Term Project
TA Session

[CS 420] Compiler Design

TA Kyuho Son

Homeworks in this course

- HW1 (Predictive parser) – done

- Term project
  - Presentation : Management plan (29th Oct)
  - Report : Internal data structure (12th Nov or 19th Nov)
  - Final presentation (right before the final exam)
  - Final report : document and program (end of semester)

- HW5 (Memory management) - canceled

Debugger Implementation

- Goal
  - Mini-C language debugger
    - Mini-C?
    - Scope : enough to handle the sample code (subset of C89)
  - Features
    - Interpretation
    - Built-in function (printf without header include)
    - Debug CLI commands

Interpreter Implementation

- Interpretation
  - Typical interpreter
    - Building AST in run-time and execution
    - No trace feature
  - For term project scope
    - AST building : Your choice
    - Should have the feature of tracing values of variables
    - More like ‘debugger’

Interpreter Implementation

Example input code

```
1  int avg(int count, int *value) {
2    int i, total;;
3    for (i = 1; i < count; i++) {
4      total = total + value[i];
5     }
6
7    return (total / count);
8  }
9
10 int main(void) {
11   int studentNumber, count, i, sum;
12   int mark[4];
13   float average;
14
15   count = 4;
16   sum = 0;
17
18   for (i = 0; i < count; i++) {
19     mark[i] = i * 30;
20     sum = sum + mark[i];
21     average = avg(i + 1, mark);
22     if (average > 40) {
23       printf("%f\n", average);
24     }
25   }
26
27 }
```

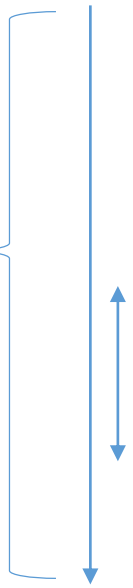
- Implementation scope
  - Variable types (int, float)
  - Variable declaration
  - Variable assignment
  - Calculation (+ , - , *, /, + +)
  - Comparison (>, <)
  - Type casting (int ↔ float)
  - Flow control (for, if)
  - Pointer
  - Function call and return
  - 1-dim array
  - printf(); function (as a built-in)
  - brackets…

Interpreter Implementation

- CLI Commands
  - next [line number]
    - The line number of statements are executed
  - print [variable name]
    - print the value of the variable in current scope
  - trace [variable name]
    - print the history of values of the variable in current scope

Interpreter Implementation

- Terminology
  - Line number
    - ➢ Meaning ①
      : *code line*
    - ➢ Meaning ②
      : *execution lines*

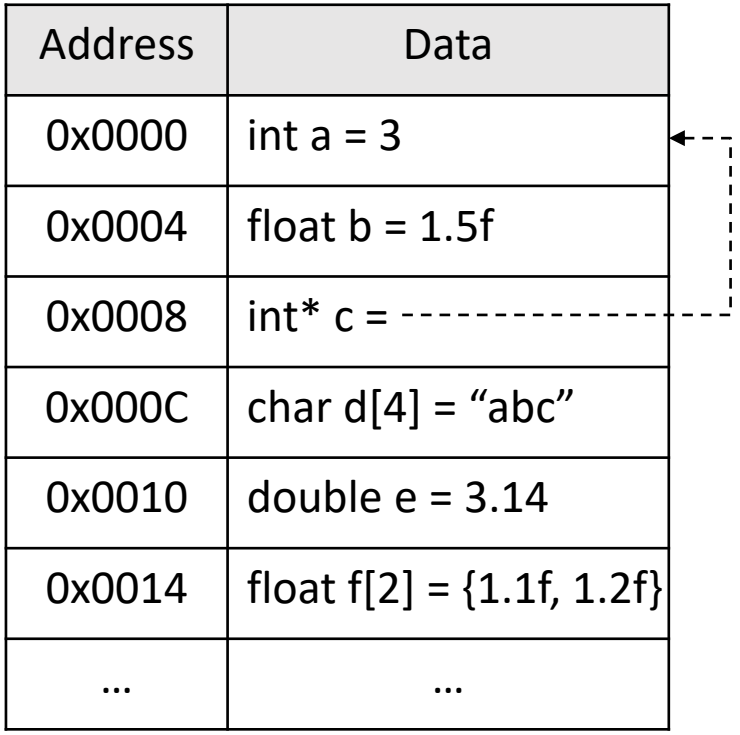


Example input code

```
1  int avg(int count, int *value) {
2    int i, total;;
3    for (i = 1; i < count; i++) {
4      total = total + value[i];
5     }
6
7    return (total / count);
8  }
9
10 int main(void) {
11   int studentNumber, count, i, sum;
12   int mark[4];
13   float average;
14
15   count = 4;
16   sum = 0;
17
18   for (i = 0; i < count; i++) {
19     mark[i] = i * 30;
20     sum = sum + mark[i];
21     average = avg(i + 1, mark);
22     if (average > 40) {
23       printf("%f\n", average);
24     }
25   }
26
27 }
```

Interpreter Implementation

- Terminology
  - Value
    - a = 3
    - b = 1.5
    - c = 0x0000
    - d = 0x000C
    - e = 3.14
    - f = 0x0014
    - *c = 3
    - d[2] = ‘c’
    - d[3] = null character
    - f[0] = 1.1



Address	Data
0x0000	int a = 3
0x0004	float b = 1.5f
0x0008	int* c =
0x000C	char d[4] = “abc”
0x0010	double e = 3.14
0x0014	float f[2] = {1.1f, 1.2f}
…	…

Interpreter Implementation

- Terminology
  - Scope
    - ➢ Visibility of the variable

      (Visible / Invisible)

Example input code

```
 1  int avg(int count, int *value) {
 2    int i, total;;
 3    for (i = 1; i < count; i++) {
 4      total = total + value[i];
 5     }
 6
 7    return (total / count);
 8  }
 9
10  int main(void) {
11    int studentNumber, count, i, sum;
12    int mark[4];
13    float average;
14
15    count = 4;
16    sum = 0;
17
18    for (i = 0; i < count; i++) {
19      mark[i] = i * 30;
20      sum = sum + mark[i];
21      average = avg(i + 1, mark);
22      if (average > 40) {
23        printf("%f\n", average);
24      }
25    }
26
27  }
```

Interpreter Implementation

- Terminology
  - Scope
    - Scope of var *i*

Example input code

```
 1  int avg(int count, int *value) {
 2    int i, total;
 3    int sum = 0;
 4    for (i = 1; i < count; i++) {
 5      total = total + value[i];
 6     }
 7
 8    return (total / count);
 9  }
10
11  int main(void) {
12    int studentNumber, count, i, sum;
13    int mark[4];
14    float average;
15
16    count = 4;
17    sum = 0;
18
19    for (i = 0; i < count; i++) {
20      mark[i] = i * 30;
21      sum = sum + mark[i];
22      average = avg(i + 1, mark);
23      if (average > 40) {
24        printf("%f\n", average);
25      }
26    }
27
28  }
```

Interpreter Implementation

- Terminology
  - Scope
    - Scope of var *total*

Invisible

Example input code

```
 1  int avg(int count, int *value) {
 2    int i, total;
 3    for (i = 1; i < count; i++) {
 4      total = total + value[i];
 5     }
 6
 7    return (total / count);
 8  }
 9
10  int main(void) {
11    int studentNumber, count, i, sum;
12    int mark[4];
13    float average;
14
15    count = 4;
16    sum = 0;
17
18    for (i = 0; i < count; i++) {
19      mark[i] = i * 30;
20      sum = sum + mark[i];
21      average = avg(i + 1, mark);
22      if (average > 40) {
23        printf("%f\n", average);
24      }
25    }
26
27  }
```

Interpreter Implementation

- Terminology
  - Scope
    - Scope of var *sum*

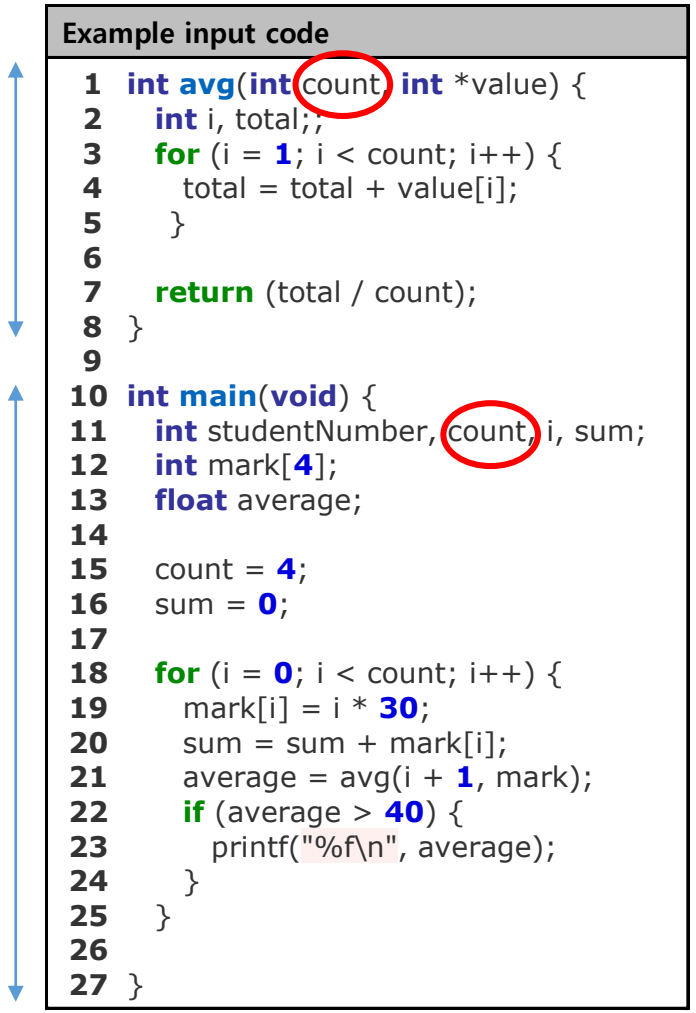
Invisible

Example input code

```
1  int avg(int count, int *value) {
2    int i, total;;
3    for (i = 1; i < count; i++) {
4      total = total + value[i];
5     }
6
7    return (total / count);
8  }
9
10 int main(void) {
11   int studentNumber, count, i, sum;
12   int mark[4];
13   float average;
14
15   count = 4;
16   sum = 0;
17
18   for (i = 0; i < count; i++) {
19     mark[i] = i * 30;
20     sum = sum + mark[i];
21     average = avg(i + 1, mark);
22     if (average > 40) {
23       printf("%f\n", average);
24     }
25   }
26
27 }
```

Interpreter Implementation

- Terminology
  - Scope
    - Scope of var *count*



Interpreter Implementation

- Terminology
  - Scope
    - Scope of var *studentNumber*

Invisible

Example input code

```
1  int avg(int count, int *value) {
2    int i, total;;
3    for (i = 1; i < count; i++) {
4      total = total + value[i];
5     }
6
7    return (total / count);
8  }
9
10 int main(void) {
11   int studentNumber, count, i, sum;
12   int mark[4];
13   float average;
14
15   count = 4;
16   sum = 0;
17
18   for (i = 0; i < count; i++) {
19     mark[i] = i * 30;
20     sum = sum + mark[i];
21     average = avg(i + 1, mark);
22     if (average > 40) {
23       printf("%f\n", average);
24     }
25   }
26
27 }
```

Interpreter Implementation

- Terminology
  - Scope
    - Scope of var *stdev*

Invisible

Example input code

```
 1  int avg(int count, int *value) {
 2    int i, total;;
 3    for (i = 1; i < count; i++) {
 4      total = total + value[i];
 5     }
 6
 7    return (total / count);
 8  }
 9
10  int main(void) {
11    int studentNumber, count, i, sum;
12    int mark[4];
13    float average;
14
15    count = 4;
16    sum = 0;
17
18    for (i = 0; i < count; i++) {
19      mark[i] = i * 30;
20      sum = sum + mark[i];
21      average = avg(i + 1, mark);
22      if (average > 40) {
23        printf("%f\n", average);
24      }
25    }
26
27  }
```

Interpreter Implementation

- Terminology
  - History
    - Lifetime of history (declaration ~ expiration)

      You do not need to maintain histories of expired variables!

    - Variable declaration (N/A on declaration w/o assignment)
    - Value assignment

Example input code

```
 1  int avg(int count, int *value) {
 2    int i, total;;
 3    for (i = 1; i < count; i++) {
 4      total = total + value[i];
 5     }
 6
 7    return (total / count);
 8  }
 9
10  int main(void) {
11    int studentNumber, count, i, sum;
12    int mark[4];
13    float average;
14
15    count = 4;
16    sum = 0;
17
18    for (i = 0; i < count; i++) {
19      mark[i] = i * 30;
20      sum = sum + mark[i];
21      average = avg(i + 1, mark);
22      if (average > 40) {
23        printf("%f\n", average);
24      }
25    }
26
27  }
```

Interpreter Implementation

- Terminology
  - History of *i* in this line

Meaning ②

Meaning ①

Code line	Value
2	N/A
4	1
4	2
4	3
…	…

Example input code

```
 1  int avg(int count, int *value) {
 2    int i, total;;
 3    for (i = 1; i < count; i++) {
 4      total = total + value[i];
 5     }
 6
 7    return (total / count);
 8  }
 9
10  int main(void) {
11    int studentNumber, count, i, sum;
12    int mark[4];
13    float average;
14
15    count = 4;
16    sum = 0;
17
18    for (i = 0; i < count; i++) {
19      mark[i] = i * 30;
20      sum = sum + mark[i];
21      average = avg(i + 1, mark);
22      if (average > 40) {
23        printf("%f\n", average);
24      }
25    }
26
27  }
```

Optional features

- Need to knows
  - Every team should choose one option
  - At most 3 teams for each option in most cases
  - Relative grading in each option
- Choices
  - Memory management
  - Recursive function call
  - Code generation
  - Code optimization
  - Error handling

Optional features

- Memory management
  - For both of static and dynamic allocation
  - Allocating and deallocating memory spaces (1Kbyte linear memory structure)
  - Defragmentation would be needed to pass the sample and grading code
  - Additional built-in function : malloc, free
  - Additional CLI command : mem
    - Print memory allocation

Optional features

- Recursive function call
  - A correct execution of recursive function
  - At least for 2 cases : A(A(A(A(…)))) , A(B(A(B(…))))
  - If the broader scope of cases are accepted, the more score you will get
- Code generation
  - The real assembly should be generated before runtime
  - Additional output : assembly code
  - Any format is acceptable if it has a form of assembly consists of an operator and operands

Optional features

- Code optimization
  - Optimizations that can be done without code generation
  - Examples : common subexpression detection, dead code elimination, etc.
  - Additional output : optimized code to a text file
- Compile error handling
  - Continue compile after detecting syntax error
  - Detecting as many syntax errors as possible can be a goal
  - But there is not a sort of correct answer, several methods are available

Other things to say

- For all the products from the teamwork, contribution of members should be specified
- TA reviews all source code quite carefully
  (Actually it is necessary to give partial scores for all products)
- Late submission will always be better than nothing
- If your source code does not operate, you will lose most, but still much better than nothing

So, do your best!!

QnA