NDHU, CSIE Dept., 2021 Spring

Professor Chung Yung

Programming Languages and Compilers

Programming Assignment #1

- Suggested Steps
- Programming Environment
- Basics of FLEX
- Basics of Bison
- A Practical Example

HW1.1

Suggested Steps

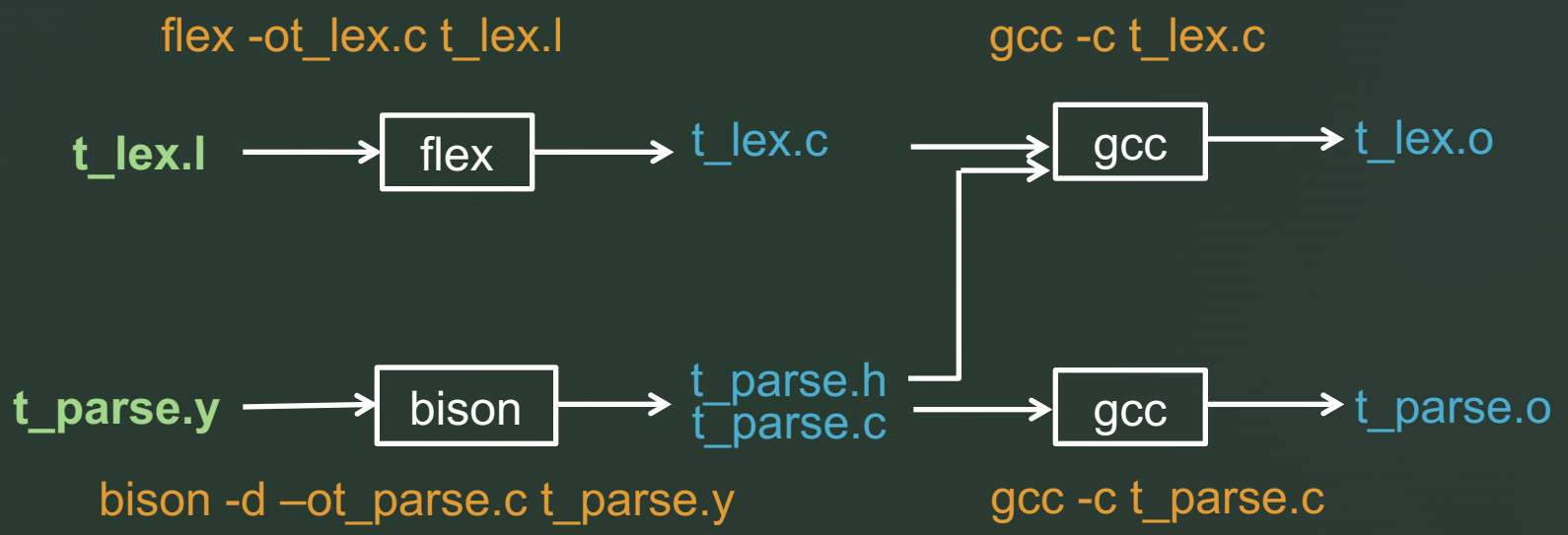
What to do
(for T Example)

- Write a lex specification **t_lex.l**.
- Write a yacc specification **t_parse.y**.
- Write a main function **t2c.c**.
- Write a header file **t2c.h**.

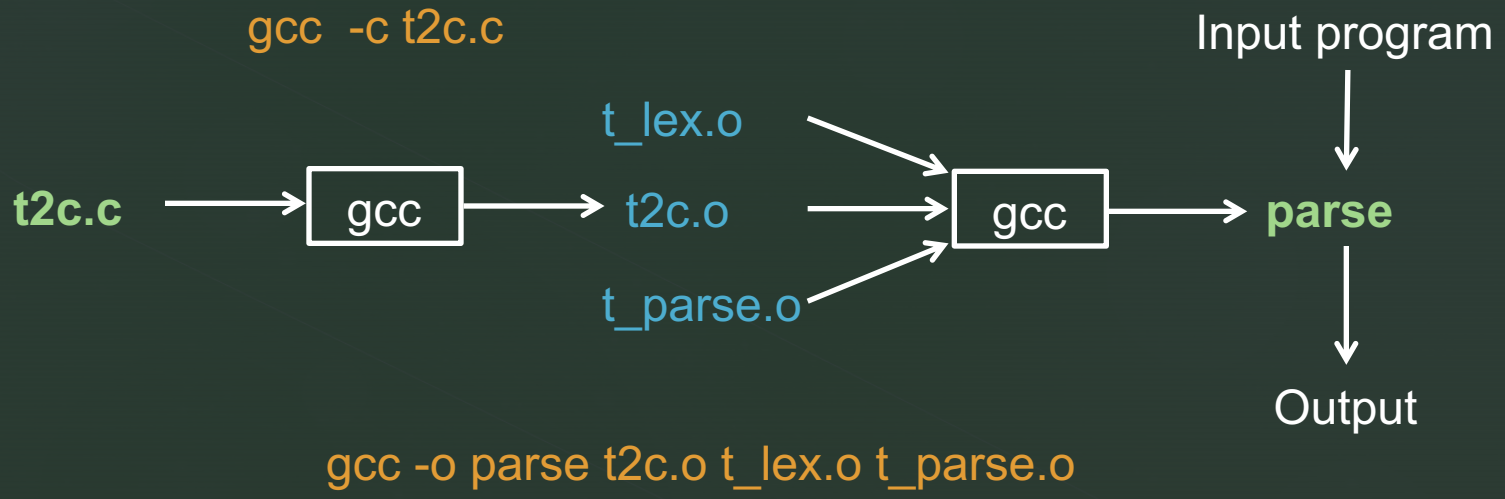
Suggested Steps

1. Use **bison** to compile **t_parse.y** into t_parse.c. The **-d** switch produces a header file t_parse.h.
2. Use **flex** to compile **t_lex.l** into t_lex.c.
3. Use **gcc** to compile t_lex.c into t_lex.o, t_parse.c into t_parse.o, and **t2c.c** into t2c.o.
4. Use gcc to link t2c.o, t_lex.o, and t_parse.o into **parse**.

Generating C Code



Building Executable



HW1.2

Programming Environment

(for Windows)

Software

Software needed

- Flex 2.5.4
- Bison 2.4.1
- MinGW 4.6.2

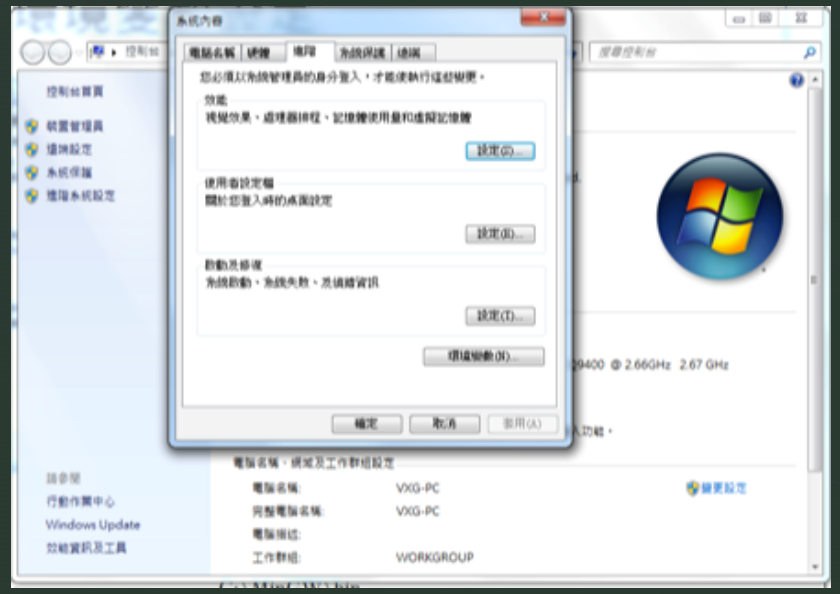
Notes

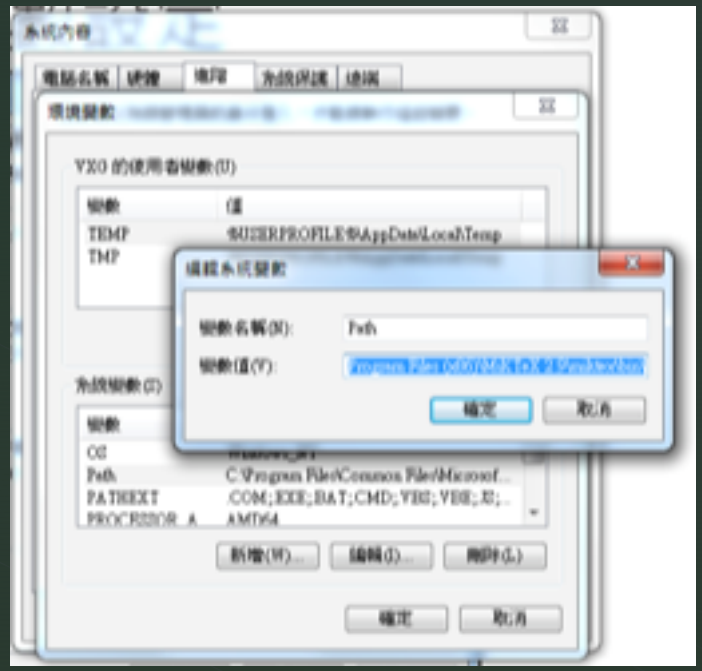
- DO NOT install both MinGW and DevC.
- Check and set up environment variables.
- Rebooting after installation is suggested.

Environment Variables

- Check whether `C:\MinGW\bin` is in `PATH`. If not, add it into `PATH`, using ; as separators.
- In case of using `DevC`, check for `C:\Dev-Cpp\bin` instead.

On Windows



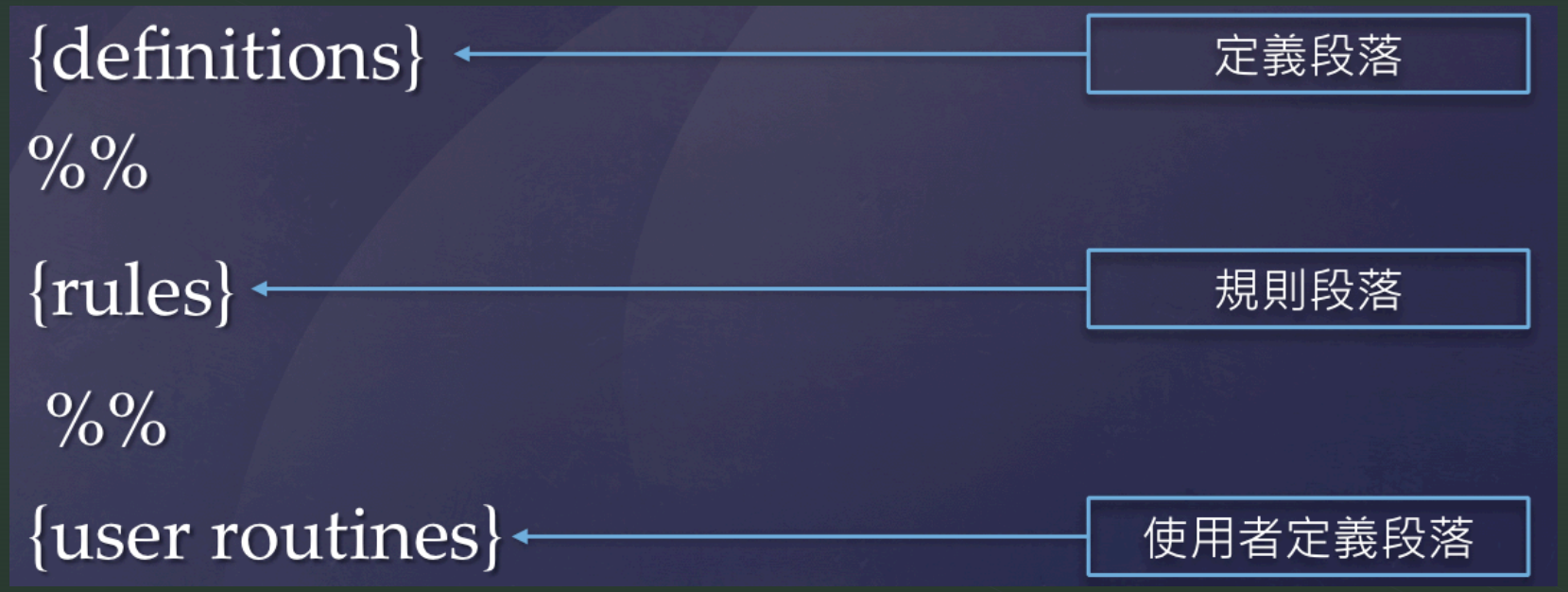


Test Installation

HW1.3

Basics of Flex

Flex Specification



Flex Special Characters

Char	Meaning
.	Any character except for '\n'
-	The range of characters, e.g., a-z
*	Repeat for 0 or more times
+	Repeat for 1 or more times
?	Appear either 0 or 1 time
^	Negation
A | B	A or B
"sth."	Exactly the characters appeared in parentheses (**`sth.`**)

Flex Compilation

- `flex -ot_lex.c t_lex.l`
- Translate `t_lex.l` into `t_lex.c`.
  (Specify the filename of your own.)
- (With old versions)
  There is NO space between `-o` and `t_lex.c`.

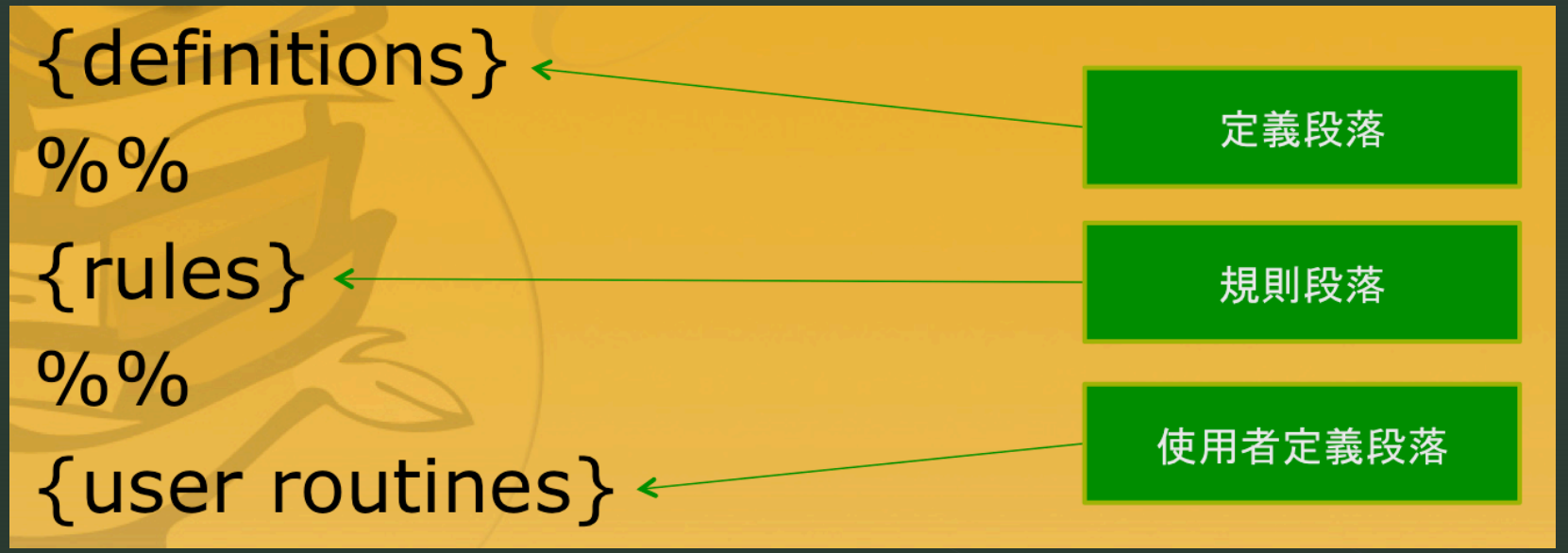
For Your Assignment

- Write your own **t_lex.l** according to the description of T lexicons.
- `flex -ot_lex.c t_lex.l`

HW1.4

Basics of Bison

A Bison Specification



(Pascal) Example: Definitions

```
1  %{
2      #include <stdio.h>
3      #include <stdlib.h>
4      #include <string.h>
5      #include "pascal.h"
6  %}
7  %token PROG PROC BG END INT BOOL TRUE
8  %token LP RP LSP RSP FALSE CC
9  %token DOT SEMI VAR ARRAY OF DOTDOT
10 %token IF THEN READ WRITE WHILE DO
11 %token ELSE ASSIGN COMMA COLON ID NUM
12 %left OR AND
13 %left NOT
14 %left EQ NE LT GT LE GE
15 %left ADD MINUS
16 %left DIV TIMES
17 %expect 1
18
19 %%
```

定義段落

(Pascal) Example: Rules

```
19  %%
20  prog    :   PROG ID SEMI block DOT {printf("prog => PROG ID SEMI block DOT \n********
21          |   {printf("********Parsing failed! \n");}
22          ;
23
24  block   :   vardecs prodecs stmts {printf("block=>vardecs prodecs stmts \n");}
25          ;
26
27  vardecs :   VAR vardec SEMI morevd {printf("vardecs => VAR vardec SEMI morevd \n");}
28          |   {printf("vardecs => Null \n");}
29          ;
30
31  morevd  :   vardec SEMI morevd  {printf("morevd => vardec SEMI morevd \n");}
32          |   {printf("morevd => Null \n");}
33          ;
34
35  vardec  :   ID moreid COLON type {printf("vardec => ID moreid COLON type\n");}
36          ;
37
38  moreid  :   COMMA ID moreid {printf("moreid =>  COMMA ID moreid \n");}
39          |   {printf("moreid => Null \n");}
40          ;
41
```

規則段落

HW1.5

A Practical Example

Commands Used

- Bison
  bison -d -o t_parse.c t_parse.y
- Flex
  flex -ot_lex.c t_lex.l
- GCC
  gcc -c t_lex.c
  gcc -c t_yacc.c
  gcc -c t2c.c
  gcc -o parse t_lex.o t2c.o t_parse.o

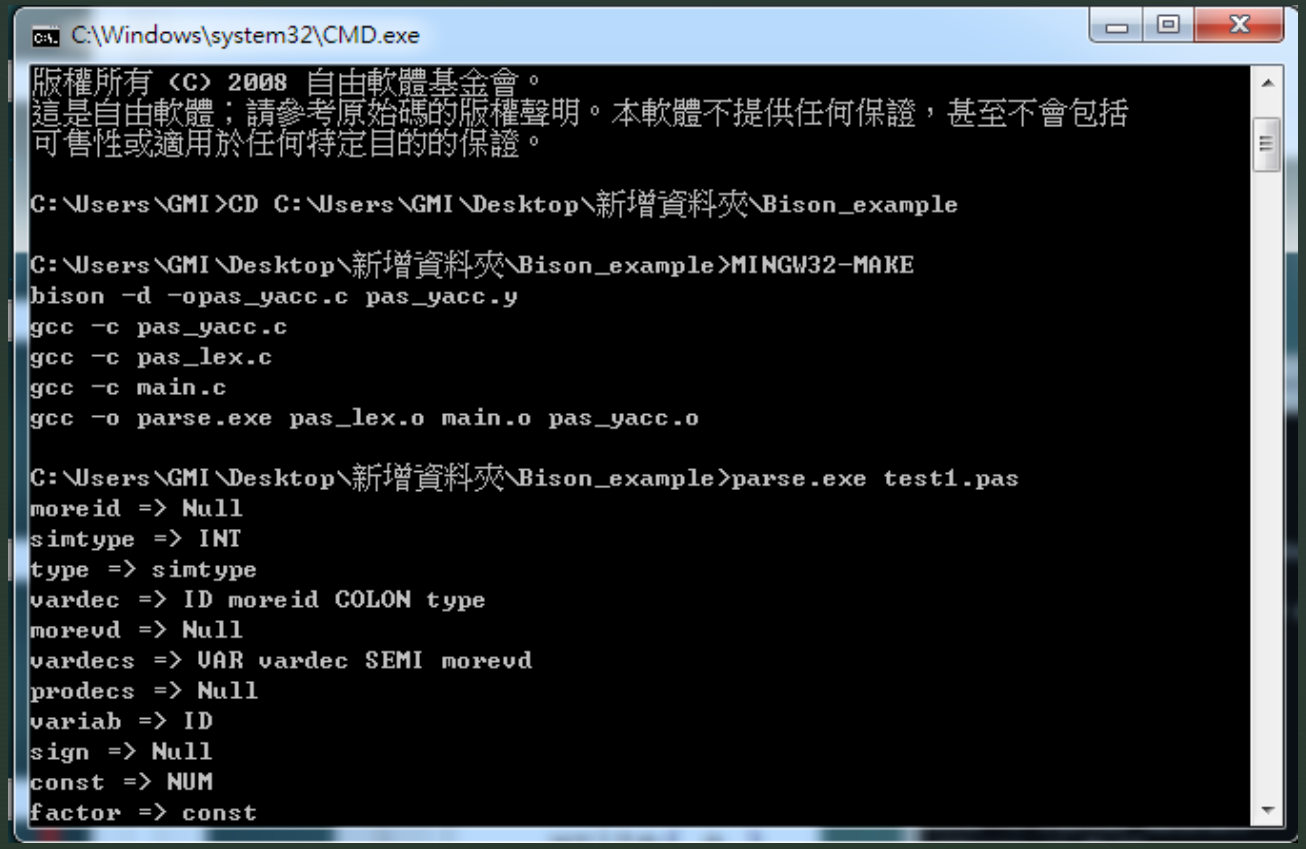
Execution

- To run your parser:
  ./parse test.t

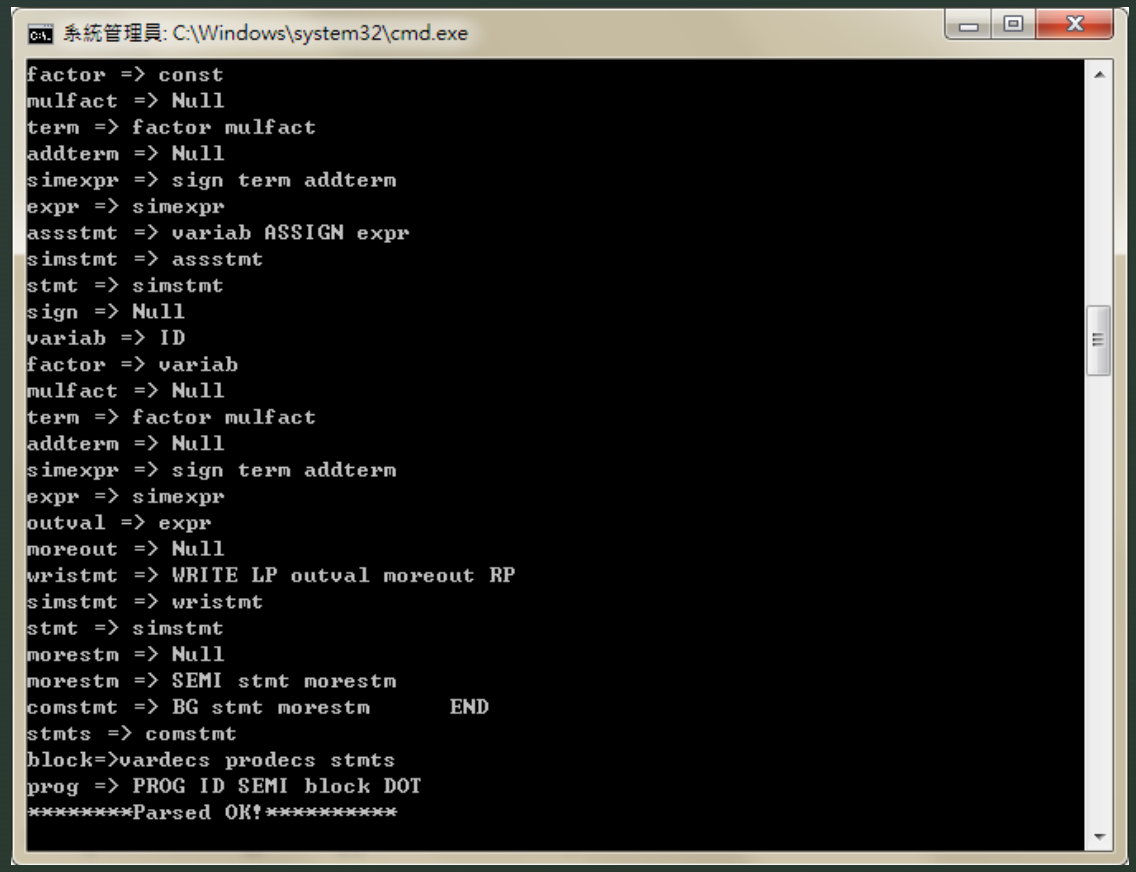
Summary:

- Write a **t_parse.y** according to the T grammar.
- Build **parse** and run
  `./parse test.t`

Building Executable
(with Pascal Example)



Executing Your Program
(with Pascal Example)




```
1  program test1;
2  var a: integer;
3  begin
4      a := 3;
5      write( a )
6  end.
```

References

- http://www.mingw.org/
- http://flex.sourceforge.net/
- http://www.gnu.org/software/bison/
- http://sourceforge.net/projects/gnuwin32

HW1.6

Reference Files

What to do?

Use lex (or flex) and yacc (or bison) to implement a front end (including a lexical analyzer and a syntax recognizer) of the compiler for the *j* programming language, which is a simplified version of Java and specially designed as a compiler construction project by Professor Chung Yung.

- See an attached file for the lexical rules and grammar rules in details.
- You are requested to separate the C code, the Lex specification, the Yacc specification into separated files.

OK, So What Exactly to Do?

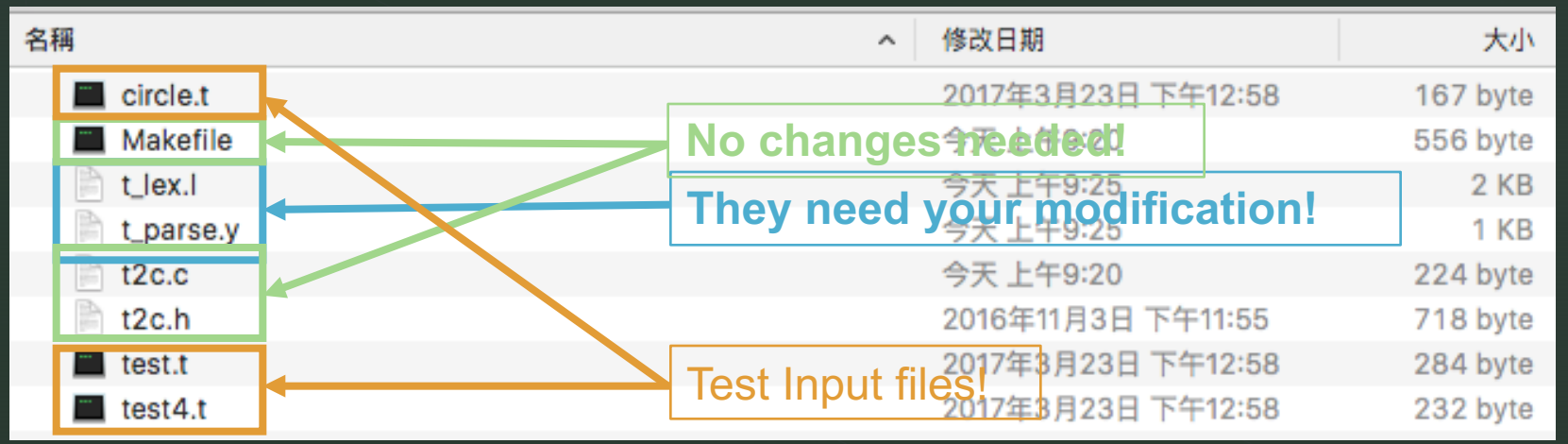
By design, the program is splitted into 4 files:

- **t_lex.l**
- **t_parse.y**
- t2c.c
- t2c.h

- Can I write a program in 4 files, if I am not sure that I may write a program in a single file?
- How about starting with partially-done programs?

www.elearn.ndhu.edu.tw

unzip the zip file



t_lex.l

```
%{
#include "t2c.h"
#include "t_parse.h"
%}

%x C_COMMENT

ID  [A-Za-z][A-Za-z0-9]*
DIG [0-9][0-9]*
RNUM {DIG}'.'{DIG}
NQUO [^"]

%%

WRITE           {return lWRITE;}
READ            {return lREAD;}
IF              {return lIF;}
ELSE            {return lELSE;}
RETURN          {return lRETURN;}
BEGIN           {return lBEGIN;}
END             {return lEND;}
MAIN            {return lMAIN;}
INT             {return lINT;}
REAL            {return lREAL;}
";"             {return lSEMI;}
","             {return lCOMMA;}
"("             {return lLP;}
")"             {return lRP;}
"+"             {return lADD;}
"-"             {return lMINUS;}
"*"             {return lTIMES;}
"/"             {return lDIVIDE;}
">"             {return lGT;}
"<"             {return lLT;}
":="            {return lASSIGN;}
"=="            {return lEQU;}
"!="            {return lNEQ;}
">="            {return lGE;}
"<="            {return lLE;}

// Identifiers, Integer numbers, Real numbers

\"{NQUO}*\"     {sscanf(yytext,"%s", qstr); return lQSTR;}
"/*"            { BEGIN(C_COMMENT); }
<C_COMMENT>"*/" { BEGIN(INITIAL); }
<C_COMMENT>\n   { }
<C_COMMENT>.    { }
[ \t\n]         {}
.               {}

%%
```

Understand the rules and write them here!

The T Lexicons

Keywords (All keywords are reserved. Each keyword can be a terminal.):

`WRITE READ IF ELSE RETURN BEGIN END MAIN INT REAL`

Single-character separators (Each operator can be a terminal.):

`; , ( )`

Single-character operators (Each operator can be a terminal.):

`+ - * / > <`

Multi-character operators (Each operator can be a terminal.):

`:= == != >= <=`

Identifiers:

An *identifier* consists of a letter followed by any number of latters or digits.

Integer numbers:

An *integer number* is a sequence of digits, where a *digit* has the following definition:

`Digit -> '0' | '1' |'2' |'3' |'4' |'5' |'6' |'7' |'8' |'9'`

Real numbers:

A *real number* is a sequence of digits followed by a dot, and followed by digits.

Comments:

A *comment* is a string between `/*` and `*/`. Comments can be longer than one line.

QStrings:

A *QString* is any sequence of characters except double quote itself, enclosed in double quotes.

t_parse.y (part 1)

```
%{
        #include <stdio.h>
        #include <stdlib.h>
        #include "t2c.h"
        #include "t_parse.h"
%}

%token lWRITE lREAD lIF lASSIGN
%token lRETURN lBEGIN lEND
%left  lEQU lNEQ lGT lLT lGE lLE
%left  lADD lMINUS
%left  lTIMES lDIVIDE
%token lLP lRP
%token lINT lREAL lSTRING
%token lELSE
%token lMAIN
%token lSEMI lCOMMA
%token lID lINUM lRNUM lQSTR

%expect 1

%%
```

Declare your tokens and delete others

No need to change!

t_parse.y (part 2)

```
prog    :       mthdcls
                { printf("Program -> MethodDecls\n");
                  printf("Parsed OK!\n"); }
        |
                { printf("****** Parsing failed!\n"); }
        ;

mthdcls :       mthdcl mthdcls
                { printf("MethodDecls -> MethodDecl MethodDecls\n"); }
        |       mthdcl
                { printf("MethodDecls -> MethodDecl\n"); }
        ;

type    :       lINT
                { printf("Type -> INT\n"); }
        |       lREAL
                { printf("Type -> REAL\n"); }
        ;

mthdcl  :       type lMAIN lID lLP formals lRP block
                { printf("MethodDecl -> Type MAIN ID LP Formals RP Block\n"); }
        |       type lID lLP formals lRP block
                { printf("MethodDecl -> Type ID LP Formals RP Block\n"); }
        ;

formals :       formal oformal
                { printf("Formals -> Formal OtherFormals\n"); }
        |
                { printf("Formals -> \n"); }
        ;

formal  :       type lID
                { printf("Formal -> Type ID\n"); }
        ;

oformal :       lCOMMA formal oformal
                { printf("OtherFormals -> COMMA Formal OtherFormals\n"); }
        |
                { printf("OtherFormals -> \n"); }
        ;
```

High-level program structures:

```
Program -> MethodDecl MethodDecl*
Type -> INT | REAL
MethodDecl -> Type [MAIN] Id '(' FormalParams ')' Block
FormalParams -> [FormalParam (',' FormalParam)*]
FormalParam -> Type Id
```

t_parse.y (part 3)

Understand the rules and write them here!

```
// Statements and Expressions

%%

int yyerror(char *s)
{
        printf("%s\n",s);
        return 1;
}
```

No need to change!

Statements:

```
Block -> BEGIN Statement+ End

Statement -> Block
           | LocalVarDecl
           | AssignStmt
           | ReturnStmt
           | IfStmt
           | WriteStmt
           | ReadStmt

LocalVarDecl -> Type Id ';' | Type AssignStmt

AssignStmt -> Id := Expression ';'

ReturnStmt -> RETURN Expression ';'

IfStmt -> IF '(' BoolExpression ')' Statement
        | IF '(' BoolExpression ')' Statement ELSE Statement

WriteStmt -> WRITE '(' Expression ',' QString ')' ';'

ReadStmt -> READ '(' Id ',' QString ')' ';'
```

Expressions:

```
Expression -> MultiplicativeExpr ( ('+' | '-') MultiplicativeExpr )*

MultiplicativeExpr -> PrimaryExpr ( ('*' | '/') PrimaryExpr )*

PrimaryExpr -> Num // Integer or Real numbers
             | Id
             | '(' Expression ')'
             | Id '(' ActualParams ')'

BoolExpr -> Expression '==' Expression
          | Expression '!=' Expression
          | Expression '>' Expression
          | Expression '>=' Expression
          | Expression '<' Expression
          | Expression '<=' Expression

ActualParams -> [Expression (',' Expression)*]
```

That’s all.

Please start now!