Compiler Project

102_Spring_Compiler Construction

Compiler Project

- Target: Compiler for C
- Deadline: 6/18(三) 23:59
- DEMO: 6/19(四)~6/25(三)(announce later)
- Upload your file to moodle
  - A zipped file(.rar, .zip, .7z, …) contain your source code and readme
  - Filename: Student1ID_Student2ID
  - Ex: F74996081_F74992163.rar
- Post your problems on moodle
- Fill in your teammate on Google Drive

Compiler Project

- Environment: Linux + GNU Compiler(**C** 、 **C++**)
- SSH Server: 140.116.246.199
- Username: Student_ID(Ex. F74996081)
- Password: 2014cpr
- Software: pietty(Windows)、 ssh(linux)
- The program must be compiled and executed on server. If we can not grade due to failure of execution, it is your own responsibility.

Compiler Project

- No 3rd party library
  - Ex. Flex、 Bison
- You should use DFA in lexical analysis.
  - Hint: Switch-Case, State

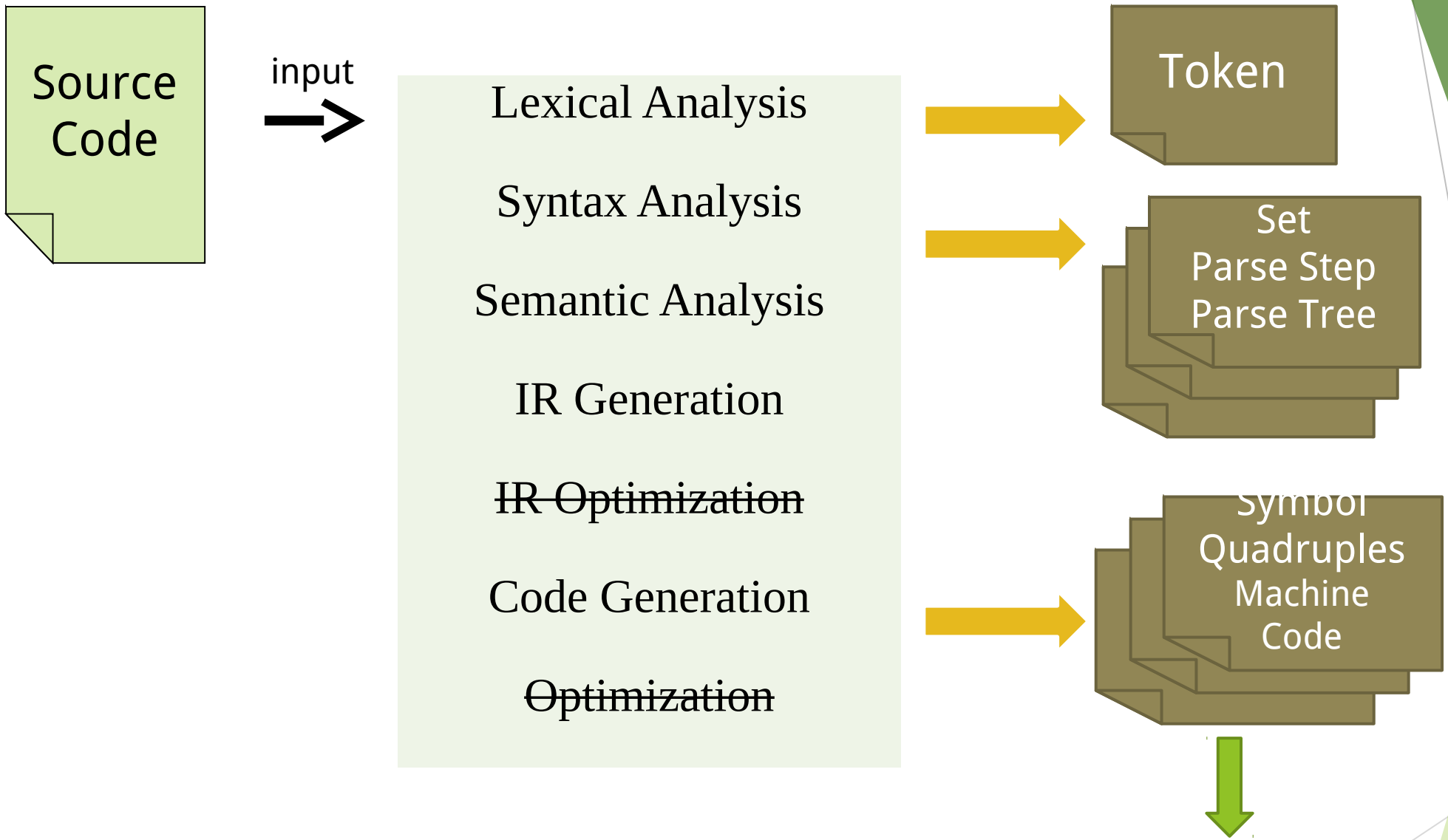
Compiler Project

- Input file:
  - main.c
  - grammar.txt
- Output file:
  - token.txt
  - set.txt
  - LLtable.txt
  - tree.txt
  - symbol.txt
  - quadruples.txt
  - code.tm

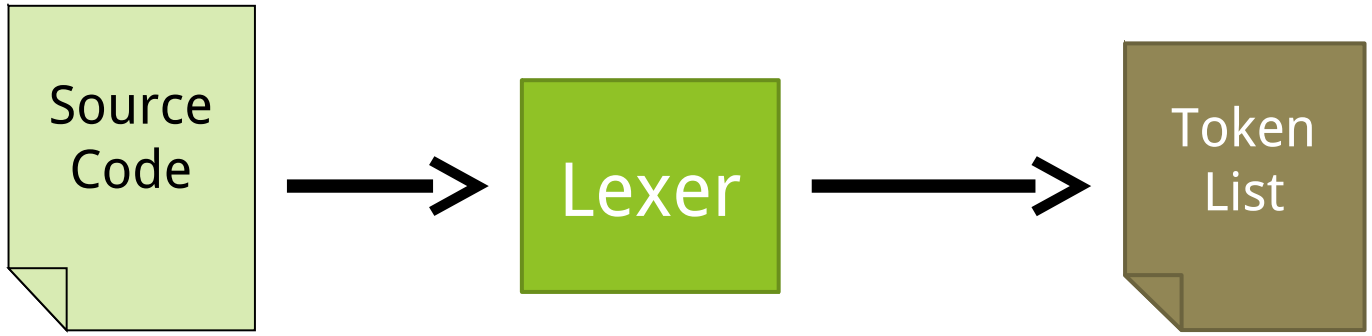
Program -> VarDeclList | FunDeclList
VarDeclList -> VarDecl VarDeclList | ε
...

grammar.txt

```
Program
    VarDeclList
    FunDeclList
VarDeclList
    VarDecl VarDeclList
    epsilon
VarDecl
    Type id ;
    Type id [ num ] ;
FunDeclList
    FunDecl
    Fundecl FundeclList
FunDecl
    Type id ( ParamDecList ) Block
```


Source Code
input
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
~~IR Optimization~~
Code Generation
~~Optimization~~
Token
Set
Parse Step
Parse Tree
Symbol
Quadruples
Machine
Code
TM

Lexical Analyzer(Lexer)



Token

- Kerwords
  - int char return if else while break
- Operators
  - = ! + - * / == != < > <= >= && ||
- Special Symbols
  - [ ] ( ) { } ; ,
- Identifier
  - [a-zA-Z_][a-zA-Z0-9_]*
- Number
  - [0-9]+
- Char
  - ‘[.|\\n|\\t|]’
  - Ex. ‘a’ 、’ \n’ 、’ ’
- Comment(no need to print)
  - //

Lexer Output

- Output file format
  - Line xx :
  - xx – Line number
  - <yyy> : zzz
  - yyy – Category(Keyword、 Operator、 Special Symbol、 Identifier、 Char、 Number、 Error)
  - zzz – token
  - Ex.
  - Line 1 :
  - <Keyword> : int
  - <Identifier> : main
  - <Special> : (
  - <Special> :)

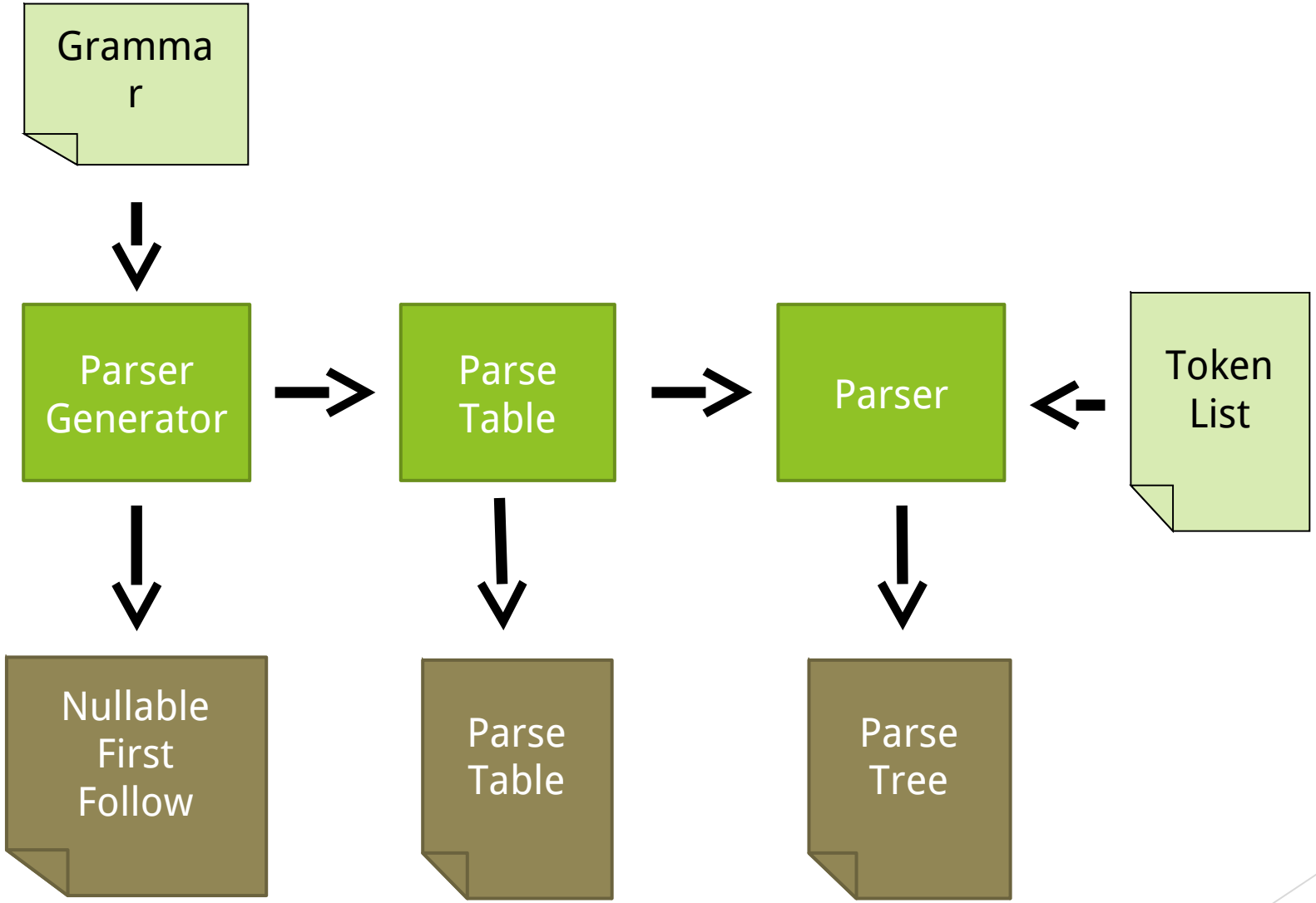
Lexer Output Example

```
1  int main ( ) {
2      int i1 = 3 ; // Comment
3      char c = 'a' ;
4      int 3asd ;
5
6      if ( )
7      else
8  }
```

token.txt

```
Line  1:
        <Keyword>    : int
        <Identifier> : main
        <Special>    : (
        <Special>    : )
        <Special>    : {
Line  2:
        <Keyword>    : int
        <Identifier> : i1
        <Operator>   : =
        <Number>     : 3
        <Special>    : ;
Line  3:
        <Keyword>    : char
        <Identifier> : c
        <Operator>   : =
        <Char>       : 'a'
        <Special>    : ;
Line  4:
        <Keyword>    : int
        <Error>      : 3asd
        <Special>    : ;
```

Syntax Analyzer



Parser Output Example 1

Display all nonterminal

- Nullable
- First
- Follow

in set.txt

Format - symbol : value

- Nullable : true or false
- First & Follow : Separate every symbol with space

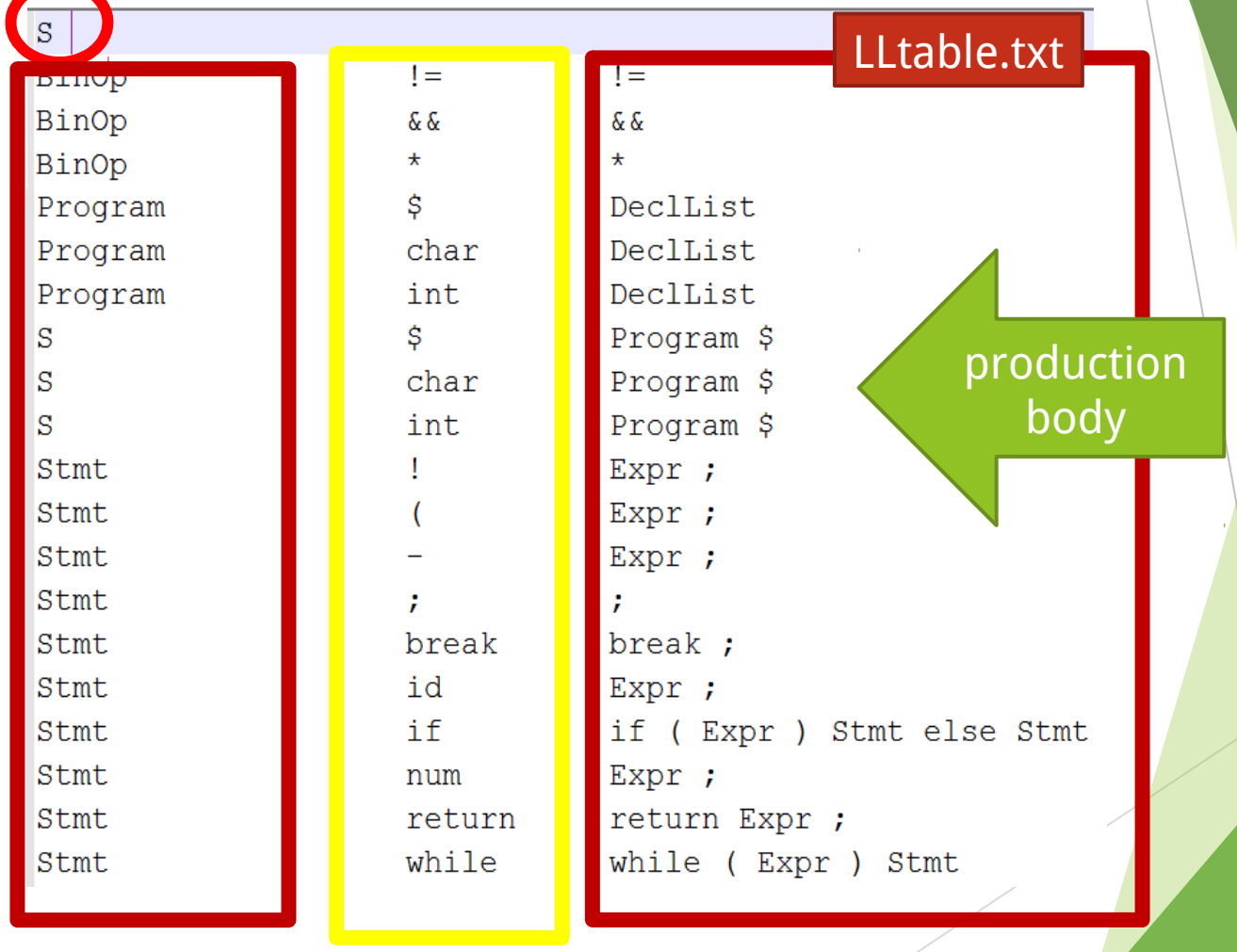
set.txt

```
Nullable
BinOp           : false
Block           : false
DeclFun         : false
DeclList        : true
.
.
.

First
BinOp           : != && * + - / < <= == > >= ||
Block           : {
DeclFun         : (
DeclList        : char int
.
.
.

Follow
BinOp           : ! ( - id num
Block           : ! $ ( - ; break char else id if int num return
while { }
DeclFun         : $ char int
DeclList        : $
.
.
.
```

Parser Output Example 2

1. print your start symbol in first line.
2. print
   - nonterminal
   - terminal
   - prdouction body

   in a line



Parser Output Example 3

```
1  int x ;
2  char y ;
3
4  int double ( int value ) {
5    return value * 2 ;
6  }
7
8  char c ;
9
10 int main ( ) {
11    int z ;
12
13    z = double ( x ) ;
14 }
```

Display parse tree

tree.txt

```
1 Program
  2 DeclList
    3 Type
      4 int
    3 id
      4 x
    3 DeclList'
      4 DeclVar
        5 ;
    3 DeclList
      4 Type
        5 char
      4 id
        5 y
      4 DeclList'
        5 DeclVar
          6 ;
      4 DeclList
        5 Type
          6 int
        5 id
          6 double
        5 DeclList'
          6 DeclFun
            7 (
            7 ParamDeclList
              8 ParamDeclListTail
```

ID、 Num、 Char display value

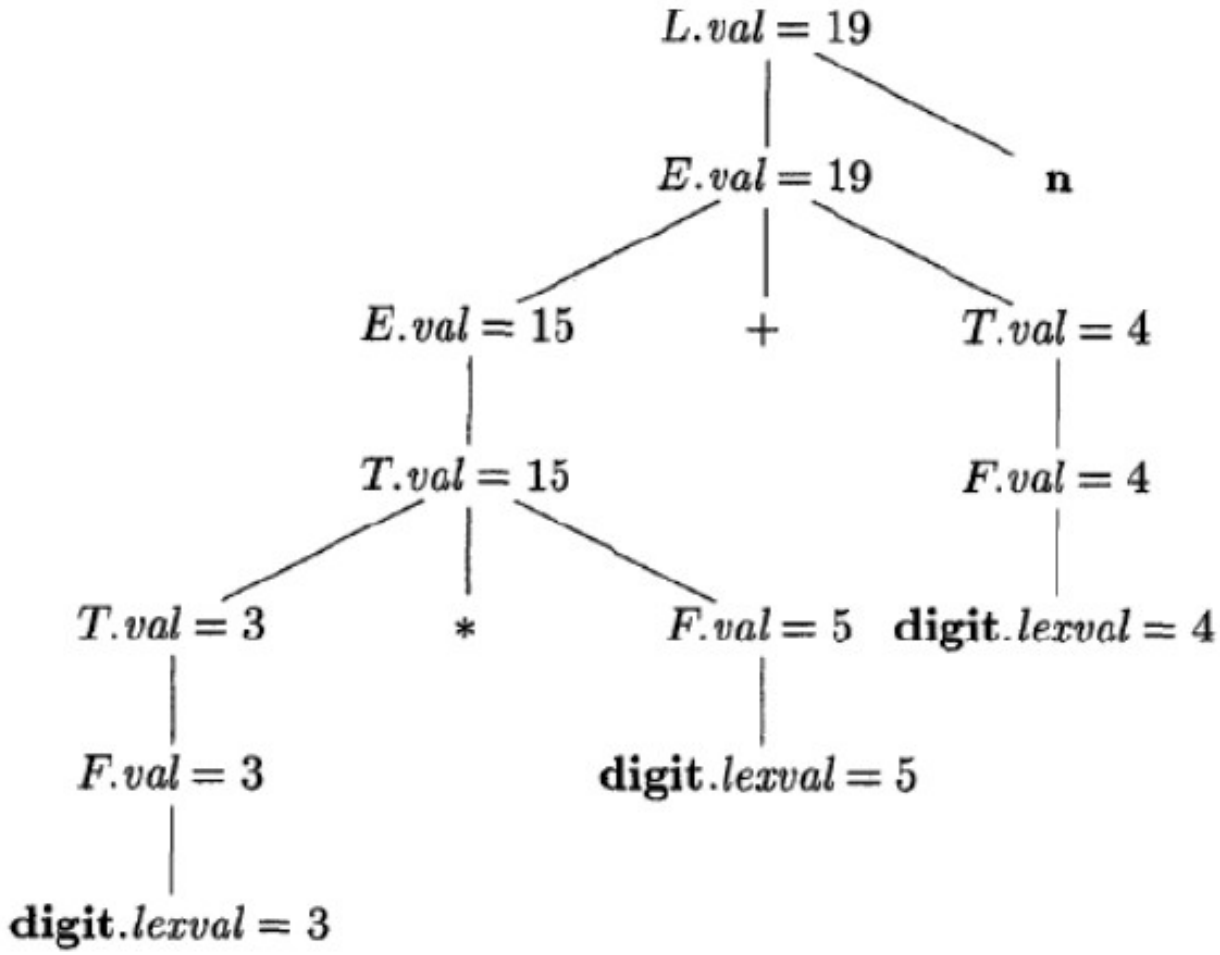
Machine Code Generator

- 1st: Build Semantic Rules
- 2nd: Build Single Symbol Table (symbol.txt)
- 3rd: Generate Three Address Code
- 4th: Generate Quadruples (quadruples.txt)
- 5th: Generate Machine Code (Tiny Machine Simulator Code) (code.tm)

semantic rule

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

3*5+4n



Machine Code Generator

- Target -> 3 output files
  - symbol.txt : Symbol Table
  - quadruples.txt : Quadruples
  - code.tm : “Machine Code”
  - Machine Code that Tiny Machine can read & run
- Semantic Rules NOT Provided
  - Write by yourself

Symbol Table

```
1   int x ;
2   char y ;
3
4   int double ( int value ) {
5     return value * 2 ;
6   }
7
8   char c ;
9
10  int main ( ) {
11     int z ;
12
13     z = double ( x ) ;
14  }
```

Symbol	Token	Type	Scope
x	id	int	0
y	id	char	0
double	id	int	0
value	id	int	1
c	id	char	0
main	id	int	0
z	id	int	2

Quadruples

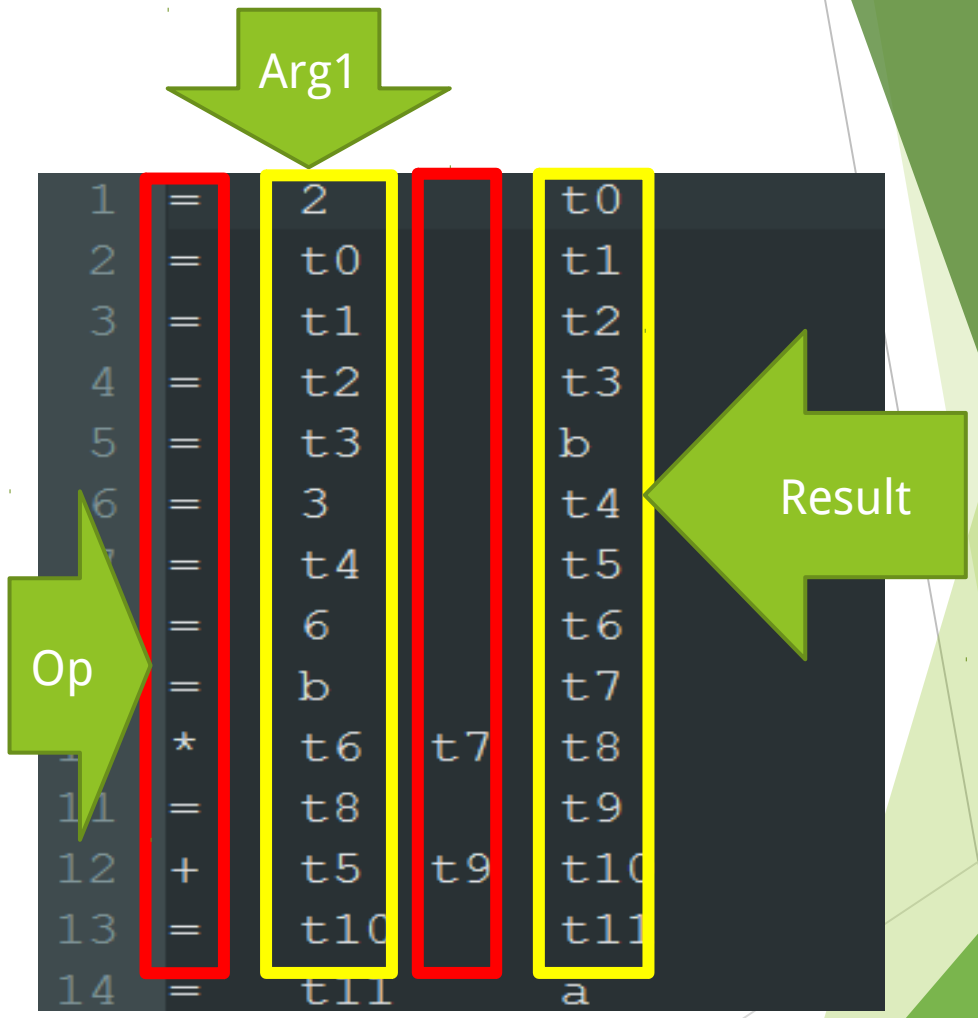
- Op Arg1 Arg2 Result

- no need to consider function call.
- your program will start at int main(){

  ……

  ……

  }



Machine Code

```
1  int main ( ) {
2      int a ;
3      int b ;
4
5      a = 2 ;
6      b = 6 * b ;
7  }
```

```
0:  LDC 1,2,0
1:  ST 1,5(0)
2:  LD 1,5(0)
3:  ST 1,6(0)
```

```
14:  LDC 1,6,0
15:  ST 1,12(0)
16:  LD 1,9(0)
17:  ST 1,13(0)
18:  LD 1,12(0)
19:  LD 2,13(0)
20:  MUL 1,1,2
21:  ST 1,14(0)
```

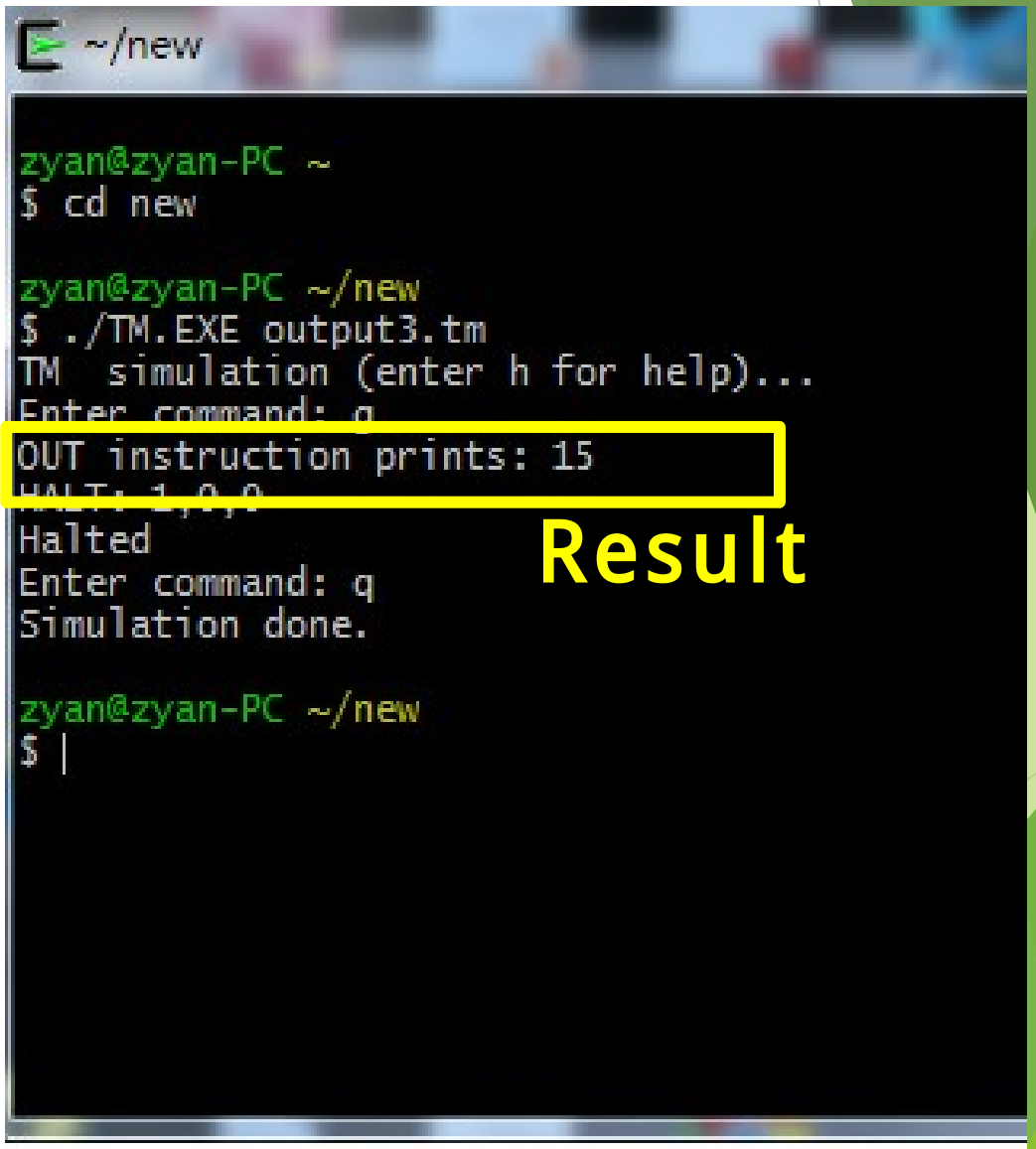
```
32:  OUT 1,0,0
33:  HALT 1,0,0
```

Two things before end

1. Add “OUT 1,0,0” at the last to output the result.
2. Add “HALT 1,0,0” to end the TM.

TM - Simulation

- ./TM.EXE <file.tm>
- “g” for execute
- “q” for quit



TM - Code

RO (register only) Instructions	
Format:	*opcode r, s, t*
Opcode	Effect
HALT	stop execution (operands ignored)
IN	reg[*r*] ← integer value read from the standard input (*s* and *t* ignored)
OUT	reg[*r*] → the standard output (*s* and *t* ignored)
ADD	reg[*r*] = reg[*s*] + reg[*t*]
SUB	reg[*r*] = reg[*s*] - reg[*t*]
MUL	reg[*r*] = reg[*s*] * reg[*t*]
DIV	reg[*r*] = reg[*s*] / reg[*t*]
LDC	**reg[*r*]** = ***d*** (load constant ***d*** directly into ***r***. /***s*** is ignored*/)

TM - Code

RM (register-memory) Instructions	
Format:	*opcode r, d(s)*
Opcode	Effect
LD	**reg[*r*]** = **dMem[a]** (load ***r*** with memory value at ***a***)
LDA	**reg[*r*]** = ***a*** (load address ***a*** directly into ***r***)
ST	**dMem[*a*]** = **reg[*r*]** (store value in ***r*** to memory location ***a***)
JLT	if (reg[r] < 0) reg[PC_REG] = a /*jump to instruction a if r is negative, similarly for the following*/
JLE	if (reg[r] <= 0) reg[PC_REG] = a
JGE	if (reg[r] >= 0) reg[PC_REG] = a
JGT	if (reg[r] > 0) reg[PC_REG] = a
JEQ	if (reg[r] == 0) reg[PC_REG] = a
JNE	if (reg[r] != 0) reg[PC_REG] = a

TM - Others

- The TM has only 8 registers. reg[0] ~ reg[7].
- reg[7] = PC_REG.

Example:	
LDA 7, d(s)	This instruction has the effect of jumping to location ***a = d + reg[s]***.

Example 2:	
The conditional jump instructions (JLT, etc.) can be made relative to the current position in the program by using the pc as the second register.	
JEQ 0, 4(7)	Causes the TM to jump five instructions forward in the code if register 0 is 0.
LDA 7, -4(7)	Performs an unconditional jump three instructions backward.

Grade

- Every single output file take 10%
  - Token List
  - Set(Nullable, First, Follow)
  - LL Table
  - Parse Tree
  - Symbol
  - Quadruples
  - Machine Code
- Readme – 10%
- Coding Style – 5%
- Demo – 15%

Q&A