

# Mini Project 1

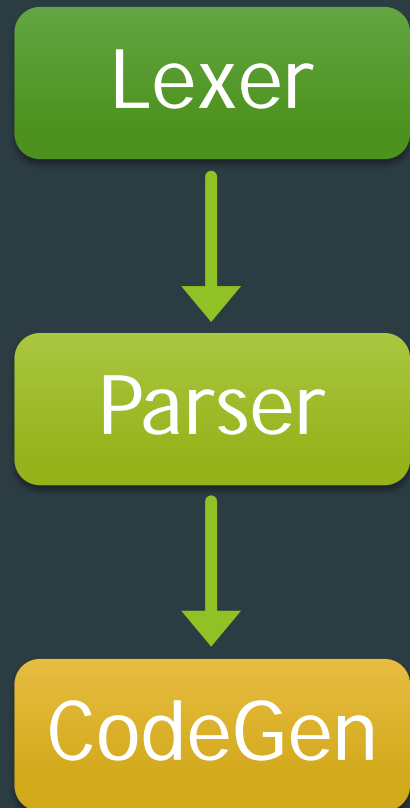
# Goal

大致了解compiler的架構

了解從高階語言至組合語言的過程

Trace code能力與實作能力

# Compiler 架構



# Compiler 架構

Lexer



Parser



CodeGen



## Lexer (Lexical Analyzer)

▶ 將input切成一個一個的token以利後續處理。

▶ Input: `x=y+++5`

▶ Token:

Identifier (`x`)

Assign (`=`)

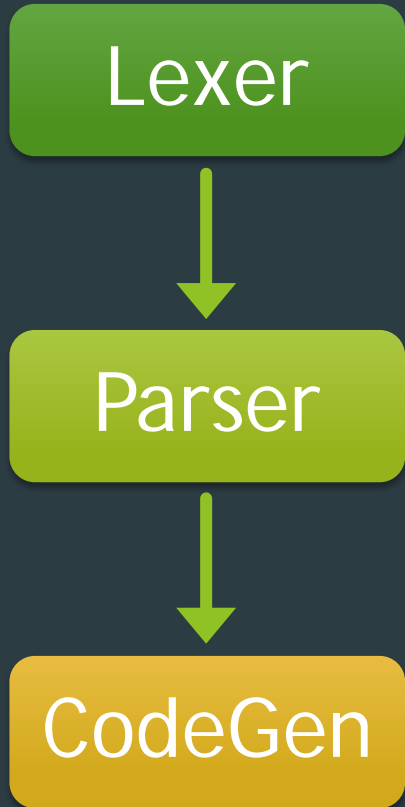
Identifier (`y`)

PostInc (`++`)

Add (`+`)

Constant (`5`)

# Compiler 架構



## Parser

- ▶ 根據grammar rule，利用tokens建立出正確的AST(Abstract Syntax Tree)。

- ▶ Token:

```
Identifier (x)
Assign      (=)
Identifier  (y)
PostInc     (++)
Add         (+)
Constant    (5)
```

- ▶ AST:

```
Assign (=)
├── Identifier (x)
└── Add (+)
    ├── PostInc (++)
    │   └── Identifier (y)
    └── Constant (5)
```

--root是"="

--"="的左邊是"x"

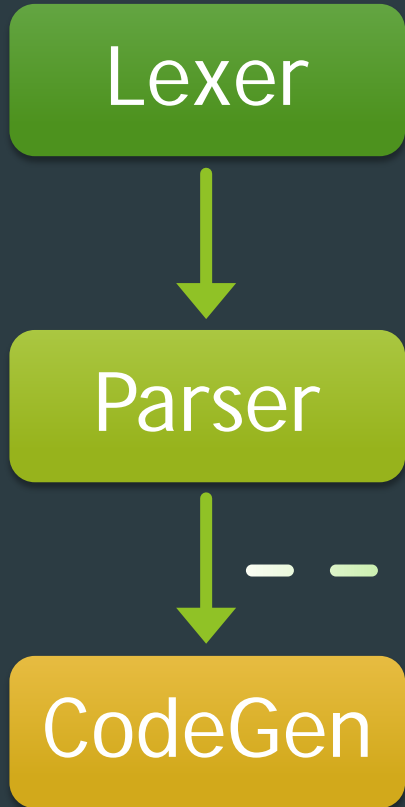
--"="的右邊是"+"

--"+"的左邊是"++"

--"++"跟著"y"

--"+"的右邊是"5"

# Compiler 架構



## semantic check

► 確認生成的AST是否有語意錯誤

► AST:

```
Assign (=)
├ Identifier (x)
└ Add (+)
  ├── PostInc (++) → PostInc必須接Identifier
  │   └ Identifier (y) → OK
  └ Constant (5)
```

► AST:

```
Assign (=)
├ Identifier (x)
└ Add (+)
  ├── PostInc (++) → PostInc必須接Identifier
  │   └ Identifier (9) → Semantical error!
  └ Constant (5)
```

# Compiler 架構

Lexer



Parser



CodeGen



## CodeGen

► 利用建立的AST，把ASM生出來。

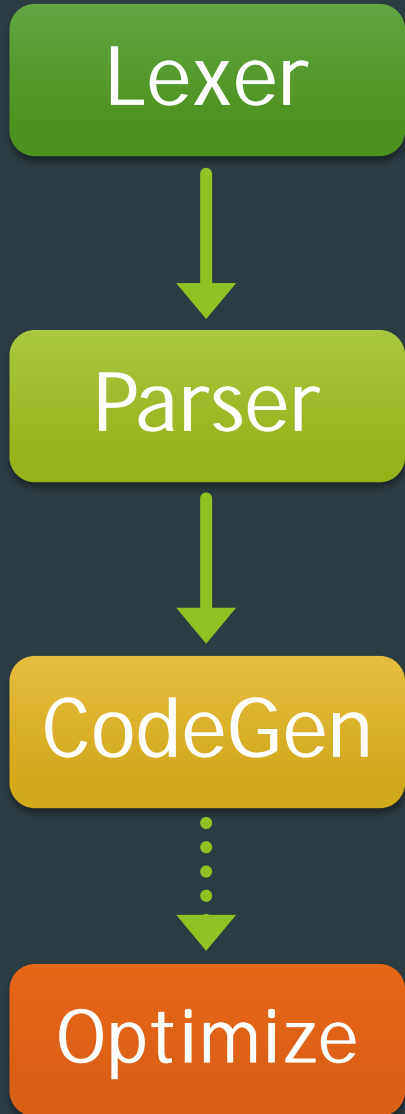
► AST:

```
Assign
├── Identifier (x)
└── Add (+)
    ├── PostInc (++)
    │   └── Identifier (y)
    └── Constant (5)
```

► ASM:

load "y" into r0	
add r0 and 1 into r1	y++
store r1 into "y"	
add 0 and 5 into r1	5
add r0 and r1 into r0	y+++5
store r0 into "x"	x=y+++5

# Compiler 架構



## Optimization

► 優化生出來的ASM，使其所耗的cycle數更少。

► ASM:

```
load "y" into r0
add r0 and 1 into r1
store r1 into "y"
add 0 and 5 into r1
add r0 and r1 into r0
store r0 into "x"
```

► After optimization:

```
load "y" into r0
add r0 and 1 into r1
store r1 into "y"
add r0 and 5 into r0
store r0 into "x"
```



# Code Trace - main()

```
int main() {  
    while (fgets(input, MAX_LENGTH, stdin) != NULL) {  
        Token *content = lexer(input);  
        size_t len = token_list_to_arr(&content);  
        AST *ast_root = parser(content, len);  
        semantic_check(ast_root);  
        codegen(ast_root);  
        free(content);  
        freeAST(ast_root);  
    }  
    return 0;  
}
```

逐行吃輸入(進input)&處理

將input轉為Token linked list

方便起見，linked list轉為array

用Token array建AST

執行semantic check

Generate ASM code

釋放記憶體: Token array

釋放記憶體: AST tree

Optimization在這裡!

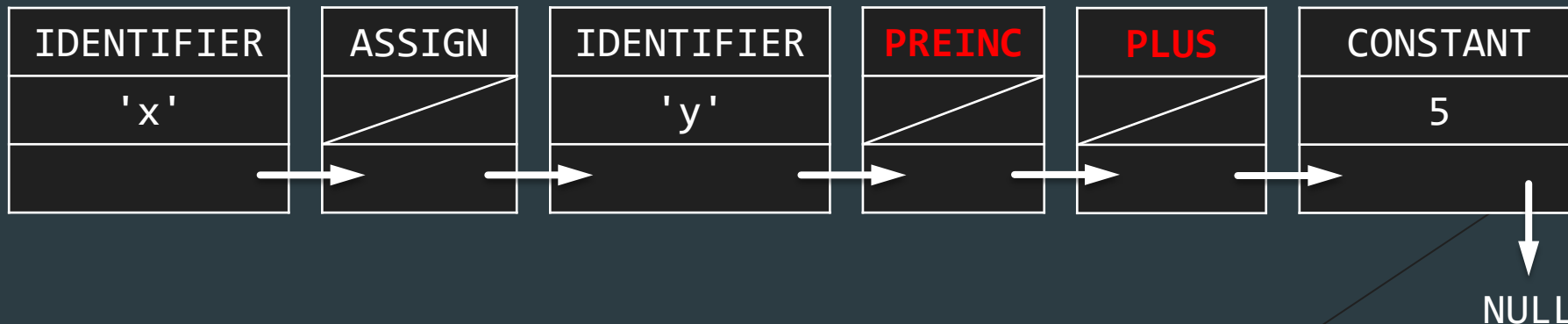
# Code Trace - lexer(const char\*)

## ► Token:

- Identifier(x), Assign(=), Identifier(y), PostInc(++), Add(+), Constant(5)

```
typedef struct TokenUnit {  
    Kind kind;  
    int val;  
    struct TokenUnit *next;  
} Token;
```

## ► struct Token (linked list):

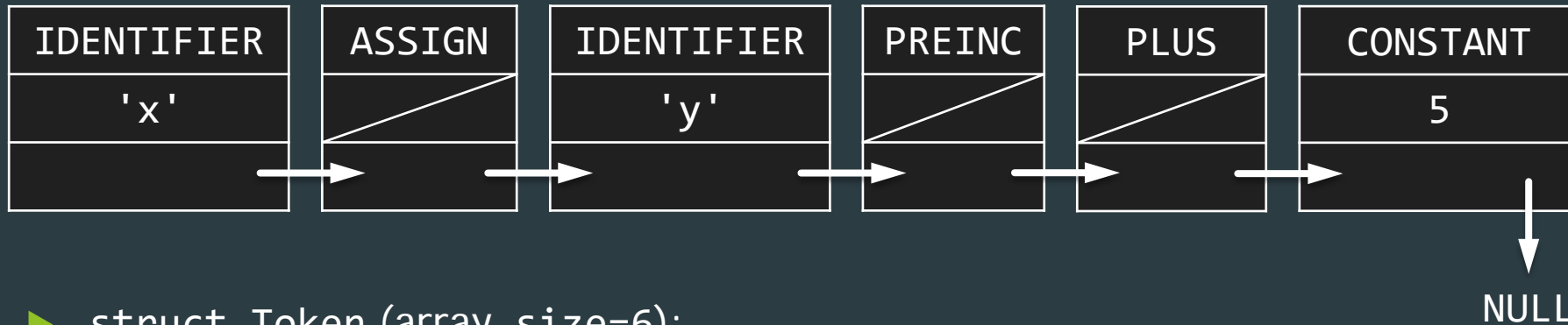


# Code Trace - `lexer(const char*)`

- ▶ `lexer`不會辨認是`PostInc`還是`PreInc`，只會辨認為`"++"`。
  - ▶ 方便起見，統一設`kind = PreInc`，`PostDec/PreDec`同理。
- ▶ `lexer`不會辨認是`Plus`還是`Add`，只會辨認為`"+"`。
  - ▶ 方便起見，統一設`kind = Plus`，`Minus/Sub`同理。
- ▶ 在`parser`建完AST的時候，這些就會全部被定位出來。

# Code Trace - token\_list\_to\_arr(Token\*\*)

► struct Token (linked list):



► struct Token (array, size=6):

Index	0	1	2	3	4	5
kind	IDENTIFER	ASSIGN	IDENTIFER	PREINC	PLUS	CONSTANT
val	'x'		'y'			5
next						

# Code Trace – parser(Token\*, size\_t)

- ▶ lexer無法辨別的PLUS/ADD和MINUS/SUB，在parser先行辨別。
- ▶ 辨別後再呼叫parse正式開始構建AST。

```
AST *parser(Token *arr, size_t len)
```

回傳構建完成的AST的root

傳入先前建好的Token array

傳入array的長度

# Code Trace - parse(Token\*, int, int, GrammarState)

- ▶ 遵照grammar rule構建AST。

```
AST *parse(Token *arr, int l, int r, GrammarState S)
```

回傳當前範圍建出的AST

整個Token array

arr的範圍：左界

arr的範圍：右界

當前的state

```
switch (S) {  
    case STMT:  
        .....  
    case EXPR:  
        .....  
    case ASSIGN_EXPR:  
        .....  
    case ADD_EXPR:  
        .....  
    case .....  
        .....  
    case PRI_EXPR:  
        .....  
}
```

# Code Trace – parse(Token\*, int, int, GrammarState)

- ▶ 先嘗試走 ASSIGN\_EXPR -> UNARY\_EXPR ASSIGN ASSIGN\_EXPR 的路線。
- ▶ 找不到(if (nxt != -1))則當作 ADD\_EXPR 往下走。

- ▶ 其餘state類似。

```
ASSIGN_EXPR  
  → ADD_EXPR  
  | UNARY_EXPR ASSIGN ASSIGN_EXPR  
  ;
```

```
case ASSIGN_EXPR:  
    if ((nxt = findNextSection(arr, l, r, condASSIGN)) != -1) {  
        now = new_AST(arr[nxt].kind, 0);  
        now->lhs = parse(arr, l, nxt - 1, UNARY_EXPR);  
        now->rhs = parse(arr, nxt + 1, r, ASSIGN_EXPR);  
        return now;  
    }  
    return parse(arr, l, r, ADD_EXPR);
```

# Code Trace – parse(Token\*, int, int, GrammarState)

- ▶ 第1/2個TODO在這裡！
  - ▶ 完成MUL\_EXPR的parse
  - ▶ 完成UNARY\_EXPR的parse

```
MUL_EXPR  
    → ...  
    ;  
UNARY_EXPR  
    → ...  
    ;
```

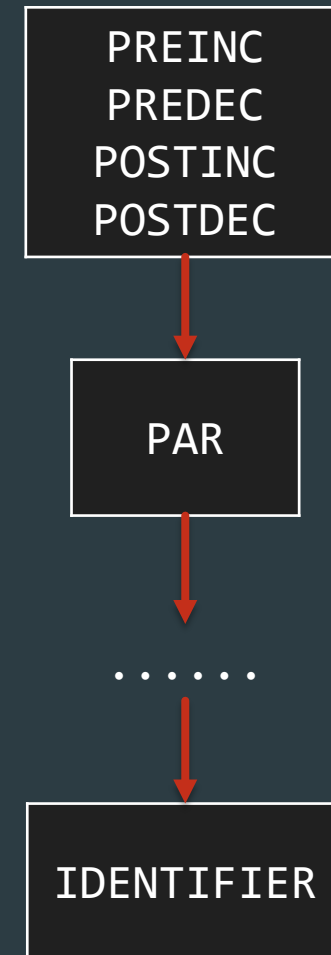
- ▶ Hint:
  - ▶ 參考grammar rule
  - ▶ 參考其他state如何進行

```
case MUL_EXPR:  
    // TODO: Implement MUL_EXPR.  
    // hint: Take ADD_EXPR as reference.  
case UNARY_EXPR:  
    // TODO: Implement UNARY_EXPR.  
    // hint: Take POSTFIX_EXPR as reference.
```



# Code Trace – semantic\_check(AST\*)

- ▶ 檢查有沒有semantical error。
  - ▶ '='的左邊只能是Identifier或是被括號層層包圍的Identifier。
    - ▶ ((((((z))))))=x+y
  - ▶ '++'/'--'的底下只能是Identifier或是被括號層層包圍的Identifier。
    - ▶ (((((x))))++) , --((((y)))) , ++z



# Code Trace – semantic\_check(AST\*)

- ▶ 第3個TODO在這裡！
  - ▶ ASSIGN已經完成了，完成剩餘部份的semantic\_check code。
- ▶ Hint:
  - ▶ 還剩下INC/DEC的部份
  - ▶ semantic\_check需要檢查所有AST node，如何做到？

```
// Operand of INC/DEC must be an identifier or identifier with one or more parentheses.  
// TODO: Implement the remaining semantic_check code.  
// hint: Follow the instruction above and ASSIGN-part code to implement.  
// hint: Semantic of each node needs to be checked recursively (from the current node to lhs/mid/rhs node).
```

# Code Trace – codegen(AST\*)

- ▶ 給定AST root，生出對應的ASM。
- ▶ 第4個TODO在這裡！
  - ▶ 完成整個codegen
  - ▶ codegen形式自由，你可以改parameter、return type，甚至整個結構
    - ▶ 其實你想重寫一份都可以
  - ▶ 直接輸出ASM，或是寫個ASM structure方便管理?
- ▶ Hint:
  - ▶ 整顆AST這麼大，我應該從哪開始生起?

```
void codegen(AST *root) {  
    // TODO: Implement your codegen in your own way.  
    // You may modify the function parameter or the return type,  
    even the whole structure as you wish.  
}
```

# Code Trace - utility interfaces

- ▶ 提供各式各樣的function以利實作。
- ▶ `err(x)`:
  - ▶ 輸入的expression不符合grammar / 無法通過semantic check時呼叫。
  - ▶ `DEBUG`改為1可以方便看到錯誤產生在code的哪一行，上傳前要改回0。

```
#define err(x) {\n    puts("Compile Error!");\n    if(DEBUG) {\n        fprintf(stderr, "Error at line: %d\\n", __LINE__); \n        fprintf(stderr, "Error message: %s\\n", x);\n    }\n    exit(0);\n}\n#define DEBUG 0
```

# Code Trace - debug interfaces

- ▶ 提供debug function以利除錯。

```
// Print token array.  
void token_print(Token *in, size_t len);  
// Print AST tree.  
void AST_print(AST *head);
```

將Token array與其長度輸入，就會自動輸出所有Token。

將建立完成的AST的root，就會自動以樹狀結構輸出AST。

# ISA Introduction

- ▶ 包含7種instruction:

- ▶ add      加法運算
- ▶ sub      減法運算
- ▶ mul      乘法運算
- ▶ div      除法運算
- ▶ rem      模運算(%)
- ▶ load    將指定位置的memory資料存進指定位置的register中
- ▶ store   將指定位置的register資料存進指定位置的memory中

# ISA Introduction

- ▶ add: 花費10 cycle。
  - ▶ add r0 r2 r7: 將r2與r7的值相加，並存在r0中。
    - ▶ 若r2=10, r7=1024, add執行完後， $r0=r2+r7=1034$ 。
  - ▶ add r0 1 2: 將r0的值設為1+2。
    - ▶ 輸入的數值只能是**非負整數**(register和memory存的value沒關係)。
- ▶ 其他的(sub, mul, div, rem)也類似。
- ▶ sub: 花費10 cycle
- ▶ mul: 花費30 cycle
- ▶ div: 花費50 cycle
- ▶ rem: 花費60 cycle

# ISA Introduction

- ▶ load: 花費200 cycle
  - ▶ load r1 [4]: 將memory [4]的值存進register r1中。
    - ▶ 若[4]=18, load執行完後, r1=18。
- ▶ store: 花費200 cycle
  - ▶ store [8] r3: 將register r3的值存進memory [8]中。
    - ▶ 若r3=-12, store執行完後, [8]=-12。



# ISA Introduction

- ▶ 變數: 僅x, y, z三個, 分別存在memory [0], [4], [8]中。
- ▶ register限制: 可使用r0 - r255, 但使用r8或以上的register將會有penalty。
  - ▶ mul cycle為30, 但若是mul r0 r12 r13, 則cycle倍增為60。
- ▶ "Compile Error!": 程式吃到錯誤的expression時應輸出此instruction。
  - ▶ 可以直接使用Macro err(x)來輸出並結束程式。

# Assembly Compiler

- ▶ ASMC - Assembly Compiler，逐行輸入ASM，直到EOF時會吐出x, y, z最後的值與總耗費cycle數。
  - ▶ x, y, z初始設定為2, 3, 5
  - ▶ x, y, z可自訂初值，詳細設定參考github page。
- ▶ 由於是用C++編寫，編譯請照github page上的步驟做。
- ▶ 務必利用ASMC做debug，會有極大的幫助。

```
load r0 [4]
add r0 r0 5
store [8] r0
x, y, z = 2, 3, 8
Total cycle = 410
```

# 評分

- ▶ 占**總成績 8%**
- ▶ 一共24個testcases，其中有6個basic testcases公布在github page上。
- ▶ 24筆testcases全過的同學中，總耗費cycle數最低的前10%會拿到**總成績2分**的bonus。
- ▶ 使用ASMC搭配python script做評分，一切以ASMC的輸出為準。
  - ▶ 評分時 $(x, y, z)$ 初始值不會是 $(2, 3, 5)$ ，而是 $[-100, 100]$ 的隨機值。偷吃步不用想了

# Submission / Demo

- ▶ 11/06(三) 晚上11:59前繳交code
  - ▶ 上傳到iLMS並命名為 **mini1.c**
- ▶ demo time: 11/07(四) 晚上06:30起
  - ▶ 務必出席，若有事無法前來請事先告知。
  - ▶ Demo結束後會在一定時間內公布通過的測資數與cycle數比賽的結果。
- ▶ 若違反上述規則(未命名成mini1.c / 未出席demo ...etc)，TA有權斟酌扣分。情節重大者(抄襲 ...etc)可能會重罰或上報校方。



Questions?

# 補充 - optimization

- ▶ 盡可能地透過重新編排、計算已知等手段，使ASM得以能夠在減少總耗費cycle數的同時保持著運算的正確性。
- ▶ 通常在codegen生完ASM時才進行。

optimization通常在這裡

```
Token *content = lexer(input);
size_t len = token_list_to_arr(&content);
AST *ast_root = parser(content, len);
semantic_check(ast_root);
codegen(ast_root);
free(content);
freeAST(ast_root);
```

# 補充 - optimization

```
x=(2+3)/10*5%2-31
```

codegen

```
add r0 2 3  
div r0 r0 10  
mul r0 r0 5  
rem r0 r0 2  
sub r0 r0 31  
store [0] r0
```

optimization

```
sub r0 0 31  
store [0] r0
```

可以算的先算掉，能少掉很多inst

```
x=(x+(y-  
(z*(x/(y%(z+(x-  
(y*(z/(x%(y+(z-  
(x*(y/(z%5)))))))  
))))))
```

```
x=5+x  
y=x*x-(12*12)
```

# 補充 - optimization

```
x=(2+3)/10*5%2-31
```

```
x=(x+(y-  
(z*(x/(y%(z+(x-  
(y*(z/(x%(y+(z-  
(x*(y/(z%5)))))))  
))))))
```

```
x=5+x  
y=x*x-(12*12)
```

codegen



```
load r0 [0]  
load r1 [4]  
.....  
load r14 [8]  
rem r14 r14 5  
.....
```



# 補充 - optimization

```
x=(2+3)/10*5%2-31
```

```
x=(x+(y-  
(z*(x/(y%(z+(x-  
(y*(z/(x%(y+(z-  
(x*(y/(z%5)))))))  
))))))
```

```
x=5+x  
y=x*x-(12*12)
```

codegen

```
load r0 [0]  
load r1 [4]  
.....  
load r14 [8]  
rem r14 r14 5  
.....
```

optimization

```
load r0 [8]  
rem r0 r0 5  
load r1 [4]  
div r0 r1 r0  
.....
```

不相關的insts可以交換，而不影響結果

# 補充 - optimization

```
x=(2+3)/10*5%2-31
```

```
x=(x+(y-  
(z*(x/(y%(z+(x-  
(y*(z/(x%(y+(z-  
(x*(y/(z%5)))))))  
))))))
```

```
x=5+x  
y=x*x-(12*12)
```

codegen

```
load r0 [0]  
add r0 r0 5  
store [0] r0  
load r0 [0]  
load r1 [0]  
mul r0 r0 r1  
.....
```

optimization

```
load r0 [0]  
add r0 r0 5  
store [0] r0  
mul r0 r0 r1  
.....
```

已經存在register上的value可以重複利用

# 補充 - optimization

```
x=(x+(y-  
(z*(x/(y%(z+(x-  
(y*(z/(x%(y+(z-  
(x*(y/(z%5)))))))  
))))))  
x=3
```

codegen

```
.....  
add r0 0 3  
store [0] r0
```

optimization

```
add r0 0 3  
store [0] r0
```

不影響最後結果的expression可以拿掉

```
z=z*(z-z)
```

```
x=y++  
x=3
```

```
x=y+10  
z=x+3  
x=3
```

# 補充 - optimization

```
x=(x+(y-  
(z*(x/(y%(z+(x-  
(y*(z/(x%(y+(z-  
(x*(y/(z%5)))))))  
))))))  
x=3
```

```
z=z*(z-z)
```

```
x=y++  
x=3
```

```
x=y+10  
z=x+3  
x=3
```

codegen

```
load r0 [8]  
load r1 [8]  
load r2 [8]  
sub r1 r1 r2  
.....
```

optimization

```
add r0 0 0  
store [8] r0
```

未知的variable也可以得到已知的結果

# 補充 - optimization

```
x=(x+(y-  
(z*(x/(y%(z+(x-  
(y*(z/(x%(y+(z-  
(x*(y/(z%5)))))))  
))))))  
x=3
```

codegen

```
load r0 [4]  
add r0 r0 1  
store [4] r0  
store [0] r0  
add r0 0 3  
store [0] r0
```

optimization 錯誤

```
z=z*(z-z)
```

```
x=y++  
x=3
```

```
add r0 0 3  
store [0] r0
```

看似不影響，但其實影響了y的值

```
x=y+10  
z=x+3  
x=3
```

# 補充 - optimization

```
x=(x+(y-  
(z*(x/(y%(z+(x-  
(y*(z/(x%(y+(z-  
(x*(y/(z%5)))))))  
))))))  
x=3
```

```
z=z*(z-z)
```

```
x=y++  
x=3
```

```
x=y+10  
z=x+3  
x=3
```

codegen

```
load r0 [4]  
add r0 r0 1  
store [4] r0  
store [0] r0  
add r0 0 3  
store [0] r0
```

optimization

```
load r0 [4]  
add r0 r0 1  
store [4] r0  
add r0 0 3  
store [0] r0
```

依然能優化，存值最後做就好  
如果中間還有用到x，務必要在register留下紀錄

# 補充 - optimization

```
x=(x+(y-  
(z*(x/(y%(z+(x-  
(y*(z/(x%(y+(z-  
(x*(y/(z%5)))))))  
))))))  
x=3
```

```
z=z*(z-z)
```

```
x=y++  
x=3
```

```
x=y+10  
z=x+3  
x=3
```

codegen

```
load r0 [4]  
add r0 r0 10  
store [0] r0  
load r0 [0]  
.....
```

optimization

```
load r0 [4]  
add r0 r0 10  
add r0 r0 3  
store [8] r0  
.....
```

中間用到了x，只能部分優化

# 補充 - optimization

```
x=(x+(y-  
(z*(x/(y%(z+(x-  
(y*(z/(x%(y+(z-  
(x*(y/(z%5)))))))  
))))))  
x=3
```

```
z=z*(z-z)
```

```
x=y++  
x=3
```

```
x=y+10  
z=x+3  
x=3
```

codegen

```
load r0 [4]  
add r0 r0 10  
store [0] r0  
load r0 [0]  
.....
```

optimization

```
load r0 [4]  
add r0 r0 10  
add r0 r0 3  
store [8] r0  
.....
```

optimization

```
load r0 [4]  
add r0 r0 13  
store [8] r0  
add r0 0 3  
store [0] r0
```

還可以更進一步多做一點



# 補充 - optimization

- ▶ 哪些可以省略? 哪裡可以縮短? reg用得太多怎麼處理?
- ▶ Compiler課程會講更多有關optimization的細節。
- ▶ Instruction數量減少與cycle減少之餘，最重要的是保持正確性。



Questions?