Abstract Syntax Tree

컴파일러, 6번째 시간

경민기, 2025-10-14

시험 문제에 대해

- 컴파일러 전 과정과 도구에 대한 간략한 이해
- 계산기 코드에 문법 추가
- 정규표현식
- BNF

지금까지의 내용

- 인공지능이 ‘사람을 만든다’는 목표가 명확한 학문입니다.
  - 1960년대, 1980년대와 같이 ‘겨울’이 와도 목표가 명확하기에 살아남았습니다.
- 컴파일러도 ‘컴파일러를 만든다’는 목표에 매우 충실한 과목입니다.
- Mini C 컴파일러를 만듭니다.

수업 변경사항

이론 커리큘럼 다 따라가면 텀 프로젝트를 할 수 없어 변경합니다.

- 이번 주: Abstract Syntax Tree
- 다음 주: 중간고사
- C 컴파일러 만들기
  - 언어 설계
    - Mini C 언어 설계 (int, float, if, while, return)
    - 확장된 scanner.l, parser.y 작성
  - 심볼테이블 및 타입검사
    - 변수 선언/사용 관리
    - 심볼테이블 구현 (이름, 타입, 주소)
    - 타입 일치 검사
  - 중간 코드 생성 (IR)
    - AST → 3주소 코드(3-address code)
    - 임시 변수(t1, t2) 생성
    - 중간 코드 출력
  - 기초 코드 생성 (Assembly)
    - IR → x86_64 어셈블리
    - 산술연산, 비교연산, 분기(if, while) 코드 생성
  - 코드 최적화
    - 상수 폴딩(Constant Folding)
    - Dead Code 제거
    - 단순한 Copy Propagation
  - 함수 호출 및 스택 프레임
    - 함수 정의 및 호출(call, ret)
    - 스택 프레임 구성, 인자 전달
    - 지역변수/매개변수 관리
  - 링킹 및 실행 테스트
    - 어셈블리 → 실행파일(gcc로 컴파일)
    - 간단한 Mini C 프로그램 실행(fib(10), sum(5))

오늘 수업 내용: AST

- AST 개념
- AST 구조 예제
- AST 계산기 예제
- 0으로 나누기 검사 예제

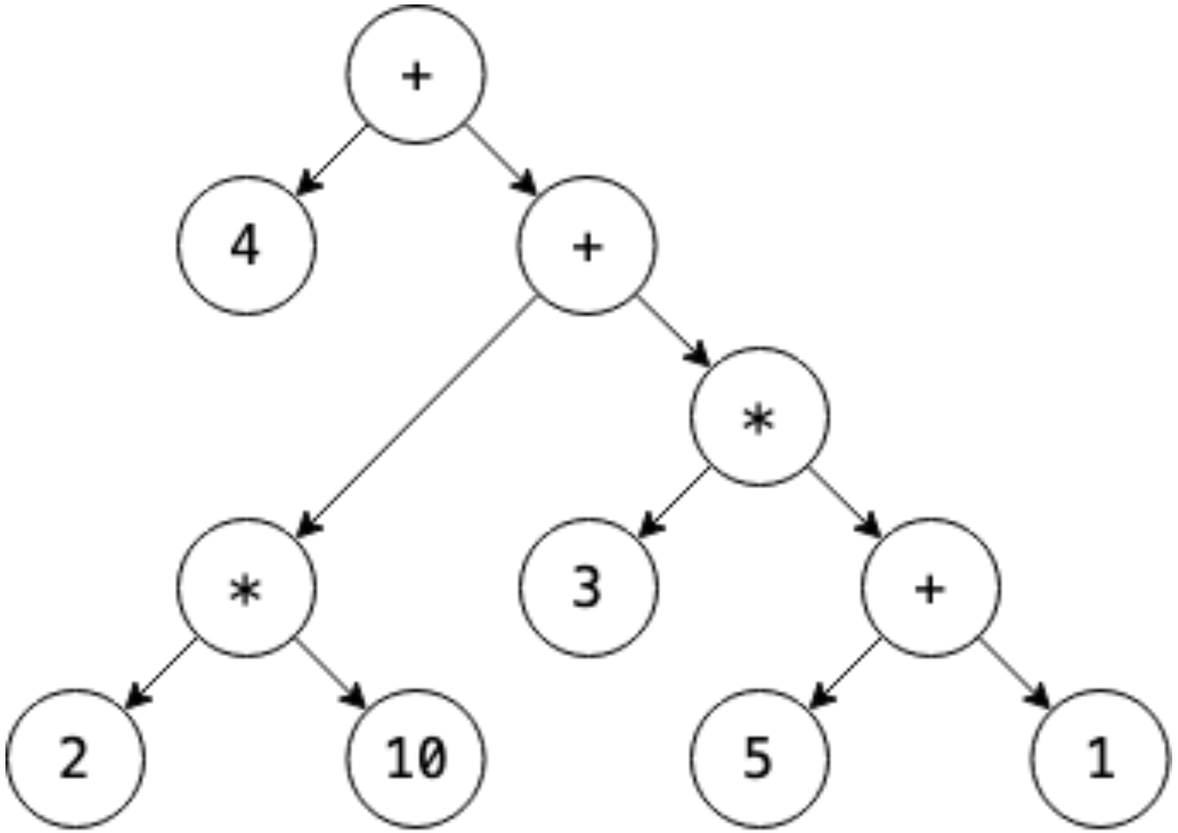
AST (Abstract Syntax Tree)

AST란?

- AST는 파싱(Parsing) 이후 단계에서 만들어집니다.
- flex(어휘 분석기)가 토큰(token)을 만들고, bison(구문 분석기)이 이 토큰들을 문법에 맞게 조합해 AST를 생성합니다.
- 이 트리는 문법 구조를 간결하게 표현하여 이후의 타입 검사(type checking), 중간 코드 생성, 최적화, 기계어 코드 생성 등에 사용됩니다.

- 요약
  - 파싱된 프로그램의 구문 구조를 트리 형태로 표현
  - 구체 문법과 달리 의미만 남긴 추상 트리
  - 각 노드는 연산자, 피연산자, 리터럴 등을 나타냄

- 예:
  - 4 + 2 * 10 + 3 * (5 + 1)
  - expr → expr '+' term, term → term '-' factor 등의 규칙에 따라 파싱됨
  - AST는 다음과 같이 표현 가능



AST를 코드로 바꾸면

1 + 2 * 3

```
1   typedef enum { NODE_NUM, NODE_ADD, NODE_SUB, NODE_MUL, NODE_DIV } NodeType;
2
3   typedef struct Node {
4       NodeType type;
5       double value;           // 숫자일 경우
6       struct Node *left;      // 왼쪽 자식
7       struct Node *right;     // 오른쪽 자식
8   } Node;
```

```
Node *mul = new_node(NODE_MUL, new_num(2), new_num(3));
Node *add = new_node(NODE_ADD, new_num(1), mul);
```

- 트리에 대한 Evaluation 작업을 수행하면 됩니다.

계산기에 AST 적용한 예제

scanner.l

6wk > calc_ast_1 > scanner.l

```
%option noyywrap

%{
#include <stdlib.h>      /* atof */
#include "parser.tab.h"  /* yylval, tokens */
%}

%%
[0-9]+(\.[0-9]+)?   { yylval.num = atof(yytext); return NUMBER; }
[ \t\n]+             ;
"+"                 return '+';
"-"                 return '-';
"*"                 return '*';
"/"                 return '/';
";"                 return ';';
.                   return yytext[0];
%%
```

parser.y (1)

```
1   %{
2   #include <stdio.h>
3   #include "ast.h"
4   int yylex(void);
5   void yyerror(const char *s);
6   AST *root;
7   %}
8
9   %code requires {  /* parser.tab.h에도 AST 타입 포함 */
10    #include "ast.h"
11  }
12
13  %union {
14      double num;
15      AST *node;
16  }
17
18  %token <num> NUMBER
19  %type  <node> expr term factor
20
21  %right UPLUS UMINUS
22  %left '+' '-'
23  %left '*' '/'
24
25  %start input
```

yylex()는 flex 스캐너에서 오는 토큰 공급자.
yyerror()는 구문 오류 메시지 출력.
root는 한 줄 파싱이 끝났을 때의 AST 루트 저장소.

생성되는 parser.tab.h에도 AST 타입 선언이 포함되도록 하는 부분.
다른 소스(scanner.l, main.c, 등)가 parser.tab.h만 include 해도 AST*를 알 수 있게 함

파서의 yylval이 담을 수 있는 타입 집합
숫자 토큰용 double, 구문 규칙 결과(식 노드)용 AST*.

NUMBER 토큰은 double 값
비단말 expr/term/factor는 AST*를 반환.

* /가 + -보다 우선.
단항 +/-는 가장 높고(명시적 %prec로 적용), 우결합(중첩 단항에 대비).

문법의 진입점은 input.

parser.y (2)

```
/* 여러 줄을 EOF까지 파싱 */
input
  : /* empty */
  | input line
  ;

line
  : expr ';'      { root = $1; print_ast(root, 0); free_ast(root); }
  | error ';'     { yyerrok; }          /* 에러 복구: 해당 줄만 건너뜀 */
  ;
```

EOF까지 line 들을 연속 파싱. 비어 있어도 OK.

한 줄은 expr + 세미콜론.

- 줄이 끝나면 root에 AST 저장
- print_ast()로 보기 좋게 출력
- free_ast()로 메모리 해제

에러 복구: error ';' 규칙 + yyerrok; 으로 그 줄만 건너뛰고 다음 줄로 진행.

parser.y (3)

```
/* 중위 표기 + 우선순위/결합규칙 */
expr
  : expr '+' term  { $$ = new_op(NODE_ADD, $1, $3); }
  | expr '-' term  { $$ = new_op(NODE_SUB, $1, $3); }
  | term
  ;

term
  : term '*' factor { $$ = new_op(NODE_MUL, $1, $3); }
  | term '/' factor { $$ = new_op(NODE_DIV, $1, $3); }
  | factor
  ;

factor
  : NUMBER                        { $$ = new_num($1); }
  | '(' expr ')'                  { $$ = $2; }
  | '-' factor   %prec UMINUS     { $$ = new_op(NODE_SUB, new_num(0), $2); }
  | '+' factor   %prec UPLUS      { $$ = $2; }
  ;

%%
```

좌재귀로 구현 → 자연스러운 좌결합(+/-) 파싱
new_op(kind, lhs, rhs)로 이항 연산 노드 생성

기본 원자: 숫자, 괄호식.
단항 부호:
-x는 0 - x로 변환해 이항 연산과 동일한 AST 형태를 재사용.
%prec UMINUS/UPLUS로 단항의 우선순위를 올려 다른 연산보다 먼저 결합되도록 함.

ast.h

6wk > calc_ast_1 > C ast.h

```c
// ast.h
#ifndef AST_H
#define AST_H

#include <stdio.h>

typedef enum {
    NODE_NUM,
    NODE_ADD,
    NODE_SUB,
    NODE_MUL,
    NODE_DIV
} NodeType;

typedef struct AST {
    NodeType type;
    double value;
    struct AST *left;
    struct AST *right;
} AST;

/* 함수 선언만 둔다 */
AST *new_num(double val);
AST *new_op(NodeType type, AST *left, AST *right);
void  print_ast(AST *node, int depth);
void  free_ast(AST *node);

#endif
```

ast.c (1)

6wk > calc_ast_1 > C ast.c

```c
// ast.c
#include <stdio.h>
#include <stdlib.h>
#include "ast.h"

AST *new_num(double val) {
    AST *node = (AST *)malloc(sizeof(AST));
    node->type = NODE_NUM;
    node->value = val;
    node->left = node->right = NULL;
    return node;
}

AST *new_op(NodeType type, AST *left, AST *right) {
    AST *node = (AST *)malloc(sizeof(AST));
    node->type = type;
    node->value = 0.0;
    node->left = left;
    node->right = right;
    return node;
}
```

숫자(리터럴) 노드를 만드는 함수

예를 들어, 파서가 3.14를 읽으면 new_num(3.14)가 호출됨
AST 구조체의 각 필드는 이렇게 채워짐
- type: NODE_NUM (숫자형 노드임을 나타냄)
- value: 실제 숫자값 (3.14)
- left, right: 자식이 없으므로 NULL

연산자 노드를 만드는 함수 (예: +, -, *, /)
left, right는 각각 왼쪽 피연산자, 오른쪽 피연산자를 가리키는 하위 노드
type은 NODE_ADD, NODE_SUB, NODE_MUL, NODE_DIV 중 하나

ast.c (2)

6wk > calc_ast_1 > C ast.c

```c
void print_ast(AST *node, int depth) {
    if (!node) return;
    for (int i = 0; i < depth; i++) printf("  ");
    switch (node->type) {
        case NODE_NUM: printf("NUM(%.2f)\n", node->value); break;
        case NODE_ADD: printf("ADD\n"); break;
        case NODE_SUB: printf("SUB\n"); break;
        case NODE_MUL: printf("MUL\n"); break;
        case NODE_DIV: printf("DIV\n"); break;
    }
    print_ast(node->left, depth + 1);
    print_ast(node->right, depth + 1);
}

void free_ast(AST *node) {
    if (!node) return;
    free_ast(node->left);
    free_ast(node->right);
    free(node);
}
```

실행 결과

```
mingi_kyung@mg-tpx240:~/compiler_class/6wk/calc_ast_1$ ./calc_ast
1 +2 +3 *4 /5
;
ADD
  ADD
    NUM(1.00)
    NUM(2.00)
  DIV
    MUL
      NUM(3.00)
      NUM(4.00)
    NUM(5.00)
```

계산기에 AST 적용하고, 계산한 예제

계산을 수행하기 위해 다음 사항을 변경합니다.

- scanner.l 파일은 변경 안 함
- parser.y 파일은 문법 약간 변경
- ast.h / ast.c 파일에는 평가함수 추가

parser.y 변경사항

```
33  line
34    : expr ';'          { printf("= %.10g\n", $1); fflush(stdout); }
35    | ';'               { fprintf(stderr, "[parse] 빈 식입니다 (line %d, col %d)\n",
36                                  @$.first_line, @$.first_column); }
37    | error ';'         { yyerrok; }
38    /* 여는 괄호 없이 ')' 가 등장한 경우: 결과는 그대로 출력하되 경고 */
39    | expr ')' ';'       {
40                          fprintf(stderr,
41                                  "[parse] 여는 괄호 없이 닫는 괄호가 있습니다 (line %d, col %d)\n",
42                                  @2.first_line, @2.first_column);
43                          printf("= %.10g\n", $1); fflush(stdout);
44                        }
45
```

ast.h 변경사항

- 마지막 줄에 다음 함수 선언 추가
  - double eval_ast(AST *node, int *err);

ast.c 변경사항

```c
/* → 추가: AST 평가 */
double eval_ast(AST *node, int *err) {
    if (!node) { if (err) *err = 1; return 0.0; }

    switch (node->type) {
        case NODE_NUM:
            return node->value;

        case NODE_ADD: {
            double l = eval_ast(node->left, err);
            if (err && *err) return 0.0;
            double r = eval_ast(node->right, err);
            if (err && *err) return 0.0;
            return l + r;
        }

        case NODE_SUB: {
            double l = eval_ast(node->left, err);
            if (err && *err) return 0.0;
            double r = eval_ast(node->right, err);
            if (err && *err) return 0.0;
            return l - r;
        }

        case NODE_MUL: {
            double l = eval_ast(node->left, err);
            if (err && *err) return 0.0;
            double r = eval_ast(node->right, err);
            if (err && *err) return 0.0;
            return l * r;
        }

        case NODE_DIV: {
            double l = eval_ast(node->left, err);
            if (err && *err) return 0.0;
            double r = eval_ast(node->right, err);
            if (err && *err) return 0.0;
            if (r == 0.0) {
                fprintf(stderr, "[sem] 0으로 나눌 수 없습니다\n");
                if (err) *err = 1;
                return 0.0;
            }
            return l / r;
        }
    }

    if (err) *err = 1;
    return 0.0;
}
```

실행결과

```
mingi_kyung@mg-tpx240:~/compiler_class/mid/calc_ast_2$ ./calc_ast
23 + 12 / 3 *7;
ADD
  NUM(23.00)
  MUL
    DIV
      NUM(12.00)
      NUM(3.00)
    NUM(7.00)
= 51
(1+2)*3;
MUL
  ADD
    NUM(1.00)
    NUM(2.00)
  NUM(3.00)
= 9
```

계산기에 0으로 나누기 예외를 추가한 예제

수정사항

- ast와 parser.y

parser.y 수정

```
35  line
36    : expr ';' {
37        root = $1;
38        print_ast(root, 0);
39
40        int err = 0;
41        double v = eval_ast(root, &err);
42        if (!err) {
43          printf("= %.10g\n", v);
44        } else {
45          /* 에러 메시지는 eval_ast 내부에서 출력됨 */
46          printf("= <error>\n");
47        }
48        free_ast(root);
49      }
50    | error ';' { yyerrok; }
51    ;
```

ast.h 수정

```
22    /* 함수 선언만 둔다 */
23    AST *new_num(double val);
24    AST *new_op(NodeType type, AST *left, AST *right);
25    void  print_ast(AST *node, int depth);
26    void  free_ast(AST *node);
27
28    int     check(AST *node);           // 오류 없으면 0, 있으면 1
29    double  eval(AST *node, int *err); // 간편 평가 래퍼
30    double eval_ast(AST *node, int *err);
```

ast.c 수정

```
 94  int check(AST *node) {
 95      if (!node) return 0;
 96      if (check(node->left))  return 1;
 97      if (check(node->right)) return 1;
 98
 99      if (node->type == NODE_DIV) {
100          int err = 0;
101          double r = eval_ast(node->right, &err);
102          if (!err && r == 0.0) {
103              fprintf(stderr, "[sem] 0으로 나눌 수 없습니다 (사전 점검)\n");
104              return 1;
105          }
106      }
107      return 0;
108  }
109
110  double eval(AST *node, int *err) {
111      if (check(node)) {
112          if (err) *err = 1;
113          return 0.0;
114      }
115      if (err) *err = 0;
116      return eval_ast(node, err);
117  }
```

실행결과

```
⊗ mingi_kyung@mg-tpx240:~/compiler_class/mid/calc_ast_3$ ./calc_ast
  123+456*5;
  ADD
    NUM(123.00)
    MUL
      NUM(456.00)
      NUM(5.00)
  = 2403
  123/0;
  DIV
    NUM(123.00)
    NUM(0.00)
  [sem] 0으로 나눌 수 없습니다 (사전 점검)
```