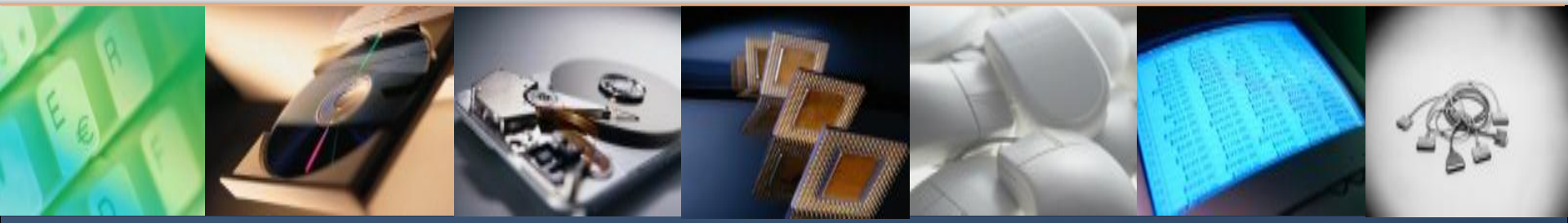


제 12 장

중간코드와 미니컴파일러

교육목표

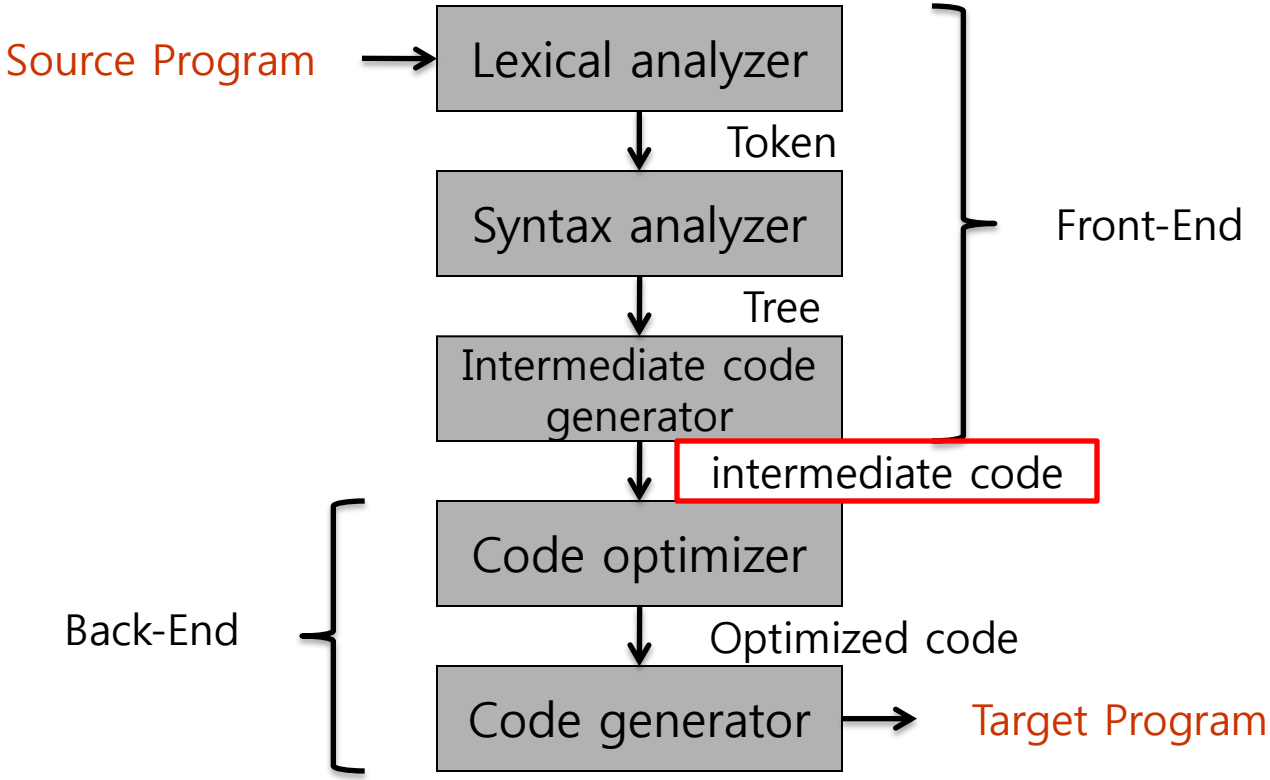
- 중간 코드의 개념 이해
- 중간코드를 사용한 Mini C 컴파일러 작성

교과내용

- 중간 코드의 개념
- 중간 코드 예시 : U-Code
- 텀 프로젝트 : Mini C 컴파일러 개발
- Mini C의 개념

중간 코드 (Intermediate code)

- **컴파일러의 전반부와 후반부를 연결**
  - 자동화된 전반부와 플랫폼 의존적인 후반부의 중간 역할



중간 코드 사용의 특징

- **장점**
  - 컴파일러를 독립적인 여러 모듈들로 구성
  - 이식성 증가
  - 번역과정을 쉽게 표현 가능 (고급 언어 -> 저급 언어)
  - 기계에 독립적이고 효율적인 최적화 가능
  - 인터프리티브 컴파일링 사용 가능
    - 중간언어를 직접 실행하는 시스템

- **단점**
  - 컴파일 시간 증가
  - 비효율적 코드 생성 가능

중간 코드의 종류

- **Polish 표기법**
  - prefix : 연산자가 먼저 나오고 피연산자가 뒤에 나옴
    - A = B + C * D / E → = + / E * D C B A
  - postfix : 피연산자가 먼저 나오고 연산자가 나옴
    - A = B + C * D / E → A B C D * E / + =

- **3-address 코드 (N-tuple 표기법)**
  - 상용 컴파일러에서 가장 널리 사용
    - 특히 최적화 컴파일러에 많이 사용
    - GNU C/C++의 중간언어인 RTL(Register Transfer Language), 단 내부 표현은 트리
  - 레지스터 기계를 위한 코드로 쉽게 번역
  - 실제 실현 방법
    - triple : (operator, $operand_1$, $operand_2$)
    - quadruple : (operator, $operand_1$, $operand_2$, 결과)

중간 코드의 종류

- **트리 구조 코드 (Tree structured code)**
  - 프로그램 의미를 효과적으로 표현하며 재구성이 쉬움
    - 최적화 컴파일러에 가장 적합한 표현
  - AST, TCOL (Tree structured Common Language), Diana
  - AST는 high-level 트리 코드로서, 이후 의미 분석 과정에 사용
- **가상 기계 코드 (Abstract machine code)**
  - Retargeting을 하기 위한 가상 스택 머신
  - 컴파일러의 전단부와 후단부의 완전한 분리 가능
  - P-code, EM, U-code, Bytecode, .NET IL
- **특징에 따라 여러 단계의 중간 코드 도입하는 추세**

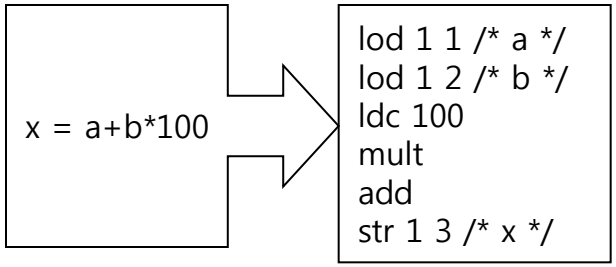
U-code의 개요

- **포터블 파스칼 컴파일러에서 사용한 중간코드**
  - 스탠포드 대학 제작
- **가상 스택 기계 언어**
  - 모든 연산을 스택 기준으로 처리
  - 변수(스택 주소를 참조) ::= 스택의주소 (B, O)
    - (B(블록 번호), O(블록 시작점으로부터의 위치)
- **파스칼 언어 기반**
  - Mini C 를 기준으로 수정해 사용
- **변수의 효력범위 (scope)**
  - compound statement ({ … }) 내에 변수 선언 불가라면 블록 번호는 고정
    - 전역 변수 : 1, 지역 변수 : 2

U-code의 명령어(1/6)

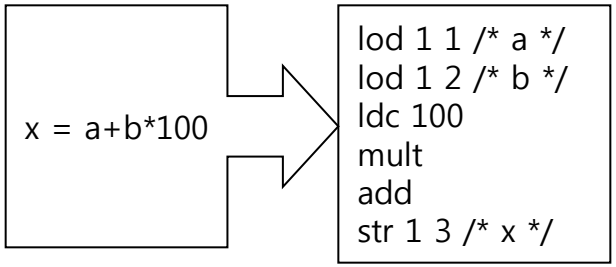
- **단일 명령어 : 피연산자를 하나만 사용, 피연산자는 스택의 top**
  - notop : 논리 값의 역을 계산 (not)
  - neg : 정수값의 음수를 계산
  - inc, dec : 값을 증가, 감소
  - dup : 스택 top의 값을 복사하여 스택에 추가

- **이진 명령어 : 두개의 피연산자를 사용**
  - 피연산자는 스택 top-1, top
  - 스택에서 두 피연산자를 제거한 후 1개의 결과만 저장(swp 은 예외)
  - add, sub, mult, div, mod : 두 값의 연산을 수행
  - swp : 두 피연산자의 위치를 교환
  - and, or, gt, lt, ge, le, eq, ne : 두 논리값의 논리 연산을 수행

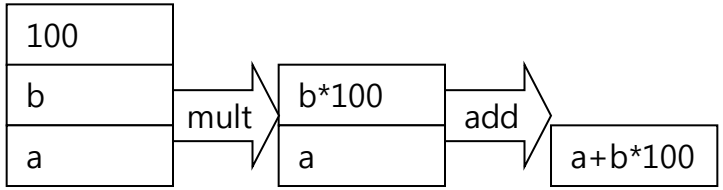


U-code의 명령어(2/6)

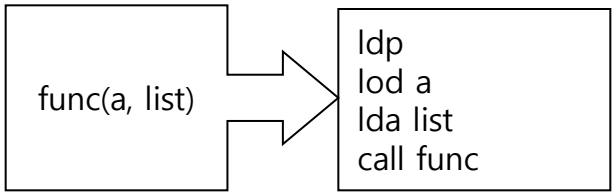
- **스택 운영 명령어 : 하나의 인자를 사용**
  - lod : 변수의 값을 스택의 top에 적재 (lod <var>)
  - str : 스택 top의 내용을 변수에 적제 (str <var>)
  - ldc : 스택의 top에 상수를 적재 (ldc <상수>)
  - lda : 변수의 주소를 스택의 top에 적재 (lda <var>)
  - var 는 (B, O) 로 표현



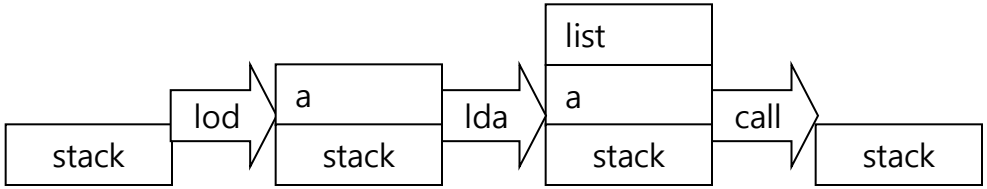
<소스> <ucode>



<연산에 따른 스택 변화>



<소스> <ucode>



<연산에 따른 스택 변화>

U-code의 명령어(3/6)

- **제어 흐름 명령어 (분기 명령)**
  - ujp <label> : 목적지로 무조건 분기
  - tjp <label> : 스택 top이 참인 경우 분기, 분기 후 값 제거
  - fjp <label> : 스택 top이 거짓인 경우 분기, 분기 후 값 제거

if (a>b) a = a + b;

```
        lod   1 1       /* a */
        lod   1 2       /* b */
        gt              /* a>b */
        fjp   next
        lod   1 1       /* a */
        lod   1 2       /* b */
        add
        str   1 1       /* a */
next    …
```

<제어 흐름 명령어의 사용>

- **범위 검사 명령어 : 실행 시간에 배열의 상한과 하한을 검사**
  - chkh <int 상수> : 상한 검사 (stack[top] <= 상수)
  - chkl <int 상수> : 하한 검사

U-code의 명령어(4/6)

- **간접 주소 명령어 : 배열의 값을 접근**
  - 주소를 먼저 계산, 주소가 가리키는 값에 접근
  - ldi(load indirect) : 스택 top을 주소로 그 값을 스택 top에 저장
  - sti(store indirect) : 스택 top의 내용을 top-1의 주소에 저장

a[i]

```
lod   Bi  Oi  /* i */
lda   Ba  Oa  /* a의 주소 */
add
ldi
```

a[i] = j;

```
lod   Bi  Oi  /* i */
lda   Ba  Oa  /* a의 주소 */
add
lod   Bj  Oj  /* j의 주소 */
sti
```

U-code의 명령어(5/6)

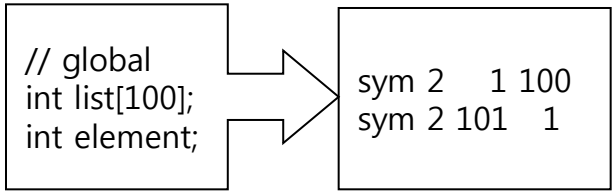
- **함수 명령어**
  - call : <label> 함수를 호출 (call <label>)
  - ret : 호출된 함수로부터 복귀
  - retv : 복귀 값과 함께 호출된 함수로부터 복귀
  - ldp : 스택에 매개변수의 값을 저장하려 할 때 사용
  - proc : 함수의 시작점, 함수의 이름
    - val1 : 지역 변수+매개 변수의 크기
    - val2 : 블록 번호 (2 고정 : 전역 함수)
    - val3 : 렉시칼 레벨 (2 고정)
  - end : 함수의 종료

```
<name>  proc    val1    val2    val3
        ....
        end
```

```
main    proc    3    2    2
        sym     2    0    1
        sym     2    1    1
        sym     2    2    1
        ....
        end
```

U-code의 명령어(6/6)

- 기타 명령어
  - nop : 레이블을 표시하기 위한 명령어
  - bgn : 주 프로그램의 시작점, 외부(전역)변수의 크기를 표시
    - (bgn <정수형 상수>)
  - sym : 심볼테이블에 관한 정보를 표시
    - (sym var1(베이스) var2(오프셋) var3(변수의 크기))



<소스> <ucode>

```
                sym             external variable
func            proc            p1          p2          p3
                sym             local variable for "func"
                ...             code of "func"
                end
                bgn
                ...             code of main
                end
```

<U-Code 프로그램 형태>

시스템 내장 함수

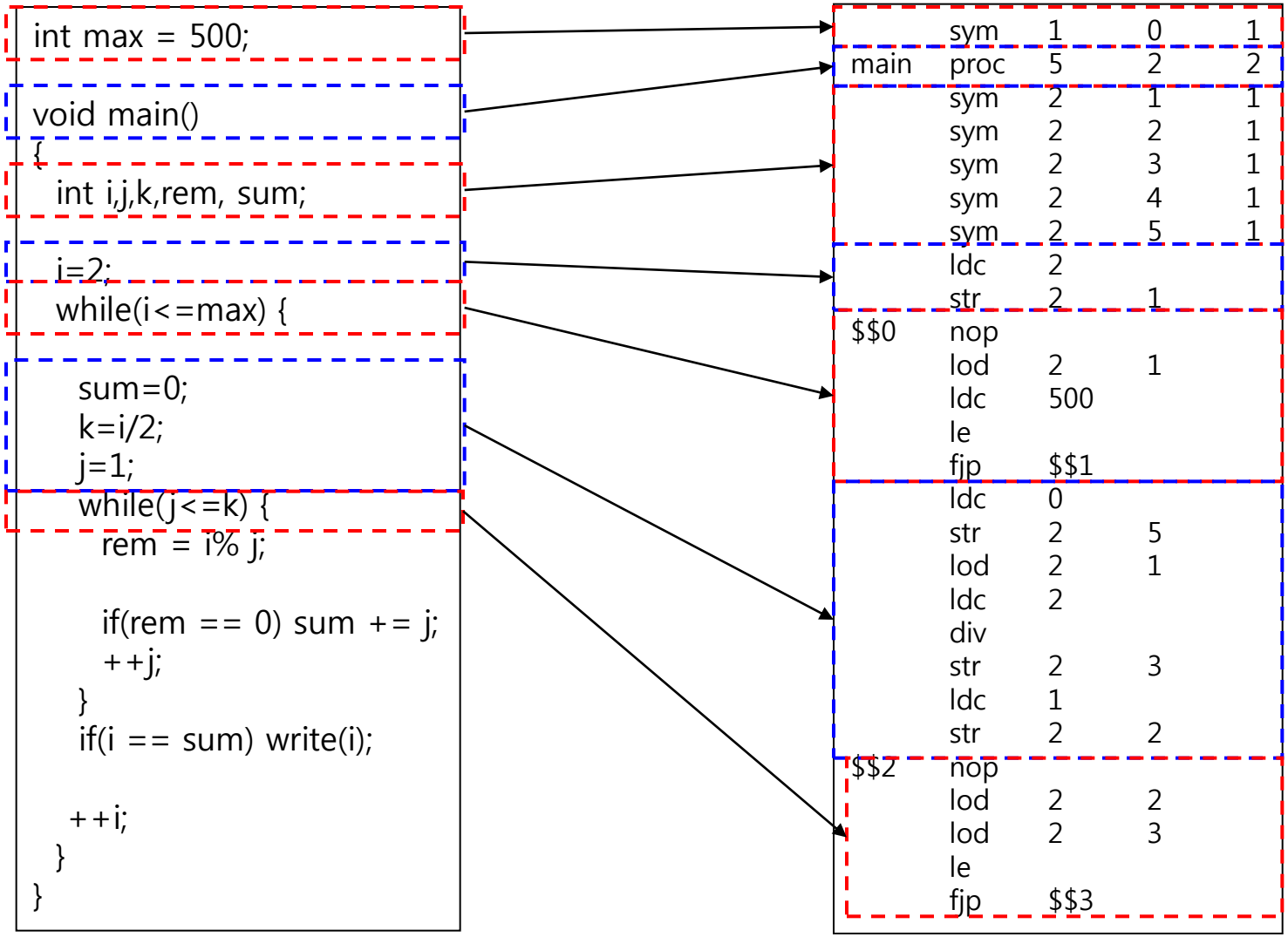
- **입/출력 처리**
  - read

```
ldp
lda     B       0
call    read
```

  - write

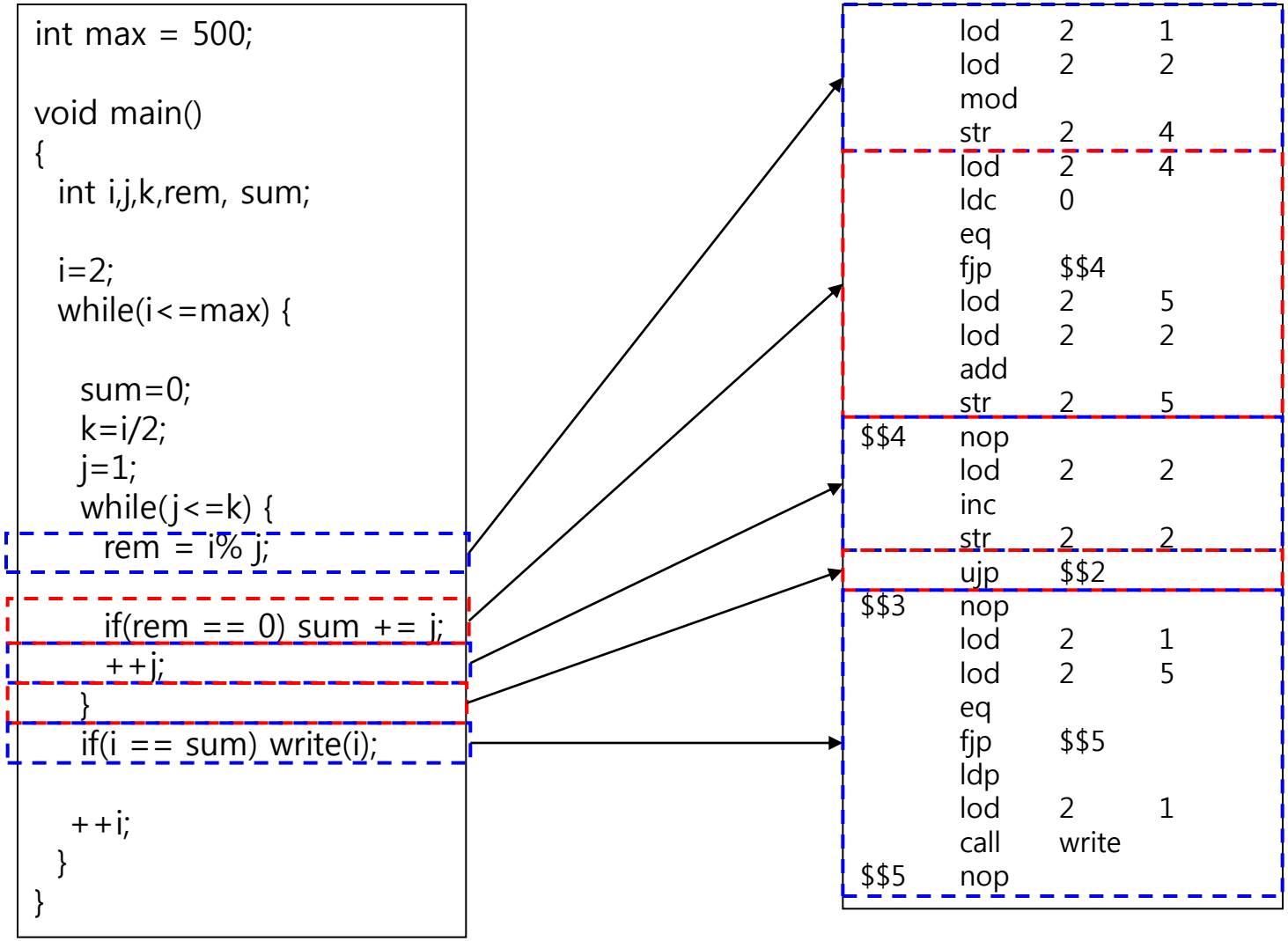
```
ldp
lod     B       0
call    write
```

U-code 번역 예 (1/3)



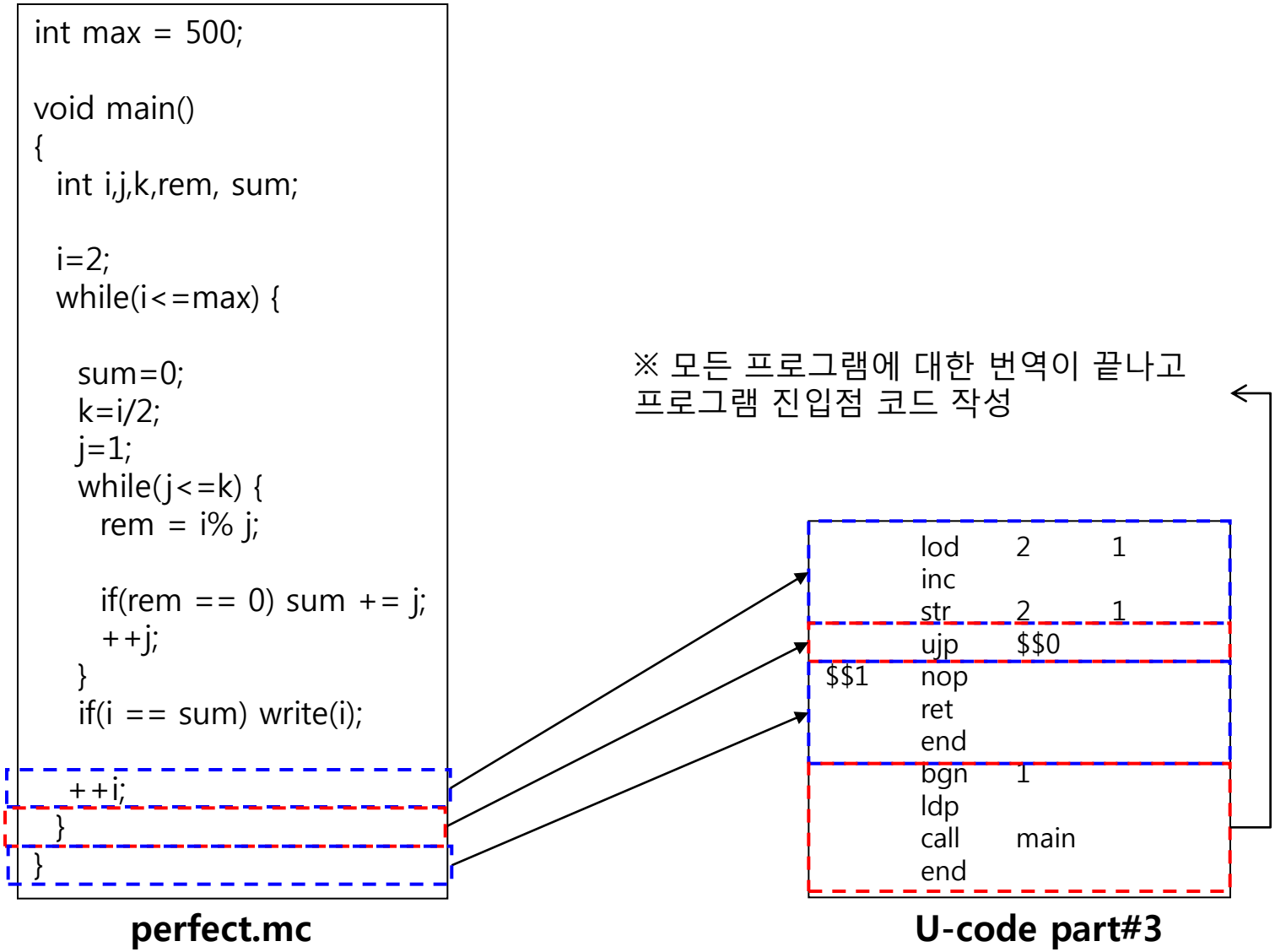
perfect.mc **U-code part#1**

U-code 번역 예 (2/3)



perfect.mc **U-code part#2**

U-code 번역 예 (3/3)


int max = 500;
void main()
{
int i,j,k,rem, sum;
i=2;
while(i<=max) {
sum=0;
k=i/2;
j=1;
while(j<=k) {
rem = i% j;
if(rem == 0) sum += j;
++j;
}
if(i == sum) write(i);
++i;
}
}
perfect.mc
※ 모든 프로그램에 대한 번역이 끝나고
프로그램 진입점 코드 작성
lod 2 1
inc
str 2 1
ujp $$0
$$1 nop
ret
end
bgn 1
ldp
call main
end
U-code part#3

U-code 생성

- **AST를 순환하여 생성**
  - 기본적으로 루트로부터 후위 순환 방식 사용
  - 후위 순환 중에 의미있는 노드 방문 시 자식, 또는 형제 AST노드를 방문하여 필요한 정보 획득

- **U-code 생성 절차**
  - 1. AST 순환
  - 2. AST 노드가 의미있는 노드인지를 검사
  - 3. AST 노드가 의미있는 노드인 경우 (선언, 연산, 호출 등)
    - 3-1. 해당, 관련 노드에 정보 획득
    - 3-2. 필요한 경우 심볼테이블 기록, 또는 참조
    - 3-3. 노드에 맞는 ucode 명령어 기록
  - 4. 모든 노드를 순회할 때까지 반복

U-code 생성

- **U-Code 생성을 위한 라이브러리 함수**
  - void emit0(int opcode)
  - void emit1(int opcode, int op1)
  - void emit2(int opcode, int op1, int op2)
  - void emit3(int opcode, int op1, int op2, int op3)
    - 인자의 개수에 따라 명령어와 인자 출력
  - void emitJump(int opcode, int label)
    - 입력된 label 위치로 점프하는 명령어 출력
  - void emitLabel(int label)
    - 점프를 위한 label 출력
  - void emitFunc(char * name, int op1, int op2, int op3)
    - 함수의 이름과 인자 출력
  - void genLabel()
    - 점프를 위한 새로운 label 생성
  - void rv_emit()
    - 함수의 매개변수가 변수인지 상수인지를 구분해서 lod와 ldc 명령을 구분하여 출력
    - pp.439 참조

실습 예제

- 함수 호출 및 배열 테스트 프로그램 번역

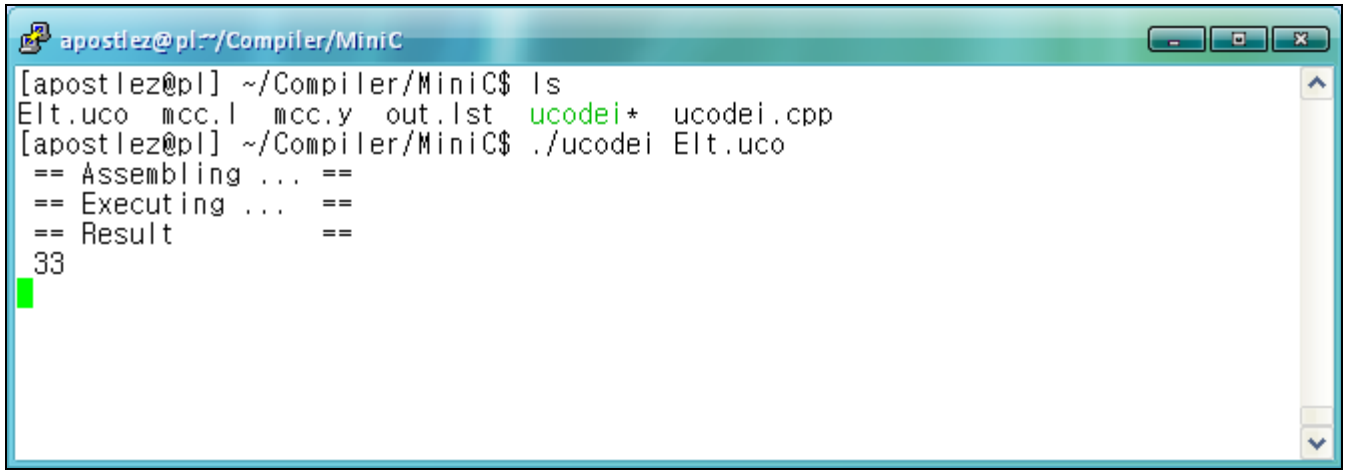
Elt.uco

```
void main()
{
	int list[100];
	int element;
	setElt(list,100);
	element=getElt(33,list);
	write(element);
}
void setElt(int array[], int size)
{
	int i;
	i = 0;
	while(i<size) {
		array[i]=i;
		i++;
	}
}
void getElt(int pos, int array[])
{
	return array[pos];
}
```

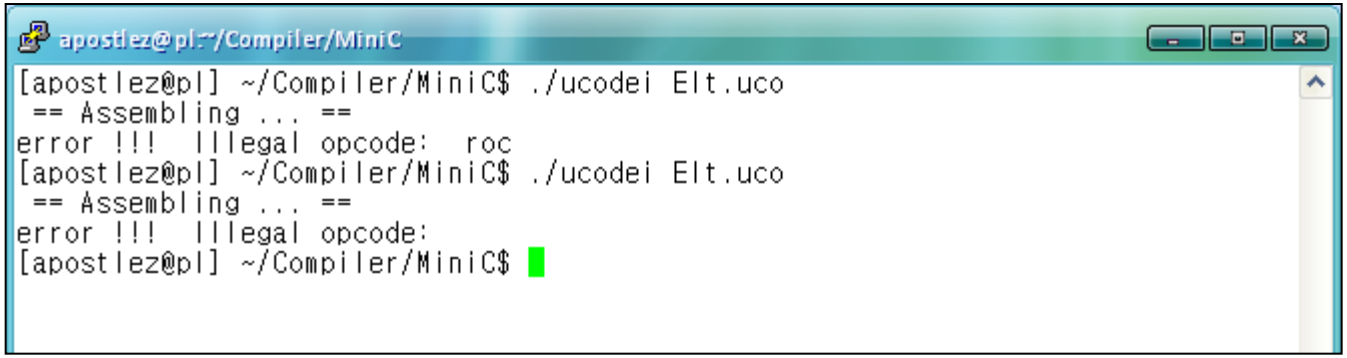
※ U-code 작성시 유의사항 :
Label-field 는 반드시 **1번째 칸부터** 시작
Op-code 는 **12번째 칸부터** 시작

실습 예제

- 실습 예제 실행 결과



- 12번째 칸이 틀린 경우



Mini C 컴파일러 개발

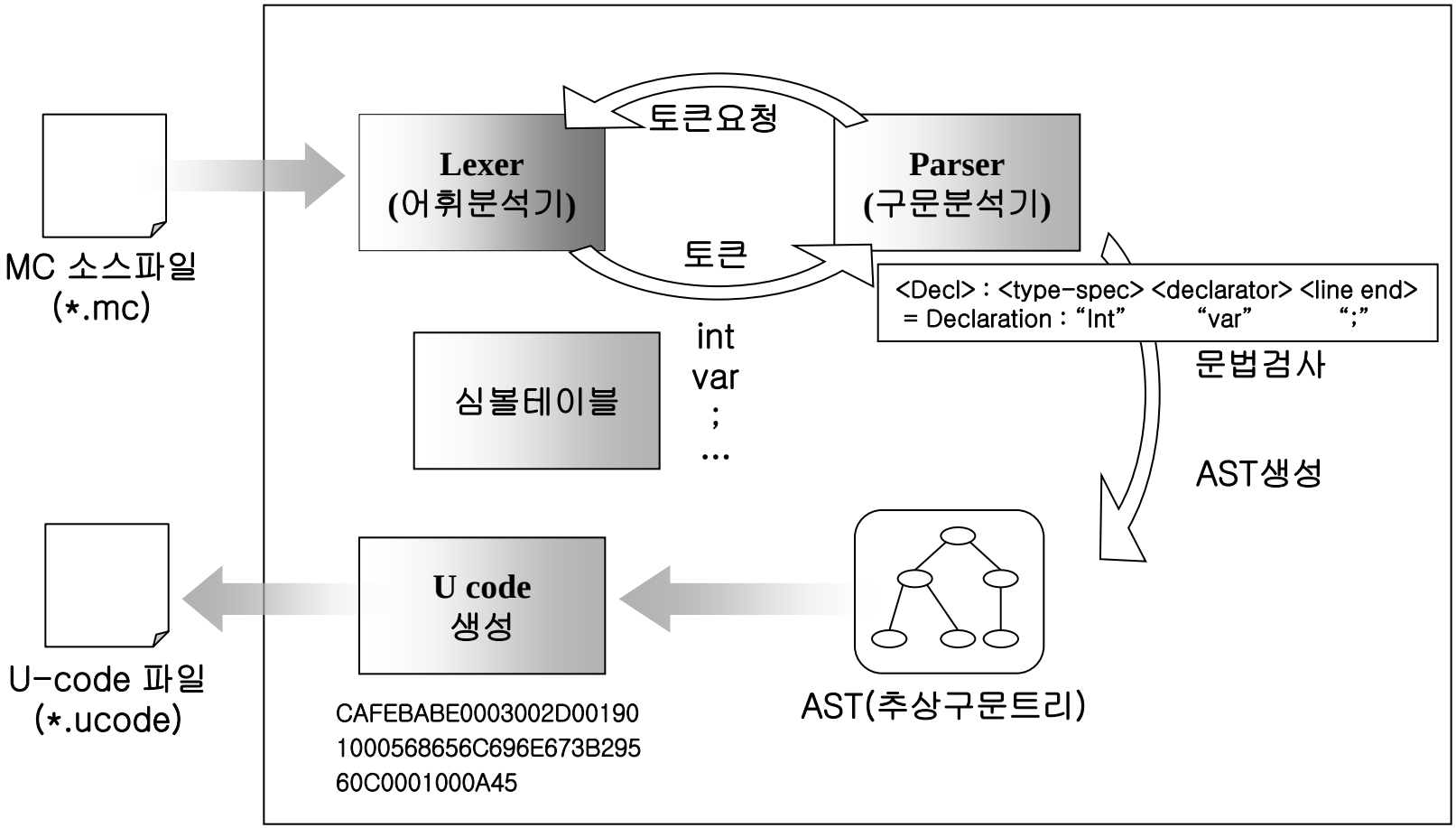
컴파일러 텀 프로젝트

Mini C Compiler 개요

- **프로젝트 목표 및 내용**
  - 원시 소스로부터 U-code를 작성하는 **미니 C 컴파일러** 작성
  - 미니 C 소스로부터 AST 작성
  - AST로부터 U-code를 작성
  - 작성한 **미니 C 컴파일러**를 통해 생성된 U-code 파일은 U-code 인터프리터를 통해 실행

- **미니 C 컴파일러**
  - Lex, Yacc 사용
  - 추상 구문 트리(Abstract Syntax Tree) 사용
  - 심볼 테이블 사용
  - 입력 : mc 파일
  - 출력 : u-code 파일(확장자는 .ucode)

Mini C Compiler의 구조



Mini C Compiler의 구현 과정

- **어휘분석기 구현**
  - Mini C 소스의 토큰을 분리
- **심볼테이블 구현**
  - 변수, 함수의 반환형 인자, 영역에 대한 정보를 저장
- **구문분석기 구현**
  - 일련의 토큰의 순서가 정해진 문법에 맞는지 검사
  - 문법 트리에 따라 소스의 정보를 AST 생성
- **중간코드 생성기 구현**
  - U code 생성기
  - AST를 순환하며 U code로 출력
  - AST 노드 종류에 따라 심볼테이블의 내용과 결합하여 U code 생성
- **테스트 및 실행**
  - 예제로 주어진 mini C 소스를 컴파일 및 실행

Mini C Compiler 작성방법

- 문법에 대한 Lex, Yacc 파일작성
- 심볼테이블 작성
- Yacc의 액션 규칙에 AST 생성 루틴 추가
- AST 노드에 따른 U code 생성 루틴 추가
- *.mc 파일작성 및 컴파일
- 결과 확인

텀 프로젝트 개요

- **기간 : 12월 7일(수요일) 까지**

- **보고서 및 목차 : PPT 형식의 발표 자료(5장 내외)**
  - 개요 및 작성한 컴파일러의 구성 요소 및 구조
  - AST 구조 및 활용 방법
    - AST 의미있는 non-terminal의 정의 등...
  - 다른 사람들과의 차이점! (나는 이런 부분을 더 신경 썼다!) ⇒ 추가점
    - 오류 메시지 처리 추가
    - AST의 활용
    - 성능 향상
    - ....
  - 소감

- **제출 자료**
  - 보고서
  - 개발한 컴파일러 소스

- **심사 방법 : 12월 7일(수요일) 5:00 ~ 7:00**
  - PPT 발표

텀 프로젝트 심사 포인트

- **실행 및 동작 여부**
  - AST 생성 후 Ucode 생성 까지 이상이 있는가?
- **오류 발생 여부**
  - 생성된 Ucode를 실행했을 때 이상이 있는가?
- **이해도**
  - 어휘분석기, 구문분석기, 심볼테이블의 각 기능을 이해하는가?
  - 컴파일러를 구성하는 각 구조의 연관 관계를 이해하는가?
  - 목적코드 생성 과정을 이해하고 있는가?

Mini C

소개 및 예제, 문법 구조

Mini C 소개 (1/3)

- 실험용 컴파일러 구현을 목적으로 ANSI C를 단순화 시킨 언어
- C 언어와 비슷한 형태를 갖는 언어
- 기존의 ANSI C 문법에서 언어 구조를 축소
- ANSI C의 부분으로 들어가므로 Mini C 로 작성된 프로그램은 일반 C 컴파일러로 컴파일 및 실행 가능

Mini C 소개 (1/3)

- 어휘 구조
  - 주석문
    - 라인 주석 : // 부터 라인 끝까지
    - 텍스트 주석 : /*와 */ 사이를 주석
  - 명칭의 형태
    - letter = 'a' | 'b' | .... | 'z'
    - digit = '0' | '1' | .... | '9'
    - <ident> = (letter + _)(letter + digit + _)*
  - 상수
    - 정수 상수만이 존재
  - 연산자
    - 사칙연산 : +, -, *, /, %
    - 배정연산 : =, +=, -=, *=, /=, %=
    - 논리연산 : !, &&, ||
    - 관계연산 : ==, !=, <, >, <=, >=
    - 증감연산 : ++, --
  - 예약어 : const, else, if, int, return, void, while

Mini C 소개 (3/3)

- **프로그램 구조**
  - 선언부 : 외부 선언 (ex. 전역변수)
  - 함수정의부 : C 프로그램과 동일
- **변수의 형**
  - 정수형
  - 배열 : 1차원 배열만을 허용
  - 매개변수 전달방법
    - 단순변수 : 값을 전달
    - 복합변수 : 주소 전달 (ex. 배열)
- **for 문은 지원하지 않음**

Mini C 예제 (1/5)

SimpleFibonacci.mc

```
/** 간단한 피보나치 수열 출력 예제 **/
void main(void)
{
	int i=0;
	int f0=0, f1=1, temp;
	int bounds = 10;
	while(i<bounds) {
		if(i<2) {
			write(i);
			continue; // 제거
		} else {
			temp = f0 + f1;
			f0 = f1;
			f1 = temp;
			write(f1);
		}
		i++;
	}
}
```

Mini C 예제 (2/5)

sort.mc

```
void sort(int array[], int size);   /* 정렬함수의 원형(prototype) 선언 */
void main(void)          {
      int array[10];   /* 배열 array의 배열크기를 10으로 설정 */
      int i=0;
      while(i<10)    /* 배열항목 입력 */ { read(array[i]); i++; }
      sort(array);   /* 정렬함수 호출*/
      i=0;while(i<10)    /* 정렬된 배열의 내용을 화면에 출력 */ { write( array[i]); i++; }
}
void sort(int array[])   /* 정렬함수 정의(선택 정렬 알고리즘)*/          {
      int i=0,j,temp;
      while(i<10-1)   /* 항목을 선택 */
      {j=i+1; while(j<10)   /* 선택된 항목과의 비교대상 선택 */
            {              if(array[i]>array[j])   /* 오름차순 정렬 */
                           {              temp = array[i];
                                          array[i] = array[j];
                                          array[j] = temp;
                           } j++;
            } i++;
}     }
```

Mini C 예제 (3/5)

simplefactorial.mc

```
/** 간단한 factorial, 1부터 10까지의 팩토리얼 계산 값을 출력 **/
void main()
{
	int i;
	int product;
	i=1; product=1;
	while(i<=10)
	{
		product *= i;
		write(product);
		i++;
	}
}
```

출력 결과

```
1
2
6
24
120
720
5040
40320
362880
3628800
```

Mini C 예제 (4/5)

recursion_factorial.mc

```
/**
재귀 함수로 구현한 fibonacci 함수
**/
void main(void)
{
	int i=0;
	while(i<=9) {
		write( fibonacci(i) );
		i++;
	}
}

int fibonacci(int n)
{
	if(n <= 1)
		return n;
	else
		return ( fibonacci(n-1) + fibonacci(n-2) );
}
```

Mini C 예제 (5/5)

scope_rule.mc

```
/**
변수의 수명, 심볼테이블을 테스트하기 위한 예제
**/
void main(void)
{
	int a = 1, b = 2; // A
	write(a);
	write(b);
	{
		int a = 5; // B
		write(a);
		write(b);
		a++; b++;   /* B의 a와 A의 b를 1씩 증가 */
		{
			write(a);
			write(b);
		}
	}
	write(a);
	write(b);  /* A의 a와 b를 출력 */
}
```

출력 결과

```
1
2
5
2
6
3
1
3
```

문법 형태 (1/5)

- **syntax of Mini C**

```
mini_c              -> external_dcls;
external_dcls       -> external_dcl;
                    -> external_dcls external_dcl;
external_dcl        -> function_def;
                    -> declaration;
function_def        -> function_header compound_st;
function_header     -> dcl_specifiers function_name formal_param
dcl_specifiers      -> dcl_specifier;
                    -> dcl_specifiers dcl_specifier;
dcl_specifier       -> type_qualifier;
                    -> type_specifier;
type_qualifier      -> 'const';
type_specifier      -> 'int';
                    -> 'void';
function_name       -> '%ident';
formal_param        -> '(' opt_formal_param ')';
opt_formal_param    -> formal_param_list;
                    -> ;
formal_param_list-  > param_dcl;
                    -> formal_param_list ',' param_dcl;
param_dcl           -> dcl_spec declarator;
```

문법 형태 (2/5)

- **syntax of Mini C**

```
compound_st         -> '{' declaration_list statement_list '}';
declaration_list    -> declaration;
                    -> declaration_list declaration;
declaration         -> dcl_spec init_dcl_list ';';
init_dcl_list       -> init_declarator;
                    -> init_dcl_list ',' init_declarator;
init_declarator     -> declarator;
                    -> declarator '=' '%number';
declarator          -> '%ident';
                    -> '%ident' '[' opt_number ']';
opt_number          -> '%number';
                    -> ;
statement_list      -> statement_list statement;
                    -> ;
statement           -> compound_st;
                    -> expression_st;
                    -> if_st;
                    -> while_st;
                    -> return_st;
```

문법 형태 (3/5)

- **syntax of Mini C**

```
expression_st       -> expression ';';
if_st               -> 'if' '(' expression ')' statement;
                    -> 'if' '(' expression ')' statement 'else' statement;
while_st            -> 'while' '(' expression ')' statement;
return_st           -> 'return' opt_expression ';';
expression          -> assignment_exp;
assignment_exp      -> logical_or_exp;
                    -> unary_exp '=' assignment_exp;
                    -> unary_exp '+=' assignment_exp;
                    -> unary_exp '-=' assignment_exp;
                    -> unary_exp '*=' assignment_exp;
                    -> unary_exp '/=' assignment_exp;
                    -> unary_exp '%=' assignment_exp;
logical_or_exp      -> logical_and_exp;
                    -> logical_or_exp '||' logical_and_exp;
logical_and_exp     -> equality_exp;
                    -> logical_and_exp '&&' equality_exp;
```

문법 형태 (4/5)

- **syntax of Mini C**

```
equality_exp            -> relational_exp;
                        -> equality_exp '==' relational_exp;
                        -> equality_exp '!=' relational_exp;
relational_exp          -> additive_exp;
                        -> relational_exp '>' additive_exp;
                        -> relational_exp '<' additive_exp;
                        -> relational_exp '>=' additive_exp;
                        -> relational_exp '<=' additive_exp;
additive_exp            -> multiplicative_exp;
                        -> additive_exp '+' multiplicative_exp;
                        -> additive_exp '-' multiplicative_exp;
multiplicative_exp      -> unary_exp;
                        -> multiplicative_exp '*' unary_exp;
                        -> multiplicative_exp '/' unary_exp;
                        -> multiplicative_exp '%' unary_exp;
unary_exp               -> postfix_exp;
                        -> '-' unary_exp;
                        -> '!' unary_exp;
                        -> '++' unary_exp;
                        -> '--' unary_exp;
```

문법 형태 (5/5)

- **syntax of Mini C**

```
postfix_exp          -> primary_exp;
                     -> postfix_exp '[' expression ']';
                     -> postfix_exp '(' opt_actual_param ')';
                     -> postfix_exp '++';
                     -> postfix_exp '--';
opt_actual_param     -> actual_param_list;
                     -> ;
actual_param_list    -> assignment_exp;
                     -> actual_param_list ',' assignment_exp;
primary_exp          -> '%ident';
                     -> '%number';
                     -> '(' expression ')';
```

※ 참조 : 컴파일러 입문, 오세만 저, 정익사