Ucode-Interpreter
분석과제

과 목 : 컴파일러 02
학 과 : 컴퓨터공학
학 번 : 2020112011
이 름 : 정찬형
제 출 일 : 2025.11.30.

1. 개요

본 보고서는 U-Code Interpreter(ucodei.cpp)의 컴파일 오류를 수정하고, 인터프리터의 구조와 U-Code 문법을 분석하며, 이를 바탕으로 마방진 생성 프로그램을 구현 및 실험한 결과를 기술합니다.

주요 수행 내용은 다음과 같습니다.

1. 소스 코드 수정: ucodei.cpp의 C++ 표준 호환성 문제(문자열 상수 처리, main 함수 반환형)를 해결하여 정상적으로 컴파일되도록 수정하였습니다.

2. 인터프리터 분석: 소스 코드를 분석하여 U-Code 명령어 집합과 동작 방식을 파악하고 문법을 추론하였습니다.

3. 샘플 실험: 제공된 샘플 코드(factorial, bubble, perfect)를 실행하여 인터프리터의 정상 동작을 검증하였습니다.

4. 마방진 구현: 분석된 U-Code 문법을 활용하여 홀수 차수의 마방진을 생성하는 프로그램을 작성하고 실험하였습니다.

2. 수정 사항

2.1. 문자열 상수 타입 불일치 수정 (-Wwrite-strings)

문제점: ISO C++에서는 문자열 상수(string constant)를 char*로 변환하는 것을 금지합니다. 기존 코드에서는 opcodeName 배열과 errmsg 함수에서 문자열 상수를 char* 타입으로 다루고 있어 경고가 발생했습니다.

수정 내용: opcodeName 배열의 선언을 char*에서 const char*로 변경하였습니다.

```
// 변경 전
char* opcodeName[NO_OPCODES] = { ... };

// 변경 후
const char* opcodeName[NO_OPCODES] = { ... };
```

errmsg 함수의 매개변수 타입을 char*에서 const char*로 변경하였습니다.

```
// 변경 전
void errmsg(char* s, char* s2 = "") { ... }

// 변경 후
void errmsg(const char* s, const char* s2 = "") { ... }
```

2.2. main 함수 반환 타입 수정

문제점: 표준 C++에서 main 함수는 반드시 int를 반환해야 합니다. 기존 코드의 void main은 비표준이므로 오류가 발생했습니다.

수정 내용: main 함수의 반환 타입을 void에서 int로 변경하고, 함수 종료 시 return 0; 를 추가하였습니다.

```
// 변경 전
void main(int argc, char* argv[]) {
    // ...
}

// 변경 후
int main(int argc, char* argv[]) {
    // ...
    return 0;
}
```

3. U-Code Interpreter 분석 및 문법 추론

3.1. ucodei.cpp 분석

ucodei.cpp는 U-Code 어셈블리어를 읽어 기계어(가상 코드)로 변환하는 **Assemble** 단계와, 이를 실행하는 **Interpret** 단계로 구성되어 있습니다.

명령어 정의 (opcodeName, executable, opcodeCycle):

opcodeName: U-Code 명령어의 니모닉(mnemonic) 문자열 배열입니다. (예: “notop”, “add”, “lod” 등)
executable: 각 명령어가 실행 가능한지 여부를 나타냅니다. nop, sym 등은 실행되지 않는 명령어입니다.
opcodeCycle: 각 명령어 실행에 소요되는 가상 사이클 수입니다. 성능 측정에 사용됩니다.

Assemble 클래스:
소스 파일을 한 줄씩 읽어 파싱합니다.
getLabel() : 라벨을 파싱하고 심볼 테이블(Label 클래스)에 등록합니다.
getOpcode() : 명령어 니모닉을 읽어 내부 정수 코드(enum opcode)로 변환합니다.
getOperand(): 명령어에 따라 필요한 피연산자를 읽어옵니다.

Interpret 클래스:
스택 기반 가상 머신입니다.
Stack : 데이터 및 제어 정보를 저장하는 스택입니다.
execute(): 변환된 명령어를 순차적으로 실행합니다. switch-case 문을 통해 각 Opcode에 맞는 동작을 수행합니다.

3.2. 문법 추론 과정 및 결과

소스 코드의 Assemble::assemble() 함수와 Assemble::getOpcode(), Assemble::getOperand() 함수를 분석하여 U-Code의 문법을 추측했습니다.

3.2.1. 명령어 형식

assemble() 함수의 파싱 로직에 따르면, U-Code 명령어는 다음과 같은 형식을 가집니다.

Label Opcode Operand1 Operand2 Operand3

- **Label**: 선택 사항이며, 점프 명령어의 대상이 됩니다.
- **Opcode**: 명령어 코드 (예: lod, str, add 등)
- **Operand**: 명령어에 따라 0개에서 3개까지의 피연산자를 가집니다.

3.2.2. 명령어 상세 목록 (분석결과)

switch(n) 문과 execute() 함수의 동작을 분석하여 각 명령어의 피연산자 개수와 동작을 정리하였습니다.

1) 산술 및 논리 연산 (Arithmetic & Logical) 스택의 상위 요소들을 사용하여 연산을 수행하고 결과를 스택에 푸시합니다.

명령어	피연산자	설명	동작 (Stack)
notop	0	논리 부정 (Logical NOT)	push(!pop())
neg	0	부호 반전 (Negation)	push(-pop())
add	0	덧셈	v2=pop, v1=pop, push(v1 + v2)
sub	0	뺄셈	v2=pop, v1=pop, push(v1 - v2)
mult	0	곱셈	v2=pop, v1=pop, push(v1 * v2)
div	0	나눗셈 (몫)	v2=pop, v1=pop, push(v1 / v2)
mod	0	나머지 연산	v2=pop, v1=pop, push(v1 % v2)
and	0	논리 곱 (AND)	v2=pop, v1=pop, push(v1 & v2)
or	0	논리 합 (OR)	v2=pop, v1=pop, push(v1 ¦ v2)
gt	0	크다 (Greater Than)	v2=pop, v1=pop, push(v1 > v2)
lt	0	작다 (Less Than)	v2=pop, v1=pop, push(v1 < v2)
ge	0	크거나 같다 (Greater Equal)	v2=pop, v1=pop, push(v1 >= v2)
le	0	작거나 같다 (Less Equal)	v2=pop, v1=pop, push(v1 <= v2)
eq	0	같다 (Equal)	v2=pop, v1=pop, push(v1 == v2)
ne	0	같지 않다 (Not Equal)	v2=pop, v1=pop, push(v1 != v2)
inc	0	증가 (Increment)	v=pop, push(++v)
dec	0	감소 (Decrement)	v=pop, push(--v)

2) 스택 조작 (Stack Manipulation)

명령어	피연산자	설명	동작
dup	0	스택 상단 요소 복제	v=pop, push(v), push(v)
swp	0	스택 상단 두 요소 교환	v2=pop, v1=pop, push(v2), push(v1)
ldc	1	상수 로드 (Load Constant)	push(Operand1)

3) 메모리 및 변수 접근 (Memory & Variable Access)

변수는 정적 깊이 차이와 Offset으로 접근합니다.

명령어	피연산자	설명	동작
lod	2	변수 값 로드 (Load)	push(Stack[Addr(Op1, Op2)])
str	2	변수 값 저장 (Store)	Stack[Addr(Op1, Op2)] = pop()
lda	2	변수 주소 로드 (Load Address)	push(Addr(Op1, Op2))
ldi	0	간접 로드 (Load Indirect)	addr=pop, push(Stack[addr])
sti	0	간접 저장 (Store Indirect)	val=pop, addr=pop, Stack[addr]=val

4) 제어 흐름 (Control Flow)

명령어	피연산자	설명	동작
ujp	1	무조건 점프 (Unconditional Jump)	PC = LabelAddress
tjp	1	참일 때 점프 (True Jump)	if (pop()) PC = LabelAddress
fjp	1	거짓일 때 점프 (False Jump)	if (!pop()) PC = LabelAddress

chkh	1	상한 체크 (Check High)	v=pop, if(v > Op1) Error, push(v)
chkl	1	하한 체크 (Check Low)	v=pop, if(v < Op1) Error, push(v)
nop	0	아무 동작 안 함 (No Operation)	-

5) 프로시저 및 함수 (Procedure & Function)

명령어	피연산자	설명	동작
ldp	0	프로시저 호출 준비 (Load Parameters)	스택 프레임 준비
call	1	프로시저 호출 (Call)	PC = LabelAddress (Return Addr 저장)
ret	0	프로시저 복귀 (Return)	호출자에게 복귀, 스택 프레임 해제
retv	0	값 반환 복귀 (Return Value)	val=pop, 복귀 후 push(val)
proc	3	프로시저 시작 선언	스택 프레임 초기화
bgn	1	프로그램 시작	SP = SP + Op1 (전역 변수 공간 확보)
end	0	프로그램 종료	실행 종료

6) 미리 정의된 프로시저 (Predefined Procedures)

call 명령어로 호출할 수 있는 내장 함수들입니다.

read: 정수 입력 (call read)

write: 정수 출력 (call write)

lf: 줄바꿈 출력 (call lf)

4. 샘플 프로그램 실험

제공된 Samples 디렉토리의 예제 코드 중 3개를 선택하여 실행하고 결과를 확인하였습니다.

4.1. 실험 1: factorial.uco (팩토리얼 계산)

- **개요**: 재귀 호출을 사용하여 입력된 수의 팩토리얼을 계산합니다.
- **입력**: 5
- **실행 명령**: ./intp Samples/factorial.uco output_fact.txt < input_fact.txt
- **결과**: 5 120 5! = 120이 정확히 계산되었습니다.

4.2. 실험 2: bubble.uco (버블 정렬)

- **개요**: 입력된 숫자들을 버블 정렬 알고리즘으로 오름차순 정렬합니다. 0이 입력되면 입력을 종료하고 정렬을 시작합니다.
- **입력**: 5 3 8 1 2 0
- **실행 명령**: ./intp Samples/bubble.uco output_bubble.txt < input_bubble.txt
- **결과**: 1 2 3 5 8 입력된 숫자들이 올바르게 정렬되었습니다.

4.3. 실험 3: perfect.uco (완전수 찾기)

- **개요**: 1부터 500까지의 수 중에서 완전수(자신을 제외한 약수의 합이 자신과 같은 수)를 찾아 출력합니다.
- **입력**: 없음 (상수 max = 500 사용)
- **실행 명령**: ./intp Samples/perfect.uco output_perfect.txt
- **결과**: 6 28 496 500 이하의 완전수인 6, 28, 496이 정확히 출력되었습니다.

5. 마방진(Magic Square) 프로그램 작성 및 실험

5.1. 개요

U-Code를 사용하여 홀수 크기(3~15)의 마방진을 생성하는 프로그램을 작성하고 실험하였습니다.

5.2. 프로그램 로직 (magic.mc)

Mini C 언어로 작성된 마방진 생성 알고리즘은 다음과 같습니다.

1. 사용자로부터 정수 n을 입력받습니다.
2. n이 0이면 프로그램을 종료합니다.
3. n이 짝수이면 입력을 무시하고 다시 입력받습니다.
4. n이 홀수이면 Siamese method를 사용하여 마방진을 생성합니다.

5.2.1. 마방진 생성 알고리즘

홀수 차수 n의 마방진을 생성하는 Siamese method는 다음과 같은 규칙을 따릅니다.

1. **시작 위치**: 첫 번째 숫자(1)를 첫 행의 가운데 열(row = 0, col = n/2)에 배치합니다.

2. **이동 규칙**: 다음 숫자는 현재 위치에서 **오른쪽 위 대각선 방향**(row - 1, col + 1)으로 이동하여 배치합니다.

3. **경계 처리 (Wrap-around)**: * 행이 0보다 작아지면 마지막 행(n - 1)으로 이동합니다. * 열이 n 이상이 되면 첫 번째 열(0)로 이동합니다.

4. **충돌 처리**: 이동하려는 위치에 이미 숫자가 채워져 있다면, 이전 위치의 **바로 아래 행**(row + 1, col)에 숫자를 배치합니다. 5. 위 과정을 n^2까지 반복합니다.

5.3. U-Code 구현 (magic.uco)

위의 로직을 U-Code로 변환하여 magic.uco 파일을 작성하였습니다. 2차원 배열 접근은 row * 15 + col 공식을 사용하여 1차원 배열로 시뮬레이션하였습니다.

5.4. 실험 결과

다음과 같은 입력값으로 실험을 수행하였습니다. * 입력: 3 (3x3 마방진 출력 예상) * 입력: 4 (무시됨) * 입력: 5 (5x5 마방진 출력 예상) * 입력: 0 (종료)

실행 명령:

```
./intp magic.uco output.txt < input.txt
```

실행 결과:

```
 == Assembling ... ==
 == Executing ...  ==
 == Result         ==
 8 1 6
 3 5 7
 4 9 2
 17 24 1 8 15
 23 5 7 14 16
 4 6 13 20 22
 10 12 19 21 3
 11 18 25 2 9
```

3x3 및 5x5 마방진이 정상적으로 출력되었으며, 짝수 입력은 무시되고 0 입력 시 프로그램이 종료됨을 확인하였습니다.

[붙임 1] magic.uco

```
main        proc            235   2     2
            sym             2     1     1
            sym             2     2     1
            sym             2     3     1
            sym             2     4     1
            sym             2     5     1
            sym             2     6     1
            sym             2     7     1
            sym             2     8     1
            sym             2     9     1
            sym             2     10    225
$$LOOP      nop
            ldp
            lda             2     1
            call            read
            lod             2     1
            ldc             0
            eq
            tjp             $$EXIT
            lod             2     1
            ldc             2
            mod
            ldc             0
            eq
            tjp             $$LOOP
            ldc             0
            str             2     2
$$INIT      nop
            lod             2     2
```

```
            ldc             225
            ge
            tjp             $$INIT_E
            lda             2     10
            lod             2     2
            add
            ldc             0
            sti
            lod             2     2
            inc
            str             2     2
            ujp             $$INIT
$$INIT_E    nop
            ldc             0
            str             2     4
            lod             2     1
            ldc             2
            div
            str             2     5
            ldc             1
            str             2     6
            lod             2     1
            lod             2     1
            mult
            str             2     7
$$FILL      nop
            lod             2     6
            lod             2     7
            gt
            tjp             $$PRINT
```

```
            lda         2     10
            lod         2     4
            ldc         15
            mult
            lod         2     5
            add
            add
            lod         2     6
            sti
            lod         2     6
            inc
            str         2     6
            lod         2     4
            str         2     8
            lod         2     5
            str         2     9
            lod         2     4
            dec
            str         2     4
            lod         2     5
            inc
            str         2     5
            lod         2     4
            ldc         0
            lt
            fjp         $$CHK_C
            lod         2     1
            dec
            str         2     4
$$CHK_C     nop
```

```
            lod             2     5
            lod             2     1
            ge
            fjp             $$CHK_O
            ldc             0
            str             2     5
$$CHK_O     nop
            lda             2     10
            lod             2     4
            ldc             15
            mult
            lod             2     5
            add
            add
            ldi
            ldc             0
            ne
            fjp             $$FILL
            lod             2     8
            inc
            str             2     4
            lod             2     4
            lod             2     1
            eq
            fjp             $$SKIP_W
            ldc             0
            str             2     4
$$SKIP_W    nop
            lod             2     9
            str             2     5
```

```
           ujp            $$FILL
$$PRINT    nop
           ldc            0
           str            2     2
$$PR_R     nop
           lod            2     2
           lod            2     1
           ge
           tjp            $$PR_D
           ldc            0
           str            2     3
$$PR_C     nop
           lod            2     3
           lod            2     1
           ge
           tjp            $$PR_RE
           ldp
           lda            2     10
           lod            2     2
           ldc            15
           mult
           lod            2     3
           add
           add
           ldi
           call           write
           lod            2     3
           inc
           str            2     3
           ujp            $$PR_C
```

```
$$PR_RE    nop
           ldp
           call          lf
           lod           2     2
           inc
           str           2     2
           ujp           $$PR_R
$$PR_D     nop
           ujp           $$LOOP
$$EXIT     nop
           ret
           bgn           0
           ldp
           call          main
           end
```