컴파일러 입문

제 10 장

중간 코드 생성

목 차

10.1 Introduction

10.2 Syntax-Directed Translation

10.3 Code Generation

10.4 U-Code Translator

10.1 Introduction [1/3]

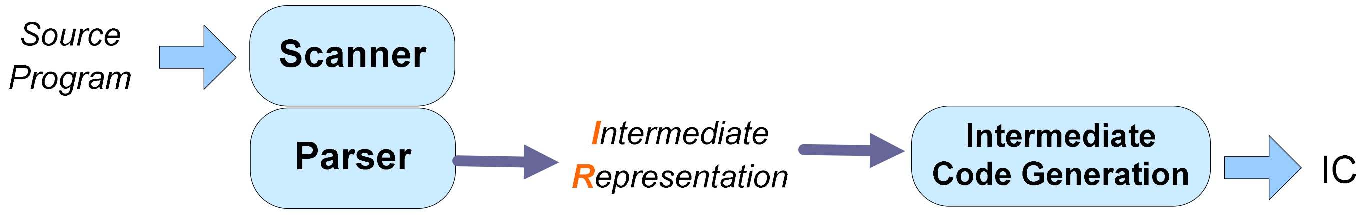
- Formal Specification
  - lexical structure :
  - syntactic structure :
  - the remaining phases of compilation : no such notations

    ⇒ but, we use a syntax-directed translation scheme which is a method semantic rules(or actions) with production.

- SDTS ::= cfg +
  - cfg의 production rule에 있는 grammar symbol을 이용하여 직접 semantic action을 기술하는 방법.
  - AST generation
  - Attribute grammar

Introduction [2/3]

- Intermediate code generation
  - the phase that generates an explicit intermediate code of the source program.
  - after syntax and semantic analysis.
  - ♣ A Model for Intermediate code generation

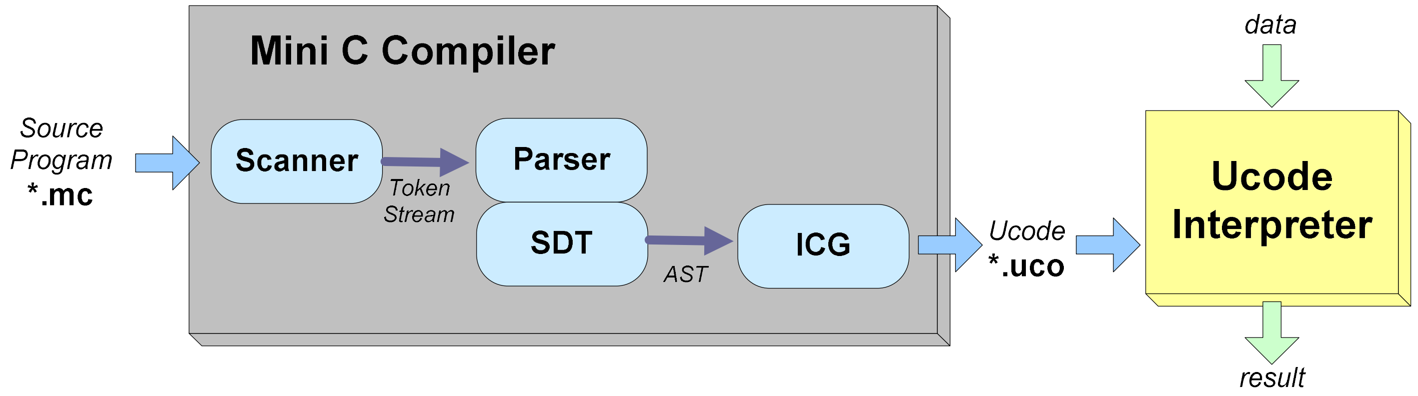


- Our implementations:

Source program	:	Mini C 프로그램
Intermediate Representation	:	Abstract Syntax Tree (AST)
Intermediate code	:	U-Code
Execution	:	U-Code Interpreter

Introduction [3/3]

Implementation Model



- **scanner** : action of parser
- **parser** : main program (LR parser)
- **SDT** : action of parser (AST generation)
- **ICG** : Intermediate code generation by traversing AST.

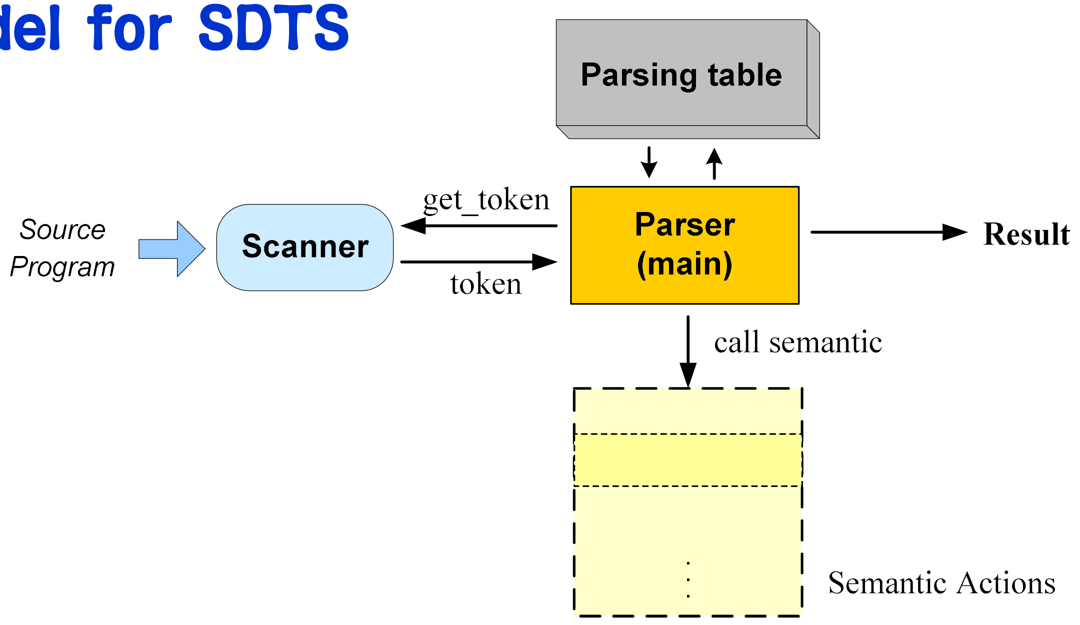
※ Semantic Analysis와 Intermediate Code Generation을 효율적으로 처리하기 위해서 **AST의 design**은 매우 중요.

10.2 Syntax-Directed Translation

- **Syntax-Directed Translation Scheme(SDTS)**

  ::= a production rule + semantic action(no widely-accepted formalism)

- **A Model for SDTS**



♣ **whenever a reduction takes place, the semantic rule corresponding to the applied syntactic rule is activated.**

- Advantages of SDT
  - Providing a method describing semantic rules and that description is independent of any particular implementation.
  - Easy to modify - new productions and semantic actions can be added without disturbing the existing ones.

- Disadvantages of SDT
  - 파싱 도중에 error가 일어난 경우 이제까지 행한 semantic action이 모두 무의미해 진다.
  - input에 대해 one pass이면서 syntax-directed하게 처리하기 때문에 어떤 경우에는 정보가 부족하여 후에 필요한 정보가 나타났을 때 backpatching 등 복잡하게 처리해야 한다.

  - Solution

$\longrightarrow$ Syntax-directed한 방법으로는 의미 분석과 코드 생성시에 필요한 정보만을 구성하고 다음 단계에서 그것을 이용하여 의미 분석과 코드 생성을 한다.

Description of Semantic Actions [1/3]

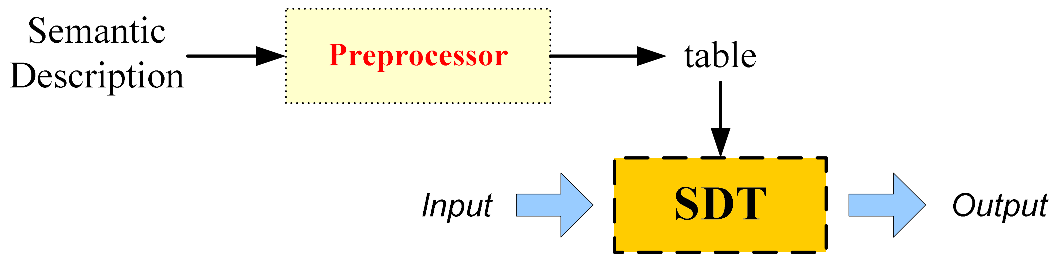
- **SDTS(Syntax-Directed Translation Scheme)**

  ::= production rules + semantic actions

- **Description of Semantic Actions**

  (1) .

  (2) Meta Language - Formal Semantic Language(FSL)



Description of Semantic Actions [2/3]

- **Semantic Description using Attributes of the Grammar Symbol**

  ::= We associate information with a programming language construct by attaching to the grammar symbols representing the construct. Values for attributes are computed by "semantic rules" associated with the grammar productions.

- **An attribute of symbol**

  ::= A value associated with a symbol. Each grammar symbol has an associated set of attributes. An attribute can represent we choose: a string, a number, a type, a memory location, or whatever.

ex)

Production	Semantic Rules
$L \rightarrow E\$$	print(E.val)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow digit$	$F.val := digit.lexval$

Description of Semantic Actions [3/3]

- **Synthesized attribute**

  ::= the value of the attribute of the nonterminal on the left side of the production is defined as a function of the grammar symbols on the right side.

  ex) $A \longrightarrow XYZ$ $A := f(X,Y,Z)$

- **Inherited attribute**

  ::= the value of the attribute of a nonterminal on the right side of the production is defined in terms of an attribute of the nonterminal on the left.

  ex) $A \longrightarrow XYZ$ $Y.val := 2 * X.val$

※ **Synthesized attribute is more natural than inherited attribute for mapping most programming language constructs into intermediate code.**

Implementation of SDT

▶ **Designing steps**

① Input design - language construct에 대한 grammar를 cfg를 이용하여 design.

② Scanner, Parser의 작성.

③ Semantic Specification - conventional PL.
⇒ SDT

④ Translator의 완성 - interconnection.

▶ **Examples** : 1. Desk Calculator
2. Conversion infix into postfix
3. Construction of AST

1. Desk Calculator [1/4]

- Step 1: Input design

  0. S -> E $
  1. E -> E + E
  2. E -> E * E
  3. E -> (E)
  4. E -> num

- Step 2: Parsing table

states \ symbols	num	+	*	(	)	$	E
0	s_3			s_2			1
1		s_4	s_5			acc	
2	s_3			s_2			6
3		r_4	r_4		r_4	r_4	
4	s_3			s_2			7
5	s_3			s_2			8
6		s_4	s_5		s_8		
7		r_1	s_5		r_1	r_1	
8		r_2	r_2		r_2	r_2	
9		r_3	r_3		r_3	r_3	

1. Desk Calculator [2/4]

- Step 3: Semantic Specification

Production	Semantic Rules
$L \rightarrow E\$$	print E.val
$E \rightarrow E_1 + E_2$	$E.val := E_1.val + E_2.val$
$E \rightarrow E_1 * E_2$	$E.val := E_1.val * E_2.val$
$E \rightarrow (E_1)$	$E.val := E_1.val$
$E \rightarrow num$	E.val := num.lexval

- Step 4: Implementation of Desk Calculator
  - Parsing stack : stack + stack + stack
  - Value stack : the values of the corresponding attribute.

1. Desk Calculator [3/4]

- Code fragments corresponding to semantic actions

Production	Code Fragment
$S \rightarrow E\$$	**print (val[top])**
$E \rightarrow E + E$	val[top-2] := val[top-2] + val[top]
$E \rightarrow E * E$	val[top-2] := val[top-2] * val[top]
$E \rightarrow (E)$	val[top-2] := val[top-1]
$E \rightarrow$ num	val[top] := num.lexval

  - the code fragments do not show how variable top is managed.
  - lexval : value
  - the code fragments are executed before a reduction takes place.

1. Desk Calculator [4/4]

ex) 23 * 5 + 4 $

		state	input	symbol	value	parse
		(0	23 * 5 + 4$,			)
s3	==>	(0 3	* 5 + 4$,	num		)
r4	==>	(0 1	* 5 + 4$,	E	23	4)
s5	==>	(0 1 5	5 + 4$,	E *	23_	4)
s3	==>	(0 1 5 3	+ 4$,	E * num	23__	4)
r4	==>	(0 1 5 8	+ 4$,	E * E	23_5	4 4)
r2	==>	(0 1	+ 4$,	E	115	4 4 2)
s4	==>	(0 1 4	4$,	E +	115_	4 4 2)
s3	==>	(0 1 4 3	$,	E + num	115__	4 4 2)
r4	==>	(0 1 4 7	$,	E + E	115_4	4 4 2 4)
r1	==>	(0 1	$,	E	119	**4 4 2 4 1**)
	==>	**accept**				

2. Conversion infix into postfix

- Code fragments

Production	Code Fragment
E → E + E	print ‘+’
E → E * E	print ‘*’
E → E / E	print ‘/’
E → (E)	no action
E → a	print ‘a’

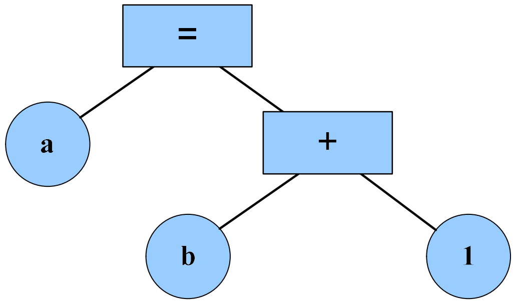
- ex) a + (a + a) * a

  ⟶ a a a + a * +

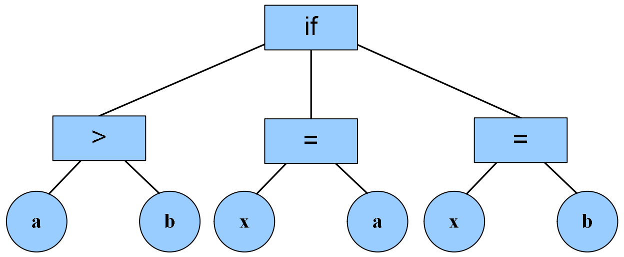
3. Construction of AST [1/3]

- AST is a condensed form of parse tree useful for representing language constructs.

  ex) a = b + 1;



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



3. Construction of AST [2/3]

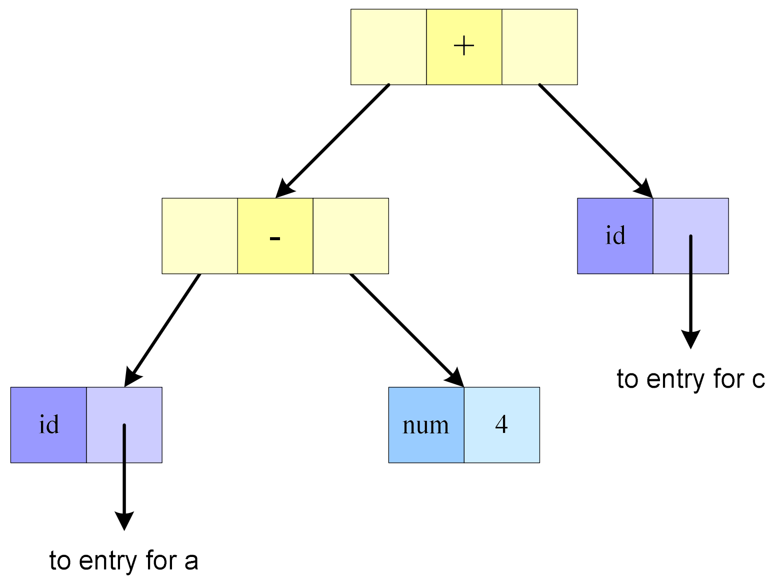
- Functions to create the nodes of AST for expressions with binary operators. Each function returns a pointer to a newly created node.
  1. mktree(op,left,right) creates an operator node with label op and two fields containing pointers to left and right.
  2. mknode(a) creates a terminal node for a and returns the node pointer.
- Semantic Specification

Production	Semantic Rules
$E \rightarrow E_1 + T$	E.nptr := mktree('+', E_1.nptr, T.nptr)
$E \rightarrow E_1 - T$	E.nptr := mktree('–', E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow (E)$	T.nptr := E.nptr
$T \rightarrow a$	T.nptr := mknode(a)

※ The synthesized attribute nptr for E and T keeps track of the pointers returned by the function calls.

3. Construction of AST [3/3]

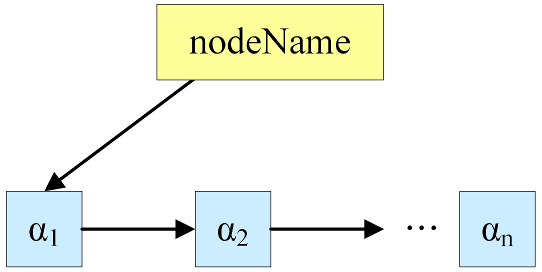
- AST for **a - 4 + c**



AST Design and Generation [1/11]

- **AST design**
  - **Grammar form :** **production rule [=> nodeName] ;**

$\mathbf{A} \rightarrow \alpha \Rightarrow$ **nodeName ;**



Note

node name의 생략 시에는 부 트리를 구성하지 않음.

AST Design and Generation [2/11]

- **Mini C Grammar with AST**

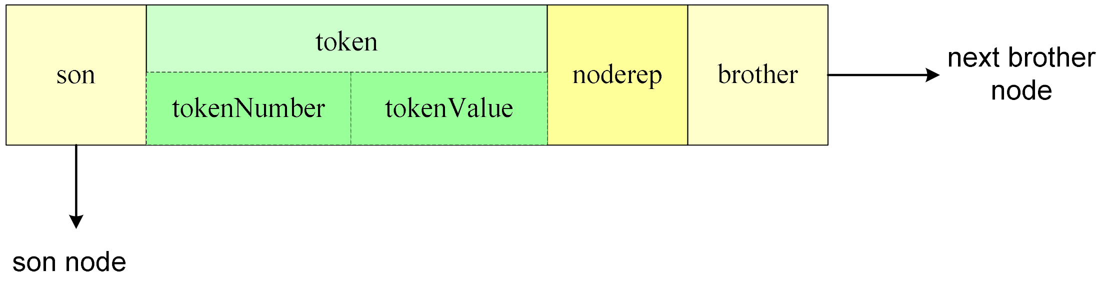
mini_c	→	translation_unit	⇒	PROGRAM;
translation_unit	→	external_dcl;		
	→	translation_unit external_dcl;		
external_dcl	→	function_def;		
	→	declaration;		
function_def	→	function_header compound_st	⇒	FUNC_DEF;
function_header	→	dcl_spec function_name formal_param	⇒	FUNC_HEAD;
dcl_spec	→	dcl_specifiers	⇒	DCL_SPEC;
dcl_specifiers	→	dcl_specifier;		
	→	dcl_specifiers dcl_specifier;		
dcl_specifier	→	type_qualifier;		
	→	type_specifier;		

Text p. 434-437 참조

AST Design and Generation [3/11]

Data Structures

A node form of AST



Node structure

```
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 Design and Generation [4/11]

- **Production rule name**

```
enum nodeNumber {
        ACTUAL_PARAM, ADD, ADD_ASSIGN, ARRAY_VAR, ASSIGN_OP,
        … ,   WHILE_ST
};
char *nodeName[] = {
        "ACTUAL_PARAM", "ADD", "ADD_ASSIGN", "ARRAY_VAR", "ASSIGN_OP",
        …   "WHILE_ST"
};
int ruleName[] = {
        /* 0        1               2        3        4        */
           0,       PROGRAM,        0,       0,       0,
        …
        /* 95       96              97                         */
           0,       0,              0
};
```

AST Design and Generation [5/11]

- **AST Generation**
  - **Shift → buildNode(simple and easy)**
  - **Reduce → buildTree(complex and difficult)**

- **Shift action of parsing :**
  - **if the token is meaningful, then call buildNode.**

```
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;
}
```

AST Design and Generation [6/11]

- Reduce action of parsing :
  - Basic concept
    - if the production rule is meaningful

```
    1. build subtree
        - linking brothers
        - making a subtree
else
    2. only linking brothers
```

  - buildTree() function
    - 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 Design and Generation [7/11]

- **Node *buildTree(int nodeNumber, int rhsLength)**

```
Node *buildTree(int nodeNumber, int rhsLength)
{ //…
    i = sp - rhsLength + 1;
    // step 1: find a first index with node in value stack
    while (i <= sp && valueStack[i] == NULL) i++;                    // …… ①
    if (!nodeNumber && i > sp) return NULL;                          // …… ②
    start = i;
    // step 2: linking brothers
    while (i <= sp-1) {
        j = i + 1;
        while (j <= sp && valueStack[j] == NULL) j++;                // …… ③
        if (j <= sp) {                                               // …… ④
                ptr = valueStack[i];
                while (ptr->brother) ptr = ptr->brother;
                ptr->brother=valueStack[j];
        }
        i = j;                                                       // …… ⑤
    }
    first = (start > sp) ? NULL : valueStack[start];                 // …… ⑥
    // step 3: making subtree root and linking son
    if (nodeNumber) {                                                // …… ⑦
        //… memory allocation for ptr
        ptr->token.tokenNumber = nodeNumber;
        ptr->token.tokenValue = NULL;
        ptr->noderep = nonterm;
        ptr->son = first;
        ptr->brother = NULL;
        return ptr;
    }
    else return first;
}
```

AST Design and Generation [8/11]

- **buildTree() 함수의 설명**

  ① 현재 reduce되는 생성 규칙의 rhs에 노드가 매달려 있는 인덱스를 값 스택에서 찾는다. 형제 노드로 연결할 노드의 첫 번째 인덱스를 찾은 것이다.

  ② 의미있는 생성 규칙이 아니고 연결할 형제 노드도 없으면 그냥 복귀한다.

  ③ 형제 노드로 연결할 노드의 다음 인덱스를 ①과 같은 방법으로 찾는다.

  ④ 만약 다음 인덱스를 찾았으면, 형제 노드로 연결한다.

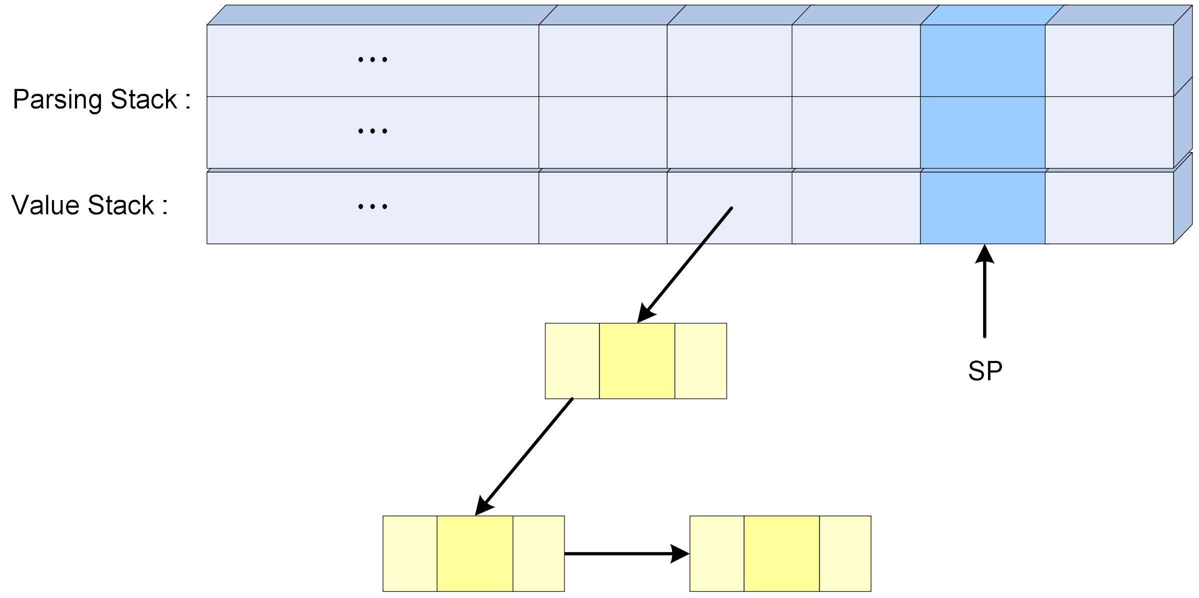
  ⑤ 연속해서 다음 인덱스를 찾기 위해 위치를 앞으로 이동한다.

  ⑥ 연결된 형제 노드들의 첫 번째 노드의 포인터를 first에 저장한다.

  ⑦ 의미있는 생성 규칙이면, nonterminal 노드를 만든 후에 연결된 형제 노드를 son으로 연결하고 새로 만든 노드의 포인터를 복귀한다. 의미있는 생성 규칙이 아니면, 연결된 형제 노드의 포인터만을 복귀한다.

AST Design and Generation [9/11]

Parsing Stack and Value Stack



Note

Parsing Stack과 Value Stack은 병렬로 운행

AST Design and Generation [10/11]

- Confirming the AST structures
  - Printing an AST using indentation
  - Traversing an AST in depth-first order
- Two functions
  - printTree() – printing an AST structure
  - printNode() – printing a node information
- printTree() function

```
void printTree(Node *pt, int indent)
{
  Node *p = pt;
  while (p != NULL) {
     printNode(p, indent);
     if (p->noderep == nonterm) printTree(p->son, indent+5);
     p = p->brother;
  }
}
```

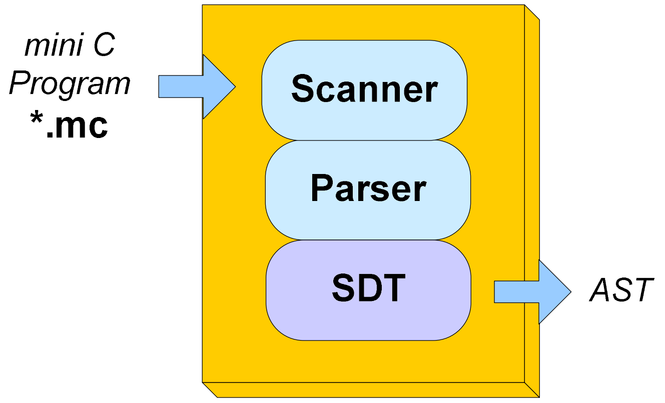
AST Design and Generation [11/11]

- **printNode() function**

```
void printNode(Node *pt, int indent)
{
    extern FILE * astFile; int i;

    for (i=1; i<=indent; i++) fprintf(astFile," ");
    if (pt->noderep == terminal) {
            if (pt->token.number == tident)
                    fprintf(astFile," Terminal: %s", pt->token.value.id);
            else if (pt->token.number == tnumber)
                    fprintf(astFile," Terminal: %d", pt->token.value.num);
    }
    else { // nonterminal node
            int i;
            i = (int) (pt->token.number);
            fprintf(astFile," Nonterminal: %s", nodeName[i]);
    }
    fprintf(astFile,"\n");
}
```

Programming Assignment #4

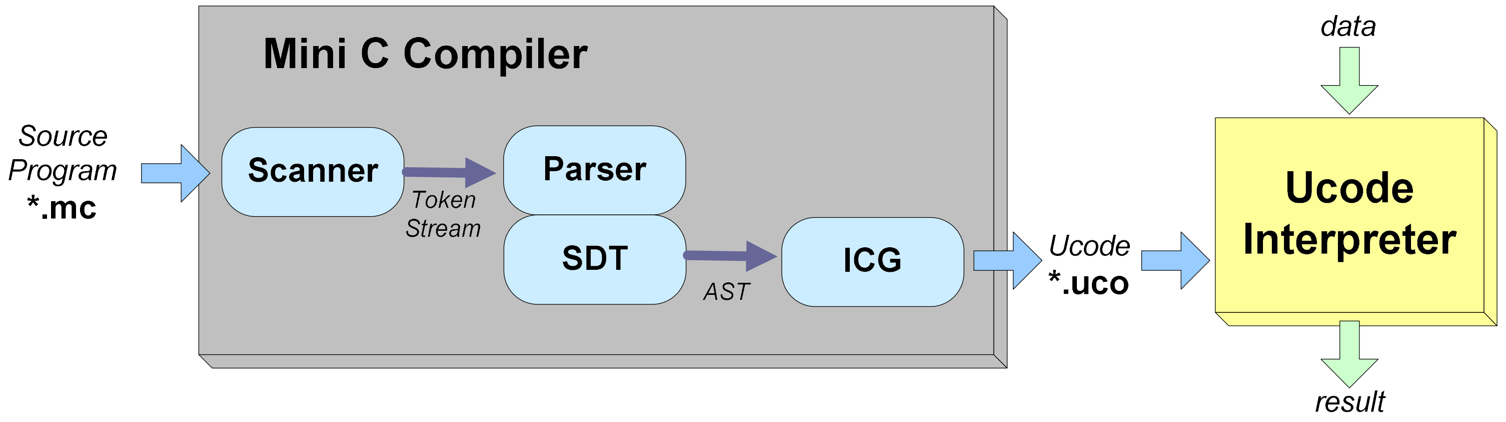
- Implement a syntax-directed translator producing an AST for Mini C program.



- Mini C Program : Perfect.mc(Text pp.447)
- The Output form of AST using printtree() : Text pp.443-444

10.3 Code Generation

A Model for ICG



Source language	:	Mini C
Intermediate Representation	:	Abstract Syntax Tree(AST)
Intermediate code	:	Ucode
Execution	:	Ucode Interpreter

AST structure for Mini C program [1/2]

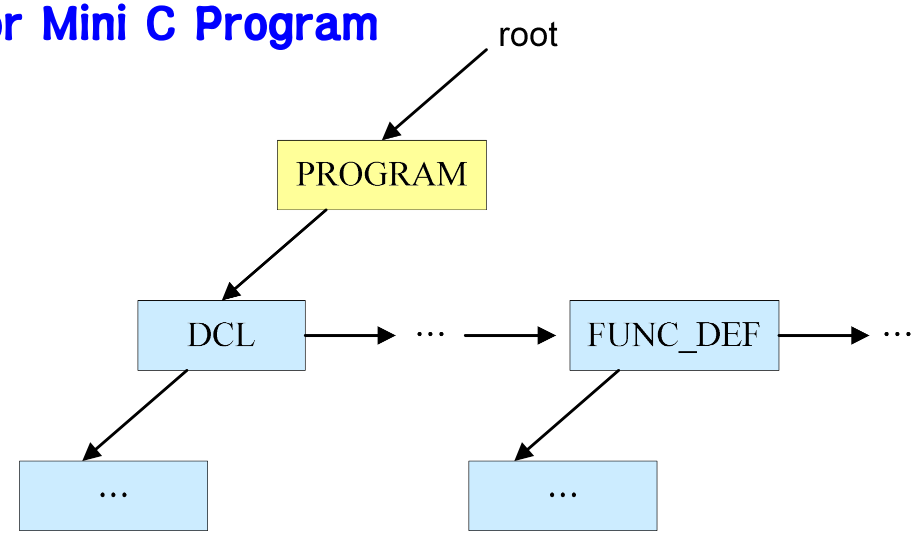
- Mini C Program structure
  - External declaration and Function definition
- Declaration
  - External declaration
  - Local declaration
- Function definition
  - Function header
  - Function body – statements
- Statement
  - expression
  - Statement
    - return statement
    - compound statement
    - expression statement
    - control statement - if, if - else, while

AST structure for Mini C program [2/2]

Mini C grammar with AST

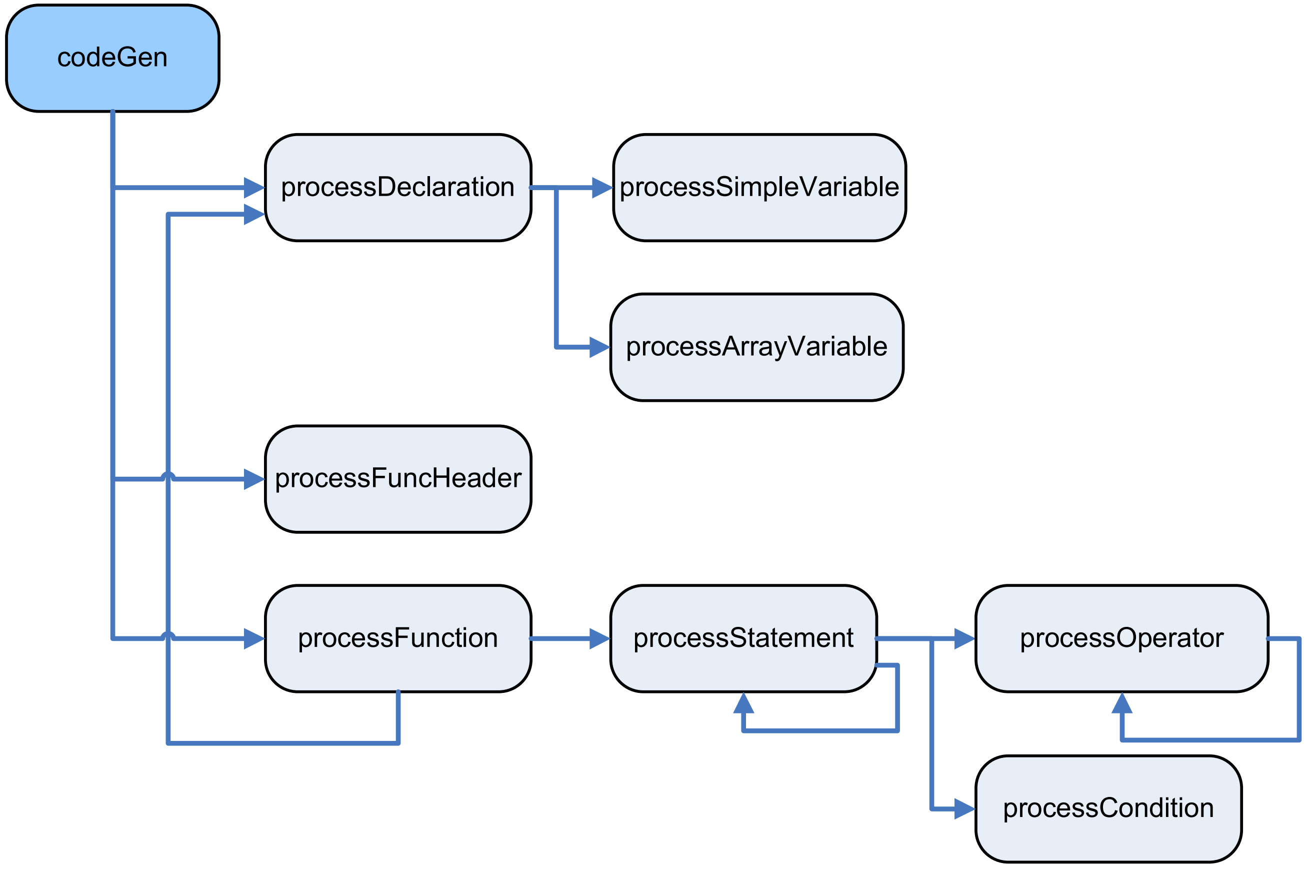
```
mini_c             →  translation_unit                    => PROGRAM;
translation_unit   →  external_dcl;
                   →  translation_unit external_dcl;
external_dcl       →  function_def;
                   →  declaration;
function_def       →  function_header compound_st         => FUNC_DEF;
declaration        →  dcl_spec init_dcl_list ';'          => DCL;
…
```

AST for Mini C Program



Code Generating Routines [1/2]

- **Relationship between code generating functions**



Code Generating Routines [2/2]

- codeGen()
  - 코드 생성의 핵심 함수
  - main 에서, codeGen(root) // root of AST

- codeGen()의 기능

```
step 1: process the declaration part
        1. process external variables
        2. process function headers
step 2: process the function part
        1. process local  variables
        2. process statements
step 3: generate starting code of U-Code interpreter
        1. before main
        2. main
        3. after main
```

Code Generating Routines [2/2]

- **codeGen() function**

```
void codeGen(Node *ptr)
{
    //…
    // step 1: process the declaration part
    for (p=ptr->son; p; p=p->brother) {
                if (p->token.number == DCL) processDeclaration(p->son);
                else if (p->token.number == FUNC_DEF) processFuncHeader(p->son);
                else icg_error(3);
    }
    //…
    // step 2: process the function part
    for (p=ptr->son; p; p=p->brother)
                if (p->token.number == FUNC_DEF) processFunction(p);
    //…
    // step 3: generate codes for starting routine
    emit1(bgn, globalSize);
    emit0(ldp);
    emitJump(call, "main");
    emit0(endop);
}
```

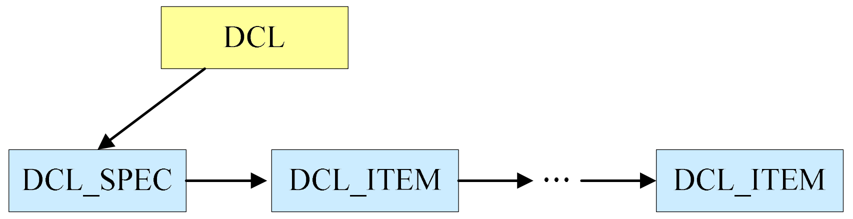
Declaration [1/3]

Grammar

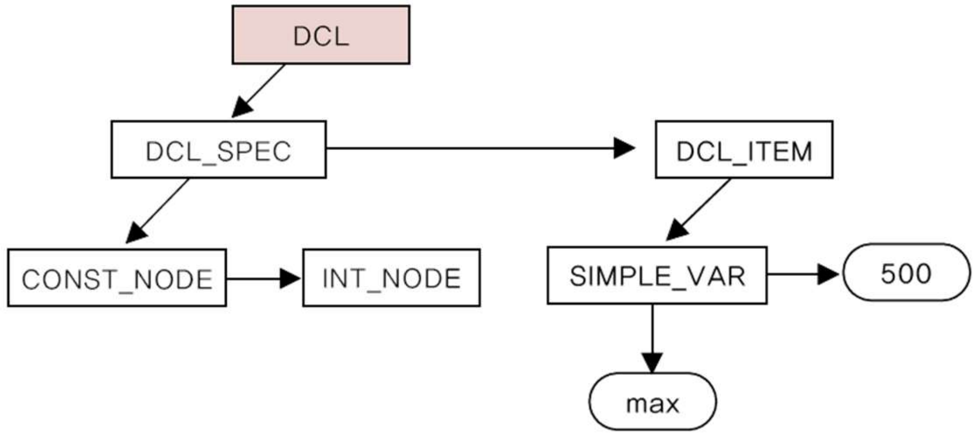
declaration_list	→	declaration;	
	→	declaration_list declaration;	
declaration	→	dcl_spec init_dcl_list ';‘	=> DCL;
dcl_spec	→	dcl_specifiers	=> DCL_SPEC;
dcl_specifiers	→	dcl_specifier;	
	→	dcl_specifiers dcl_specifier;	
dcl_specifier	→	type_qualifier;	
	→	type_specifier;	
type_qualifier	→	'const‘	=> CONST_NODE;
type_specifier	→	'int‘	=> INT_NODE;
	→	'void‘	=> VOID_NODE;
init_dcl_list	→	init_declarator;	
	→	init_dcl_list ',' init_declarator;	
init_declarator	→	declarator	=> DCL_ITEM;
	→	declarator '=' '%number‘	=> DCL_ITEM;
declarator	→	'%ident‘	=> SIMPLE_VAR;
	→	'%ident' '[' opt_number ']‘	=> ARRAY_VAR;
opt_number	→	'%number';	
	→	;	

Declaration [2/3]

- AST



- 예제 10.5
  - const int max = 500;



Declaration [3/3]

- **Process declaration part**

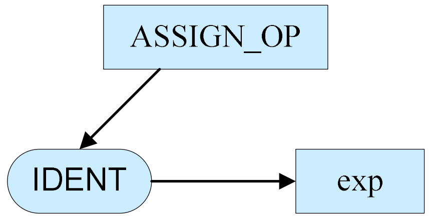
```
void processDeclaration(Node *ptr)
{  //…
   // step 1: process DCL_SPEC
   //…
   // step 2: process DCL_ITEM
   while (p) {
               q = p->son;   // SIMPLE_VAR or ARRAY_VAR
               switch (q->token.number) {
                  case SIMPLE_VAR:              // simple variable
                              processSimpleVariable(q, typeSpecifier, typeQualifier);
                              break;
                  case ARRAY_VAR:               // array variable
                              processArrayVariable(q, typeSpecifier, typeQualifier);
                              break;
                  default: printf("error in SIMPLE_VAR or ARRAY_VAR\n"); break;
               } // end switch
               p = p->brother;
   } // end while
}
```

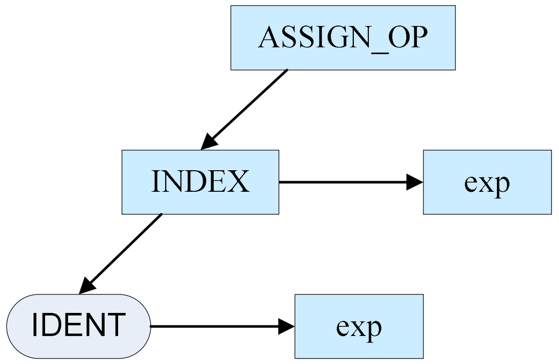
Assignment [1/3]

Grammar

```
expression          →  assignment_exp;
assignment_exp      →  logical_or_exp;
                    →  unary_exp '=' assignment_exp      => ASSIGN_OP;
                    →  unary_exp '+=' assignment_exp     => ADD_ASSIGN;
                    →  unary_exp '-=' assignment_exp     => SUB_ASSIGN;
                    →  unary_exp '*=' assignment_exp     => MUL_ASSIGN;
                    →  unary_exp '/=' assignment_exp     => DIV_ASSIGN;
                    →  unary_exp '%=' assignment_exp     => MOD_ASSIGN;
```

AST





Assignment [2/3]

- **Process assignment**

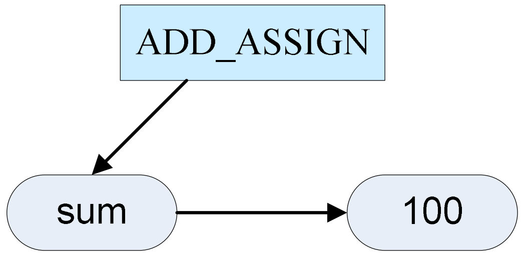
```
void processOperator(Node *ptr)
{
    switch (ptr->token.number) {
        // assignment operator
        case ASSIGN_OP:
            // ...
            // step 1: generate instructions for left-hand side if array variable
            // step 2: generate instructions for right-hand side
            // step 3: generate a store instruction
        // complex assignment operators
        case ADD_ASSIGN: case SUB_ASSIGN: case MUL_ASSIGN:
        case DIV_ASSIGN: case MOD_ASSIGN:
            // ...
            // step 1: code generation for left hand side
            // step 2: code generation for repeating part
            // step 3: code generation for right hand side
            // step 4: emit the corresponding operation code
            // step 5: code generation for store code
        // ...
    } // end switch
}
```

Assignment [3/3]

- **Example**
  - **program**

```
sum += 100;
```

  - **AST**



  - **Ucode**

```
lod     1    1      // load sum
ldc     100
add
str     1    1      // store sum
```

Binary operators [1/3]

Grammar

```
logical_or_exp        →  logical_and_exp;
                      →  logical_or_exp '||' logical_and_exp        => LOGICAL_OR;
logical_and_exp       →  equality_exp;
                      →  logical_and_exp '&&' equality_exp          => LOGICAL_AND;
equality_exp          →  relational_exp;
                      →  equality_exp '==' relational_exp           => EQ;
                      →  equality_exp '!=' relational_exp           => NE;
relational_exp        →  additive_exp;
                      →  relational_exp '>' additive_exp            => GT;
                      →  relational_exp '<' additive_exp            => LT;
                      →  relational_exp '>=' additive_exp           => GE;
                      →  relational_exp '<=' additive_exp           => LE;
additive_exp          →  multiplicative_exp;
                      →  additive_exp '+' multiplicative_exp        => ADD;
                      →  additive_exp '-' multiplicative_exp        => SUB;
multiplicative_exp    →  unary_exp;
                      →  multiplicative_exp '*' unary_exp           => MUL;
                      →  multiplicative_exp '/' unary_exp           => DIV;
                      →  multiplicative_exp '%' unary_exp           => MOD;
```

Binary operators [2/3]

- **Process binary operators**

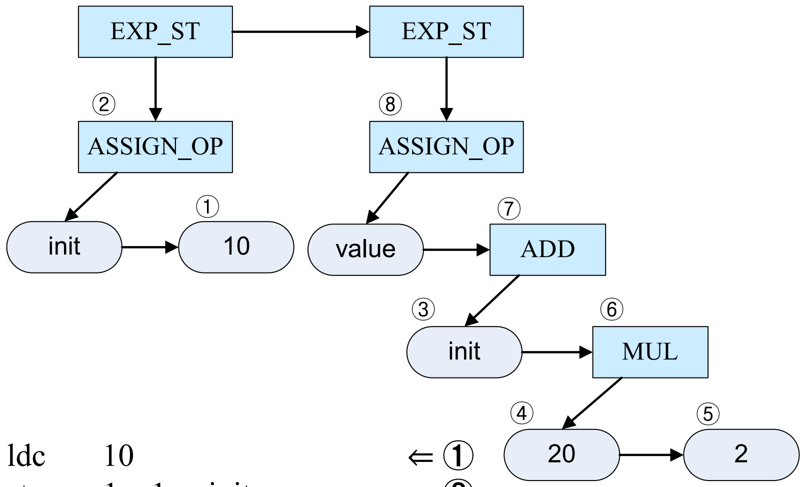
```
void processOperator(Node *ptr)
{
    switch (ptr->token.number) {
        //…
        // binary(arithmetic/relational/logical) operators
        case ADD: case SUB: case MUL: case DIV: case MOD:
        case EQ:  case NE: case GT: case LT: case GE: case LE:
        case LOGICAL_AND: case LOGICAL_OR:
        {
                    // 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 operators
                        // relational operators
                        // logical operators
                    }
        }
        // ...
    } // end switch
}
```

Binary operators [3/3]

- **Example**
  - **program**

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

  - **AST**



  - **Ucode**

```
ldc      10                    ⇐ ①
str      1    1   ; init       ⇐ ②
lod      1    1   ; init       ⇐ ③
ldc       20                   ⇐ ④
ldc       2                    ⇐ ⑤
mult                           ⇐ ⑥
add                  ⇐ ⑦
str       1    2   ; value     ⇐ ⑧
```

Unary operators [1/6]

- **Grammar**

```
unary_exp    →  postfix_exp;
             →  '-' unary_exp                         => UNARY_MINUS;
             →  '!' unary_exp                         => LOGICAL_NOT;
             →  '++' unary_exp                        => PRE_INC;
             →  '--' unary_exp                        => PRE_DEC;
postfix_exp  →  primary_exp;
             →  postfix_exp '[' expression ']'        => INDEX;
             →  postfix_exp '(' opt_actual_param')'   => CALL;
             →  postfix_exp '++'                      => POST_INC;
             →  postfix_exp '--'                      => POST_DEC;
```

Unary operators [2/6]

- Process unary operators
  - Unary -, ~

```
// unary operators
case UNARY_MINUS: case LOGICAL_NOT:
{
        Node *p = ptr->son;
        if (p->noderep == nonterm) processOperator(p);
        else rv_emit(p);
        switch (ptr->token.number) {
            case UNARY_MINUS: emit0(neg); break;
            case LOGICAL_NOT: emit0(notop); break;
        }
        break;
}
```

Unary operators [3/6]

- **Array variable**
  - In one-dimensional array, location of i's element = Base + (i - Low) * W
    where, Low : lower bound of array
    Base : start address of array
  - In C programming language, Low is always 0.
    ∴ Address of A[i] = Base + i*W
    - Assume that the size of integer is 1. W = 1(∵ word machine)
      For example, Location of list[10] = (start address of list array) + 10 * 1
- **Process array(index)**

```
case INDEX:
{
        Node *indexExp = ptr->son->brother;
        if (indexExp->noderep == nonterm) processOperator(indexExp);
        else rv_emit(indexExp);
        stIndex = lookup(ptr->son->token.value.id);
        if (stIndex == -1) {
                printf("undefined variable : %s\n", ptr->son->token.value.id);
                return;
        }
        emit2(lda, symbolTable[stIndex].base, symbolTable[stIndex].offset);
        emit0(add);
        if (!lvalue) emit0(ldi);        // rvalue
        break;
}
```

Unary operators [4/6]

【예제 10.9】 다음은 Mini C 언어에서 배열 참조에 대한 Ucode 이다.

```
int vector[100];
void main()
{
            int temp;
            // ...
            vector[5] = 10;          // ………. ①
            // ...
            temp = vector[20];       // ………. ②
            // ...
}
```

① 에 해당하는 U-코드 :

```
ldc     5
lda     1   1   /* base address(vector)의 적재 */
add
ldc     10
sti
```

② 에 해당하는 U-코드 :

```
ldc     20
lda     1   1   /* base address(vector)의 적재 */
add
ldi
str     2   1   /* temp */
```

Unary operators [5/6]

- **Process unary operators : ++, --**

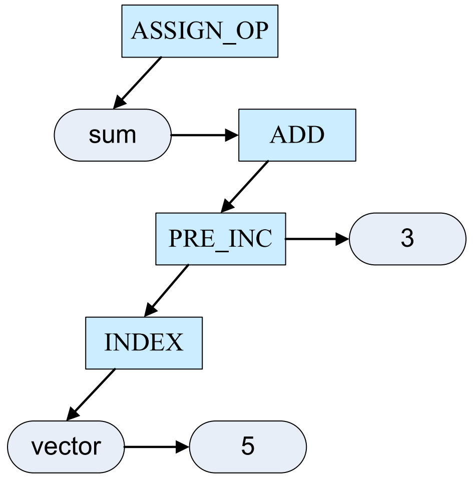
```
// increment/decrement operators
case PRE_INC: case PRE_DEC: case POST_INC: case POST_DEC:
{
        //…
        // compute an operand
        //…
        switch (ptr->token.number) {
              case PRE_INC: emit0(incop);
                        // if (isOperation(ptr)) emit0(dup);
                        break;
              case PRE_DEC: emit0(decop);
                        // if (isOperation(ptr)) emit0(dup);
                        break;
              case POST_INC:
                        // if (isOperation(ptr)) emit0(dup);
                        emit0(incop); break;
              case POST_DEC:
                        // if (isOperation(ptr)) emit0(dup);
                        emit0(decop); break;
        }
        //…
        // compute index
        //…
}
```

Unary operators [6/6]

- **Example**
  - **program**

    sum = ++vector[5] + 3;

  - **AST**



  - **U-Code**

```
ldc     5
lda     1    2     // vector
add
ldi
inc
dup
ldc     5
lda     1    2     // vector
add
swp
sti
ldc     3
add
str     1    1     // sum
```

Statement [1/2]

- **Grammar**

$$
\begin{aligned}
\text{statement} &\rightarrow \text{compound\_st;} \\
&\rightarrow \text{expression\_st;} \\
&\rightarrow \text{if\_st;} \\
&\rightarrow \text{while\_st;} \\
&\rightarrow \text{return\_st;}
\end{aligned}
$$

Statement [2/2]

- **Process statement**

```
void processStatement(Node *ptr)
{
        switch (ptr->token.number) {
            // process COMPOUND_ST ...
            // process EXP_ST …
            case RETURN_ST:
                    if (ptr->son != NULL) {
                            returnWithValue = 1;
                            p = ptr->son;
                            if (p->noderep == nonterm) processOperator(p); // return value
                            else rv_emit(p);
                            emit0(retv);
                    } else emit0(ret);
            break;
            // process IF_ST, IF_ELSE_ST, WHILE_ST …
        } //end switch
}
```

※ **Code skeleton and return statement**

Compound statement [1/2]

Grammar

```
compound_st     →  '{' opt_dcl_list opt_stat_list '}’  =>  COMPOUND_ST;
opt_dcl_list    →  declaration_list                    =>  DCL_LIST;
                →                                      =>  DCL_LIST;
opt_stat_list   →  statement_list                      =>  STAT_LIST;
                →  ;
statement_list  →  statement;
                →  statement_list statement;
```

※ Mini C 언어에서, 함수 내에서는 지역 변수를 선언할 수 있지만
복합문 내에서는 지역 변수를 선언할 수 없다.
따라서, 복합문 내에서 지역 변수를 선언하더라도 무시하고
문장들만 처리한다.

Compound statement [2/2]

- **Process compound statement**

```
void processStatement(Node *ptr)
{
        //…
        case COMPOUND_ST:
            p = ptr->son->brother;      // STAT_LIST
            p = p->son;
                while (p) {
                    processStatement(p);
                    p = p->brother;
                }
            break;
        //…
}
```

Expression statement

- **Grammar**

expression_st	→	opt_expression ';'	=> EXP_ST;
opt_expression	→	expression;	
	→	;	

- **Process expression statement**

```
void processStatement(Node *ptr)
{
        //…
        case EXP_ST:
            if (ptr->son != NULL) processOperator(ptr->son);
            break;
        //…
}
```

Control statement [1/11]

- **Control Statements**
  1. conditional statement - if, case, switch
  2. iteration statement - for, while, do-while, loop, repeat-until
  3. branch statement - goto

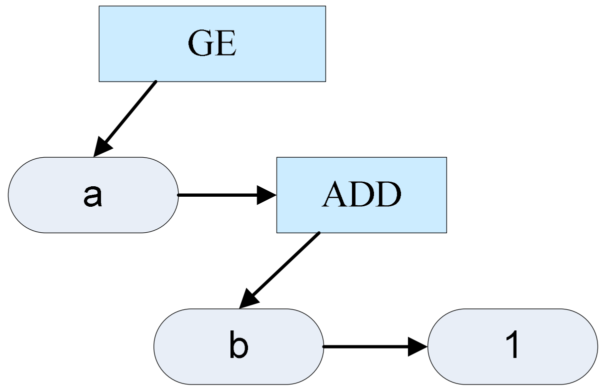
- **Logical expression**
  1. use calculation of logical values
  2. use control expression in control statements

- **Expression of logical values**
  1. true와 false를 숫자로 변환, 산술식의 연산과 유사한 방법으로 계산
  2. 값에 따라 선택적인 실행이 가능

Control statement [2/11]

【예제 10.12】 관계식 a >= b + 1에 대한 AST와 U-코드는 다음과 같다.

- AST 형태 :



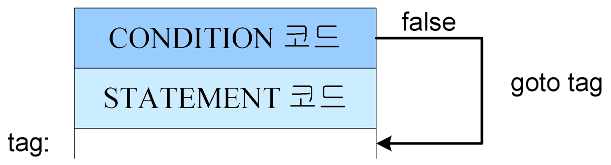
- U-코드 :

```
lod    Ba  Oa    // Ba: 변수 a의 base, Oa: 변수 a의 offset
lod    Bb  Ob    // Bb: 변수 b의 base, Ob: 변수 b의 offset
loc    1
add
ge
```

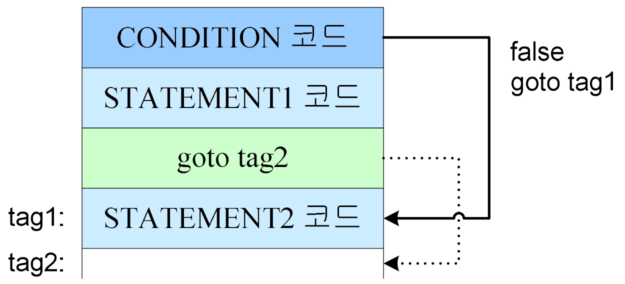
Control statement [3/11]

- **Scheme for control statements**

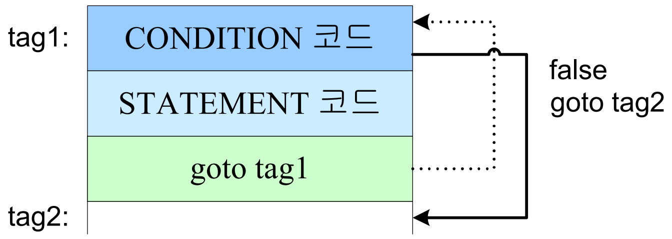
  - **if 구조**



  - **if – else 구조**



  - **while 구조**



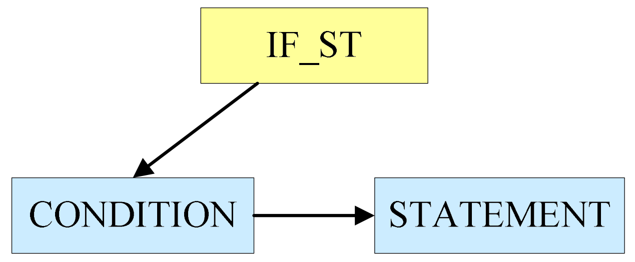
Control statement [4/11]

- **Grammar**

if_st	→	'if' '(' expression ')' statement	=> IF_ST;
	→	'if' '(' expression ')' statement 'else' statement	=> IF_ELSE_ST;
while_st	→	'while' '(' expression ')' statement	=> WHILE_ST;

Control statement [5/11]

- if statement
  - AST

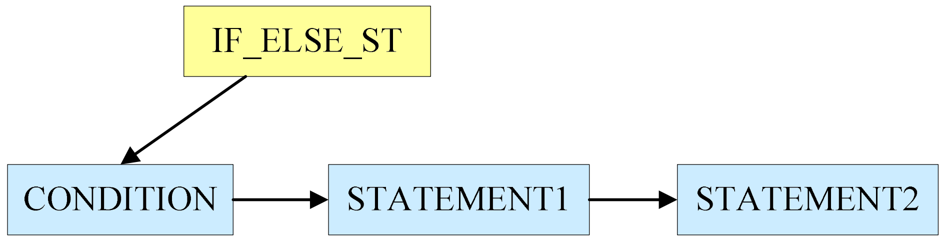


  - Code segment

```
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);
    }
    //…
}
```

Control statement [6/11]

- if-else statement
  - AST



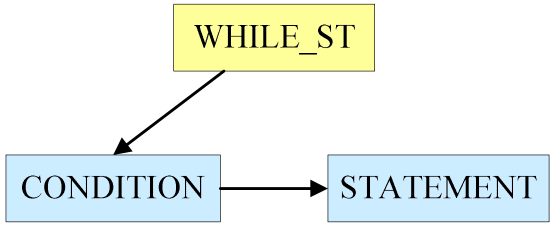
$$\begin{tikzcd}
 & \text{IF\_ELSE\_ST} \arrow[dl] & \\
\text{CONDITION} \arrow[r] & \text{STATEMENT1} \arrow[r] & \text{STATEMENT2}
\end{tikzcd}$$

  - Code segment

```
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);
    }
    //…
}
```

Control statement [7/11]

- **while statement**
  - **AST**



  - **Code segment**

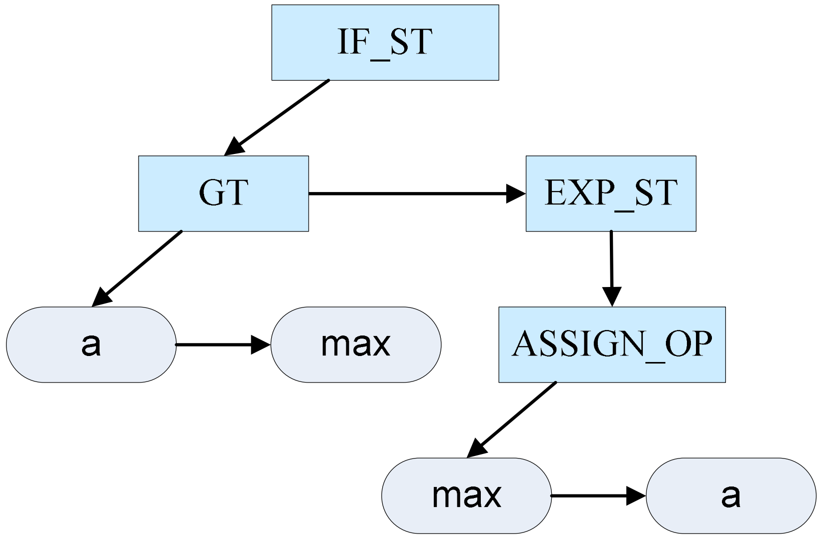
```
void processStatement(Node *ptr)
{
    //…
    case WHILE_ST:
    {
        char label1[LABEL_SIZE], label2[LABEL_SIZE];

        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);
    }
    //…
}
```

Control statement [8/11]

- **Example 1**
  - **code**

    ```
    if (a > max) max = a;
    ```

  - **AST**



Control statement [9/11]

- Example 1(계속)

  - U-Code

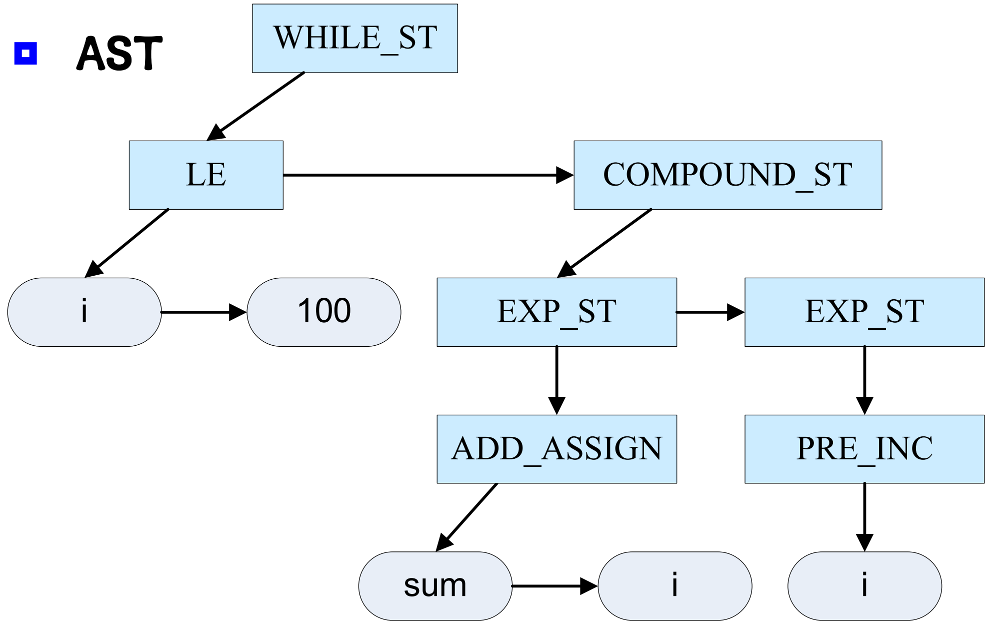
```
        lod     1  1    // a
        lod     1  2    // max
        gt              // a > max
        fjp     $$1
        lod     1  1
        str     1  2    // max = a
$$1     nop
```

Control statement [10/11]

- **Example 2**
  - **code**

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

  - **AST**



Control statement [11/11]

- Example 2(계속)
  - U-Code

```
$$1      nop
         lod        1   1
         loc        100          // i <= 100
         le
         fjp        $$2
         lod        sum
         lod        i            // sum += i;
         add
         str        sum
         lod        i
         inc                     // ++i;
         str        i
         ujp        $$1
$$2      nop
```

Function

1 Function Call

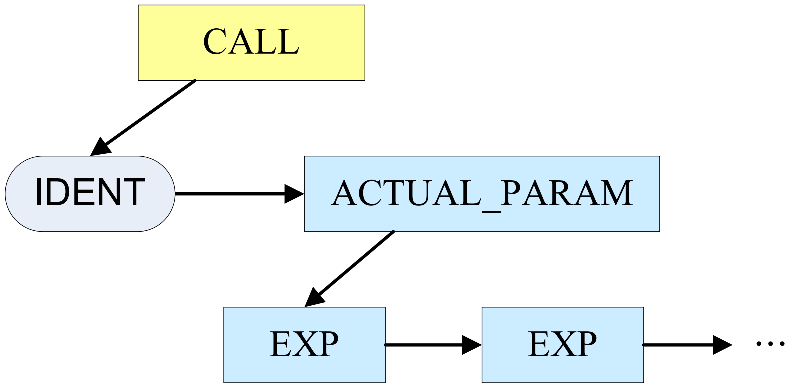
2 Function Definition

Function – Function call [1/2]

- **Grammar**

```
postfix_exp       →  primary_exp;
                  →  postfix_exp '(' opt_actual_param ')'‘   => CALL;
opt_actual_param  →  actual_param;
                  →  ;
actual_param      →  actual_param_list                     => ACTUAL_PARAM;
actual_param_list →  assignment_exp;
                  →  actual_param_list ',' assignment_exp;
```

- **AST**



Function – Function call [2/2]

- **Process function call**

```
void processStatement(Node *ptr)
{
        //…
        case CALL:
        {
             //…
             // predefined(Library) functions
             //…
             // handle for user function
             functionName = p->token.value.id;
             stIndex = lookup(functionName);
             if (stIndex == -1) break; // undefined function !!!
             noArguments = symbolTable[stIndex].width;

             emit0(ldp);
             p = p->brother;        // ACTUAL_PARAM
             while (p) {            // processing actual arguments
                       if (p->noderep == nonterm) processOperator(p);
                       else rv_emit(p);
                       noArguments--;
                       p = p->brother;
             }
             //…
             emitJump(call, ptr->son->token.value.id);
             break;
        }
         //…
}
```

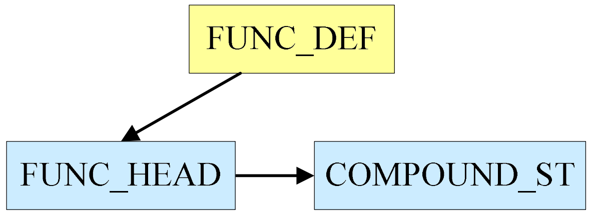
Function – Function definition [1/4]

- **Grammar**

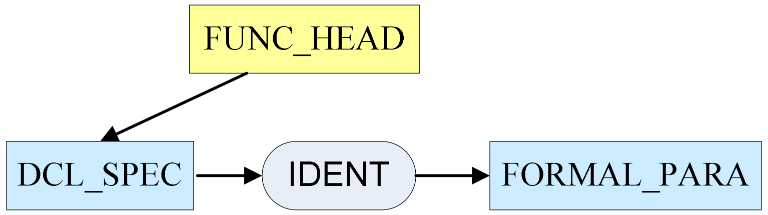
```
function_def          →  function_header compound_st          => FUNC_DEF;
function_header       →  dcl_spec function_name formal_param  => FUNC_HEAD;
function_name         →  '%ident';
formal_param          →  '(' opt_formal_param ')'             => FORMAL_PARA;
opt_formal_param      →  formal_param_list;
                      →  ;
formal_param_list     →  param_dcl;
                      →  formal_param_list ',' param_dcl;
param_dcl             →  dcl_spec declarator                  => PARAM_DCL;
```

Function – Function definition [2/4]

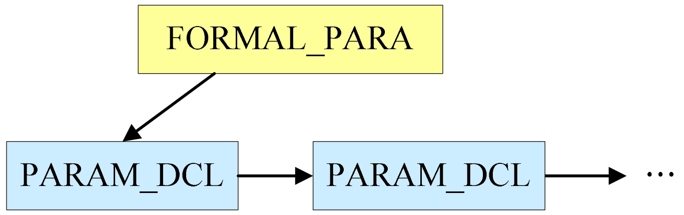
- **AST**
  - **Function definition**



  - **Function head**



  - **Formal parameter**



Function – Function definition [3/4]

- **Process function header**

```
void processFuncHeader(Node *ptr)
{
    //…
    // step 1: determine return type
    p = ptr->son->son;
    while (p) {
        if (p->token.number == INT_NODE) returnType = INT_TYPE;
        else if (p->token.number == VOID_NODE) returnType = VOID_TYPE;
        else printf("invalid function return type\n");
        p = p->brother;
    }
    // step 2: count the number of formal parameters
    p = ptr->son->brother->brother;          // FORMAL_PARA
    p = p->son;                              // PARAM_DCL
    noArguments = 0;
    while (p) {
        noArguments++;
        p = p->brother;
    }
    // step 3: insert function name
    stIndex = insert(ptr->son->brother->token.value.id, returnType, FUNC_TYPE,
                  1/*base*/, 0/*offset*/, noArguments/*width*/, 0/*initialValue*/);
    //if (!strcmp("main", functionName)) mainExist = 1;
}
```

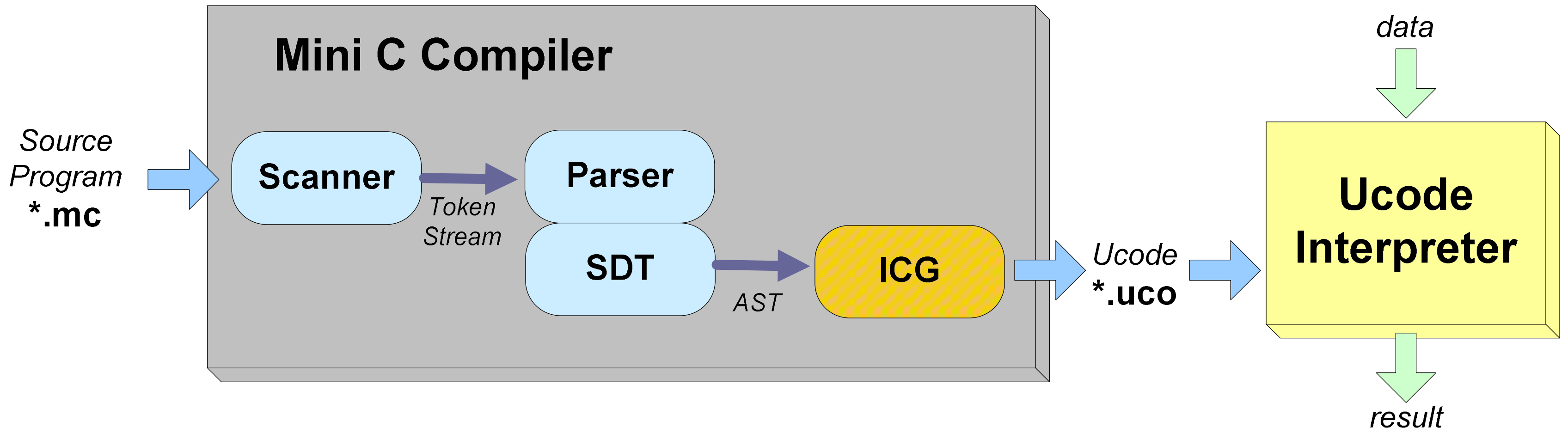
Function – Function definition [4/4]

- **Main routine for processing a function definition**

```
void processFunction(Node *ptr)
{
    // …
    // step 1: process function header        // already explained
    // step 2: process function body
    // …
}

void processFunctionBody(Node *ptr)
{
    // …
    // step 1: process the declaration part in function body
    // step 2: emit the function start code
    // step 3: process the statement part in function body
    // step 4: check if return type and return value
    // step 5: generate the ending codes
    // …
}
```

10.4 Ucode Translator [1/2]

- Design and Implementation of Ucode Translator
  - scanner, parser, SDT, ICG



Ucode Translator [2/2]

- **Execution sequence of perfect.mc**

  ① **Mini C program : Text pp.446**

  ② **The Output form of AST using printtree() : Text pp.443-444**

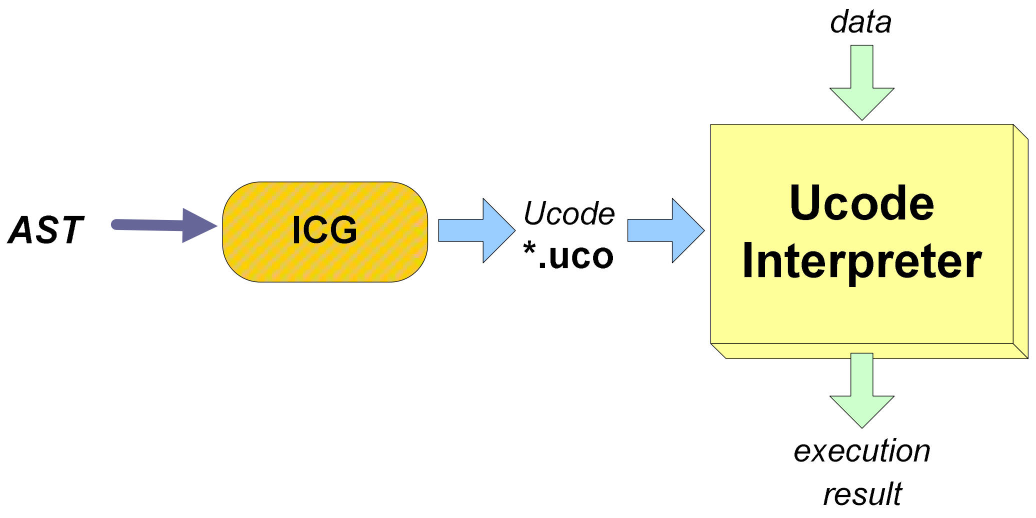
  ③ **Ucode that generated by code generator : Text pp.493-495**

  ④ **The execution of Ucode using Ucode Interpreter**

```
ucodei perfect.uco

Result filename is perfect.lst

    -- Assembling...
     -- Executing...
     -- Result Data

     6   28   496
```

Programming Assignment #5 [1/2]

- Mini C 언어에 대한 Ucode Translator를 작성하시오. 생성된 Ucode는 Interpreter를 사용하여 실행하시오.



Programming Assignment #5 [2/2]

- 예제 프로그램: perfect.mc

```
const int max = 500;
void main()
{
     int i, j, k;
     int 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;
     }
}
```