

Project #2: Parser

2024 Fall

Hunjun Lee

Hanyang University

Project Goal

- **C-Minus Parser Implementation using Yacc (Bison)**

- The Parser reads an input source code string, tokenizes and parses it with C-Minus grammar, and returns (prints) abstract syntax tree (AST).
- C-Minus scanner with LEX should be used.
- Some source code should be obtained using Yacc.
 - Yacc takes a grammar in BNF form as input and generate a LALR(1) parser.
 - **Ambiguous grammar will cause conflicts.**
- cminus.y, ... -> cminus_parser

Grammar for C-Minus

1. *program* → *declaration-list*
2. *declaration-list* → *declaration-list declaration* | *declaration*
3. *declaration* → *var-declaration* | *fun-declaration*
4. *var-declaration* → *type-specifier ID ;* | *type-specifier ID [NUM] ;*
5. *type-specifier* → **int** | **void**
6. *fun-declaration* → *type-specifier ID (params) compound-stmt*
7. *params* → *param-list* | **void**
8. *param-list* → *param-list , param* | *param*
9. *param* → *type-specifier ID* | *type-specifier ID []*
10. *compound-stmt* → { *local-declarations statement-list* }
11. *local-declarations* → *local-declarations var-declarations* | *empty*
12. *statement-list* → *statement-list statement* | *empty*
13. *statement* → *expression-stmt* | *compound-stmt* | *selection-stmt* | *iteration-stmt* | *return-stmt*
14. *expression-stmt* → *expression ;* | *;*
15. *selection-stmt* → **if** (*expression*) *statement* | **if** (*expression*) *statement* **else** *statement*
16. *iteration-stmt* → **while** (*expression*) *statement*
17. *return-stmt* → **return ;** | **return** *expression ;*
18. *expression* → *var = expression* | *simple-expression*
19. *var* → **ID** | **ID** [*expression*]
20. *simple-expression* → *additive-expression relop additive-expression* | *additive-expression*
21. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**
22. *additive-expression* → *additive-expression addop term* | *term*
23. *addop* → **+** | **-**
24. *term* → *term mulop factor* | *factor*
25. *mulop* → ***** | **/**
26. *factor* → (*expression*) | *var* | *call* | **NUM**
27. *call* → **ID** (*args*)
28. *args* → *arg-list* | *empty*
29. *arg-list* → *arg-list , expression* | *expression*

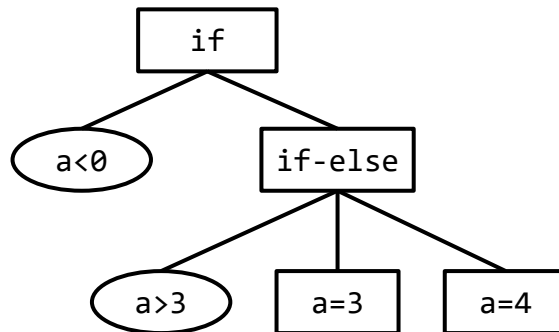
Dangling Else Problem

- Ambiguity in the grammar 15

```
/* dangling else example */  
void main(void) { if ( a < 0 ) if ( a > 3 ) a = 3; else a = 4; }
```

- (1) void main(void) { **if(a < 0)** if (a > 3) a = 3; **else a = 4;** }
- (2) void main(void) { **if(a < 0)** **if (a > 3) a = 3;** **else a = 4;** }

- Rule: Associate the else with the **nearest if**



C-MINUS COMPILATION: ./test.cm

Syntax tree:

Function Declaration: name = main, return type = void

Void Parameter

Compound Statement:

If Statement:

Op: <

Variable: name = a

Const: 0

If-Else Statement:

Op: >

Variable: name = a

Const: 3

Assign:

Variable: name = a

Const: 3

Assign:

Variable: name = a

Const: 4

AST Output Format - 1

*** Type** (*type-specifier, ...*)

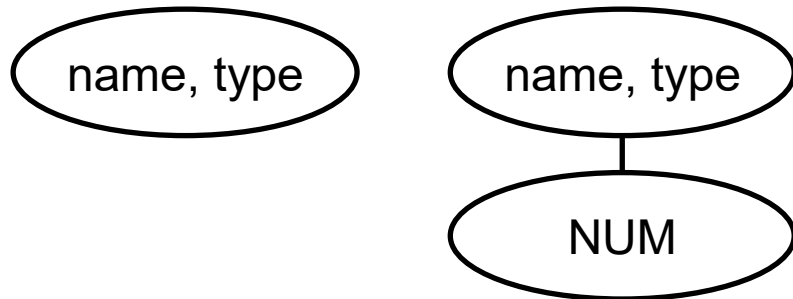
<Format: Type>

int
void
int[]
void[]

*** Variable Declaration** (*var-declaration*)

<Format: Variable Declaration>

Variable Declaration: name = %s, type = %s
/* Child Node: Array Size */



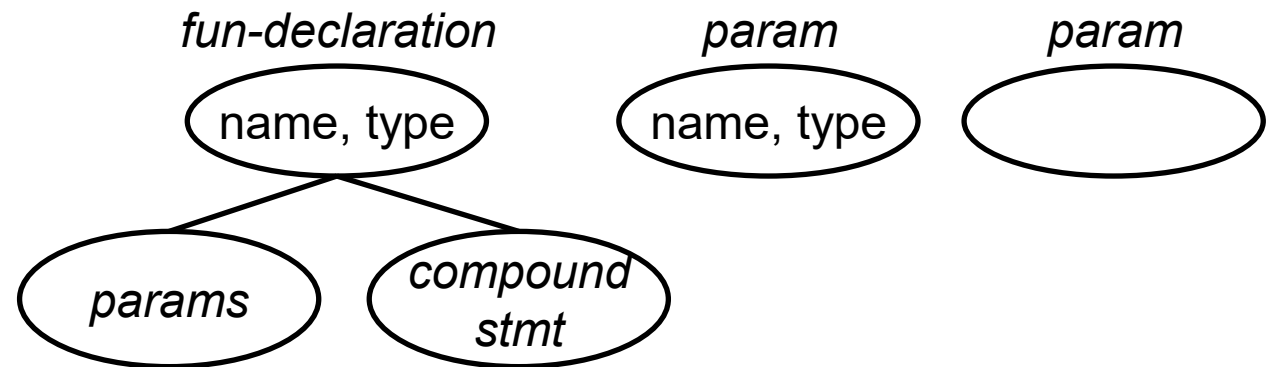
*** Function Declaration** (*fun-declaration*)

<Format: Function Declaration>

Function Declaration: name = %s, return type = %s
/* Child Node: Parameters */
/* Child Node: Compound Statement */

<Format: Parameters>

Parameter: name = %s, type = %s
Void Parameter

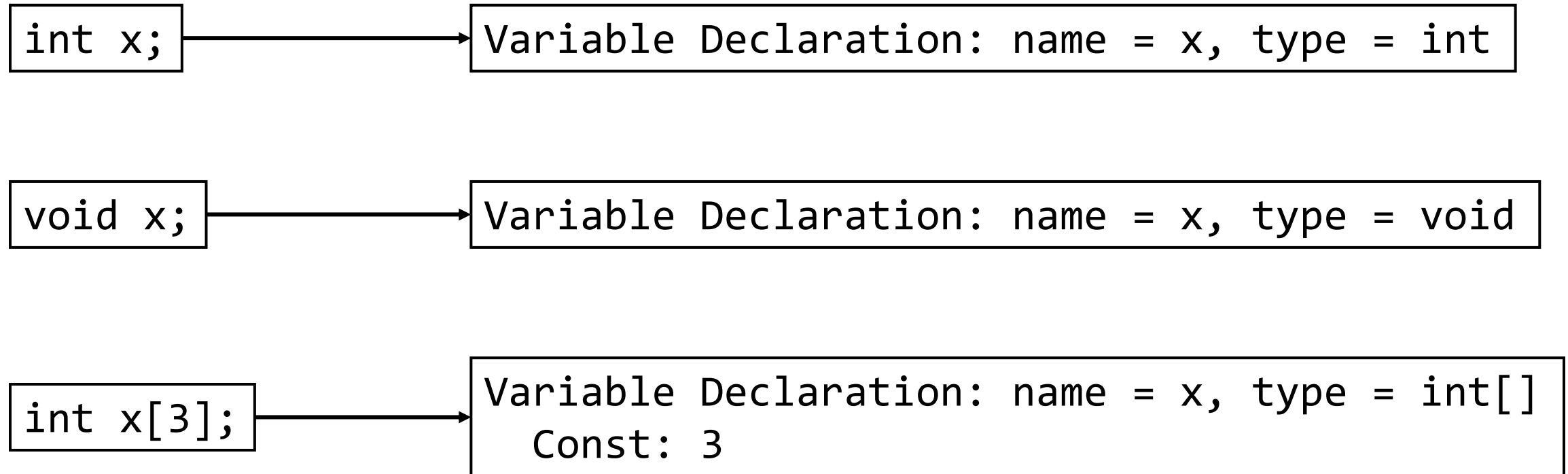


AST Output Format - 1

- Let's make an example

Source Code

AST Output



AST Output Format - 1

- Let's make an example

Source Code

AST Output

```
int func(int x, void y){}
```

Function Declaration: name = func, type = int
Parameter: name = x, type = int
Parameter: name = y, type = void
Compound Statement:

```
int func(void){}
```

Function Declaration: name = func, type = int
Void Parameter
Compound Statement:

AST Output Format - 2

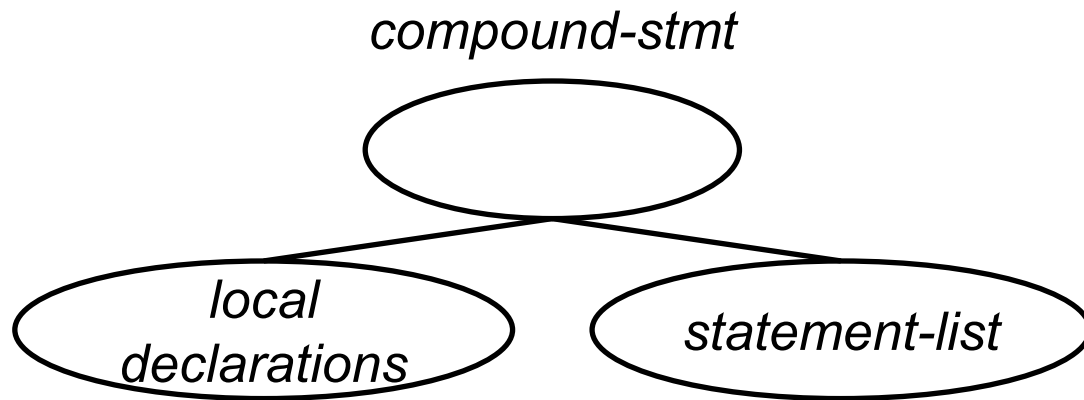
* Compound Statement (*compound-stmt*)

<Format: Compound Statement>

Compound Statement:

/* Child Node: Local Declarations */

/* Child Node: Statement Lists */



* If/If-Else Statement (*selection-stmt*)

<Format: If/If-Else Statement>

If Statement:

/* Child Node: Condition Expression */

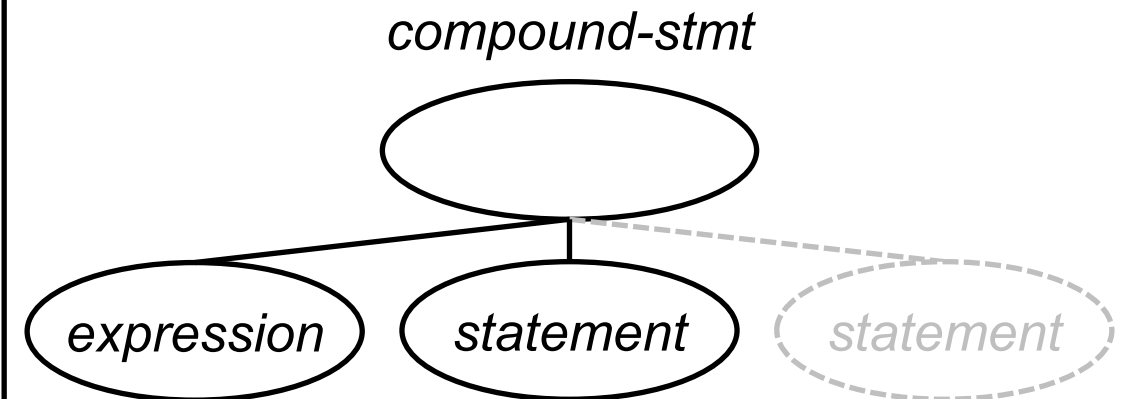
/* Child Node: Then-Statement */

If-Else Statement:

/* Child Node: Condition Expression */

/* Child Node: Then-Statement */

/* Child Node: Else-Statement */



AST Output Format - 2

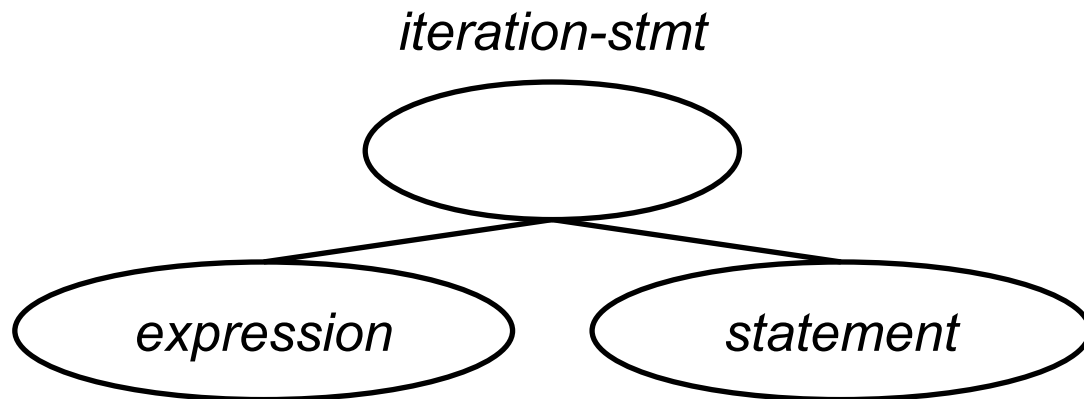
*** While Statement (*iteration-stmt*)**

<Format: While Statement>

While Statement:

/* Child Node: Condition Expression */

/* Child Node: Loop Body Statement */



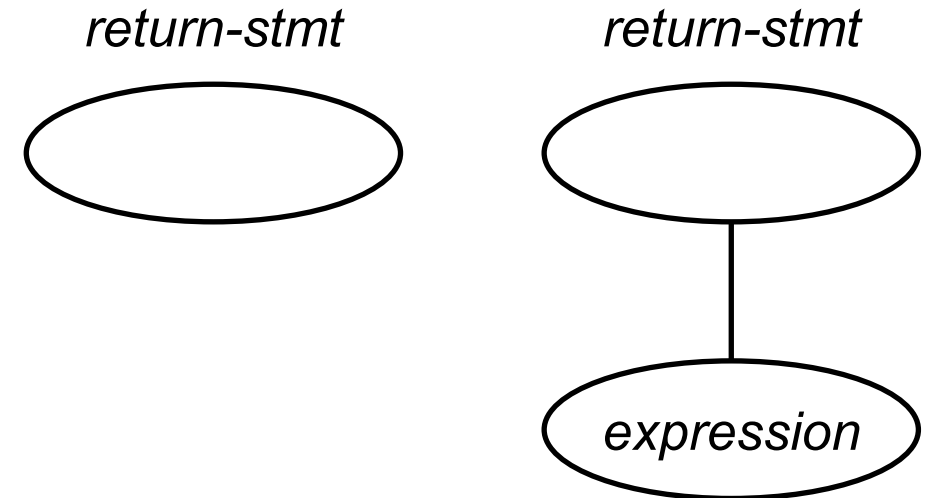
*** Return Statement (*return-stmt*)**

<Format: Return Statement>

Non-value Return Statement

Return Statement:

/* Child Node: Return Expression */



AST Output Format - 2

- Let's make an example

Source Code

```
...  
int func(void){  
    int x;  
    int y;  
    return x;  
}  
...
```

AST Output

```
Function Declaration: name = func, type = int  
Void Parameter  
Compound Statement:  
    Variable Declaration: name = x, type = int  
    Variable Declaration: name = y, type = int  
    Return Statement:  
        Variable: name = k
```

AST Output Format - 3

*** Operator (*relop*, *addop*, *mulop*)**

<Format: Operator>

+

*

<=

...

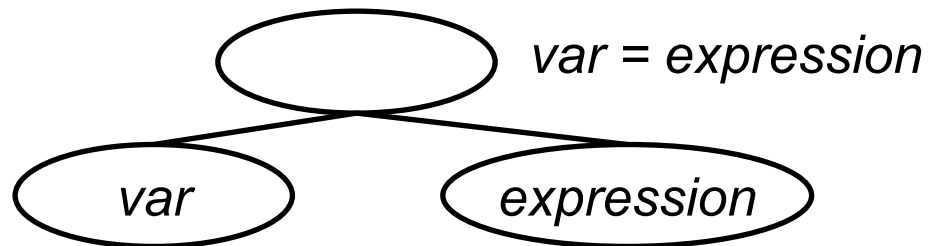
*** Assignment Expression (*var* = *expression*)**

<Format: Assignment Expression>

Assign:

/* Child Node: Variable */

/* Child Node: Expression */



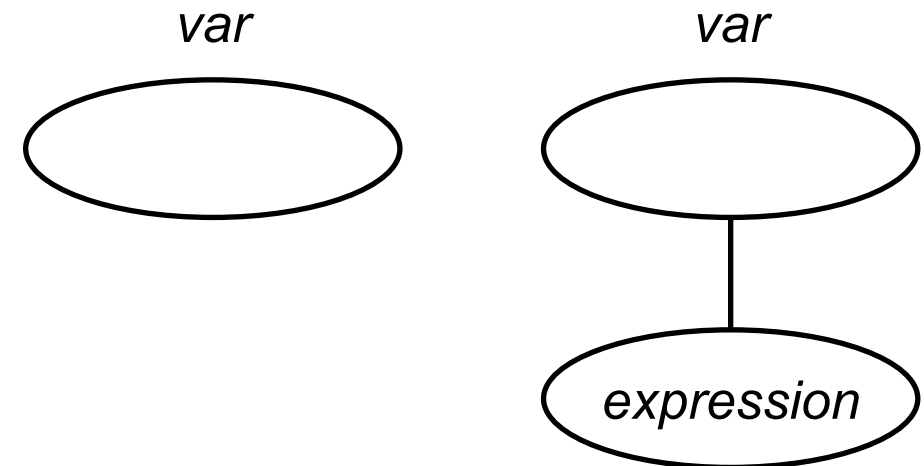
*** Variable Accessing & Array Indexing**

Expression (*var*)

<Format: Variable Accessing & Array Indexing>

Variable: name = %s

/* Child Node: Array Index Expression */



AST Output Format - 3

* Binary Operator Expression

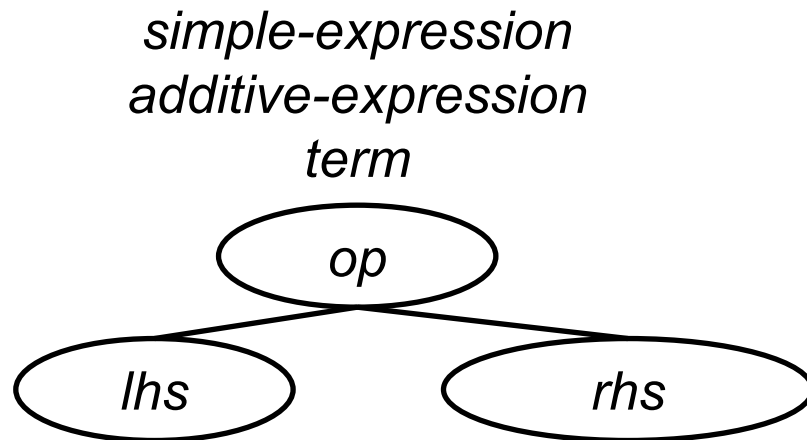
(*simple-expression, additive-expression, term*)

<Format: Binary Operator Expression>

Op: %s

/* Child Node: Left Hand Side */

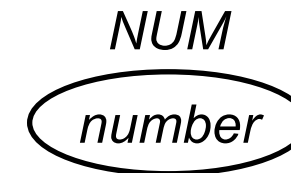
/* Child Node: Right Hand Side */



* Constant Expression (*NUM*)

<Format: Constant Expression>

Const: %d

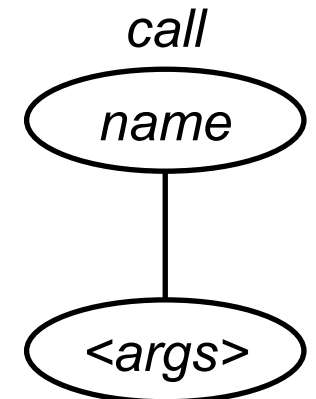


* Call Expression (*call*)

<Format: Call Expression>

Call: function name = %s

/* Child Node: Arguments */



AST Output Format - 3

- Let's make an example

Source Code

AST Output

```
...  
int temp;  
temp = 3 + 5;  
func(temp)  
...
```

Variable Declaration: name = temp, type = int

Assign:

Variable: name = temp

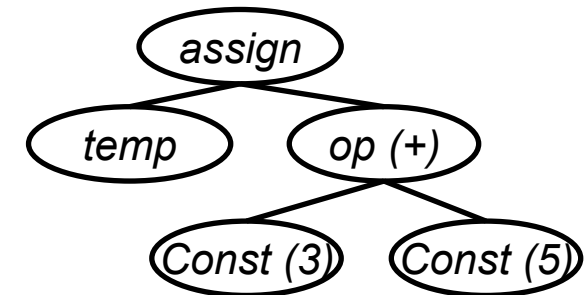
Op: +

Const: 3

Const: 5

Call: function name = func

Variable: name = temp



Yacc (Bison)

- **Parser Generator for UNIX**

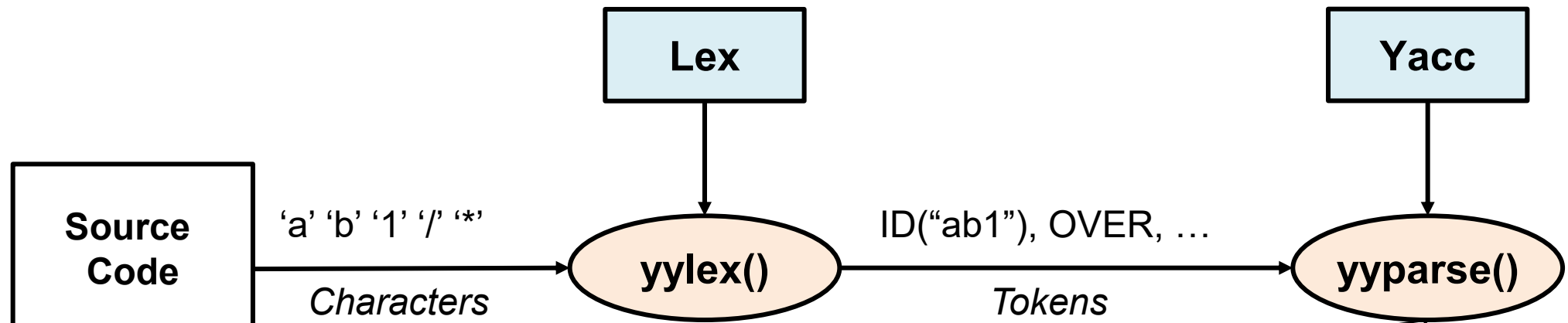
- Yacc: Yet Another Compiler Compiler
- Bison: GNU project parser generator (upward compatible with Yacc)

- **Input**

- LALR(1) grammar specification with declarations of “precedence and associativity”

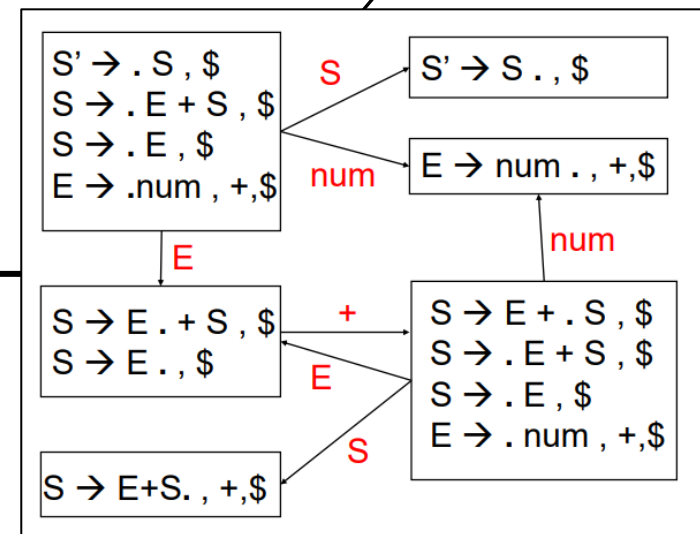
- **Output**

- LALR(1) parser program



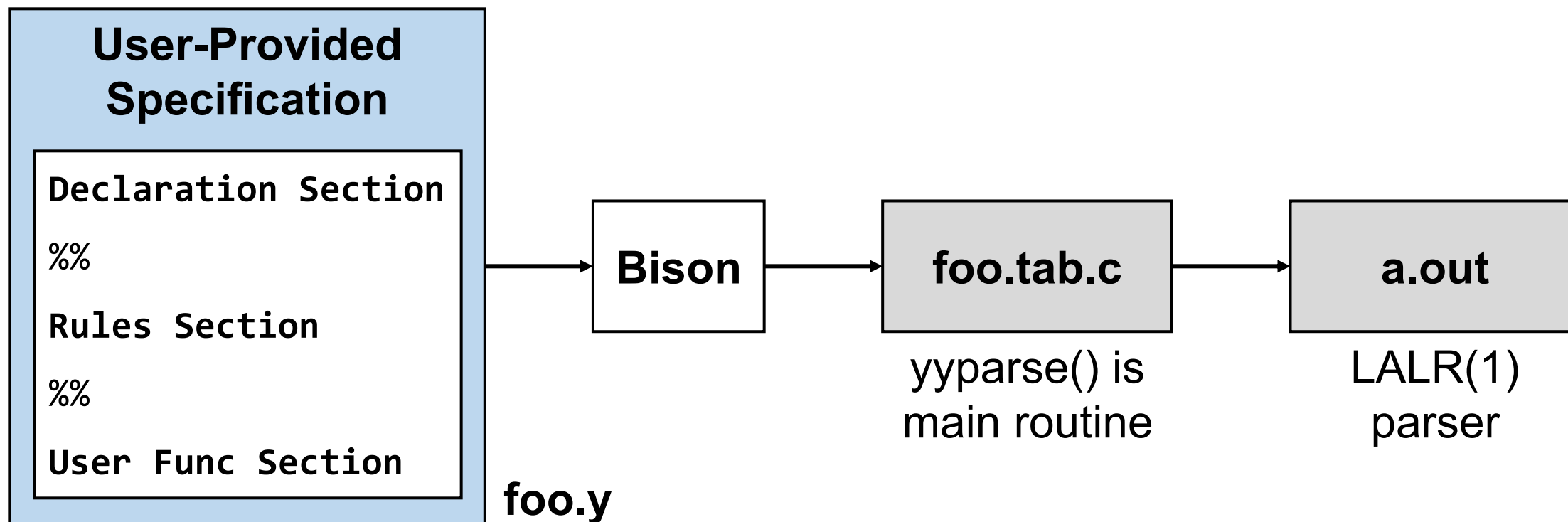
derivation	stack	input stream	action
(1+2+(3+4))+5		(1+2+(3+4))+5	shift
(1+2+(3+4))+5	(1+2+(3+4))+5	shift
(1+2+(3+4))+5	(1	+2+(3+4))+5	reduce E → num
(E+2+(3+4))+5	(E	+2+(3+4))+5	reduce S → E
(S+2+(3+4))+5	(S	+2+(3+4))+5	shift
(S+2+(3+4))+5	(S+	2+(3+4))+5	shift
(S+2+(3+4))+5	(S+2	+(3+4))+5	reduce E → num
(S+E+(3+4))+5	(S+E	+(3+4))+5	reduce S → S+E
(S+(3+4))+5	(S	+(3+4))+5	shift
(S+(3+4))+5	(S+	(3+4))+5	shift
(S+(3+4))+5	(S+(3+4))+5	shift
(S+(3+4))+5	(S+(3	+4))+5	reduce E → num

Shift-Reduce Parsing



Parsing Table

Bison / Yacc Overview



Declarations Section

```
%{  
    #include <stdio.h>  
    int yylex(void)  
    ...  
%}
```

↓

```
%nonassoc ELSE  
%token NUM  
%left MINUS PLUS  
%right EXPONENT
```

Higher
Precedence

- Users can declare or include variables, enumeration, etc using the code in between “%{“ and “%}”
- Define terminal symbols (tokens) of the grammar
 - **General:** %token terminal1 terminal2 ...
 - **Left-associative:** %left terminal1 terminal 2 ...
 - **Right-associative:** %right terminal1 terminal 2 ...
 - **No-associative:** %nonassoc terminal1 terminal 2 ...
- The declaration order determines precedence

Rules Section - 1

- **Every name in the rules section that has not been declared is a non-terminal**
- **Productions format**
 - non-terminal : first production RHS | second production RHS ...;
 - ϵ production is a blank (i.e., non-terminal : ;)
 - Use braces after RHS to indicate actions to take

Rules Section - 2

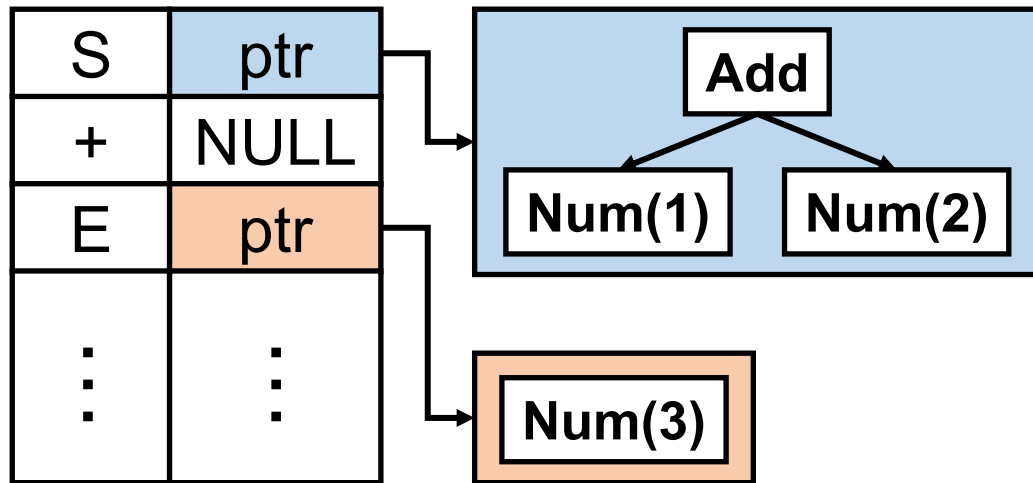
- We use special format to indicate LHS and n-th term in the action of each production
 - \$\$: LHS
 - \$n : nth symbol in RHS
- The precedence follows the patterns defined in the declaration section
 - We can also force precedence for a given production using %prec

expr: NUM	{ \$\$ = \$1; }
exp PLUS exp	{ \$\$ = \$1 + \$3; }
exp MINUS exp	{ \$\$ = \$1 - \$3; }
MINUS exp %prec MINUS	{ \$\$ = -\$2; }

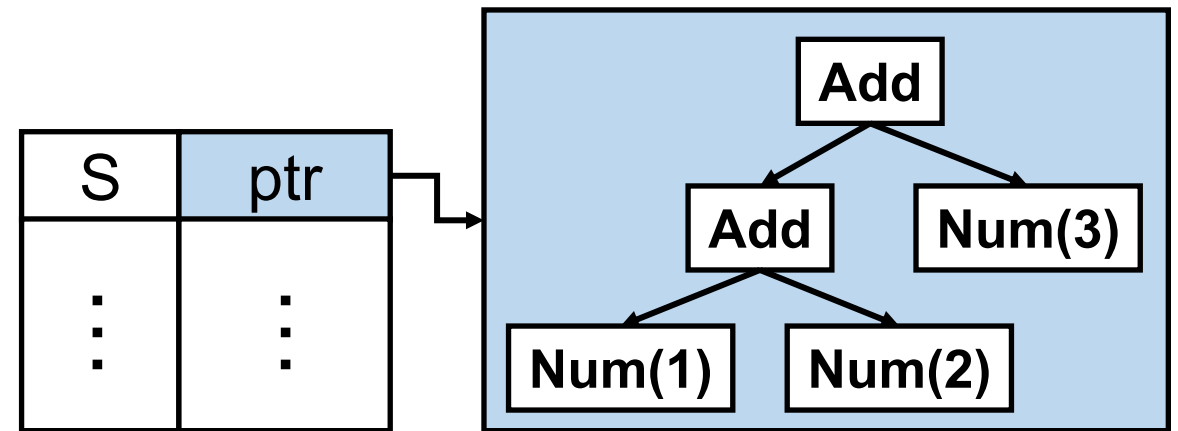
Recall: AST Construction

- **AST construction mechanism**

- Store parts of the tree on the stack
- For each nonterminal X on the stack, store the sub-tree for X on the stack
- After reduce operation for a production $X \rightarrow \alpha$, create an AST node for X



Before $S \rightarrow E + S$ Reduction



After $S \rightarrow E + S$ Reduction

Yacc Example: tiny.y

Pointer to
LHS
(non-terminal)

Pointer to
YYSTYPE
(*TreeNode**)

\$\$

\$1

\$2

\$3

\$4

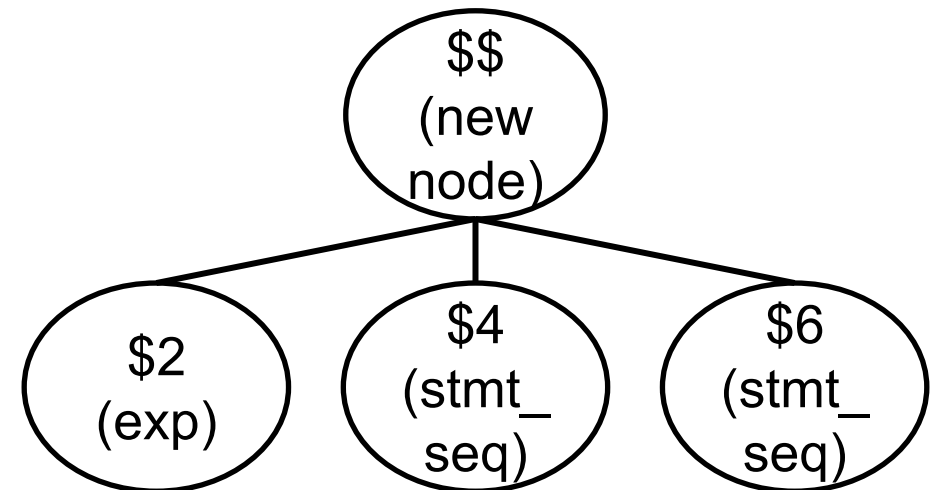
\$5

```
if_stmt : IF exp THEN stmt_seq END
        { $$ = newStmtNode(IfK);
          $$->child[0] = $2;
          $$->child[1] = $4;
        }
        | IF exp THEN stmt_seq ELSE stmt_seq END
        { $$ = newStmtNode(IfK);
          $$->child[0] = $2;
          $$->child[1] = $4;
          $$->child[2] = $6;
        }
        ;
```

Executed at
REDUCE

```
#define MAXCHILDREN 3

typedef struct treeNode
{ struct treeNode * child[MAXCHILDREN];
  struct treeNode * sibling;
  int lineno;
  NodeKind nodekind;
  union { StmtKind stmt; ExpKind exp; } kind;
  union { TokenType op;
          int val;
          char * name; } attr;
  ExpType type; /* for type checking of exps */
} TreeNode;
```



Yacc Usages

- **Usage**

- yacc [options] filename

- **Options**

- d write definitions (y.tab.h)
 - o [output_file] (default: y.tab.c)
 - t add debugging support
 - v write description (y.output)

- **Manual**

- <https://www.gnu.org/software/bison/manual>

Modify main.c File

- main.c

- Modify code to print only syntax tree
- NO_ANALYZE, TraceParse

```
1 /* File: main.c
2 /* Main program for TINY compiler
3 /* Compiler Construction: Principles and Practice
4 /* Kenneth C. Louden
5 /*
6 /*
7
8 #include "globals.h"
9
10 /* set NO_PARSE to TRUE to get a scanner-only compiler */
11 #define NO_PARSE FLASE
12 /* set NO_ANALYZE to TRUE to get a parser-only compiler */
13 #define NO_ANALYZE TRUE
14
15 /* set NO_CODE to TRUE to get a compiler that does not
16 * generate code
17 */
18 #define NO_CODE FALSE
19
20 #include "util.h"
21 #if NO_PARSE
22 #include "scan.h"
23 #else
24 #include "parse.h"
25 #if NO_ANALYZE
26 #include "analyze.h"
27 #if NO_CODE
28 #include "cgen.h"
29 #endif
30 #endif
31 #endif
32
33 /* allocate global variables */
34 int lineno = 0;
35 FILE * source;
36 FILE * listing;
37 FILE * code;
38
39 /* allocate and set tracing flags */
40 int EchoSource = FALSE;
41 int TraceScan = FALSE;
42 int TraceParse = TRUE;
43 int TraceAnalyze = FALSE;
44 int TraceCode = FALSE;
45
46 int Error = FALSE;
```

```
10 /* set NO_PARSE to TRUE to ge
11 #define NO_PARSE FLASE
12 /* set NO_ANALYZE to TRUE to
13 #define NO_ANALYZE TRUE
```

```
39 /* allocate and set tracing flags */
40 int EchoSource = FALSE;
41 int TraceScan = FALSE;
42 int TraceParse = TRUE;
43 int TraceAnalyze = FALSE;
44 int TraceCode = FALSE;
45
46 int Error = FALSE;
```

Modify globals.h File

- **Overwrite your globals.h with yacc/globals.h**
- **Modify the “Syntax tree for parsing” part in the global.h file**
 - Free to modify/add/remove NodeKind, StmtKind, ExpKind, ExpType, and TreeNode
→ Modify them according to the C-Minus grammar in page 3
 - Free to modify the kind, attr, and type if you want (TreeNode* is used to define YYSTYPE in cminus.y), following the modified NodeKind, StmtKind, ExpKind ...

Modify global.h File

```
/* ****  
/* **** Syntax tree for parsing ****  
/* ****  
  
typedef enum {StmtK,ExpK} NodeKind;  
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK} StmtKind;  
typedef enum {OpK,ConstK,IdK} ExpKind;  
  
/* ExpType is used for type checking */  
typedef enum {Void,Integer,Boolean} ExpType;  
  
#define MAXCHILDREN 3  
  
typedef struct treeNode  
{ struct treeNode * child[MAXCHILDREN];  
  struct treeNode * sibling;  
  int lineno;  
  NodeKind nodekind;  
  union { StmtKind stmt; ExpKind exp;} kind;  
  union { TokenType op;  
          int val;  
          char * name; } attr;  
  ExpType type; /* for type checking of exps */  
} TreeNode;
```

Modify util.c File

- **util.c**

- printTree() function should be updated to print C-Minus Syntax Tree.
 - Refer to the AST output format to determine what to print (for each node type you defined in global.h)
 - INDENT and UNINDENT macros automatically control the indentation (for the tree structure)
 - printTree() traverses child and sibling fields in TreeNode
- Update newStmtNode(), newExprNode() or other function to allocate and initialize new Node type (according to the one you defined in global.h file)

Modify cminus.y File

- **cminus.y**
 - Copy yacc/tiny.y to cminus.y.
 - Write C-Minus tokens in the definition section.
 - Consider priority and associativity (and remove ambiguity)
 - Define a C-Minus grammar and reduce actions for each rules.
 - YYSTYPE (the type of \$\$, \$1, ...) is defined as TreeNode*.

Example Syntax Tree

```
/* A program to perform Euclid's
```

```
Algorithm to computer gcd */
```

```
int gcd (int u, int v)
```

```
{
```

```
    if (v == 0) return u;
```

```
    else return gcd(v,u-u/v*v);
```

```
    /* u-u/v*v == u mod v */
```

```
}
```

```
void main(void)
```

```
{
```

```
    int x; int y;
```

```
    x = input(); y = input();
```

```
    output(gcd(x,y));
```

```
}
```



C-MINUS COMPILATION: ./test.1.txt

Syntax tree:

Function Declaration: name = gcd, return type = int

Parameter: name = u, type = int

Parameter: name = v, type = int

Compound Statement:

If-Else Statement:

Op: ==

Variable: name = v

Const: 0

Return Statement:

Variable: name = u

Return Statement:

Call: function name = gcd

Variable: name = v

Op: -

Variable: name = u

Op: *

Op: /

Variable: name = u

Variable: name = v

Variable: name = v

Function Declaration: name = main, return type = void

Void Parameter

Compound Statement:

Variable Declaration: name = x, type = int

Variable Declaration: name = y, type = int

Assign:

Variable: name = x

Call: function name = input

Assign:

Variable: name = y

Call: function name = input

Call: function name = output

Call: function name = gcd

Variable: name = x

Variable: name = y

Some Hints - 1

- You should generate the exactly same output as the reference and output format
- Remove all YACC conflicts even if it is just a warning
 - **PENALTIES FOR EACH CONFLICT**: Shift/Shift, Shift/Reduce, Reduce/Reduce
- There are some potential errors
 - *If* without *Else* statement and *If-Else* Statement
 - No Parameter (*void*) and Parameters
 - *Return* statement without value and *return* statement with value

Some Hints - 2

- **How to implement Lists?** (*declaration-list, statement-list, param-list, ...*)
 - Hint: see stmt_seq in tiny.y
- **Store attributes of TreeNode such as ID (=name), type and op**
 - Consideration: TokenString may not contain “string of the ID token” when reduce
 - Intra-Rule action (performed at shift) such as [assign_stmt] in tiny is not recommended (This may cause unexpected actions)
 - Do not directly update variables handled by scanner such as TokenString. Use copyString()

Some Hints - 3

- You should properly set the line number (This will be important when doing a semantic analysis)
- In `var_declaration`, `fun_declaration`, and `...` : the `lineno` is set according to `lineno` of the identifier

```
fun_declaration      : type_specifier identifier LPAREN params RPAREN compound_stmt
                      {
                        $$ = newTreeNode(FunctionDecl);
                        $$->lineno = $2->lineno;
                        $$->type = $1->type;
                      }
```

- Remaining cases: `lineno` is set using the last symbol `lineno`

```
compound_stmt       : LCURLY local_declarations statement_list RCURLY
                      {
                        $$ = newTreeNode(CompoundStmt);
                        $$->lineno = lineno;
                      }
```

Some Hints - 4

- You don't need to care about Semantics, just Syntax analyzer will be okay. (Analyzing semantics is for Project 3.)
- For this example, this code will be parsed correctly even though the code has some semantic error.

```
/* Semantic Error Example */
/* (1) void-type variable a, b
 * (2) uninitialized variable c (and b)
 * (3) undefined variable d */

int main ( void a[] )
{
    void b;
    int c;
    d[1] = b + c;
}
```



C-MINUS COMPILATION: ./error_test.cm

Syntax tree:

Function Declaration: name = main, return type = int

Parameter: name = a, type = void[]

Compound Statement:

Variable Declaration: name = b, type = void

Variable Declaration: name = c, type = int

Assign:

Variable: name = d

Const: 1

Op: +

Variable: name = b

Variable: name = c

Build with Makefile

- I will upload a proper makefile for you
 - Execute `make cminus_parser`

Evaluation

- **Evaluation Items**

- **Compilation** (Success / Fail): **20%**

- Please describe in the report how to build your project.

- **Correctness** check for several testcases: **70%**

- Note: **Comments** are also one of key check point.
 - Note: Make sure there are no **segmentation fault** or **infinite loop** on any inputs.

- **Report** : **10%**

Report

- **Guideline (≤ 5 pages)**

- Compilation environment and method
- Brief explanations about how to implement and how it operates
- Examples and corresponding result screenshots

- **Format**

- Use PDF with the filename as follows

Submission

- **Deadline: 10/28 23:59:00**
- **Submission**
 - Submit all the source codes in a single zip file and report as a pdf file
 - Format + Name:
 - Report: [Student No]_Project2.pdf
 - Code: do not modify any name and compress all the codes into a single zip file and the name should be
 - [Student No]_Project2.zip
- **Questions**
 - Upload any questions to the LMS