컴파일러

서문

이 코드는 SysY라는 c언어의 부분집합인 언어를 컴파일하는 컴파일러입니다. 코드는 c++로 작성되었습니다. 이 컴파일러는 크게 형태소 분석, 구문 분석, 의미 분석 및 코드 생성기로 구분되어 있습니다. 개발 언어는 c++17로 작성하였습니다.

파일

1. 컴파일을 위한 제 코드는 `compile.cpp` 파일에 있습니다.
2. 컴파일 하고자 하는 코드는 `testfile.txt`에 넣으시면 됩니다. 문법 요구사항을 준수하는 파일과, 준수하지 않은 파일이 모두 있고, 문법에 오류가 있다면, `error.txt`에 오류를 생성합니다.
3. 입력 하고자 하는 코드는 `input.txt` 파일에 추가하시면 됩니다.
4. 문법에 의해 구문분석된 파일은 `output.txt` 파일에 추가됩니다.
5. 문법에 오류가 있다면, `error.txt`에 오류를 생성합니다.
6. 코드 생성이 완료된 부분, 즉 printf로 출력이 되는 코드들은 `pcoderesult.txt` 파일에 추가됩니다.

빠른 코드실행

문서를 읽지 않고, 빠르게 실행시키고 싶으시다면 마지막 부분인 코드 생성 부분의 코드를 가져와서, `input.txt`와 `testfile.txt`에 위에 업로드해둔 `'예시'` 파일의 값을 각각 하나씩 넣고, 코드 실행 결과인 `pcoderesult.txt`와 문법 분석 결과인 `output.txt`, 에러는`error.txt`의 결과를 보시면 됩니다.

설명

각 단계마다 코드들을 모두 분류해서 업로드 하였으며, 모든 단계의 의미와 과제를 적어두었습니다.

형태소 분석

소스 프로그램에서 단어를 인식하고, 단어의 카테고리와 값을 기록하는 어휘 분석 프로그램을 설계하고 구현하였습니다. 입력, 출력 및 처리 요구사항은 다음과 같습니다.

1. 코드를 읽고, `//` 과 `/* */` 등의 주석을 파싱하며 원본 텍스트를 문법 분석에 필요한 최소 단위인 token으로 분리하여 배열에 저장합니다. 배열은 아래의 wordList에 저장하였습니다.

```
struct Word
{
    string label;
    string idenfr;
    int lineId;
};
Word wordList[10000];
```

2. label과 idenfr은 아래 표로 구분하였습니다. 기존의 코드에서 사용하는 `* / % <` 등의 문법은 idenfr, 이를 각각 `MULT, DIV, MOD, LSS` 등으로 번역한 문자열을 label에 저장하였습니다.

Ident	IDENFR	!	NOT	*	MULT	=	ASSIGN
IntConst	INTCON	&&	AND	/	DIV	;	SEMICN
FormatString	STRCON	||	OR	%	MOD	,	COMMA
main	MAINTK	for	FORTK	<	LSS	(	LPARENT
const	CONSTTK	getint	GETINTTK	<=	LEQ	)	RPARENT
int	INTTK	printf	PRINTFTK	>	GRE	[	LBRACK
break	BREAKTK	return	RETURNTK	>=	GEQ	]	RBRACK
continue	CONTINUETK	+	PLUS	==	EQL	{	LBRACE
if	IFTK	-	MINU	!=	NEQ	}	RBRACE
else	ELSETK	void	VOIDTK				

3. 문법에 사용하지 않는 오류들은 삭제합니다. 예를들어 변수명이 아닌 `&&A` 와 같이 c언어에서 사용하지 않은 문법등 입니다.

구문 분석

문법 문서(아래의 '문법' 파트)를 참고하여 구문 분석 프로그램을 설계하고 구현하였습니다. 소스 코드에서 해당하는 구문 요소를 식별하는데 있어 다음과 같은 입력, 출력 및 처리 요구사항이 있습니다:

1. 제시된 문법 규칙에 따라 재귀 하위 프로그램 방법을 사용하여 정의된 구문 요소를 분석해야 합니다.
2. 어휘 분석을 통해 식별된 단어의 순서에 따라, 각 단어의 정보를 한 줄에 하나씩 출력해야 합니다. (예: 단어 유형 코드 단어 문자열 (간격은 하나의 공백으로 구분))

3. 문법에서 등장하는 (<BlockItem>, <Decl>, <BType>을 제외한) 구문 분석 요소가 분석되기 전에, 현재 구문 요소의 이름을 새로운 줄로 출력해야 합니다. (예: "<Stmt>")
4. 참고: 출력이 요구되지 않는 구문 요소들도 분석을 수행해야 하지만 출력은 필요하지 않습니다.

문법

SysY 언어의 문법은 확장된 Backus-Naur Form(EBNF)을 사용하여 표현되며 다음과 같습니다:

- [...] : 대괄호 안에 포함된 것이 선택 사항임을 나타냅니다.
- {...} : 중괄호 안에 포함된 항목이 0회 이상 반복될 수 있음을 나타냅니다.
- 작은 따옴표로 묶인 문자열이나 Ident, InstConst는 **터미널** 기호입니다.

1. 컴파일 번역 단위

```
//컴파일 번역 단위
CompUnit → {Decl} {FuncDef} MainFuncDef

//선언 : 두 가지 선언 종류 : 상수 선언, 변수 선언
Decl → ConstDecl | VarDecl
//기본 타입
BType → 'int'
```

2. 상수

```
//상수 생성
ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'
//상수 정의 : 변수, 1차원, 2차원 배열을 포함
ConstDef → Ident { '[' ConstExp ']' } '=' ConstInitVal
// 상수 초기값 : 위에서 선언한 변수, 1차원, 2차원 배열을 초기화
ConstInitVal → ConstExp | '{' [ ConstInitVal { ',' ConstInitVal }
```

3. 변수

```
//변수 선언 : 중괄호 내부 0회 반복 또는 중괄호 내부 다수 반복
VarDecl → BType VarDef { ',' VarDef } ';'
// 변수 정의 : 배열 선언 가능. 변수 초기화를 해도, 안해도 가능.
// ex) int arr[3]={1,2,3}; int arr[3];
VarDef → Ident { '[' ConstExp ']' } | Ident { '[' ConstExp ']' }
```

```
// 표현식 초기값, 1차원 배열 초기값, 2차원 배열 초기값
InitVal → Exp | '{' [ InitVal { ',' InitVal } ] '}'
```

4. 함수

```
//  메인 함수 정의
MainFuncDef → 'int' 'main' '(' ')' Block

// 일반 함수
// 일반 함수 정의 : 매개변수 있을수도 있고, 없을 수도 있다.
FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
// 일반 함수는 void 함수와 int 함수가 있습니다.
FuncType → 'void' | 'int'

// 함수 매개 변수 목록
FuncFParams → FuncFParam { ',' FuncFParam }
// 함수 매개 변수 : 일반 매개변수, 1차원 배열 변수, 2차원 배열 변수
FuncFParam → BType Ident ['[' ']' { '[' ConstExp ']' }]
```

5. 문법 : `{}, if, for, break, return, getint(), printf` 문을 모두 다룹니다.

```
// {} 등의 불록
语句块 Block → '{' { BlockItem } '}'
// 블록 안에서는 변수나 stmt 선언 가능
语句块项 BlockItem → Decl | Stmt

// stmt 선언
语句 Stmt →
LVal '=' Exp ';'
| [Exp] ';' //exp는 있어도, 없어도 된다.
| Block
| 'if' '(' Cond ')' Stmt [ 'else' Stmt ] // 1.else문은 있을수도 없을
// for의 모든 경우를 다룹니다. for(;;), for(i=0;;) for(;i<5;)등 모두 가
| 'for' '(' [ForStmt] ';' [Cond] ';' [ForStmt] ')' Stmt
| 'break' ';' | 'continue' ';'
| 'return' [Exp] ';' // return 문은 함수 타입에 따라 반환값이 없을수도 있
| LVal '=' 'getint''('')'';' // scanf 대신에 getint()문을 사용합니다.
| 'printf''('FormatString{','Exp}')'';' // printf("%d",a), printf
```

```
语句 ForStmt → LVal '=' Exp
表达式 Exp → AddExp 注:SysY 表达式是int 型表达式
条件表达式 Cond → LOrExp
// 모든 배열이 가능합니다.
左值表达式 LVal → Ident {'[' Exp ']'}


//기본 표현식
PrimaryExp → '(' Exp ')' | LVal | Number
//숫자
Number → IntConst


// FuncRParams는 int c = func2(a,b,10)처럼 func2()가 붙었을때 호출
//단항 연산자
UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp Una
UnaryOp → '+' | '−' | '!'
// 함수 인수 목록
FuncRParams → Exp { ',' Exp }


// 곱셈, 나눗셈 나머지(%) 표현식
MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
// 덧셈, 뺄셈 표현식
AddExp → MulExp | AddExp ('+' | '−') MulExp
//관계식
RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
//등식
EqExp → RelExp | EqExp ('==' | '!=') RelExp
// 논리 AND 표현식
LAndExp → EqExp | LAndExp '&&' EqExp
//논리 OR 표현식
LOrExp → LAndExp | LOrExp '||' LAndExp
//상수 표현식
ConstExp → AddExp //사용하는 Ident은 반드시 상수여야 함
```

이해하기 어려울수 있지만, c언어의 while문, 변수는 int형과 const형만 가능하다는 점 빼고 거의 동일하다고 생각하시면 됩니다.

입력 형식

문법 이해 파일에 있는 **testfile.txt에는 문법 요구 사항을 준수하는 테스트 프로그램이 있습니다.**

출력 형식

어휘 분석 결과를 output.txt에 출력해야 합니다.

입력 예시

```
int main(){
    int c;
    c= getint();
    printf("%d",c);
    return c;
}
```

출력 예시

```
INTTK int
MAINTK main
LPARENT (
RPARENT )
LBRACE {
INTTK int
IDENFR c
<VarDef>
SEMICN ;
<VarDecl>
IDENFR c
<LVal>
ASSIGN =
GETINTTK getint
LPARENT (
RPARENT )
SEMICN ;
```

```
<Stmt>
PRINTFTK printf
LPARENT (
STRCON "%d"
COMMA ,
IDENFR c
<LVal>
<PrimaryExp>
<UnaryExp>
<MulExp>
<AddExp>
<Exp>
RPARENT )
SEMICN ;
<Stmt>
RETURNTK return
IDENFR c
<LVal>
<PrimaryExp>
<UnaryExp>
<MulExp>
<AddExp>
<Exp>
SEMICN ;
<Stmt>
RBRACE }
<Block>
<MainFuncDef>
<CompUnit>
```

의미 분석

의미 분석이란?

프로그램의 요소가 서로 의미적으로 적당한지를 확인하는 검사를 수행합니다. 또한 심볼 테이블 메이저를 생성 합니다.

프로그램의 요소가 서로 의미적으로 적당한지를 확인하는 검사를 수행한다. 의미 분석 단계에서는 소스 프로그램의 의미 오류를 검사하고 다음에 이어질 코드 생성 단계를 위해 타입 정보를 모아서 정리

해 준다. 이 단계에서는 수식과 문장의 연산자와 피연산자를 인식하기 이해 구문 분석에서 생성된 계층 구조를 사용한다.

심볼 테이블 메니저(symbol table manager)

심벌테이블은 각 식별자에 해당하는 레코드를 포함하는 데이터 구조이다. 각 레코드는 식별자의 특성들로 된 필드로 구성된다. 이때 데이터 구조(심벌 테이블)는 **각 식별자에 대한 레코드를 빨리 찾게 해주고 레코드로부터 데이터를 빠르게 저장 또는 탐색하게 해준다**.

오류 출력과 3번째 과제 설명

결과 파일에는 다음과 같은 두 가지 정보가 포함되어 있습니다: 오류가 있는 행 번호, 잘못된 카테고리 코드(행 번호와 카테고리 코드 사이에 공백만 있고 카테고리 코드는 표의 소문자 영문자를 엄격히 따름)

오류 범주 코드는 다음 표에 정의된 대로 출력되며 행 번호는 1부터 계산됩니다.

예를 들어 if 뒤에 여는 괄호 토큰이 와야 하는데 이게 안오면 에러를 발생시킵니다. if 다음에 바로 value를 가진 토큰이 왔기 때문입니다. 이러한 규칙을 토대로 분석을 합니다. 그리고 이 정보를 디버그 로그에 출력해주는 것이 컴파일러가 하는 역할입니다.

오류 유형	오류 코드	설명	해당 문법
틀린 기호	a	형식 문자열에서 비법적인 문자(예: &)가 등장합니다. 오류가 발생한 줄은 <FormatString>에 표시됩니다.	<FormatString> → '"' {<Char>} '"'
이름 중복 정의	b	현재 범위에서 함수 이름 또는 변수 이름이 중복으로 정의되었습니다. 변수의 경우 동일한 레벨의 범위에서만 오류로 간주됩니다. 다른 레벨의 범위에서는 내부 정의가 외부 정의를 덮어씁니다. 오류가 발생한 줄은 <Ident>에 표시됩니다.	<ConstDef> → <Ident> ... <VarDef> → <Ident> ... <FuncDef> → <FuncType><Ident> ... <FuncFParam> → <BType> <Ident> ...
정의되지 않은 이름	c	정의되지 않은 식별자가 사용되었습니	<LVal> → <Ident> ... <UnaryExp> → <Ident> ...

		다. 오류가 발생한 줄은 <Ident>에 표시됩니다.	
함수 매개변수 불일치	d	함수 호출문에서 매개변수 개수가 함수 정의에서의 매개변수 개수와 일치하지 않습니다. 오류가 발생한 줄은 함수 호출문의 함수 이름이 있는 줄에 표시됩니다.	<UnaryExp> → <Ident> '(' [FuncRParams] ')'
함수 매개변수 타입 불일치	e	함수 호출문에서 매개변수의 타입이 함수 정의에서 해당 위치의 매개변수 타입과 일치하지 않습니다. 오류가 발생한 줄은 함수 호출문의 함수 이름이 있는 줄에 표시됩니다.	<UnaryExp> → <Ident> '(' [FuncRParams] ')'
반환값이 없는 함수에 불일치한 return 문장	f	'return' 문장이 있는 줄의 오류입니다.	<Stmt> → 'return' {'[' Exp ']'} ';'
반환값이 있는 함수에서 누락된 return 문장	g	함수의 끝 부분에 'return' 문장이 누락되었습니다. 오류가 발생한 줄은 해당 함수의 마지막 '}'가 있는 줄입니다.	<FuncDef> → <FuncType><Ident> '(' [<FuncFParams>] ')' <Block> <MainFuncDef> → 'int' 'main' '(' ')' <Block>
상수의 값을 변경할 수 없음	h	<LVal>이 상수인 경우 해당 상수의 값을 변경할 수 없습니다. 오류가 발생한 줄은 <LVal>이 있는 줄입니다.	<Stmt> → <LVal> '=' <Exp> ';'
세미콜론이 누락됨	i	세미콜론이 누락된 줄의 오류입니다.	<Stmt>, <,ConstDecl> 및 <VarDecl>에 있는 ‘;’
오른쪽 소괄호가 누락됨	j	오른쪽 소괄호가 누락된 줄의 오류입니다.	함수 호출(<UnaryExp>)、함수 정의(<FuncDef>) 및 <Stmt>에 있는 ')'

오른쪽 대괄호가 누락됨	k	오른쪽 대괄호가 누락된 줄의 오류입니다.	배열 정의 (<ConstDef>, <VarDef>, <FuncFParam>) 및 사용(<LVal>)에 있는 ']'
printf 문장에서 형식 문자열과 표현식의 개수가 일치하지 않음	l	'printf' 문장에서 형식 문자열과 표현식의 개수가 일치하지 않습니다. 오류가 발생한 줄은 'printf'가 있는 줄입니다.	<Stmt> → 'printf' '(' <FormatString> {, <Exp>} ')' ';'
반복 블록 외부에서 'break'와 'continue' 문장 사용	m	'break'와 'continue' 문장이 있는 줄의 오류입니다.	<Stmt> → 'break' ';'

예시

```
int main() {
    break;
    continue;
    return 0;
}
2 m
3 m


//2
void f1(int x, int y, int z){

}

int main() {
    f1(1);
    return 0;
}
6 d


//3
int main() {
```

```
    a;
    a = 1;
    a();
    return 0;
}
2 c
3 c
4 c


//4
void f(int a, int a){

}

int main() {
    int a, a;
    return 0;
}
1 b
6 b


//5
int main() {
    printf("#:\n");
    printf("%%");
    printf("\");
    return 0;
}
2 a
3 a
4 a


//6
int main() {
    printf("%d", 1, 1);
```

```
    return 0;
}
2 l


//7
int main() {
    int a[2][2;
    return 0;
}
2 k


//8
void f1({

}

void f2(){

}

int main() {
    f2(;
    return 0;
}
1 j
10 j


//9
int main() {
    return 0
}
2 i


//10
```

```
int main() {
    const int a = 0;
    a = 1;
    return 0;
}
3 h


//11
int f1()
{}

int main() {

}
2 g
6 g


//12
void f1(){
    return 0;
}

int main() {
    return 0;
}
2 f
```

코드 생성

구문 분석, 문법 분석 및 오류 처리 작업을 기반으로 컴파일러의 의미 분석 및 코드 생성 기능을 구현하였습니다,

문법 규칙과 의미 규칙에 따라 상향식 구문 지도 번역 기술을 사용하여 의미 분석을 수행하고 대상 코드 (PCODE)를 생성하였습니다.

코드 설명

1. 최종 코드 해석기(class Interpreter) : Pcode 명령어를 해석하고 최종 코드를 생성하는 결과입니다.
2. Block

```
public class Block {
    private String type; // 전역 블록, 함수 블록, 일반 블록을 구분하는 역
    private ArrayList<Block> CBlock; // 하위 블록들
    private Block FBlock; // 상위 블록
    private ArrayList<Symbol> SymbolTable; // 블록 내 심볼 테이블
    private int level; // 블록의 깊이. 전역은 1이며, 새로운 블록이 추가될
    private boolean returnTk; // 반환 값이 있는지 여부를 판단하는 변수.
}
```

3. symbol

```
public class Symbol {
    private String name; // 이름
    private int dim; // void 함수는 -1, int 함수 및 일반 표현식은 0, 1
    private int dim1 = 0; // 첫 번째 차원의 크기 또는 2차원 배열의 두 번
    private int dim2 = 0; // 2차원 배열의 첫 번째 차원의 크기입니다.
    private int address = 0; // 주소
    private boolean isConst = false; // 상수인지 여부를 판단합니다.
    private boolean isGlobal = false; // 전역 정의인지 여부를 판단합니
}
```

4. Code

```
public class Code {
        private String name; // Pcode 명령어 이름
        private int level; // 해당 레벨
        private int addr; // 해당 주소
        private String print; // print 문의 Strcon 내용
        private Label label; // 점프 대상
        private int type = 0; // 명령어 유형
}

public class Label {
```

```
        private int point = 0;// 점프 포인트
}
```

PCode 문법 정의

아래 예시는 제가 책을 보고 만들고 추가한 예시입니다. 정형화된 문법이 아님을 참고 바랍니다.

INT x	스택 상단 포인터를 x만큼 이동시키다
DOWN x	스택의 맨 위 포인터를 x만큼 이동시킵니다
LOD x y	OD x y는 상대 위치 y에서 내용을 조회하여 스택의 맨 위에 저장합니다. x는 0 또는 1로, 0은 절대 주소에서 조회를 수행하고 1은 상대 주소에서 조회를 수행합니다

LODS	스택 상단 포인터의 주소에서 내용을 조회하여 스택 상단에 저장합니다
	
	`dstack[sp] = dstack[dstack[sp]];`
	
	`sp++`

LDA x y	**결론은 int a=10; 이라고 했을때, 변수 a가 저장될 공간을 저장한다.**
	
	相对位置가 y인 주소를 스택 상단에 저장한다.
	x는 0 | 1 이며 0은 절대주소(绝对地址)에서 조회를 수행함을 나타내고 1은 상대 주소(相对地址)에서 조회를 수행함을 나타낸다.
	
	결론은 int a=10; 이라고 했을때, 변수 a가 저장될 공간을 저장한다.
	
	`sp++;`
	
	`addr = Bddr + curCode.addr;`
	
	`dstack[sp] = addr;` dstack[sp]에 변수 a가 저장될 공간 addr을 저장한다.
	
	`at++;`

LDC	값을 스택에 저장한다. `int a = 10:` 이라고 했을때, dstack[sp]에 10을 저장한다. LDA에서 a를 저장할 주소를 저장했으면 그걸 가져와서 그 주소에 10을 저장한다.

	`sp++;`
	
	`dstack[sp] = curCode.getAddr();`
	
	`at++;`

STOS	스택 상단 포인터의 주소에서 내용을 조회하여 다음 스택 상단의 주소에 저장하고, 스택을 두 번 빼냅니다
	
	`int a=10;` 이라고 했을때, LDA에서 변수를 저장할 공간을 저장했었고, LDC에서 변수에 할당될 값을 저장했었으니까, LDA의 위치에 LDC를 저장한다.
	
	`sp—;`
	
	`dstack[dstack[sp]] = dstack[sp+1];`
	dstack[sp]는 LDA에서 저장한 위치를 말하고 dstack[sp+1] 에서 sp+1이 LDC에서 할당된 값을 가져온다.
	결국
	`dstack[LDA에서 저장한 위치] = LDA에서 할당된 위치` 라는 뜻이다.
	
	`sp—-;`
	
	`at++;`

ADD	스택 상단과 다음 스택 상단을 더한 다음 그 결과를 스택 상단에 저장합니다
SUB	-
MUL	*
DIV	/
MOD	%
MINU	-
GET	getint() 수행
PRF	printf출력
JTM x	main 함수로 이동합니다. x는 main 함수의 이전 명령어의 시퀀스 주소입니다.
CAL x	함수를 호출합니다. x는 호출하는 함수의 이전 명령어의 시퀀스 주소입니다.
RET	return
RET_TO_END	메인 함수에서 반환합니다.

INT_L x	스택 포인터를 Lable만큼 이동시킵니다. Lable은 함수 블록에 따라 결정됩니다. 예를 들어 변수가 두 개 있는 경우 Lable은 5 (3+2)입니다. 각 함수 블록마다 반환 값, 반환 값 기본 주소, 반환 값 명령어 시퀀스 번호를 저장하기 위해 세 개의 위치가 예약됩니다.
BGT	> 스택을 한 번 빼내어 이전 스택의 맨 위와 비교합니다. (>) 관계가 존재하면 현재 스택 맨 위에 1을 저장하고, 그렇지 않으면 0을 저장합니다.
BGE	스택을 한 번 빼내어 이전 스택의 맨 위와 비교합니다. (>=) 관계가 존재하면 현재 스택 맨 위에 1을 저장하고, 그렇지 않으면 0을 저장합니다.
BLT	< 스택을 한 번 빼내어 이전 스택의 맨 위와 비교합니다. (<) 관계가 존재하면 현재 스택 맨 위에 1을 저장하고, 그렇지 않으면 0을 저장합니다.
BLE	≤ 스택을 한 번 빼내어 이전 스택의 맨 위와 비교합니다. (<=) 관계가 존재하면 현재 스택 맨 위에 1을 저장하고, 그렇지 않으면 0을 저장합니다.
BEQ	== 스택을 한 번 빼내어 이전 스택의 맨 위와 비교합니다. (==) 관계가 존재하면 현재 스택 맨 위에 1을 저장하고, 그렇지 않으면 0을 저장합니다.
BNE	≠ 스택을 한 번 빼내어 이전 스택의 맨 위와 비교합니다. (!=) 관계가 존재하면 현재 스택 맨 위에 1을 저장하고, 그렇지 않으면 0을 저장합니다.
BZT x	if 0 jump 스택 맨 위가 0이면 x 위치의 명령어 시퀀스로 이동합니다. 스택을 한 번 빼내어 실행합니다. (IFTK WHILETK)
J x	jump 조건 없이 x 위치로 점프합니다.
JP0 x	스택 맨 위가 0이면 x 위치의 명령어 시퀀스로 점프합니다. 스택을 빼내지 않습니다. (LAND TK)
JP1 x	스택 맨 위가 1이면 x 위치의 명령어 시퀀스로 점프합니다. 스택을 빼내지 않습니다. (LOR TK)
NOT	!a 스택 맨 위의 값을 논리 부정합니다.

설명

1. 상수정의

```
const int a = 10;
int b = 10;
int f;
const int c[1] = {1};
const int d[2][2] = {{1,2},{3,4}};

//결과
0 INT 1
1 LDA 0 0
```

```
2 LDC 10
3 STOS//const int a = 10;
4 INT 1
5 LDA 0 1
6 LDC 10
7 STOS//int b = 10;
8 INT 1
9 LDA 0 2
10 LDC 0
11 STOS//int f;
12 INT 1
13 LDA 0 3
14 LDC 1
15 STOS//const int c[1] = {1};
16 INT 1
17 LDA 0 4
18 LDC 1
19 STOS//const int d[2][2] = {{1,2},{3,4}}; 1
20 INT 1
21 LDA 0 5
22 LDC 2
23 STOS//const int d[2][2] = {{1,2},{3,4}}; 2
24 INT 1
25 LDA 0 6
26 LDC 3
27 STOS//const int d[2][2] = {{1,2},{3,4}}; 3
28 INT 1
29 LDA 0 7
30 LDC 4
31 STOS//const int d[2][2] = {{1,2},{3,4}}; 4
```

스택의 맨 위 포인터 초기값을 -1로 설정하므로, 매번 스택의 맨 위 포인터를 1만큼 증가시킵니다(INT 1). 그런 다음 상대 주소를 스택의 맨 위에 저장합니다(LDA 0 0). 그 다음 할당된 값을 스택의 맨 위에 저장합니다(LDC). 스택의 맨 위 내용을 이전에 표시된 주소에 저장한 다음 스택을 두 번 축소합니다(STOS). 이렇게 하면 됩니다. 할당 작업이 수행되지 않은 문장의 경우, 기본적으로 0으로 할당됩니다.

배열의 경우, 순서대로 값을 저장하기만 하면 되며, 이는 상수 또는 변수와 관련된 정의와 기본 원리가 동일합니다.

2. 계산

```
1 + 2;
1 LDC 1
2 LDC 2
3 ADD
```

덧셈, 뺄셈, 곱셈, 나눗셈, 나머지와 같은 이항 연산자의 연산 작업은 다음과 같이 세 단계로 수행됩니다:

1. 첫 번째 피연산자를 스택에 넣습니다 (LDC 1).
2. 두 번째 피연산자를 스택에 넣습니다 (LDC 2).
3. 해당 연산을 수행합니다 (ADD, SUB, MUL, DIV, MOD).

예를 들어, 덧셈 연산의 경우 다음과 같이 수행됩니다:

1. 첫 번째 피연산자를 스택에 넣습니다 (LDC 1).
2. 두 번째 피연산자를 스택에 넣습니다 (LDC 2).
3. 덧셈 연산을 수행합니다 (ADD).

이와 같은 방식으로 뺄셈, 곱셈, 나눗셈, 나머지 연산도 수행됩니다.

3. scanf, printf

```
int a ;
a = getint();
printf("a is %d",a);
1 INT 1
2 LDA 0 0
3 LDC 0
4 STOS
5 LDA 0 0
6 GET
7 STOS
8 LDA 0 0
9 LODS
10 PRF "a is %d"
```

읽기(read) 작업 및 쓰기(write) 작업은 모두 다음과 같은 단계로 진행됩니다:

1. 주소를 읽습니다 (LDA x y 또는 LDC x).
2. 해당 주소에서 값을 읽거나 쓰는 작업을 수행합니다.

- 읽기 작업의 경우:
  - 주소를 읽습니다 (LDA x y).
  - 해당 주소에서 값을 읽어와 사용합니다.
- 쓰기 작업의 경우:
  - 주소를 읽습니다 (LDA x y 또는 LDC x).
  - 해당 주소에 값을 쓰거나 업데이트합니다.

이와 같은 방식으로 읽기 작업과 쓰기 작업이 수행됩니다. 주소를 읽은 후 해당 주소에서 값을 읽거나 쓰는 작업을 수행하여 데이터를 처리합니다.

4. 표현식

```
const int a = 10;
int b[2][2] = {{1,2},{3,4}};
int main(){
 int d, c;
 d = a;
 c = b[1][1];
 return 1;
}
//결과
30 LDA 0 0//d address
31 LDA 1 0//a address
32 LODS// load a value
33 STOS//store a value
//数组传值
34 LDA 0 1//c address
35 LDC 1//b dim1
36 LDC 2//b dim2
37 MUL// calculate b[1][1]'s address
38 LDC 1
39 LDA 1 1//load b address
40 ADD//calculate b baseAddr+offset
41 ADD//calculate b baseAddr+offset
```

```
42 LODS//load from b baseAddr+offset
43 STOS//store value
```

왼쪽 값 표현식의 경우, 일반적인 전달 방식은 다음과 같이 수행됩니다:

1. 등호 왼쪽의 주소를 스택의 맨 위에 읽어옵니다 (LDA 0 0).
2. 등호 오른쪽의 주소를 스택의 맨 위에 읽어옵니다 (LDA 1 0).
3. 현재 스택의 맨 위 값을 가져옵니다 (LODS).
4. 스택의 맨 위 내용을 이전에 표시된 주소에 저장한 다음 스택을 두 번 축소합니다 (STOS).

이렇게 하면 됩니다. 배열 전달의 경우, 배열 헤더의 기본 주소를 기록하고, 배열의 차원을 사용하여 필요한 오프셋을 계산한 후 값을 저장합니다. 자세한 과정은 34-43번째 줄의 Pcode에서 확인하실 수 있습니다.

5. 조건문

```
int main(){
 int a,b,c,d;
 a = 1;
 if(a){
 b = 2;
 }else{
 b = 3;
 }
}
21 LDA 0 0//load a address
22 LODS//load a value
23 BZT 28//if 0 jumpto Pcode28(进入else块)
24 LDA 0 1//load b address
25 LDC 2//load 2
26 STOS// b = 2
27 J 31//jumpto Pcode31(离开条件语句块)
28 LDA 0 1
29 LDC 3
30 STOS//b = 3
```

조건문의 경우, 먼저 조건 블록(cond block) 내부를 계산하여 최종 결과를 스택의 맨 위에 저장합니다. 스택의 맨 위 값이 1이면 if 블록으로 진입하고, 0이면 else 블록으로 진입합니다 (BZT

28). 만약 else 블록이 없다면 조건문 블록을 바로 나가게 됩니다.

6. 논리식

```
int main(){
 int a,b,c;
 a = 1;
 b = 0;
 c = 2;
 if(a && b && c){
 b = 1;
 }
 if(a || b || c){
 b = 2;
 }
}
23 LDA 0 0
24 LODS//load a value
25 JP0 29//短路求值 a为0直接跳转
26 DOWN 1
27 LDA 0 1
28 LODS//load b value
29 JP0 33//短路求值 b为0直接跳转
30 DOWN 1
31 LDA 0 2
32 LODS
33 BZT 37//if块判断
34 LDA 0 1
35 LDC 1
36 STOS
37 LDA 0 0
38 LODS
39 JP1 43//短路求值 a为1直接跳转
40 DOWN 1
41 LDA 0 1
42 LODS
43 JP1 47//短路求值 b为1直接跳转
44 DOWN 1
```

```
45 LDA 0 2
46 LODS
```

리식의 경우, 각 피연산자를 읽을 때마다 해당 값의 평가를 수행하여 직접적으로 점프 동작을 수행하고, 단락 평가(short-circuit evaluation)를 구현할 수 있습니다.

예를 들어, 논리식 `A && B`의 경우:

1. A를 읽고 평가합니다.
2. A가 거짓(false)인 경우, 논리식 전체가 거짓이므로 건너뛰고 다음 동작을 수행합니다.
3. A가 참(true)인 경우, B를 읽고 평가합니다.
4. B의 평가 결과에 따라 논리식 전체의 결과를 결정합니다.

이와 같은 방식으로 논리식의 각 피연산자를 평가하고, 직접적으로 점프 동작을 수행하여 단락 평가를 구현할 수 있습니다.

7. 함수

```
int add(int a,int b){
 return a+b;
}
int main(){
 int a,b;
 a = 1;
 b = 0;
 b = add(a,b);
 return 1;
}
0 JTM 10
1 INT_L 5//函数块内部 3个基本值(返回值 基地址 返回值指令序列)和2个参数
2 LDA 0 0
3 LDA 0 3
4 LODS
5 LDA 0 4
6 LODS
7 ADD
8 STOS
9 RET
10 INT_L 2
```

```
25 LDA 0 1//load b
26 INT 3
27 LDA 0 0//load a address
28 LODS//load a value
29 LDA 0 1//load b address
30 LODS//load b value
31 DOWN 5
32 CAL 1//cal function
33 STOS//sto return
```

함수 호출의 경우, Pcode 정의에서는 함수 블록에 진입할 때마다 반환값, 반환값 주소, 반환값 인스트럭션 시퀀스를 저장하기 위해 세 개의 공간을 요청합니다. 함수 호출은 cal (call)을 사용하여 함수 내부로 점프하여 계산을 수행합니다. 마지막으로, 스택의 맨 위에는 반환값의 크기가 있으므로 이를 직접 저장하면 됩니다.

이렇게 함수 호출 시 반환값, 반환값 주소, 반환값 인스트럭션 시퀀스를 저장하는 공간을 할당하고, cal을 사용하여 함수 내부로 점프하여 계산을 수행한 후, 스택의 맨 위에는 반환값의 크기가 있으므로 이를 직접 저장하면 됩니다.