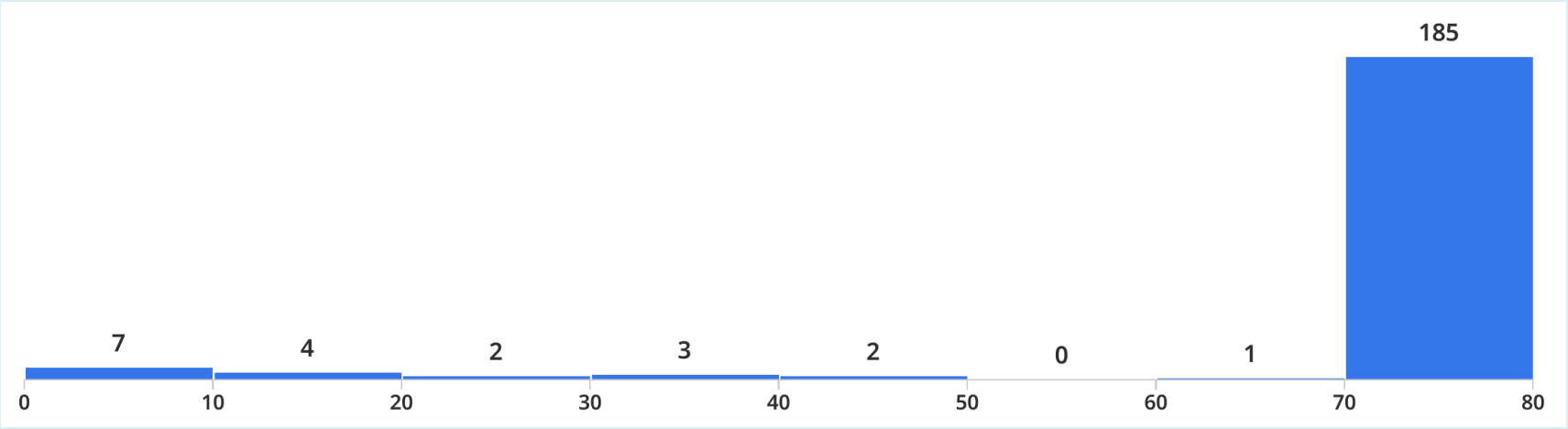
Notes:

- *Lecture slides available on Canvas*
- ***HW 02** due tonight @ 11:59pm*
- ***Project 02** due Friday @ 11:59pm, may be submitted up to 48 hours late (see syllabus). Gradescope is open for submissions (4 per day), test files are posted (Canvas/Piazza has link).*

Northwestern University

Project 01

- **Statistics:**
  - *207 students*
  - *204 submissions (99%)*
  - *185 finished with autograder score of 80/80 (91%)*



Lesson #1: one step at a time

1. Build one step at a time…

1. Recognize the remaining punctuation: +, -, /, %, =, ==, !=, <, <=, etc. Run and test.
2. Recognize keywords by checking the identifier value against the set of given keywords; the cleanest solution is to put the keywords in an array and perform a linear search. See the "TODO" in the code that recognizes identifiers. You can declare the array like this, then use C's *strcmp* function to compare the identifier **value** to a given keyword **keywords[i]**:

```
char* keywords[] = {"and", "break", …, "True", "while"};
```

3. Recognize string literals, much like identifiers are handled. Be careful to handle the case where the closing quote or double-quote may be missing. Run and test.
4. Recognize integer literals (ignore real literals for now); the isdigit(c) function is helpful. Run and test.
5. Extend your code for integer literals to look for a '.' that follows, and if so collect more digits and return a real literal instead. Run and test.
6. Turn your code for #4 and #5 above into a function, such as collect_int_or_real_literal( ). Why? For use in the next step…
7. Recognize and discard comments; nuPython supports line comments that start with #.
8. Appendix A shows examples of test input with the corresponding correct output. The test files are "test01.py", "test02.py", and "test03.py".
9. Confirm your program does not contain memory errors by running valgrind, for example:
   - **make valgrind**
   - **make valgrind < test01.py**

Lesson #2: extend, don't expand

2. Think in sub-steps, and implement that way…

```
static int collect_numeric_literal(FILE* input, ...)
{
  //
  // digits, followed possibly by '.' and
  // more digits
  //
  while (isdigit(c) || c == '.')
  {
     .
     . // trying to do everything in one loop…
     .
  }
```

```
while (isdigit(c))
{
   .
   . // initial integer part
   .
}

if (c == '.') // real literal
{
   .  // value[i] = '.'
   .  // while (isdigit(c)) { }
   .  // return real literal
}
else // integer literal
{
   .
   .
   .
}
```

Lesson #3: helper functions

3. Think in terms of functions…

```
› static void collect_identifier(FILE *input, int c, int *colNumber, char *value) {…}

› static int id_or_keyword(char *value) {…}

  static int collect_numeric_literal(FILE *input, int c, int *colNumber,
›                                    char *value, int prefix) {…}

  static void collect_string_literal(FILE *input, int c, int *colNumber,
›                                    char *value, int startLine, int startCol) {…}
```

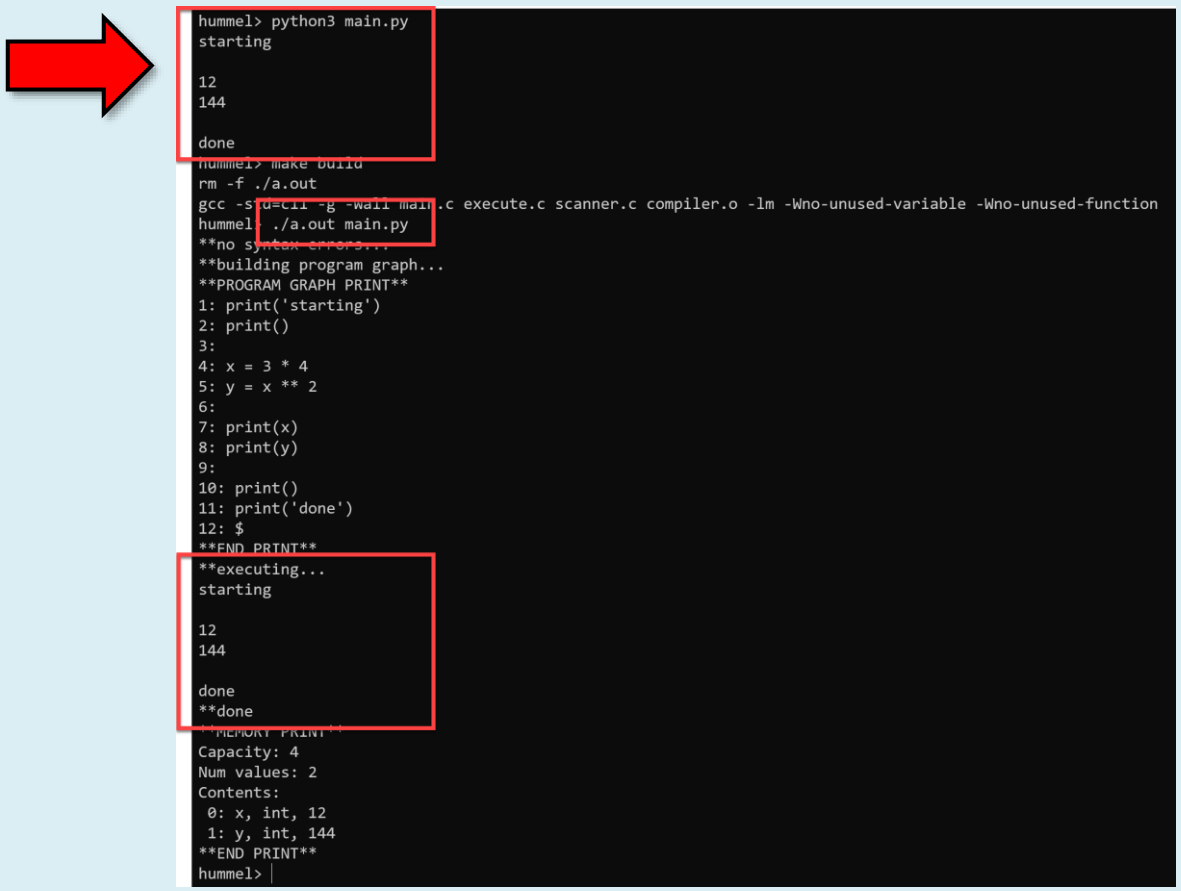
Designing functions is HARD, and takes years of experience…

nuPython

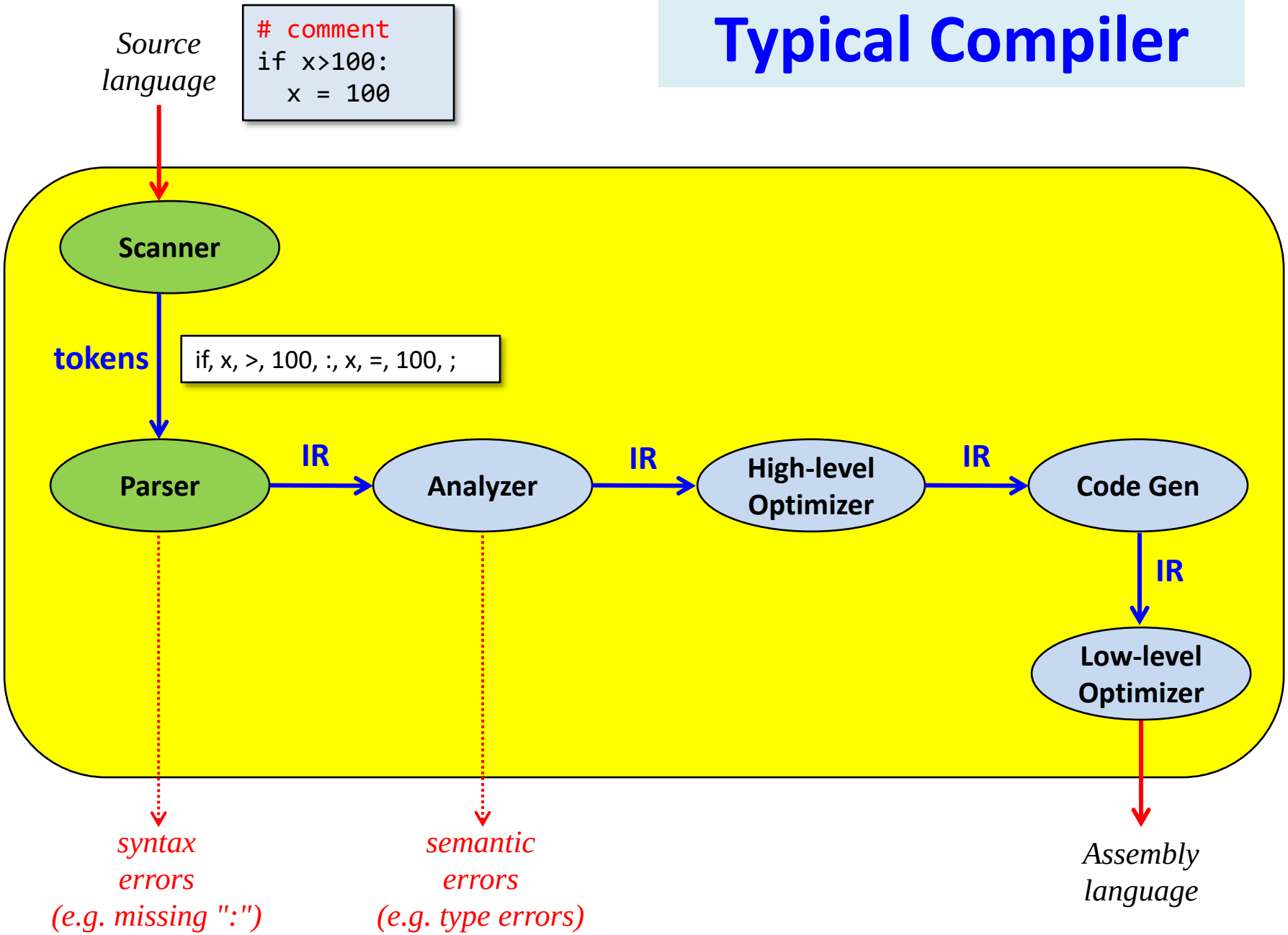
Projects 01 - 04

- **Building a program for executing Python code**

```
1  print("starting")
2  print()
3
4  x = 3 * 4
5  y = x ** 2
6
7  print(x)
8  print(y)
9
10 print()
11 print("done")
12
```

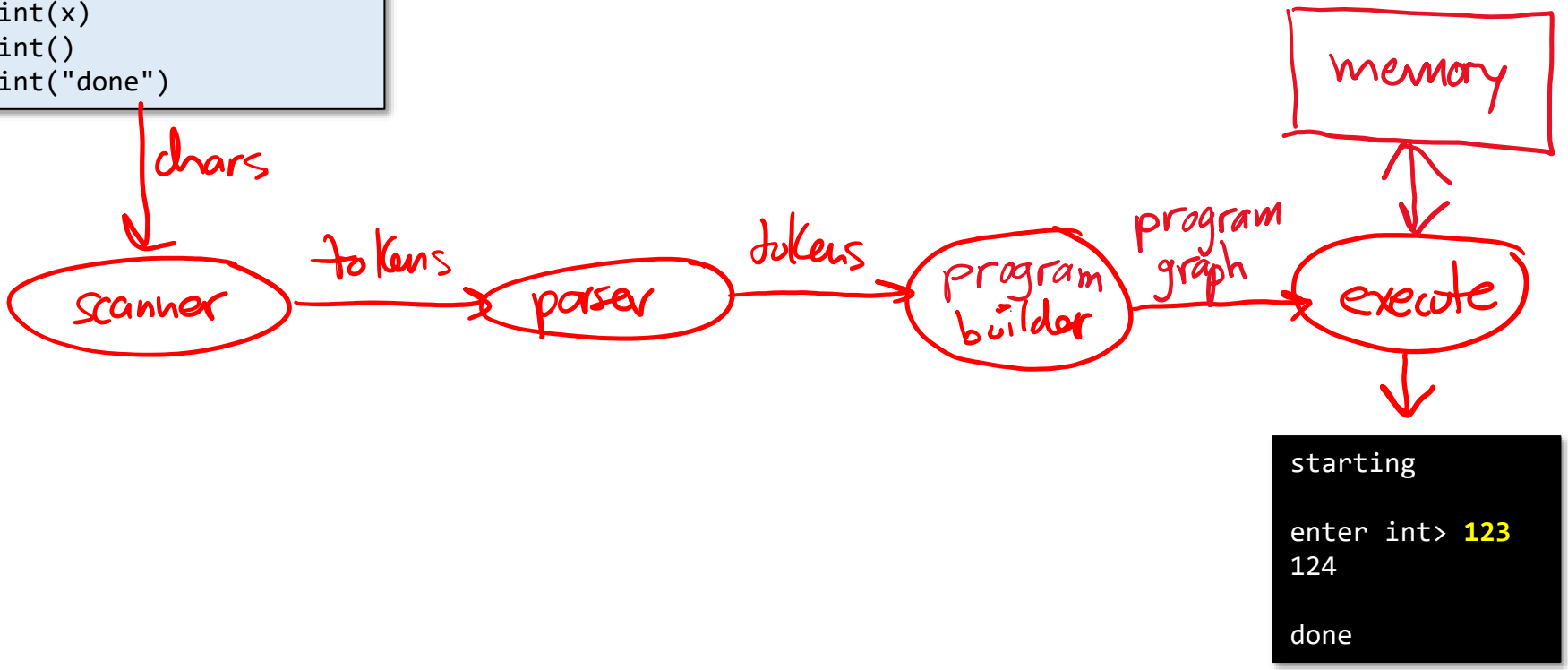


Typical Compiler



nuPython

```
print("starting")
print()
x = input("enter int> ")
x = int(x)
x = x + 1
print(x)
print()
print("done")
```




```
starting

enter int> 123
124

done
```

Structs

- **Structs are used to group data**
- **Example: Project 01…**

```
struct Token {
  int ID;
  int line;
  int col;
};
```

```
struct Token scanner_nextToken(FILE* input, ...)
{
  struct Token T;

  while (true)
  {
    int c = fgetc(input);  // get the next input char:

    if (c == EOF || c == '$')  // no more input, return EOS:
    {
       T.id   = nuPy_EOS;
       T.line = *lineNumber;
       T.col  = *colNumber;
       .
       .
       .
       return T;
```

Question #1

1) Suppose A is a dynamic array containing N tokens. A is declared as follows:

```
struct Token {
  int ID;
  int line;
  int col;
};
```

```
struct Token*  A;
```

Which loop correctly prints all the token ids?

(A)

```
for (int i=0; i<N; i++)
{
  printf("%d\n", A[i].ID);
}
```

(B)

```
for (int i=0; i<N; i++)
{
  printf("%d\n", A[i]->ID);
}
```

(C)

```
for (int i=0; i<N; i++)
{
  printf("%d\n", T.ID);
}
```

Linked-list of tokens

- **Suppose we have a linked-list of tokens…**
- **Nodes in a linked-list are defined using a struct**

```
struct Token {
  int ID;
  int line;
  int col;
};
```

```
struct Node {
  struct Token T;
  struct Node* next;
};
```

```
int main()
{
  struct Node* list;
  .
  .  // build linked-list:
  .
```

```
  // traverse linked-list:
  struct Node* cur = list;
  while (cur != NULL) {
     .
     .
     .
     cur = (*cur).next;
  }
```

->

- The -> operator is equivalent to * .

```
struct Token {
  int ID;
  int line;
  int col;
};
```

```
struct Node {
  struct Token T;
  struct Node* next;
};
```

```
int main()
{
  struct Node* list;
  .
  .
  .
```

```
  // traverse linked-list:
  struct Node* cur = list;
  while (cur != NULL) {
     .
     .
     .
     cur = (*cur).next;
  }
```

```
  // traverse linked-list:
  struct Node* cur = list;
  while (cur != NULL) {
     .
     .
     .
     cur = cur->next;
  }
```

Question #2

```
struct Token {
  int ID;
  int line;
  int col;
};
```

```
struct Node {
  struct Token T;
  struct Node* next;
};
```

2) Suppose list is a linked-list of tokens:

```
struct Node*  list;
```

Which loop correctly prints all the token ids?

(A)

```
struct Node* cur = list;

while (cur != NULL)
{
  printf("%d\n",
         cur->T->ID);

  cur = cur->next;
}
```

(B)

```
struct Node* cur = list;

while (cur != NULL)
{
  printf("%d\n",
         cur->T.ID);

  cur = cur->next;
}
```

(C)

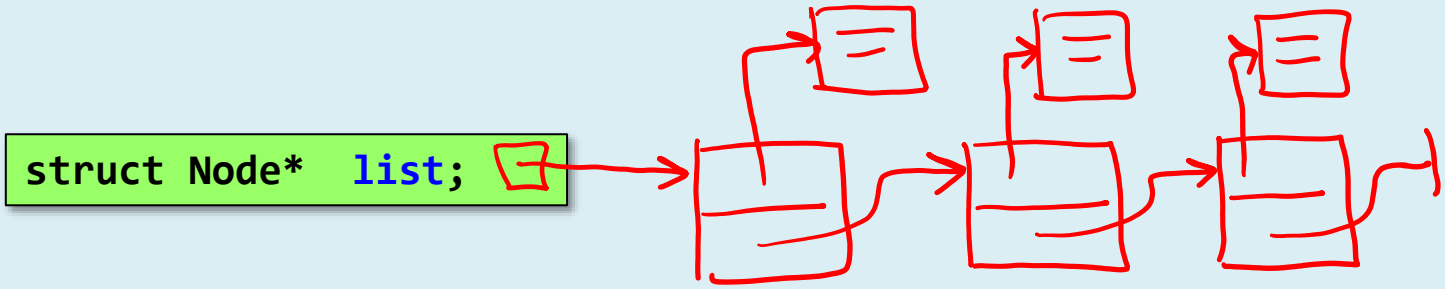
```
struct Node* cur = list;

while (cur != NULL)
{
  printf("%d\n",
         cur->(*T)->ID);

  cur = cur->next;
}
```

Question #3

```
struct Token {
  int ID;
  int line;
  int col;
};
```

```
struct Node {
  struct Token* T;
  struct Node* next;
};
```

3) Suppose the list is built this way:



Which loop correctly prints all the token ids?

(A)

```
struct Node* cur = list;

while (cur != NULL)
{
  printf("%d\n",
         cur->T->ID);

  cur = cur->next;
}
```

(B)

```
struct Node* cur = list;

while (cur != NULL)
{
  printf("%d\n",
         cur->T.ID);

  cur = cur->next;
}
```

(C)

```
struct Node* cur = list;

while (cur != NULL)
{
  printf("%d\n",
         cur->(*T)->ID);

  cur = cur->next;
}
```

Unions

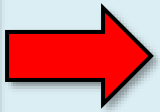
- **Unions are used when you have a set of possibilities, but only need to store one of them**
- **Example: Project 02…**

In nuPython there are 5 possible types of statements…

```
struct STMT {
  int stmt_type;

  union
  {
    struct STMT_ASSIGNMENT*    assignment;
    struct STMT_FUNCTION_CALL* function_call;
    struct STMT_IF_THEN_ELSE*  if_then_else;
    struct STMT_WHILE_LOOP*    while_loop;
    struct STMT_PASS*          pass;
  } types;

};
```

Example

```
x = 3
print(z)
y = x ** 2
print(y)
```



Question #4

*4) On a 64-bit computer, what is **sizeof(struct STMT)**?*

```
struct STMT {
  int stmt_type;

  union
  {
    struct STMT_ASSIGNMENT*    assignment;
    struct STMT_FUNCTION_CALL* function_call;
    struct STMT_IF_THEN_ELSE*  if_then_else;
    struct STMT_WHILE_LOOP*    while_loop;
    struct STMT_PASS*          pass;
  } types;

};
```

```
int main()
{
  printf("size in bytes: %lu\n", sizeof(struct STMT));

  return 0;
}
```

A) 8

B) 16

C) 24

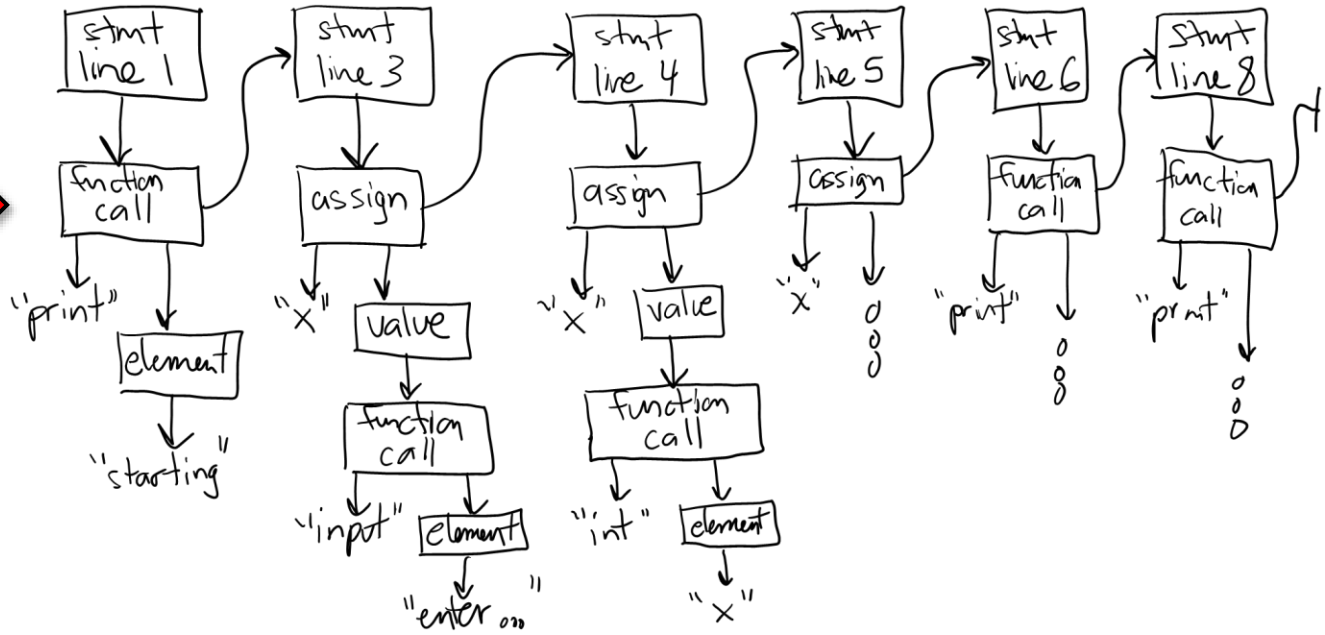
D) 48

nuPython

Program graph

- **In project 02 the representation is a linked-list…**

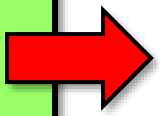
```
print("starting")

x = input("enter int> ")
x = int(x)
x = x + 1
print(x)

print("done")
```



Why a graph?

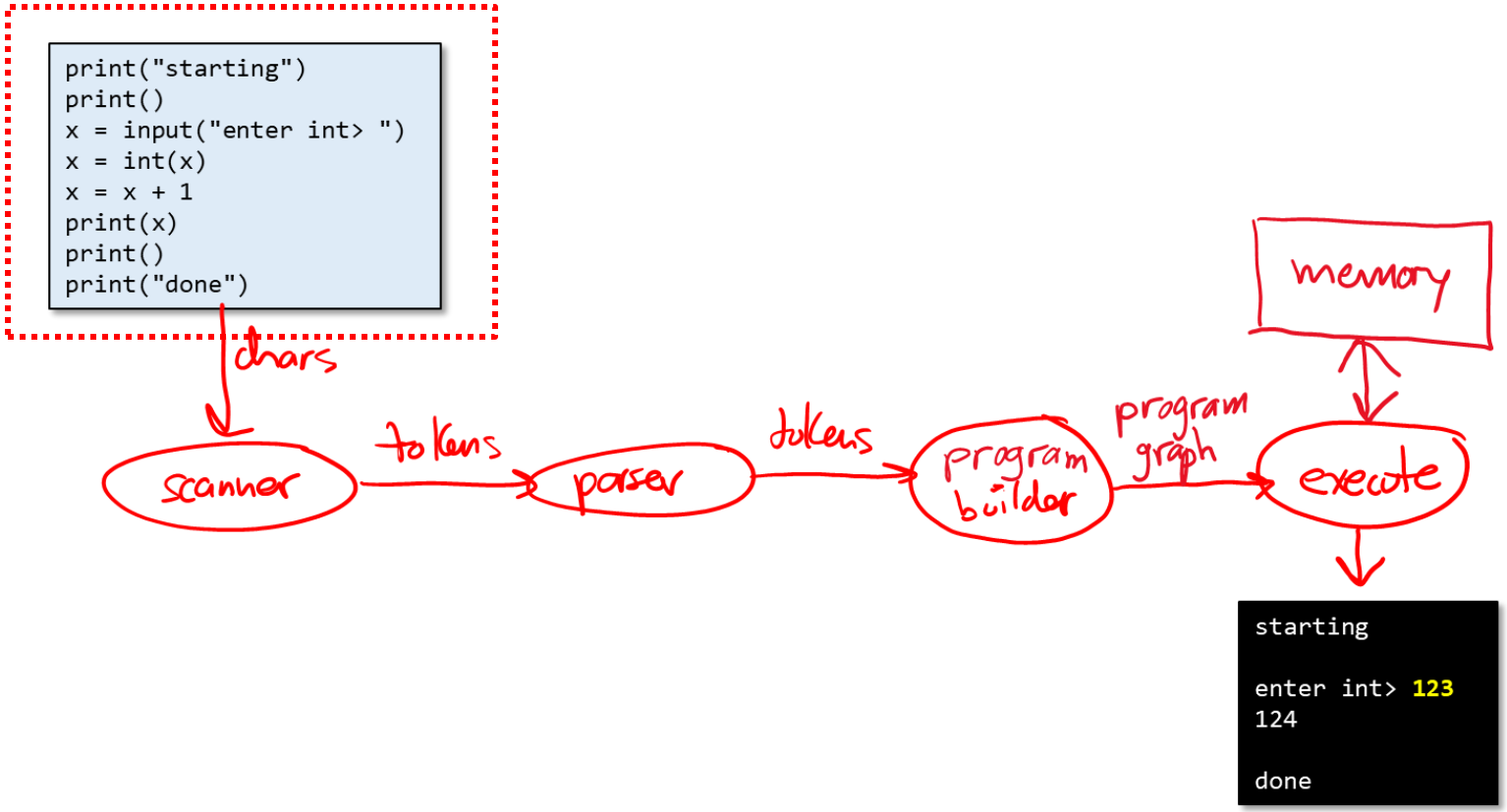
- **Why a graph? Because it may have cycles…**

```
print("starting")

x = 1
while x <= 10:
{
   print(x)
   x = x + 1
}

print("done")
```



nuPython

- **What subset of Python are we supporting?**
- **How is this subset defined?**



BNF (Backus Naur Form)

```
<program> ::= <stmts> EOS
<stmts>   ::= <stmt> [<stmts>]
<stmt>    ::= <assignment>
            | <function_call>
            | <if_then_else>
            | <while_loop>
            | pass

<assignment>    ::= ['*'] IDENTIFIER '=' <value>
<function_call> ::= IDENTIFIER '(' [<element>] ')'
<while_loop>    ::= while <expr> ':' <body>
<if_then_else>  ::= if <expr> ':' <body> [<else>]
<else>          ::= elif <expr> ':' <body> [<else>]
                  | else ':' <body>
<body>          ::= '{' <stmts> '}'

<value>       ::= <function_call>
                | <expr>
<expr>        ::= <unary_expr> [<op> <unary_expr>]
<unary_expr>  ::= '*' IDENTIFIER
                | '&' IDENTIFIER
                | '+' [IDENTIFIER | INT_LITERAL | REAL_LITERAL]
                | '-' [IDENTIFIER | INT_LITERAL | REAL_LITERAL]
                | <element>
<element>     ::= IDENTIFIER
                | INT_LITERAL
                | REAL_LITERAL
                | STR_LITERAL
                | True
                | False
                | None
```

```
print("starting")
print()

x = input("enter int> ")
x = int(x)
x = x + 1
print(x)

print()
print("done")
```

```
<op> ::= '+'
       | '-'
       | '*'
       | '**'
       | '%'
       | '/'
       | '=='
       | '!='
       | '<'
       | '<='
       | '>'
       | '>='
       | IS
       | IN
```

What should I be working on?

HW 02 *is due tonight…*

Project 02 *is due Friday night…*