CS 135601
Introduction to Programming II

Mini project 1
Compiler: A Simple Calculator

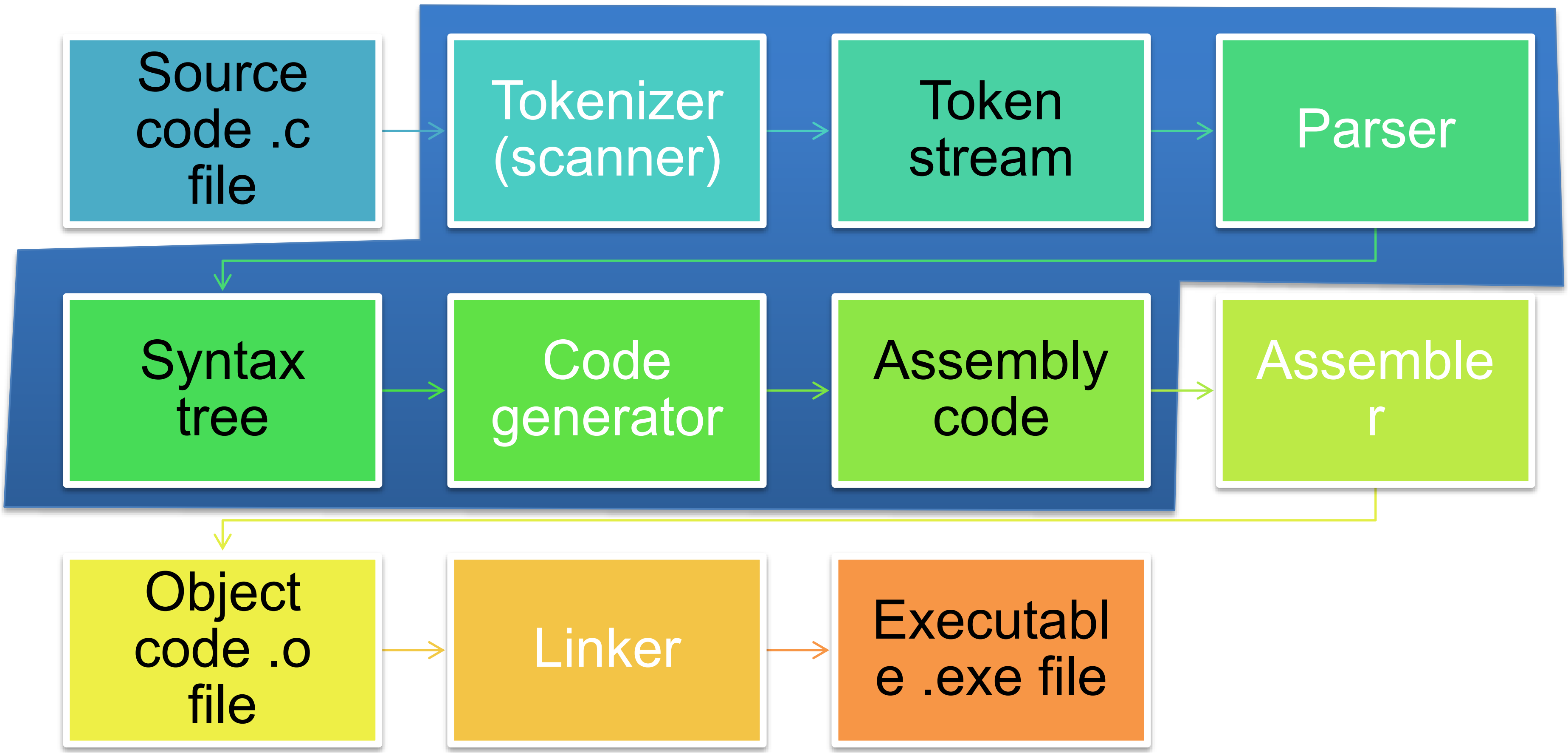
Po-Chih Kuo 郭柏志

Week	Date	Tuesday (13:20-15:10)	Friday (13:20-15:10)
1	2/23, 2/26		
2	3/2, 3/5		
3	3/9, 3/12		
4	3/16, 3/19		
5	3/23, 3/26		Midterm 1 review Mini-project 1: simple calculator
6	3/30, 4/2	Midterm I Project 1 materials released	Tomb Sweeping Day
7	4/6, 4/9	Intercollegiate Activities	
8	4/13, 4/16		
9	4/20, 4/23		
	4/24 (Sat)	Project 1 Deadline	
10	4/27, 4/30	Mini Project 1 DEMO	
11	5/4, 5/7		
12	5/11, 5/14		
13	5/18, 5/21		
	5/22 (Sat)		
14	5/25, 5/28		
	5/31 (Mon)		
15	6/1, 6/4		
16	6/8, 6/11		
17	6/15, 6/18		
18	6/22		
	6/29 (Tue)		

Compiler

- A compiler is computer software that transforms computer code written in one programming language (the source language) into another computer language (the target language).
- Example: C compiler
  - Source language: C language
  - Target language: Assembly code or intermediate code or machine code

Stages of compilation



Lexical Analysis (Tokenizer)

- Read an input string
- Identify symbols and words (Token)
  - Symbols include: `+ - * / = ( )`
  - Word includes numbers and variables。
- Each type of token is given a code.

```
typedef enum {UNKNOWN, END, INT, ID,
FLOAT, ADDSUB, MULDIV, ASSIGN, LPAREN,
RPAREN} TokenSet;
```

Example

- Input: a string of expression
  – If the expression is `20*(30000+x)`
- Output: a stream of tokens
  – Tokens are

20	*	(	3000	+	x	)
INT	MULDIV	LPAREN	INT	ADDSUB	ID	RPAREN

Scanner

- What is an integer (expressed in a string)?
  - INT: digits include 0 to 9
  - How about negative sign “-”?
- ~~What is a floating point?~~
  - ~~FLOAT: digits include 0 to 9 and a period~~

Not included in this project

- What is a variable?
  - Started with a alphabet, a-z, A_Z, or an underline_
  - Followed by a string of alphabets, number, underlines

Parser

- We have seen this before.

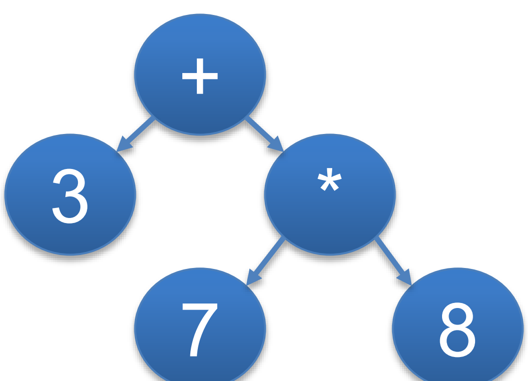
  FACTOR = ID | NUM | (EXPR)

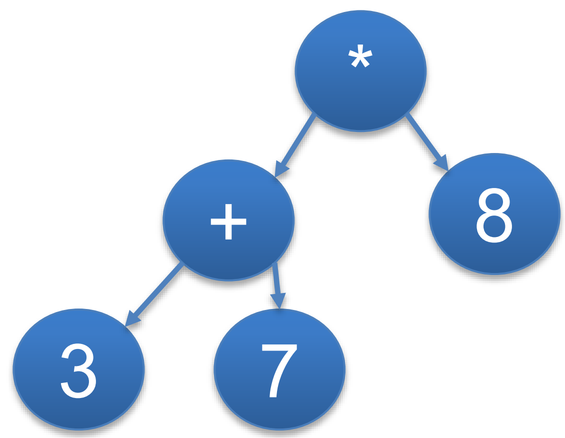
  EXPR = FACTOR | EXPR OP FACTOR

- Parsing **EXPR = FACTOR| EXPR OP FACTOR**
  1. Find a factor from the end of expression
  2. If there is an OP in front of the factor
  3. Let factor be OP's right child
  4. Parse the remaining expression recursively and make it OP's left child

Are we done yet?

- There are multiplication/division, whose priority is higher than add/sub. What is the grammar?
- Multiplication/division: Ex: 3+7*8 = 3+(7*8) not (3+7)*8





Handling mul/div

- We need to distinguish ADDSUB and MULDIV
  - If it is ADDSUB, we can use the previous grammar.
  - If it is MULDIV, we need to process it immediately, just like handing parenthesis “(expr)”.
- We need to add a new grammar rule (which means a new recursive function) to handle it.

FACTOR = ID | NUM| (EXPR)

TERM = FACTOR | TERM MULDIV FACTOR

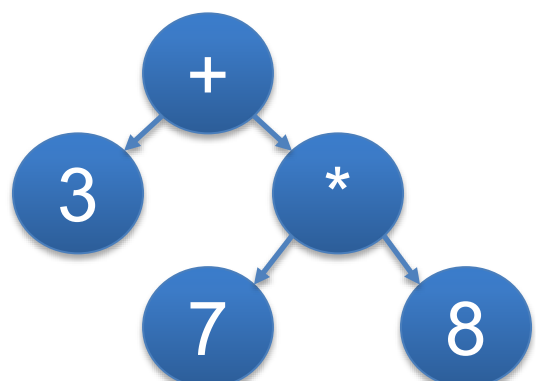
EXPR = TERM | EXPR ADDSUB TERM

Other considerations

- How to handle assignment?
  - Ex: $x = 3 + y$
- How to handle the sign of ID or NUM?
  - For example, $3 + -2$
- How to handle empty statements?
- How to handle errors?

Code Generation

- Generate assembly code from the syntax tree
- Tasks of code generation include:
  - Instruction arrangement
  - Register and memory management
- Instruction arrangement: using **post-order** traversal
  - Ex: 3+7*8




```
MOV r0, 3
MOV r1, 7
MOV r2, 8
MUL r1, r2
ADD r0, r1
```

(Post-order: left subtree, right subtree, and root)

Symbol Table

- Recall the relation between variables, memory, registers, and value.
- A symbol table which indexed by the variable is usually use to map variables to memory, register, and value, and to track the types of variables
  - Use array of struct for implementation

Symbol name	Type
bar	function, double
x	double
foo	function, double
count	int
sum	double
i	int


COMPILER ISSUES
COMPILER ISSUES EVERYWHERE
makeameme.org

Mini-project Goal

- A calculator to demonstrate how does a compiler work.
- Input: a list of expressions.
- Output: corresponding assembly codes.
- Example

```
>> x = 3
>> y = -5
>> z = 4*x + y*-6
```

```
MOV r1, 3
MOV r2, -5
MOV r0, 4
MUL r1, r0
MOV r0, -6
MUL r2, r0
ADD r1, r2
```

What will we provide?

- main.c
- lex.h / lex.c
- parser.h/parser.c
- codeGen.h/codeGen.c

D:\Course\HWCode\calculator\bin\Debug\calculator.exe

```
>> 1+2+3
6
Prefix traversal: + + 1 2 3
>> x+3
3
Prefix traversal: + x 3
>> x=y+3
3
Prefix traversal: = x + y 3
>> x
3
Prefix traversal: x
>> y
0
Prefix traversal: y
>> x = 1 + 2 +3
6
Prefix traversal: = x + + 1 2 3
>> x = 1 + 2 +3 +
error() called at C:\Users\Bruce\Dropbox\NTHU\I2P2\MiniProject1\package\calcul
ator_recursion\parser.c:123: error: number or identifier expected

Process returned 0 (0x0)   execution time : 74.324 s
Press any key to continue.
```

Lexical analysis (lex.h / lex.c)

- Tokenizer "**`getToken()`**": a function that
  - extracts the next token from the input string;
  - stores the token in "`char lexeme[MAXLEN]`";
  - identifies the token's type
- `typedef enum {UNKNOWN, END, INT, ~~FLOAT,~~ ID, ADDSUB, MULDIV, ASSIGN, LPAREN, RPAREN, ENDFILE} TokenSet;`

The parser (parser.h/parser.c)

- The parsing process:
  - group tokens into statements based on a set of rules, collectively called a grammar.
- The grammars:

```
□ statement    := ENDFILE | END | expr END
□ expr         := term expr_tail
□ expr_tail    := ADDSUB term expr_tail | NiL
□ term         := factor term_tail
□ term_tail    := MULDIV factor term_tail| NiL
□ factor       := INT | ADDSUB INT |
                  ID  | ADDSUB ID  |
                  ID ASSIGN expr |
                  LPAREN expr RPAREN |
                  ADDSUB LPAREN expr RPAREN
```

Tail Recursion -> Loop (1/2)

Recursion Version

```
BTNode* expr(void)
{
    BTNode *node;
    node = term();
    return expr_tail(node);
}
BTNode* expr_tail(BTNode *left)
{
    BTNode *node;
    if (match(ADDSUB)) {
        node = makeNode(ADDSUB, getLexeme());
        advance();
        node->left = left;
        node->right = term();
        return expr_tail(node);
    }
    else
        return left;
}
```

Loop Version

```
BTNode* expr(void)
{
    BTNode *retp, *left;
    //int retval;
    retp = left = term();
    while (match(ADDSUB)) {
        retp = makeNode(ADDSUB, getLexeme());
        advance();
        retp->right = term();
        retp->left = left;
        left = retp;
    }
    return retp;
}
```

Tail Recursion -> Loop (2/2)

Recursion Version

```
BTNode* term(void)
{
   BTNode *node;
   node = factor();
   return term_tail(node);
}
BTNode* term_tail(BTNode *left)
{
   BTNode *node;
   if (match(MULDIV)) {
      node = makeNode(MULDIV, getLexeme());
      advance();
      node->left = left;
      node->right = factor();
      return term_tail(node);
   }
   else
      return left;
}
```

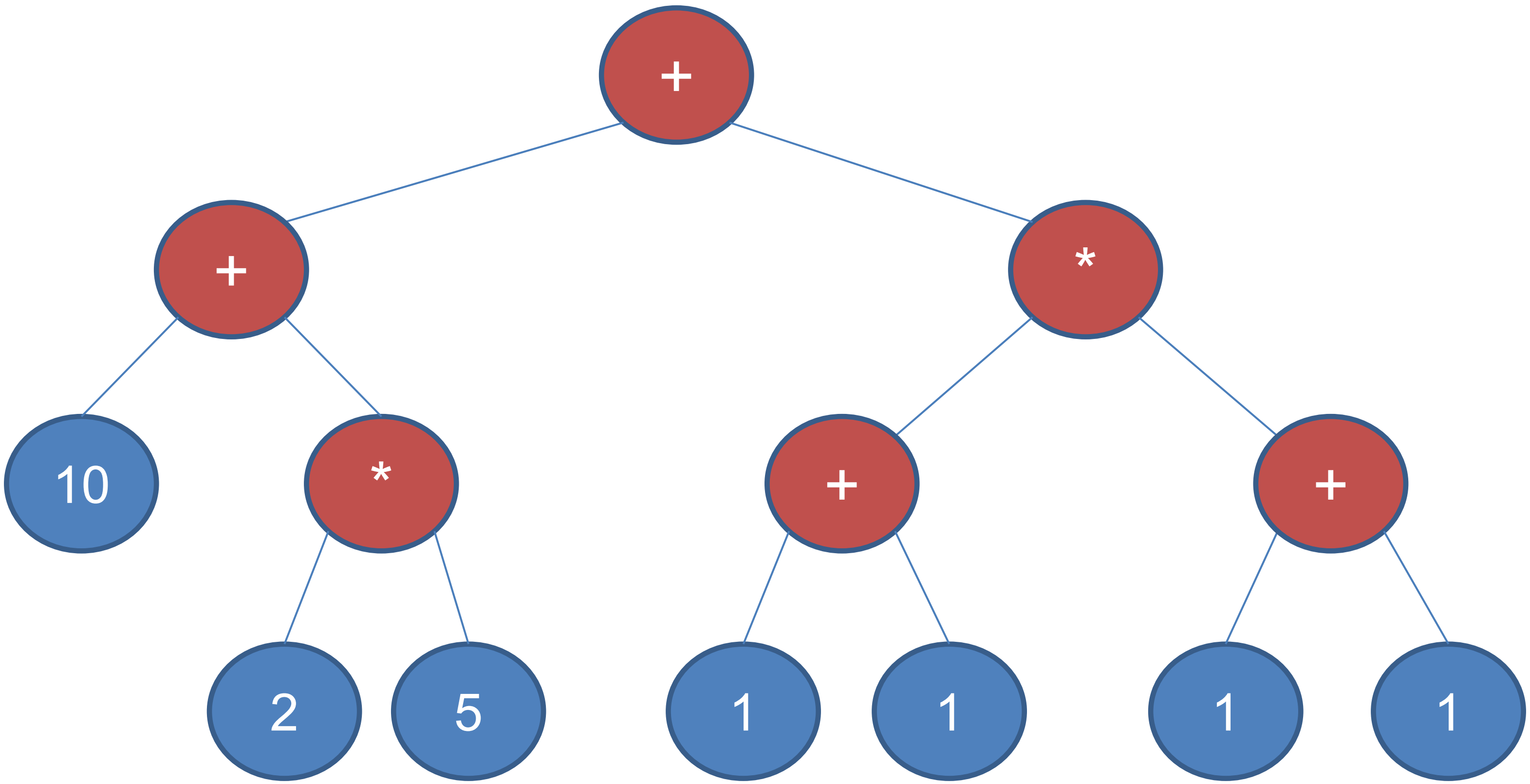
Loop Version

```
BTNode* term(void)
{
   BTNode *retp, *left;
   retp = left = factor();

   while (match(MULDIV)) {
      retp = makeNode(MULDIV, getLexeme());
      advance();
      retp->right = factor();
      retp->left = left;
      left = retp;
   }
   return retp;
}
```

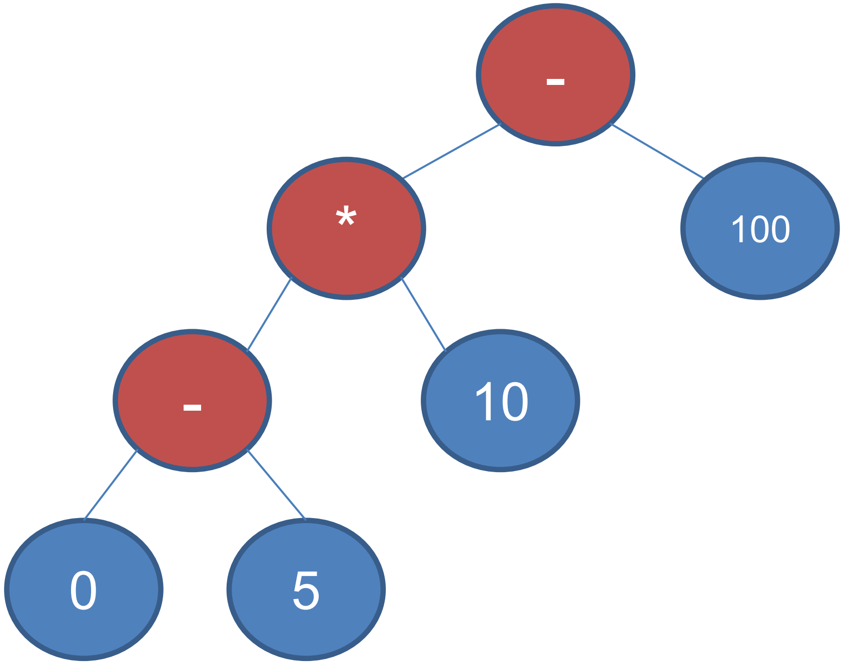
The syntax trees based on the grammars

10+2*5+(1+1)*(1+1)



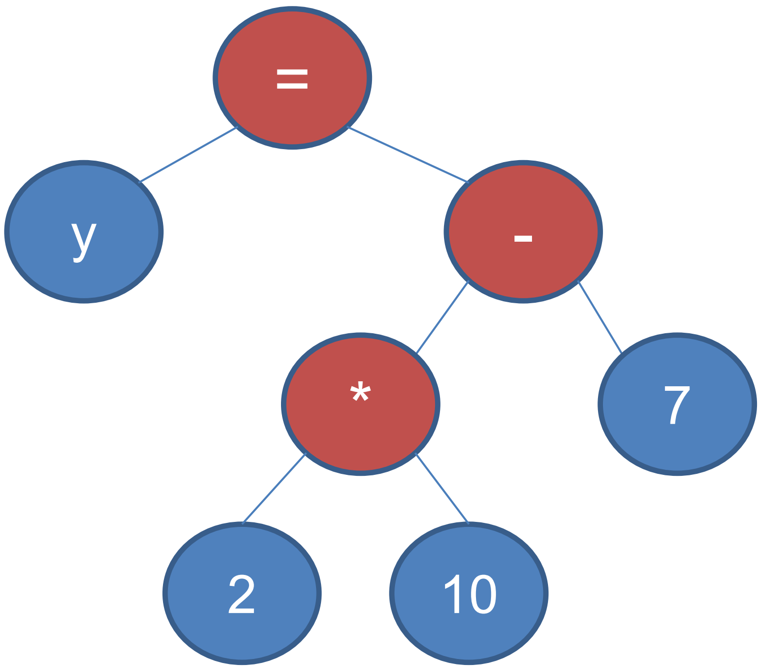
The syntax trees based on the grammars

-5*10-100



The syntax trees based on the grammars

y=2*10-7



The code generator (codeGen.h/codeGen.c)

- The code generation process:
  - constructing the machine-language instructions to implement the statements recognized by the parser and represented as syntax trees
- In codeGen.h/codeGen.c, we provide a function: evaluateTree(BTNode *root)
  - that calculates the answer by pre-order traversal of the syntax tree

Symbol table

- Note that, to deal with expressions including variables such as

  `>> x=3`

  `>> 4*x+(y=((10-2)/4))`

  – we use a symbol table to record variables' current values

  – we use getval() and setval() to manipulate the symbol table

- The symbol table are used both in the parser and the code generator.

Symbol table

```
#define TBLSIZE 65535
typedef struct {
    char name[MAXLEN];
    int val;
} Symbol;
Symbol table[TBLSIZE];
```

What you need to do in this mini-project?

1. Trace all the code if you like to
   – understand each detail and
   – review what you have learnt during the courses.
2. Modify the grammar and parser.c to accept new operators, including **&, |, ^, ++, --.**
3. Replace the current evaluateTree () in codeGen.c to generate assembly code during the pre-order tree traversal.
4. Check the error handling in the parser.c to print "EXIT 1" when encountering errors.
5. Optimize your assembly code to get extra credits!