Table of Contents

부산대 정보컴퓨터공학부 2017-2 컴파일러 과제

교수 : 권혁철

200721395 한경수

cva-script

개요

lex(flex), yacc(bison) 을 이용하여 컴파일러 과제를 수행한다.

일반적인 언어에서 쓰는 함수, 조건문, 비교연산자, 우선순위에 따른 수식평가, 출력문을 포함한다.

특히 형선언을 구현하여 강타입언어를 흉내내본다.

이 컴파일러는 lex->parse->execute 순으로 흘러가며, 인터프리터기능을 내장하고 있다.

인터넷의 tiny-c 와 yacc calculator 를 참고 했다.

if, for, while, sub-program, <, >, ==, +, -, *, print 를 지원한다.

개발환경

- ubuntu 16.04
- c++14
- flex
- yacc
- make

실행하기전 flex 와 bison 을 설치해야한다.

조건

자유롭게 하되 형선언을 넣는다.

형선언을 하면 강타입언어처럼 동작하게 했고 형선언이 없으면 자유롭게 변수의 성질이 바뀌게 했다.

코드 구조

lexer.l:

parser.y:

AST.h/AST.c:

interp_expr.c/interp.c/interp.h:

makefile:

큰 흐름은 다음과 같다.

- lexer 에서 scanning 을하고 parser 에서 파스트리를 만든다.
- parser 트리에서 만들어진 트리를 inter_expr 에서 main symbol 을 찾아서 실행한다.

파스트리를 만듦에 있어서 가장 중요한 자료구조는 AST 다.

AST 는 현재 명령어의구조(op), 트리구조(left, right), 범용데이터(val, str), Symbol 을 가진다.

Symbol 은 함수나 변수에 대한 자료구조다. Symbol 은 name, data, 함수표현구조를 가진다.

```
struct Symbol;
struct AST {
    enum code op;
    Symbol *sym;
    AST *left, *right;
    std::string str;
    int val;
} ;

struct Symbol {
    std::string name;
    any data;
    int *addr;
    AST *func_params;
    AST *func_body;
};
```

파싱 개요

파싱의 대상이 되는 예제를 살펴보자. 만만한 IF 문으로 어떻게 파스트리가 만들어지는지 살펴본다.

```
%type <val> statements statement expr primary_expr arg_list
%union {
    AST *val;
}

...

statement:
expr ';'
{ $$ = $1; }
| block
{ $$ = $1; }
| IF '(' expr ')' statement
{ $$ = makeAST(IF_STATEMENT, $3, makeList2($5, NULL)); }
| IF '(' expr ')' statement ELSE statement
{ $$ = makeAST(IF_STATEMENT, $3, makeList2($5, $7)); }
```

IF 문 설명을 위해서 일부 코드를 가져왔다.

```
| IF '(' expr ')' statement ELSE statement
{ $$ = makeAST(IF_STATEMENT, $3, makeList2($5, $7)); }
```

를 보면, $3 = expr, $5 = TRUE_STATMENT, $7 = FALSE_STATEMENT 다.

```
AST *makeAST(enum code op, AST *left, AST *right)
{
    AST *p;
    p = (AST *)malloc(sizeof(AST));
    p->op = op;
    p->right = right;
    p->left = left;
    return p;
}
```

makeAST 를 보면 위와 같이 매우 단순하다. 이 코드의 취지는

```
니가 나중에 뭐할건지 모르겠는데 일단 트리는 만들게.
```

위와 같이 코드를 돌면서 makeAST 와 같은 함수들로 parsing tree 가 만들어진다.

나중에 execute 될 때 executeStatement에 의해 root statement가 실행된다.

```
void executeStatement(AST* p)
{
    if(p == NULL) return;
    switch(p->op){
        case BLOCK_STATEMENT:
            executeBlock(p->left, p->right);
            break;
        case IF_STATEMENT:
            executeIf(p->left, getNth(p->right, 0), getNth(p->ri
ght, 1));
            break;
    }
}
void executeIf(AST* cond, AST* then_part, AST* else_part)
{
    if(executeExpr(cond))
        executeStatement(then_part);
    else
        executeStatement(else_part);
}
any executeExpr(AST *p){...}
```

if 문을 실행하는 코드의 개요는 위와 같다.

흐름제어(break, continue)

특별히 서브프로그램이 가능하게 디자인했고, 흐름제어(continue, break)가 가능하다.

```
jmp_buf* loopControlLabel;

void executeStatement(AST* p)
{
    if(p == NULL) return;
    switch(p->op){
```

```
        case BLOCK_STATEMENT:
            executeBlock(p->left, p->right);
            break;
        case FOR_STATEMENT:
            executeFor(getNth(p->left, 0), getNth(p->left, 1), g
etNth(p->left, 2),
                    p->right);
            break;
        case BREAK_STATEMENT:
            longjmp(*loopControlLabel,  1);
            error("break error\n");
            break;
        case CONTINUE_STATEMENT:
            longjmp(*loopControlLabel, 2);
            error("continue error\n");
            break;
    }
}
void executeFor(AST* init, AST* cond, AST* iter, AST* body)
{
    executeExpr(init);
    jmp_buf loopControl, *loopControlSave;
    loopControlSave = loopControlLabel;
    loopControlLabel = &loopControl;
    for(; executeExpr(cond); executeExpr(iter)) {
        int code = setjmp(loopControl);
        if(code != 0) {
            if(code == 1) {
                break;
            }
            if(code == 2) {
                continue;
            }
        }
        else {
            executeStatement(body);
        }
    }
    loopControlLabel = loopControlSave;
}
```

for 문 또한 if 문과 거의 비슷한 원리로 실행되는데 중요한점은 continue; break; 를 처리함에 있어서 특별한 처리가 필요했다.

코드에 break; 나 continue; 를 만나면 longjmp 를 실행하게 되는데 longjmp 는 setjmp 로 실행한 위치로 특정 signal value 를 함께 넘겨주면서 점프를 할 수 있다.

점프하기전에 loopControlSave 로 현재 context 를 보존시키고 끝나면 복구시켜준다. (2중 for 문 구현)

재귀

재귀는 일반적인 함수안의 함수를 구현하는 문제와 같았다.

핵심은 함수안의 심볼을 어떻게 처리할거냐가 주요이슈다.

```
int executeCallFunc(Symbol *f, AST* args)
{
    int nargs;
    int val;
    jmp_buf ret_env;
    jmp_buf *ret_env_save;

    nargs = executeFuncArgs(f->func_params, args);
    ret_env_save = funcReturnEnv;
    funcReturnEnv = &ret_env;
    if(setjmp(ret_env) != 0){
        val = funcReturnVal;
    } else {
        executeStatement(f->func_body);
    }
    envp -= nargs;
    funcReturnEnv = ret_env_save;
    return val;
}

static int executeFuncArgs(AST* params, AST* args)
{
    Symbol *var;
    any val;
    int nargs;

    if(params == NULL) return 0;
    val = executeExpr(getFirst(args));
    var = getSymbol(getFirst(params));
    nargs = executeFuncArgs(getNext(params), getNext(args));
    Env[envp].var = var;
    Env[envp].val = val;
    envp++;
    return nargs+1;
}
```

파라메터의 개수만큼 Env(Stack) 을 재귀적으로 호출할 때마다 늘어난다.

그리고 해당 심볼을 참조할일이 생기면 getSymbol 로 현재 스택에서 가장 가까운 변수를 찾아서 쓴다.

예제 프로그램

9X9단

```
function main()
{
    i=2;

    for(i=2; i<10; i=i+1)
    {
        for(j=1;j<10;j=j+1)
        {
            k=i*j;
            print(i);
            print("*");
            print(j);
            print("=");
            print(k);
            print("\t");
        }
        print("\n");
    }
}
```

```
2*1=2    2*2=4    2*3=6    2*4=8    2*5=10    2*6=12    2*7=14
  2*8=16    2*9=18
3*1=3    3*2=6    3*3=9    3*4=12    3*5=15    3*6=18    3*7=21
   3*8=24    3*9=27
4*1=4    4*2=8    4*3=12    4*4=16    4*5=20    4*6=24    4*7=28
    4*8=32    4*9=36
5*1=5    5*2=10    5*3=15    5*4=20    5*5=25    5*6=30    5*7=3
5    5*8=40    5*9=45
6*1=6    6*2=12    6*3=18    6*4=24    6*5=30    6*6=36    6*7=4
2    6*8=48    6*9=54
7*1=7    7*2=14    7*3=21    7*4=28    7*5=35    7*6=42    7*7=4
9    7*8=56    7*9=63
8*1=8    8*2=16    8*3=24    8*4=32    8*5=40    8*6=48    8*7=5
6    8*8=64    8*9=72
9*1=9    9*2=18    9*3=27    9*4=36    9*5=45    9*6=54    9*7=6
3    9*8=72    9*9=81
```

break-continue 조합 테스트

```
function printTriangle() {
    i = 0;
    for(i=0; i<10; i=i+1) {
        if(i == 7) {
            break;
        }
        if(i == 3) {
            continue;
        }
        for(j=0; j<i; j=j+1) {
            print("*");
        }
        print("\n");
    }
}

function main()
{
    printTriangle();
}
```

```
*
**
****
*****
******
```

recursive sample

```
function fibo(x)
{
    if(x == 1)
        return 1;
    if(x == 2)
        return 1;
    return fibo(x-1) + fibo(x-2);

}
function main()
{
    print("1 1 2 3 5 8 13 21...\n");
    print("fibonacci = ");
    print(fibo(7));
    print("\n");
}
```

```
1 1 2 3 5 8 13 21...
fibonacci = 13
```

강타입 테스트

```
number num = 3;
string str = "hello";
function main()
{
    if(str == 3) {
        print("compare");
    }

    num = "hello";
}
```

```
convert error!
```

어려웠던 점 & 한계

parser 를 돌리면서 트리파악은 할 수 있었는데(변수 카운팅과 같은), 파스트리를 구성하고 구성된 트리를 실행시킨다는 점에서 처음해보는 것이라 힘들었다.

지금까지 해왔던 프로그래밍방식과는 너무나 달라서 적응하는데도 꽤나 오랜 시간이 걸렸던걸로 기억한다. 흐름제어를 완성시키고 sub-program 를 구현하는데에 있어서도 적지않은 고뇌의 시간을 가졌다. 단순 구조화된 프로그래밍 방식은 그냥 루트만 실행시키면 되는건데 sub-program 은 루트에서 쭉 실행 되는 개념이 아니라 return시에 점프를 해야하거나 context 를 관리해줘야하기 때문이다.

이 cva-script 의 한계로는 symbol table 이나 environment 테이블이 정적으로 선언되어있어서 프로그램이 많이 커지면 memory fault 가 날 것이다. std::stack 이나 다른 동적 container 로 구현을 바꾸거나 하면 될 것이다. 또한, 사실상 모든 변수가 전역화되어 선언되는데 함수안에서만 변수가 선언되게 스펙을 고치고 지역변수를 stack 으로 관리하면 될 것 같다.