7. Abstract Syntax Tree (AST)

충북대학교

이 재 성

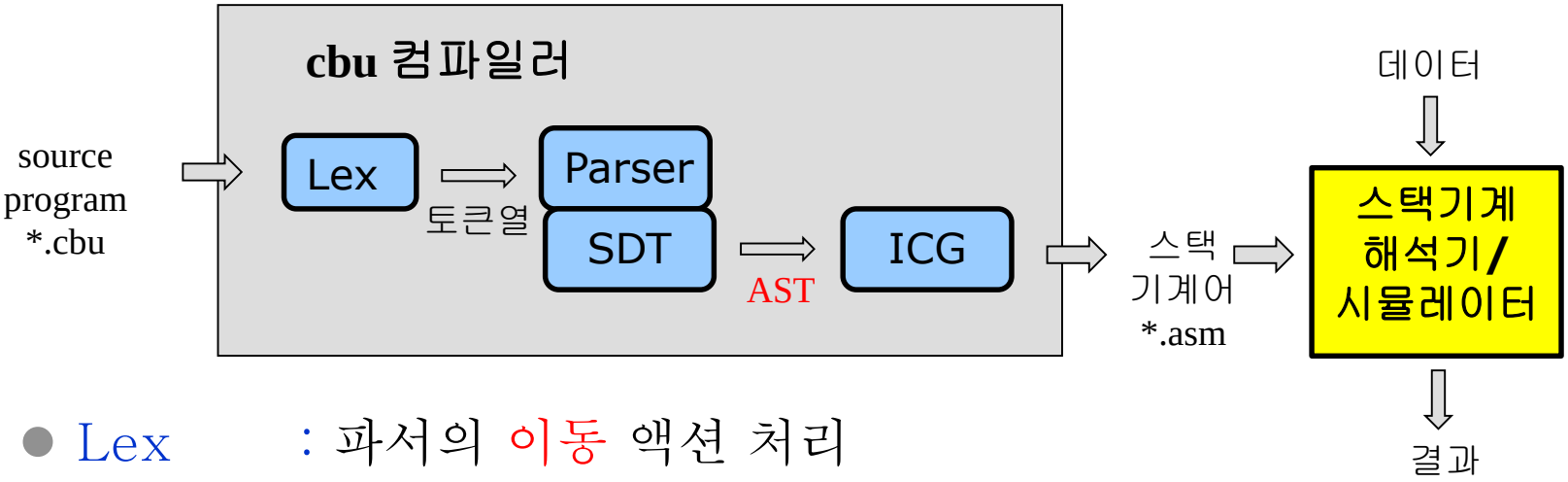
LaKE LAB

학습내용

- AST 개념
- AST 구현
- AST 및 코드생성

컴파일러 구현 구조

컴파일러 구현 모델 예



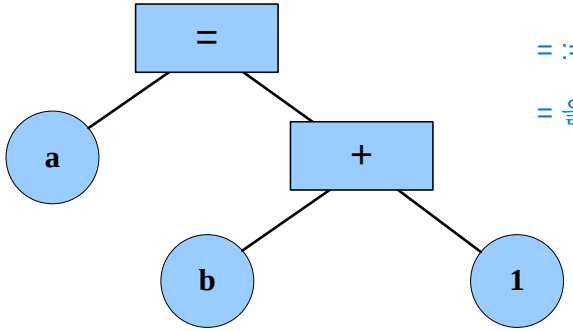
- Lex : 파서의 이동 액션 처리
- Parser : main program (LR parser)
- SDT : 파서의 축소 액션 처리(AST 생성)
- ICG : AST를 탐색하여 중간코드생성

※ 의미 분석과 중간코드 생성을 효율적으로 처리하기 위해서 AST의 design은 매우 중요.

AST 개념

- AST (Abstract Syntax Tree)
  - 언어구조 표현에 유용한 압축된 파스트리 형태

ex) a = b + 1;

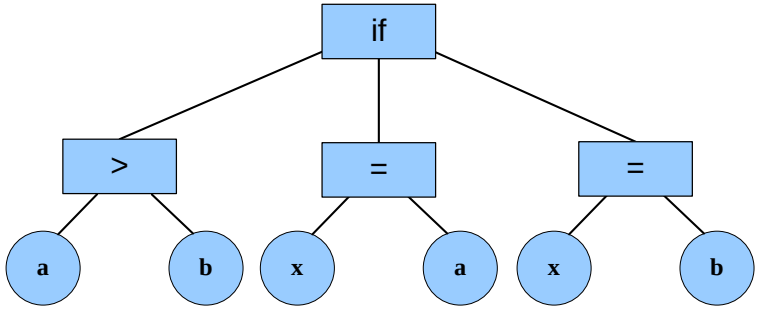


= := right sub tree

= 을 a에 넣어라.

-> operand = 피연산자

ex) if (a > b) x = a; else x = b;



internal node

AST의 구축 예

- 구축 함수
  - buildTree(op, left, right): left와 right를 좌우 자식 필드로 가진 연산자 op 노드를 생성하고, 그 노드의 포인터 리턴
  - buildNode(a): 터미널 a에 대한 노드를 생성하고 그 포인터를 리턴
- 의미 규격
  - 합성속성 nptr에 각 함수 호출에서 리턴된 포인터들을 추적 관리

Production	Semantic Rules
E -> E_1 + T	E.nptr := buildTree('+', E_1.nptr, T.nptr)
E -> E_1 –T	E.nptr := buildTree('－', E_1.nptr, T.nptr)
E -> T	E.nptr := T.nptr
T -> (E)	T.nptr := E.nptr
T -> a	T.nptr := buildNode(a)

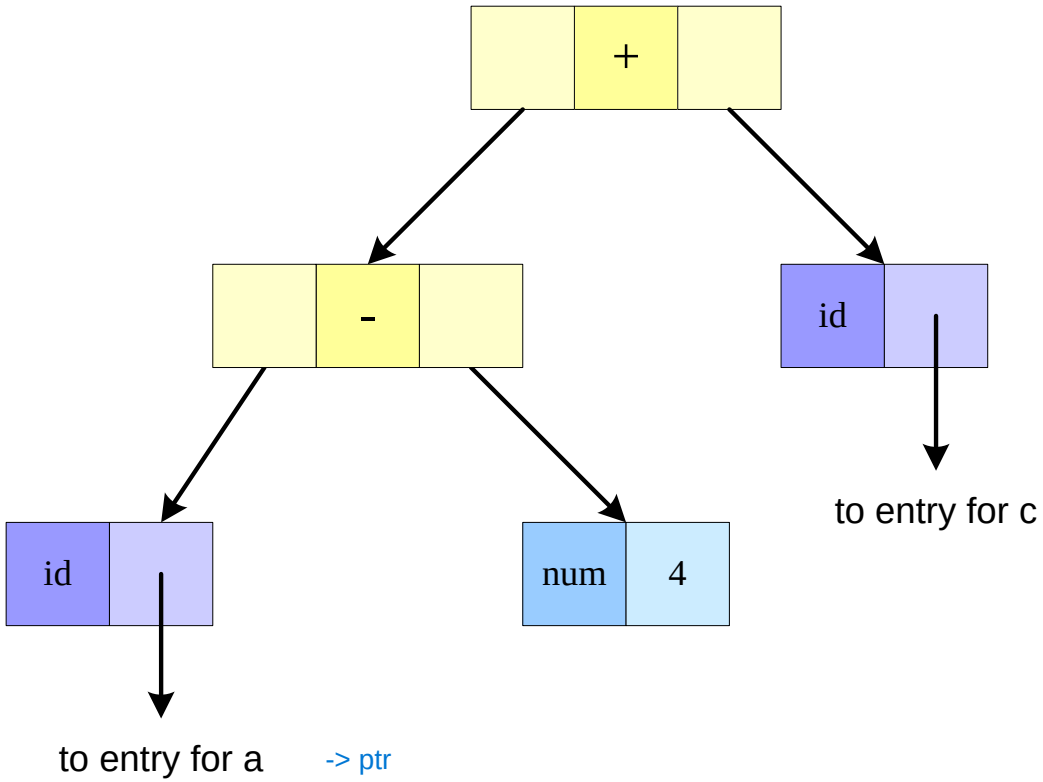
buildTree(parent node, left child, right child)

AST 표현 예

- a-4+c의 AST

a + b * c의 AST:

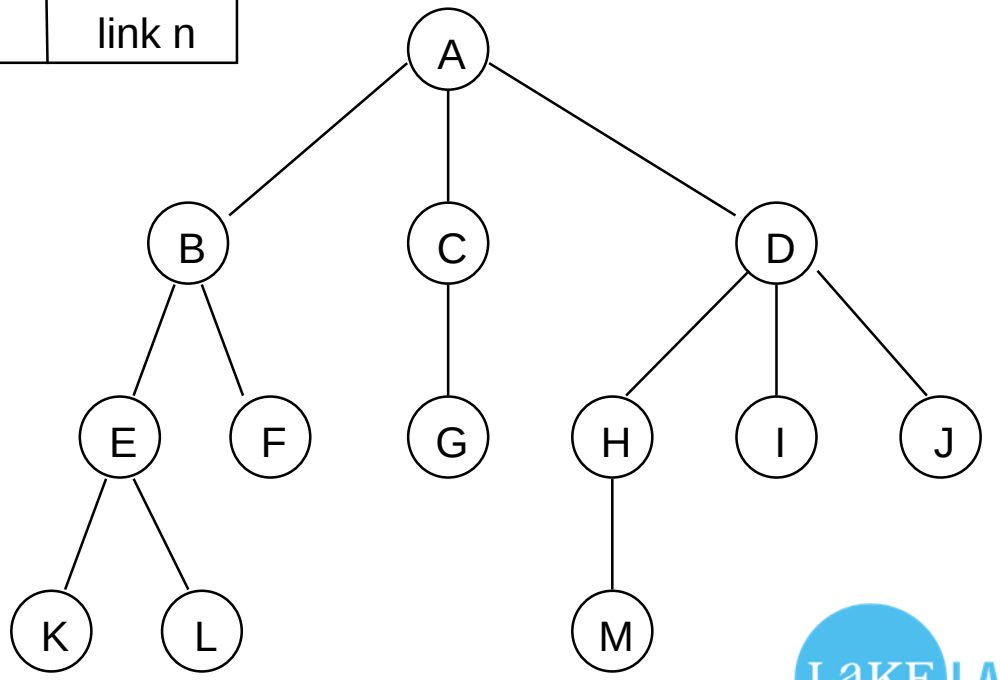
```
    +
      *
a   b   c
```



- 차수가 많은 노드이 구현은?

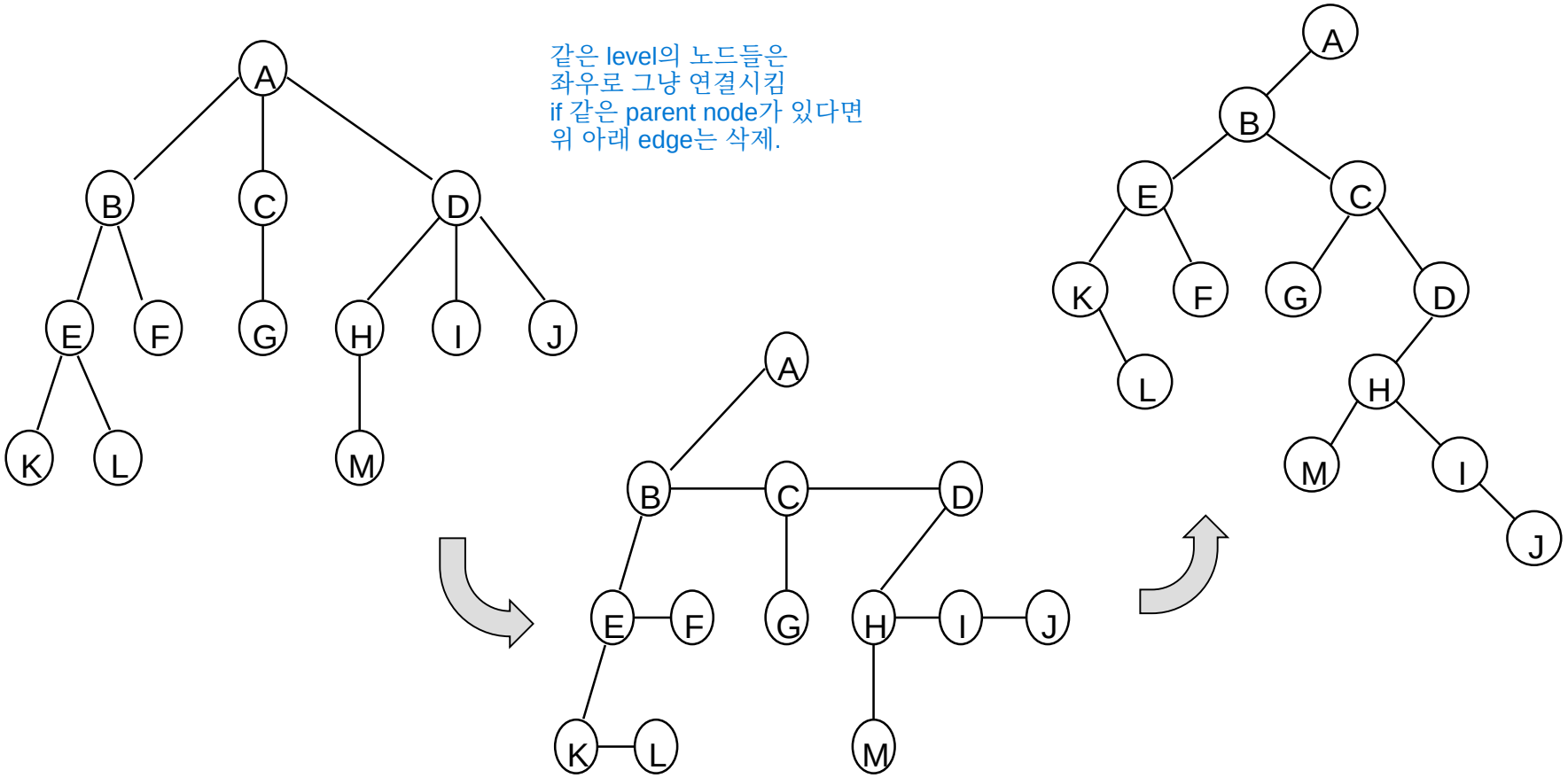
(review) 트리 표현

- 트리의 메모리 내 표현
  - 최대 차수로 표현 제한
  - 각 노드별 자식 수가 달라 메모리 낭비
  - n 차수 트리의 구조: 각 링크에 자식 연결



(review) 이진 트리로 변환

- 좌-자식 우-형제 형태의 이진 트리로 변환



(review) 이진 트리로 변환

- 트리 구현
  - 왼쪽-자식 오른쪽-형제 노드 표현
  - 노드의 크기가 고정
    - 작업 용이
    - 한 노드당 2개의 링크/포인터
  - 왼쪽-자식 오른쪽-형제 노드 구조

data	
left child	right sibling

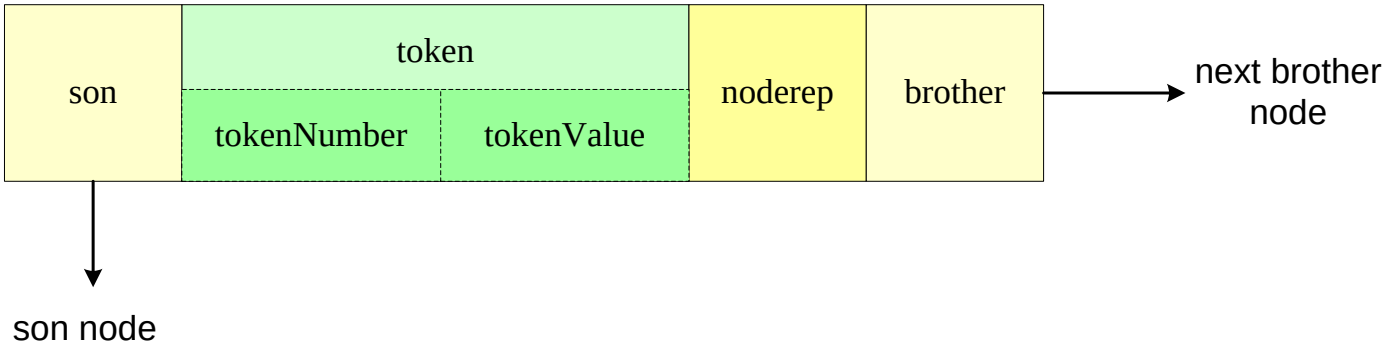
원래 노드 구조:

data link1 link2 link3 ...

이진 AST 트리

자료구조

- AST노드 형태



- Node 구조

```
struct tokenType {
        int tokenNumber;                    // 토큰 번호
        char * tokenValue;                  // 토큰 값
};
typedef struct nodeType {
        struct tokenType token;             // 토큰 종류
        enum {terminal, nonterm} noderep;   // 노드 종류
        struct nodeType *son;               // 왼쪽 링크
        struct nodeType *brother;           // 오른쪽 링크
} Node;
```

AST 생성

- 이동 → buildNode : simple and easy
- 축소 → buildTree : complex and difficult

토큰이 들어올 때

축소의 예:

expr -> expr + expr

RHS가 LHS가 됨.

이동 액션 (파싱) :

- 토큰이 의미가 있으면 buildNode 호출

```
Node *buildNode(struct tokenType token)
{
    Node *ptr;
    ptr = (Node *) malloc(sizeof(Node));
    if (!ptr) { printf("malloc error in buildNode()\n");
            exit(1);
    }
    ptr->token = token;
    ptr->noderep = terminal;
    ptr->son = ptr->brother = NULL;
    return ptr;
}
```

좌 자식, 우 형제 트리인걸 명심!!

축소 액션 (파싱):

- 기본 개념

  if the production rule is meaningful

  1. build subtree
     - linking brothers
     - making a subtree

  else

  2. only linking brothers

축소를 하면 서브 트리를 만들면서 AST가 축소된다는 느낌임.

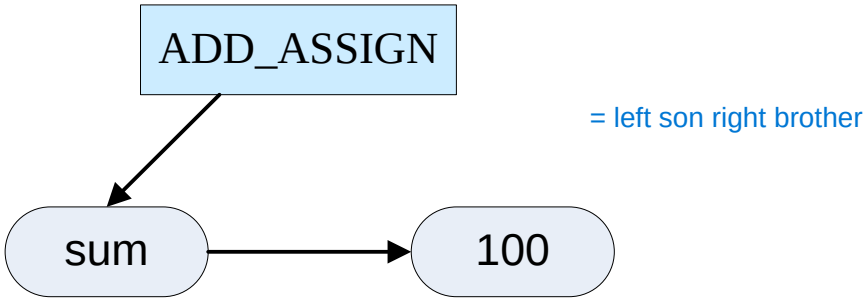
- buildTree() 함수
  - step 1: finding a first index with node in value stack.
  - step 2: linking brothers.
  - step 3: making subtree root and linking son if meaningful

AST 및 코드생성 – 배정문 처리

- 예
  - 프로그램

```
sum += 100;
```

  - AST



  - 스택기계 코드

```
lvalue sum
rvalue sum        // load sum
push    100
+
:=                // store sum
```

AST 및 코드생성– 이진 연산자 처리

- 이진 연산자 처리 코드

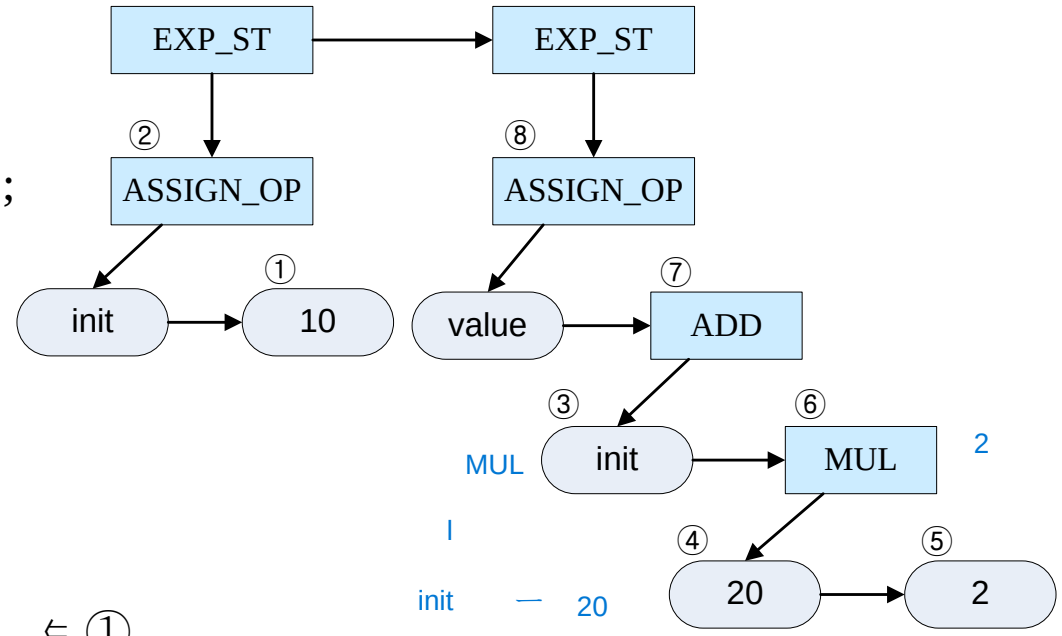
```
void processOperator(Node *ptr)  {
    switch (ptr->token.number) {
        //…
        // binary(arithmetic/relational/logical) operators
        case ADD: case SUB: case MUL: case DIV: case MOD:  case EQ:  ...:
        {
            // step 1: visit left operand
            if (lhs->noderep == nonterm) processOperator(lhs);
            else rv_emit(lhs);
            // step 2: visit right operand
            if (rhs->noderep == nonterm) processOperator(rhs);
            else rv_emit(rhs);
            // step 3: visit root
            switch (ptr->token.number) {
                // arithmetic, relational, logical operators
            }
        }
//…
```

AST 및 코드생성– 이진 연산자 처리

- 예
  - 프로그램

```
init = 10;
value = init + 20 * 2;
```

  - AST



  - 스택기계 코드

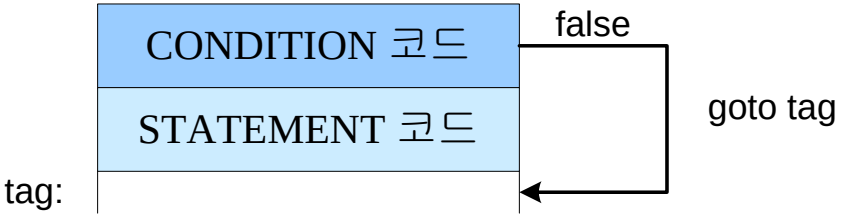
```
lvalue   init
push      10      ⇐ ①
:=                ⇐ ②
lvalue   value
rvalue   init     ⇐ ③
push      20      ⇐ ④
push      2       ⇐ ⑤
*                 ⇐ ⑥
+                 ⇐ ⑦
:=                ⇐ ⑧
```

엮에 적은건 value = init * 20 + 2 일 경우에

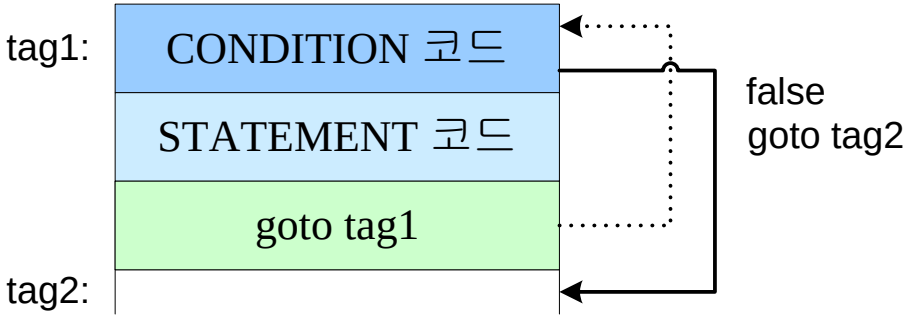
노드가 이 내용으로 대체 된다는 것을 나타냄.

AST 및 코드생성 – 제어문 처리

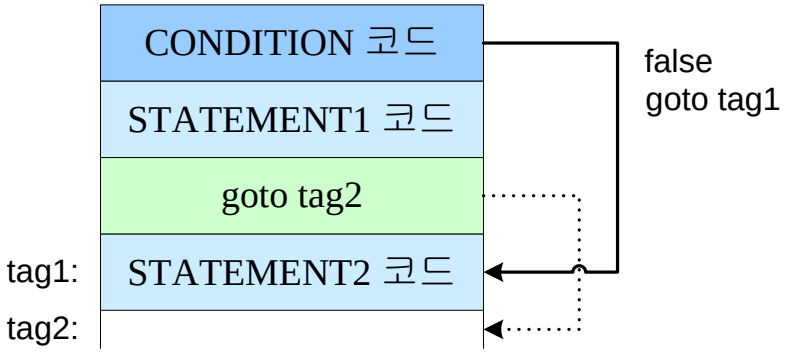
- 제어문의 스키마
  - if 구조



  - while 구조

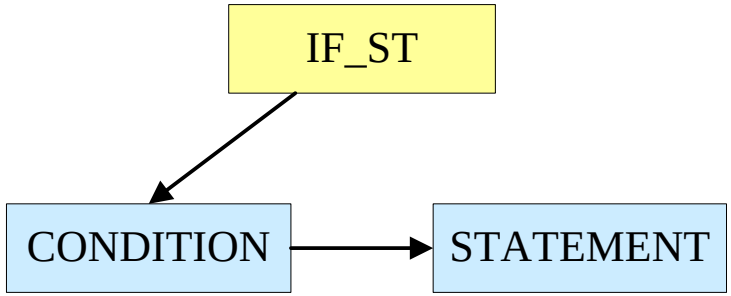


- if – else 구조



AST 및 코드생성

- if 문
  - AST

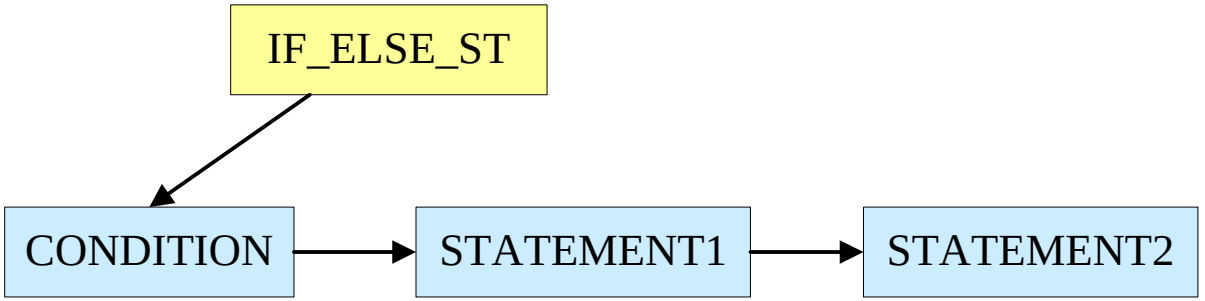


  - 코드 일부

```
void processStatement(Node *ptr)
{
    //…
      case IF_ST:
      {
        char label[LABEL_SIZE];
        genLabel(label);
        processCondition(ptr->son);       // condition part
        emitJump(fjp, label);

        processStatement(ptr->son->brother); // true part
        emitLabel(label);
      }
      //…
}
```

AST 및 코드생성

- if-else 문
  - AST



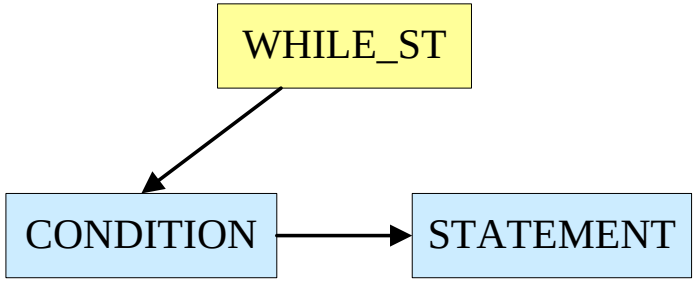
  - 코드 일부

```
void processStatement(Node *ptr)
{
    //…
     case IF_ELSE_ST:
     {
         char label1[LABEL_SIZE], label2[LABEL_SIZE];
         genLabel(label1); genLabel(label2);
         processCondition(ptr->son);                          // condition part
         emitJump(fjp, label1);

         processStatement(ptr->son->brother);                 // true part
         emitJump(ujp, label2);

         emitLabel(label1);
         processStatement(ptr->son->brother->brother);        // false part
         emitLabel(label2);
     }
        //…
}
```

AST

AST 및 코드생성

- while 문
  - AST



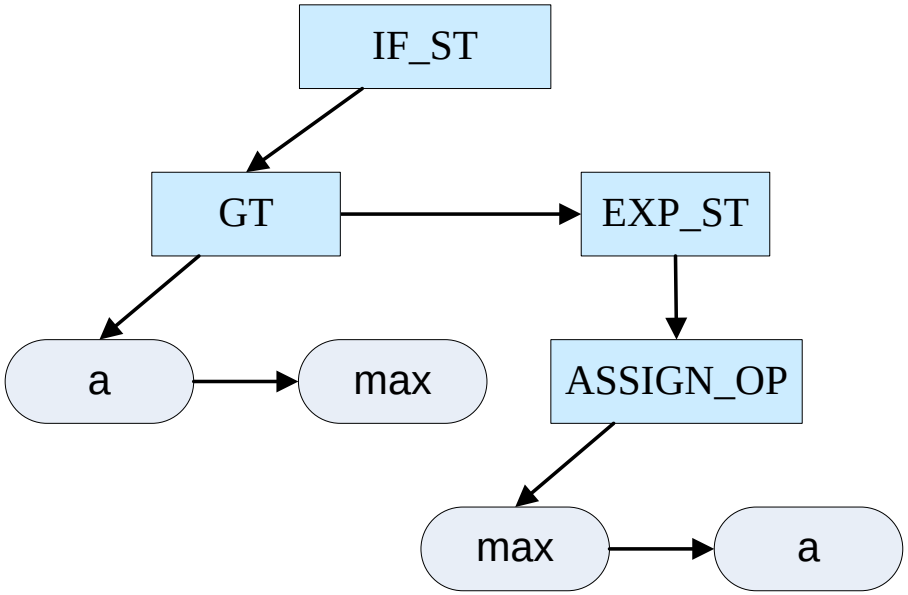
  - 코드 일부

```
void processStatement(Node *ptr)
{
    //…
     case WHILE_ST:
     {
         char label1[LABEL_SIZE], label2[LABEL_SIZE];

         genLabel(label1); genLabel(label2);
         emitLabel(label1);
         processCondition(ptr->son);                  // condition part
         emitJump(fjp, label2);

         processStatement(ptr->son->brother);        // loop body
         emitJump(ujp, label1);

         emitLabel(label2);
     }
        //…
}
```

AST 및 코드생성

- 예 1
  - 코드

    if (a > max) max = a;
  - AST



  - 스택기계 코드

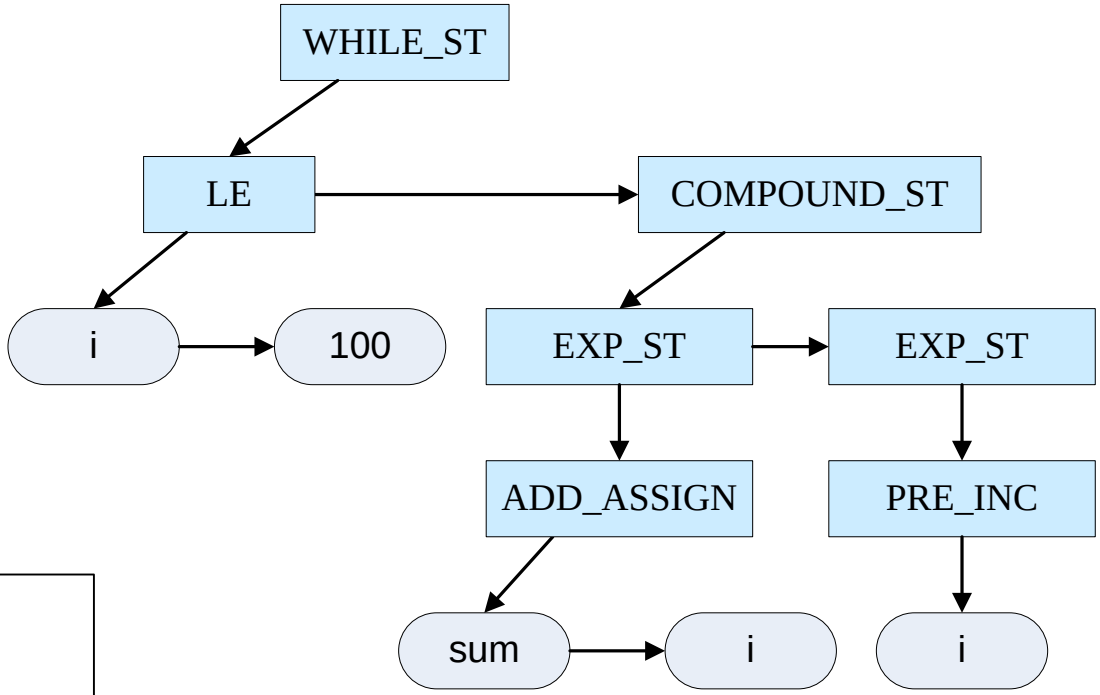
```
rvalue      a
rvalue      max
<    > 오타
gofalse     label

lvalue      max
rvalue      a
:=

label:
```

AST 및 코드생성

- 예 2
  - 코드

```
while (i <= 100) {
    sum += i;
    ++i;
}
```

  - AST



  - 스택기계 코드

```
looplabel:
rvalue    i
push      100
<=
gofalse   outlabel
```

```
lvalue    sum
rvalue    sum
rvalue    i
+
:=
```

```
lvalue    i
rvalue    i
push      1
+
:=
go        looplabel
outlabel:
```

참고 문헌

- [1] 오세만, “컴파일러 입문”, 정익사, 2004.