Group Project Part 3

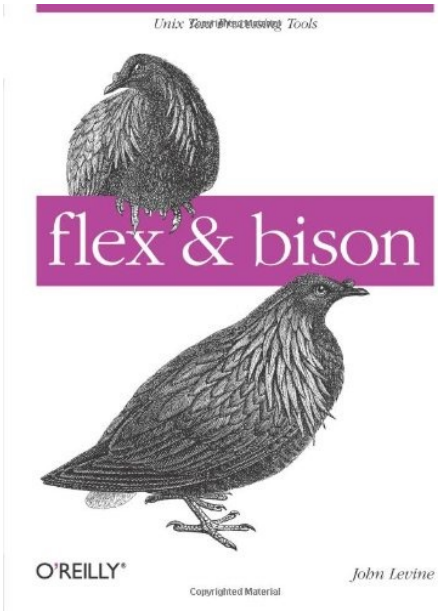
CS 3323 - Spring 2024

About Midterm 2 **Wednesday, 04/03/2024**

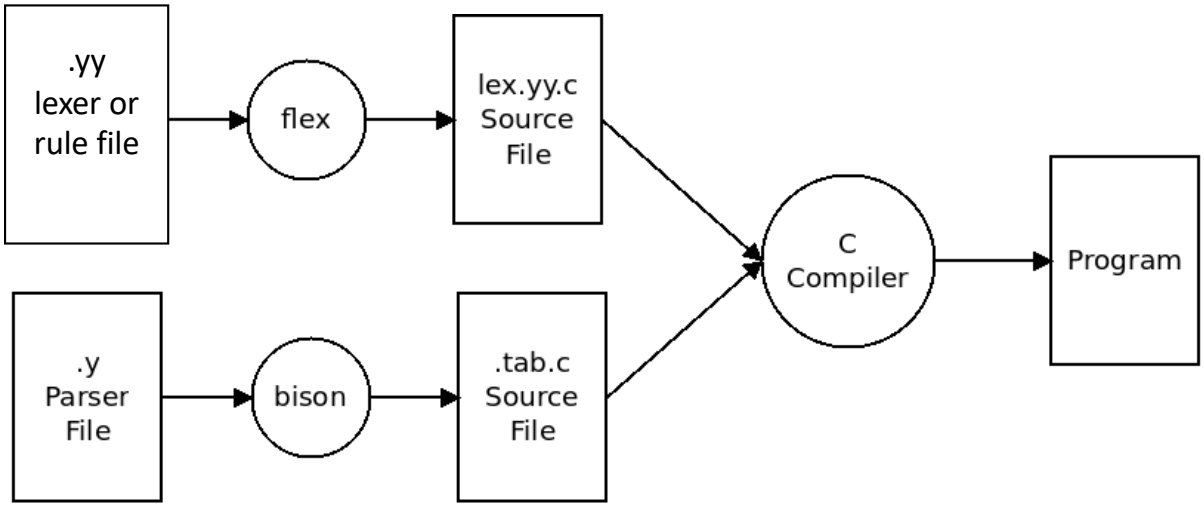
- **Midterm Score = max(Midterm 1, Midterm2)**
- **Topics:**
  - Programming Language History
  - Compilation Overview
  - *NIX Crash Course: Basic UNIX/LINUX command
  - Lexical Analysis (including regular expression, remember how it is used in Group Project Part 1)
  - Syntactic Analysis
- **Open notes/cheat sheets/books but only hard copies/papers.**
- Bring your laptop, yet during the exam **you can only open the midterm Canvas page**.
  - **No other apps or websites are allowed**

Group Project: what we have done so far (Part 1 and Part 2)

Compiler-Compiler Tools.



Using Flex and Bison to build a compiler frontend.



Picture modified from https://ianfinlayson.net/class/cpsc401/notes/08-bison

Group Project Part 3

- **We will:**
  - Build a functional compiler with basic I/O support and arithmetic computations.
- **TODO:**
  1. Download all files under **Files/Group Project/Part3** from Canvas
  2. Read the instructions described in the PDF file
  3. Compile and run the original (unmodified) code first (in local or remote machine)
  4. Read the instructions described in the PDF file
  5. Modify grammar.y and icode.cc according to the requirements (you have **1.5 weeks** to work in group)
  6. Anytime you modify the files, always compile and run it again
  7. Submit the modified **grammar.y** and **icode.cc** file **by 04/08/2024 11:59 pm** (one submission per group), please ensure no compilation error (see #5)

Compile and Run Project Part 3

❖ **Option 1 use a gpel machine (remote machine)**

1. Download and copy all Part 2 files to a gpel# machine, # = [8-13]
2. cd to the directory where the files are located
3. To compile, type and run: `make`
4. To run, type: `./simple.exe < input-file.in`

❖ **Option 2 use your local machine**

✓ Make sure Flex and Bison are installed on your machine (if you use Windows, install it on the **Windows Subsystem for Linux (WSL)**)

✓ Open terminal and type: `which flex`
`which bison`

✓ No bison installed?

- Linux/WSL, open terminal and type:
  - sudo apt-get update
  - sudo apt -y install flex (if not already done so)
  - sudo apt -y install bison
- macOS
  - You can use **MacPorts** **or** **Homebrew** to install bison.
  - Need to install MacPorts? https://www.macports.org/install.php
  - **Or** Homebrew? https://brew.sh
  - After installing MacPorts or Homebrew, open the terminal, type, and run:
    - `sudo port install bison`

    or

    - `brew install bison`

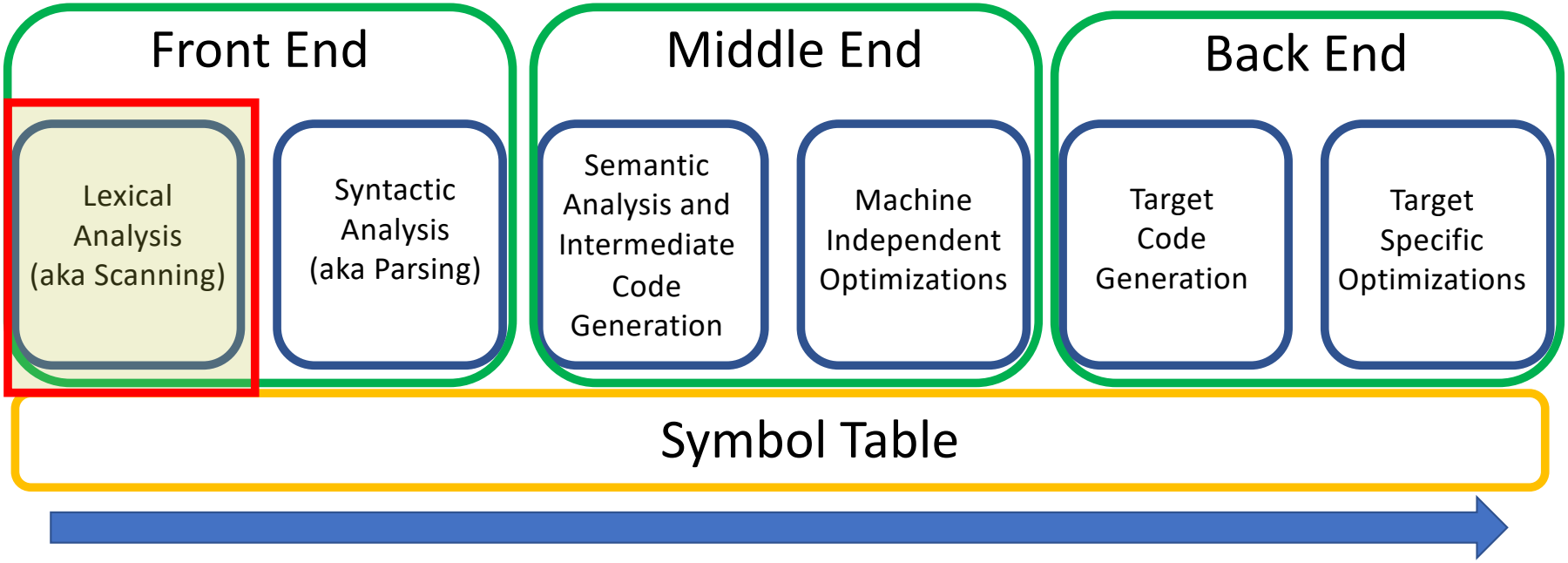
✓ Download all Part 3 files and follow Steps 2-4 described in Option 1

Group Project Part 3

```
CMD>ls
driver.cc   icode.cc   Makefile     run-smp.sh   symtab.cc
grammar.y   icode.hh   run-all.sh   scanner.yy   symtab.hh
```

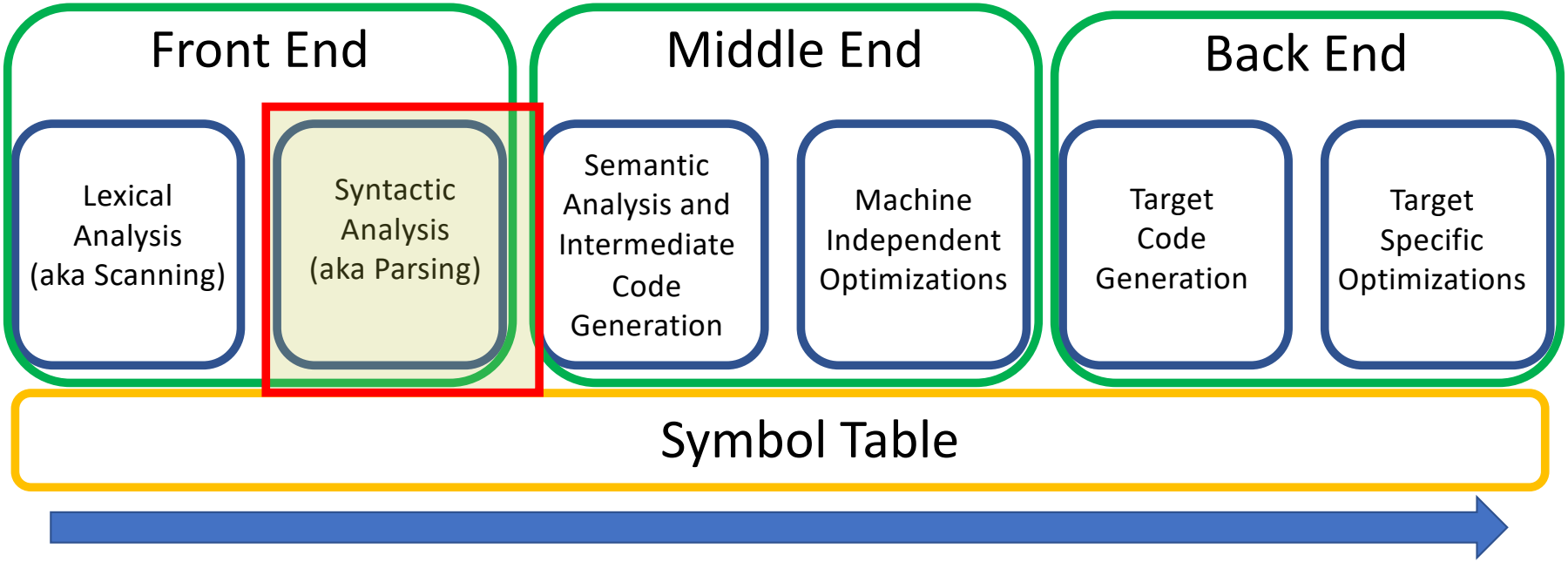
Group Project Part 3

```
CMD>ls
driver.cc   icode.cc   Makefile     run-smp.sh   symtab.cc
grammar.y   icode.hh   run-all.sh   scanner.yy   symtab.hh
```



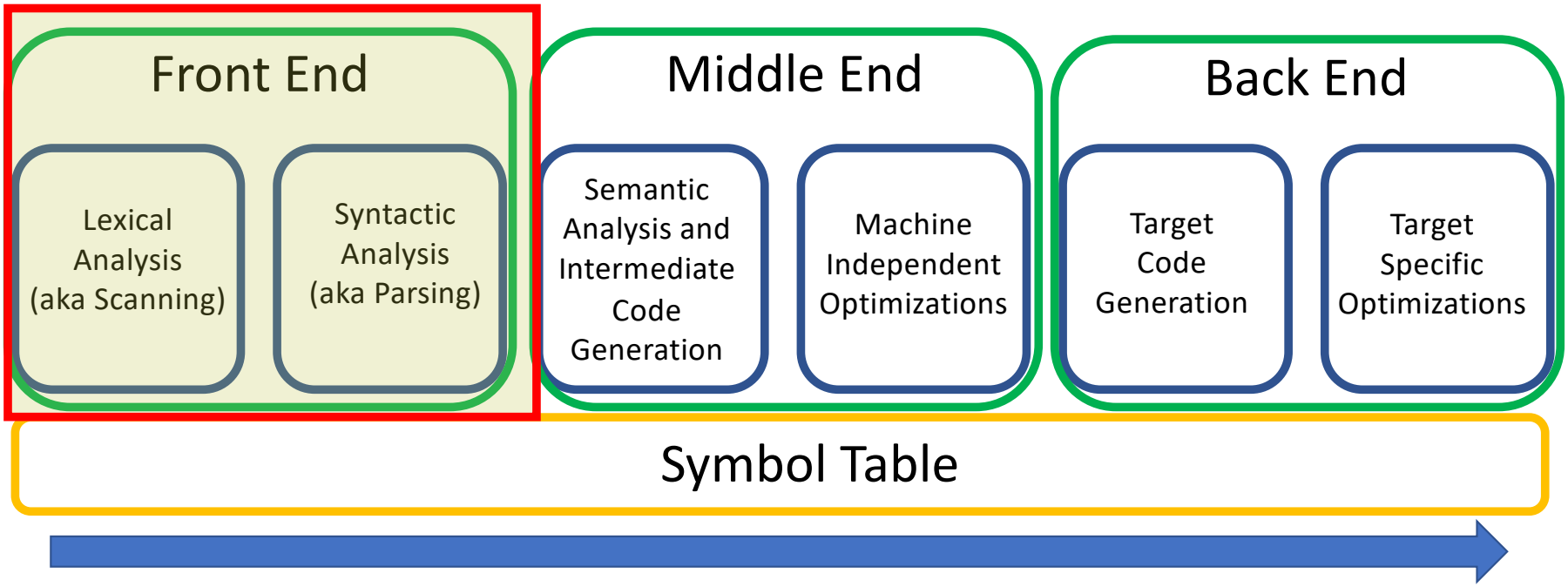
Group Project Part 3

```
CMD>ls
driver.cc   icode.cc   Makefile     run-smp.sh   symtab.cc
grammar.y   icode.hh   run-all.sh   scanner.yy   symtab.hh
```



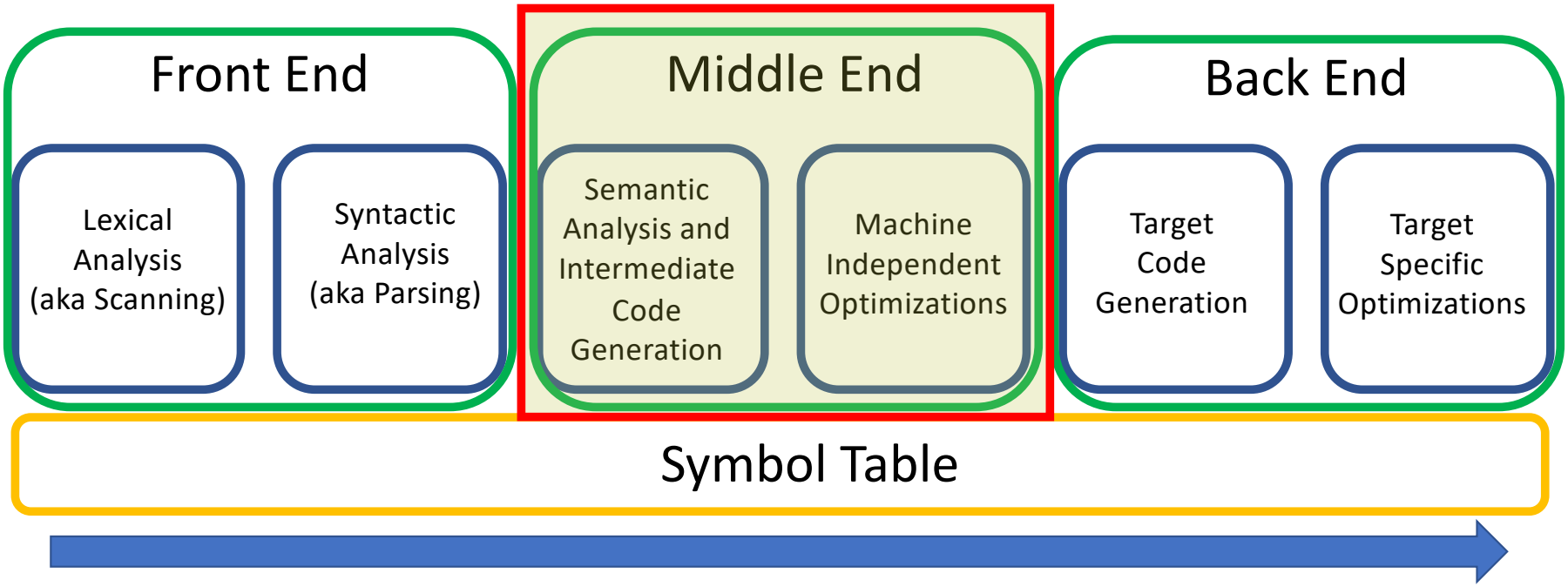
Group Project Part 3

```
CMD>ls
driver.cc   icode.cc   Makefile     run-smp.sh   symtab.cc
grammar.y   icode.hh   run-all.sh   scanner.yy   symtab.hh
```



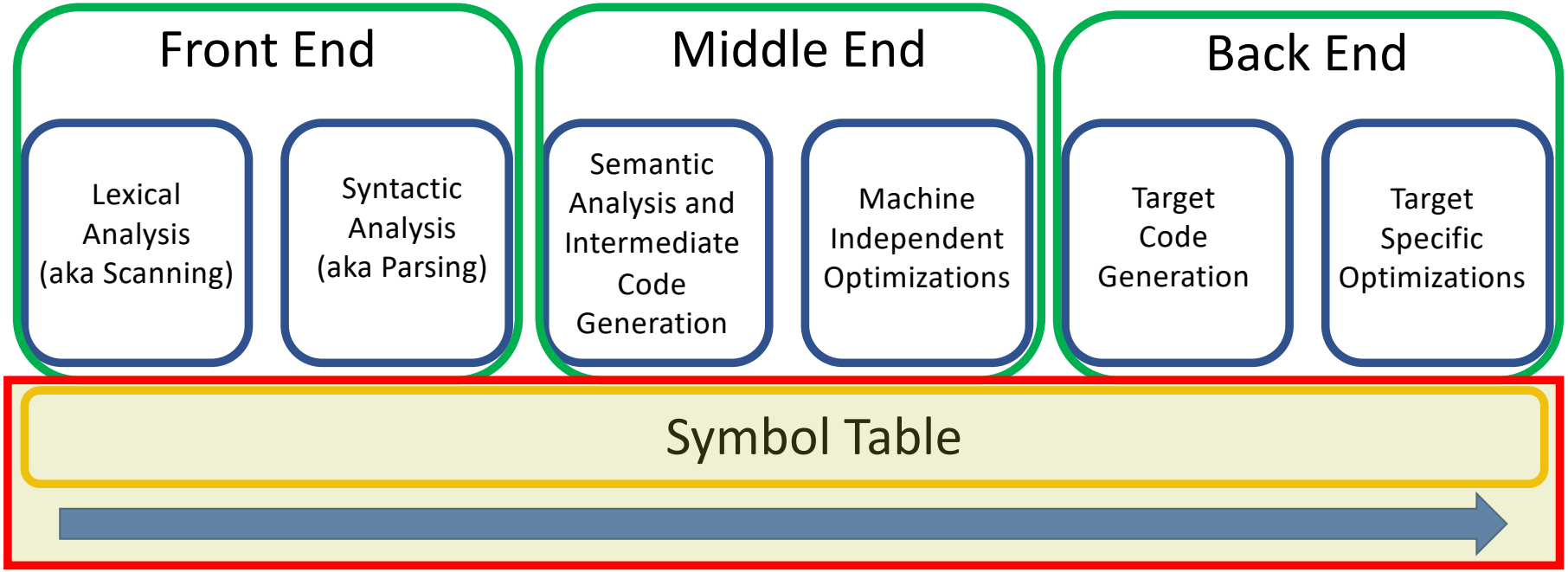
Group Project Part 3

```
CMD>ls
driver.cc   icode.cc   Makefile     run-smp.sh   symtab.cc
grammar.y   icode.hh   run-all.sh   scanner.yy   symtab.hh
```



Group Project Part 3

```
CMD>ls
driver.cc   icode.cc   Makefile     run-smp.sh   symtab.cc
grammar.y   icode.hh   run-all.sh   scanner.yy   symtab.hh
```



The Scanner File (scanner.yy)

The Scanner File (scanner.yy)

```
\/\/.*$
[ \t]+                          { loc.step (); }
[\n]+                                   { loc.lines (yyleng); loc.step (); }


"write"             {
                                                                        //yylval->build (yytext);
                                                                        return yy::simple_parser::make_T_WRITE(loc);
                    }

"read"              {
                                                                        //yylval->build (yytext);
                                                                        return yy::simple_parser::make_T_READ(loc);
                    }

":="                                    {
                                                                        //yylval->build (yytext);
                                                                        return yy::simple_parser::make_T_ASSIGN(loc);
                    }

";"                                     {
                                                                        //yylval->build (yytext);
                                                                        return yy::simple_parser::make_T_SEMICOLON(loc);
                    }

","                                     {
                                                                        //yylval->build (yytext);
                                                                        return yy::simple_parser::make_T_COMMA(loc);
                    }
```

The Scanner File (scanner.yy)

```
\/\/.*$
[ \t]+                                  { loc.step (); }
[\n]+                                   { loc.lines (yyleng); loc.step (); }


"write"            {
                                                    //yylval->build (yytext);
                                                    return yy::simple_parser::make_T_WRITE(loc);
                   }

"read"             {
                                                    //yylval->build (yytext);
                                                    return yy::simple_parser::make_T_READ(loc);
                   }

":="                                    {
                                                    //yylval->build (yytext);
                                                    return yy::simple_parser::make_T_ASSIGN(loc);
                   }

";"                                     {
                                                    //yylval->build (yytext);
                                                    return yy::simple_parser::make_T_SEMICOLON(loc);
                   }

","                                     {
                                                    //yylval->build (yytext);
                                                    return yy::simple_parser::make_T_COMMA(loc);
                   }
```

The Scanner File (scanner.yy)

```
//yylval->build (yytext);
return yy::simple_parser::make_T_ASSIGN(loc);
```

For instance, given the following declarations:

```
%define api.token.prefix {TOK_}
%token <std::string> IDENTIFIER;
%token <int> INTEGER;
%token COLON;
%token EOF 0;
```

Bison generates:

```
symbol_type make_IDENTIFIER (const std::string&, const location_type&);
symbol_type make_INTEGER (const int&, const location_type&);
symbol_type make_COLON (const location_type&);
symbol_type make_EOF (const location_type&);
```

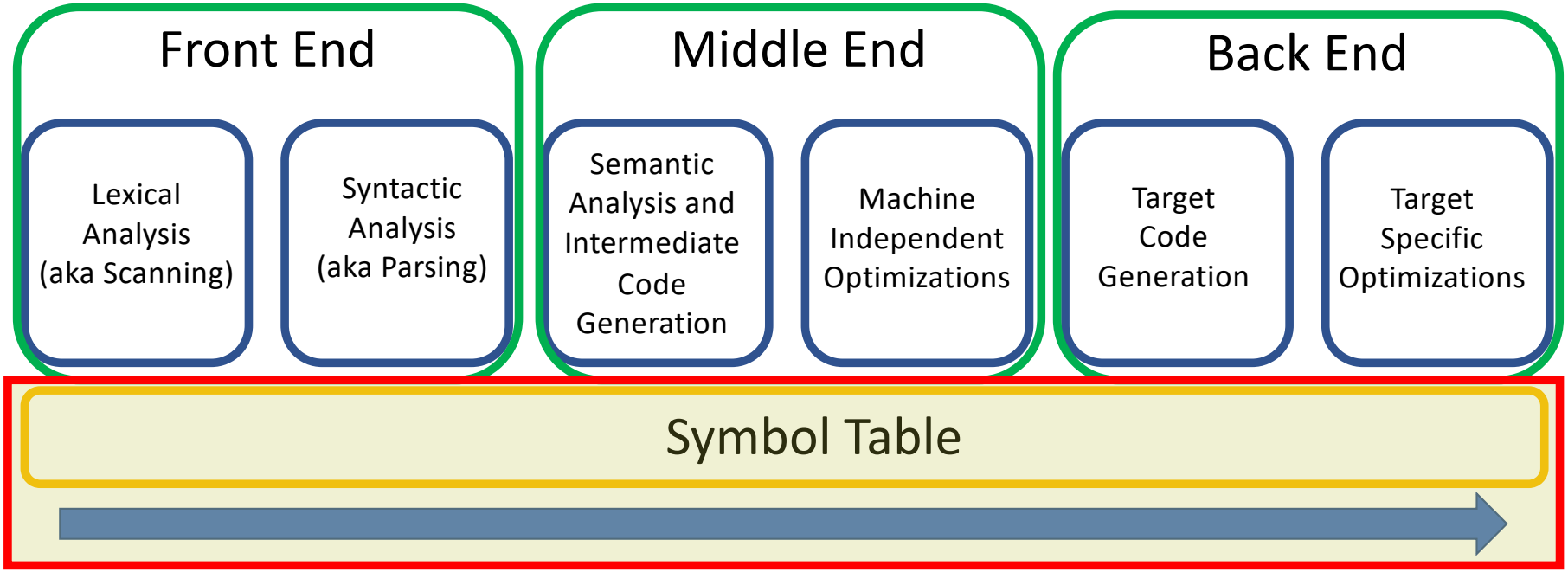
which should be used in a scanner as follows.

```
[a-z]+   return yy::parser::make_IDENTIFIER (yytext, loc);
[0-9]+   return yy::parser::make_INTEGER (text_to_int (yytext), loc);
":"      return yy::parser::make_COLON (loc);
<<EOF>>  return yy::parser::make_EOF (loc);
```

SEE: https://www.gnu.org/software/bison/manual/html_node/Complete-Symbols.html

The Symbol Table (symtab.cc)

```
CMD>ls
driver.cc   icode.cc   Makefile     run-smp.sh   symtab.cc
grammar.y   icode.hh   run-all.sh   scanner.yy   symtab.hh
```



The Symbol Table (symtab.hh)

```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;

typedef map<string,symbol_t*> mapsym_t;
typedef map<string,symbol_t*>::iterator itersym_t;
typedef vector<symbol_t*> vector_sym_t;
typedef vector<symbol_t*>::iterator vector_itersym_t;


struct symbol_table {
  mapsym_t * map;
  int offset;
  int ntemp;
};

typedef struct symbol_table symtab_t;
```

```
extern
symtab_t * symtab_create ();

extern
void symtab_free (symtab_t * symtab);

extern
symbol_t * symbol_create (symtab_t * symtab, string p_name, int p_type);

extern
void symbol_show (symbol_t * sym);

extern
void symbol_free (symbol_t * sym);

extern
int symbol_add (symtab_t * & symtab, symbol_t * & sym);

extern
symbol_t * symbol_find (symtab_t * symtab, string sym_name);

extern
void symtab_show (symtab_t * symtab);

extern
symbol_t * make_temp (symtab_t * symtab, int p_type);
```

The Symbol Table (symtab.hh)

```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;

typedef map<string,symbol_t*> mapsym_t;
typedef map<string,symbol_t*>::iterator itersym_t;
typedef vector<symbol_t*> vector_sym_t;
typedef vector<symbol_t*>::iterator vector_itersym_t;


struct symbol_table {
  mapsym_t * map;
  int offset;
  int ntemp;
};

typedef struct symbol_table symtab_t;
```

```
extern
symtab_t * symtab_create ();

extern
void symtab_free (symtab_t * symtab);

extern
symbol_t * symbol_create (symtab_t * symtab, string p_name, int p_type);

extern
void symbol_show (symbol_t * sym);

extern
void symbol_free (symbol_t * sym);

extern
int symbol_add (symtab_t * & symtab, symbol_t * & sym);

extern
symbol_t * symbol_find (symtab_t * symtab, string sym_name);

extern
void symtab_show (symtab_t * symtab);

extern
symbol_t * make_temp (symtab_t * symtab, int p_type);
```

The Symbol Table (symtab.hh)

```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;

typedef map<string,symbol_t*> mapsym_t;
typedef map<string,symbol_t*>::iterator itersym_t;
typedef vector<symbol_t*> vector_sym_t;
typedef vector<symbol_t*>::iterator vector_itersym_t;

struct symbol_table {
  mapsym_t * map;
  int offset;
  int ntemp;
};

typedef struct symbol_table symtab_t;
```

```
extern
symtab_t * symtab_create ();

extern
void symtab_free (symtab_t * symtab);

extern
symbol_t * symbol_create (symtab_t * symtab, string p_name, int p_type);

extern
void symbol_show (symbol_t * sym);

extern
void symbol_free (symbol_t * sym);

extern
int symbol_add (symtab_t * & symtab, symbol_t * & sym);

extern
symbol_t * symbol_find (symtab_t * symtab, string sym_name);

extern
void symtab_show (symtab_t * symtab);

extern
symbol_t * make_temp (symtab_t * symtab, int p_type);
```

The Symbol Table (symtab.hh)

Symbol Table

Key (name)	Value (symbol_t)

```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;
```

Stack

Offset (1 Byte)	Value (1B)
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
…	
100	

The Symbol Table (symtab.hh)

Symbol Table

Key (name)	Value (symbol_t)
"var_a"	{0,INT}

```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;
```

Stack

Offset (1 Byte)	Value (1B)
0 ("var_a")	
1 ("var_a")	
2	
3	
4	
5	
6	
7	
8	
9	
…	
100	

The Symbol Table (symtab.hh)

Symbol Table

Key (name)	Value (symbol_t)
"var_a"	{0,INT}

```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;
```

Stack

Offset (1 Byte)	Value (1B)
0 ("var_a")	
1 ("var_a")	
2	
3	
4	
5	
6	
7	
8	
9	
...	
100	

The Symbol Table (symtab.hh)

Symbol Table

Key (name)	Value (symbol_t)
"var_a"	{0,INT}
"var_b"	{2,FLOAT}

```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;
```

Stack

Offset (1 Byte)	Value (1B)
0 ("var_a")	
1 ("var_a")	
2 ("var_b")	
3 ("var_b")	
4 ("var_b")	
5 ("var_b")	
6	
7	
8	
9	
...	
100	

The Symbol Table (symtab.hh)

Symbol Table

Key (name)	Value (symbol_t)
"var_a"	{0,INT}
"var_b"	{2,FLOAT}

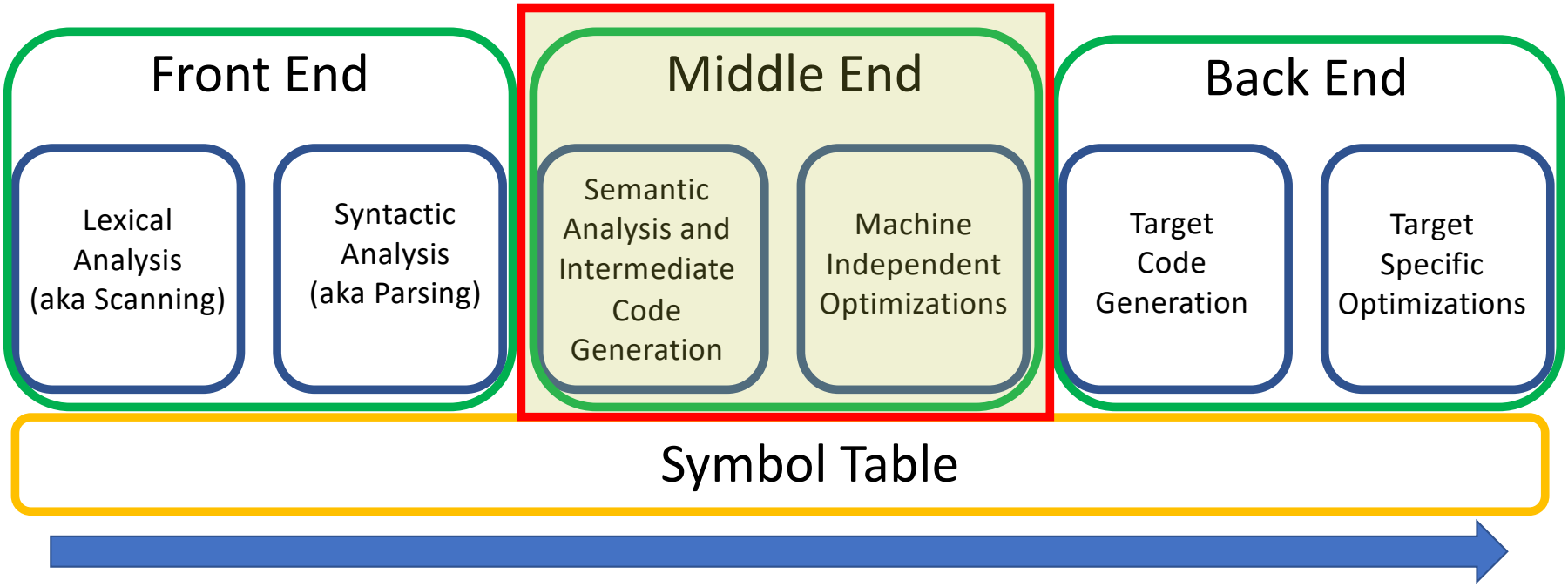
```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;
```

Stack

Offset (1 Byte)	Value (1B)
0 ("var_a")	
1 ("var_a")	
2 ("var_b")	
3 ("var_b")	
4 ("var_b")	
5 ("var_b")	
6	
7	
8	
9	
...	
100	

Intermediate Code (icode.*)

```
CMD>ls
driver.cc   icode.cc   Makefile     run-smp.sh   symtab.cc
grammar.y   icode.hh   run-all.sh   scanner.yy   symtab.hh
```



Intermediate Code (icode.*)

3 Address ICode

```
// Simple 3-address structure
struct simple_icode {
  int op_code;
  int addr1;
  int addr2;
  int addr3;
};
// a := b + c ==> addr1 := addr2 o_code addr3

typedef struct simple_icode icode_t;
```

Op codes

```
// Operations where the datatype is embedded in
#define OP_LOAD 0
#define OP_STORE 1
#define OP_LOADCST 2
#define OP_WRITE 3
#define OP_READ 4

// Arithmetic Operations specific to datatypes
#define OP_ADD 5
#define OP_SUB 6
#define OP_MUL 7
#define OP_DIV 8
#define OP_UMIN 9

// Floating point opcode
#define OP_FADD 10
#define OP_FSUB 11
#define OP_FMUL 12
#define OP_FDIV 13
#define OP_FUMIN 14

#define SMP_SUCCESS 0
#define SMP_ERROR 1
```

Intermediate Code (icode.*)

Instruction Table

```
typedef struct simple_icode icode_t;

struct simple_itab {
  vector<icode_t*> tab;
};

typedef struct simple_itab itab_t;
```

Instruction table

index	instruction
0	a := b + c
1	
2	
3	
4	
5	
6	
7	
8	
9	
...	
100	

Intermediate Code (icode.*)

Run routine in icode.c (REPL)

```
int run (itab_t * itab, char * stack, char * static_mem)
{
  int ii;
  for (ii = 0; ii < itab->tab.size (); ii++)
  {
    icode_t * op = itab->tab[ii];
    switch (op->op_code)
    {
      case OP_LOAD:
        if (op->addr2 == DTYPE_INT)
        {
          int * src = (int*)(stack + op->addr3);
          int * dst = (int*)(stack + op->addr1);
          *dst = *src;
        }
        // TASK: Complete case for DTYPE_FLOAT
        break;
      case OP_LOADCST:
        if (op->addr2 == DTYPE_INT)
        {
          int * src = (int*)(static_mem + op->addr3);
          int * dst = (int*)(stack + op->addr1);
          *dst = *src;
        }
        // TASK: Complete case for DTYPE_FLOAT
        break;
      case OP_STORE:
        if (op->addr2 == DTYPE_INT)
        {
          int * src = (int*)(stack + op->addr3);
          int * dst = (int*)(stack + op->addr1);
          *dst = *src;
        }
        // TASK: Complete case for DTYPE_FLOAT
        break;
      case OP_ADD:
        {
          int * left = (int*)(stack + op->addr2);
          int * right = (int*)(stack + op->addr3);
          int * res = (int*)(stack + op->addr1);
          *res = *left + *right;
        }
        break;
```

Zoomed in operation

```
case OP_ADD:
  {
    int * left = (int*)(stack + op->addr2);
    int * right = (int*)(stack + op->addr3);
    int * res = (int*)(stack + op->addr1);
    *res = *left + *right;
  }
  break;
```

3 Address ICode

```
// Simple 3-address structure
struct simple_icode {
  int op_code;
  int addr1;
  int addr2;
  int addr3;
};
// a := b + c ==> addr1 := addr2 o_code addr3

typedef struct simple_icode icode_t;
```

Intermediate Code (icode.*)

Symbol Table

Key (name)	Value (symbol_t)
"var_c"	{4,INT}
"var_a"	{0,INT}
"var_b"	{2,INT}

```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;
```

Stack

Offset (1 Byte)	Value (1B)
0 ("var_a")	
1 ("var_a")	
2 ("var_b")	
3 ("var_b")	
4 ("var_c")	
5 ("var_c")	
6	
7	
8	
9	
...	
100	

Instruction table

index	instruction
0	a := b + c
1	
...	
100	

```
case OP_ADD:
  {
    int * left = (int*)(stack + op->addr2);
    int * right = (int*)(stack + op->addr3);
    int * res = (int*)(stack + op->addr1);
    *res = *left + *right;
  }
  break;
```

Intermediate Code (icode.*)

Symbol Table

Key (name)	Value (symbol_t)
"var_c"	{4,INT}
"var_a"	{0,INT}
"var_b"	{2,INT}

```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;
```

Stack

Offset (1 Byte)	Value (1B)
0 ("var_a")	
1 ("var_a")	
2 ("var_b")	
3 ("var_b")	
4 ("var_c")	
5 ("var_c")	
6	
7	
8	
9	
...	
100	

Instruction table

index	instruction
0	a := b + c
1	
...	
100	

```
case OP_ADD:
  {
    int * left = (int*)(stack + op->addr2);
    int * right = (int*)(stack + op->addr3);
    int * res = (int*)(stack + op->addr1);
    *res = *left + *right;
  }
  break;
```

Intermediate Code (icode.*)

Symbol Table

Key (name)	Value (symbol_t)
"var_c"	{4,INT}
"var_a"	{0,INT}
"var_b"	{2,INT}

```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;
```

Stack

Offset (1 Byte)	Value (1B)
0 ("var_a")	
1 ("var_a")	
2 ("var_b")	
3 ("var_b")	
4 ("var_c")	
5 ("var_c")	
6	
7	
8	
9	
...	
100	

Instruction table

index	instruction
0	a := b + c
1	
...	
100	

```
case OP_ADD:
  {
    int * left = (int*)(stack + op->addr2);
    int * right = (int*)(stack + op->addr3);
    int * res = (int*)(stack + op->addr1);
    *res = *left + *right;
  }
  break;
```

Intermediate Code (icode.*)

Symbol Table

Key (name)	Value (symbol_t)
"var_c"	{4,INT}
"var_a"	{0,INT}
"var_b"	{2,INT}

```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;
```

Stack

Offset (1 Byte)	Value (1B)
0 ("var_a")	
1 ("var_a")	
2 ("var_b")	
3 ("var_b")	
4 ("var_c")	
5 ("var_c")	
6	
7	
8	
9	
...	
100	

Instruction table

index	instruction
0	a := b + c
1	
...	
100	

```
case OP_ADD:
  {
    int * left = (int*)(stack + op->addr2);
    int * right = (int*)(stack + op->addr3);
    int * res = (int*)(stack + op->addr1);
    *res = *left + *right;
  }
  break;
```

Intermediate Code (icode.*)

Symbol Table

Key (name)	Value (symbol_t)
"var_c"	{4,INT}
"var_a"	{0,INT}
"var_b"	{2,INT}

```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;
```

Stack

Offset (1 Byte)	Value (1B)
0 ("var_a")	
1 ("var_a")	
2 ("var_b")	
3 ("var_b")	
4 ("var_c")	
5 ("var_c")	
6	
7	
8	
9	
...	
100	

Instruction table

index	instruction
0	a := b + c
1	
...	
100	

```
case OP_ADD:
  {
    int * left = (int*)(stack + op->addr2);
    int * right = (int*)(stack + op->addr3);
    int * res = (int*)(stack + op->addr1);
    *res = *left + *right;
  }
  break;
```

Intermediate Code (icode.*)

Symbol Table

Key (name)	Value (symbol_t)
"var_c"	{4,INT}
"var_a"	{0,INT}
"var_b"	{2,INT}

```
struct simple_symbol {
  string name;
  int  addr;
  int  datatype; //
};

typedef struct simple_symbol symbol_t;
```

Stack

Offset (1 Byte)	Value (1B)
0 ("var_a")	
1 ("var_a")	
2 ("var_b")	
3 ("var_b")	
4 ("var_c")	
5 ("var_c")	
6	
7	
8	
9	
...	
100	

Instruction table

index	instruction
0	a := b + c
1	
...	
100	

```
case OP_ADD:
  {
    int * left = (int*)(stack + op->addr2);
    int * right = (int*)(stack + op->addr3);
    int * res = (int*)(stack + op->addr1);
    *res = *left + *right;
  }
  break;
```

Putting it all together

The Scanner File (scanner.yy)

The Grammar File (grammar.yy)

```
CMD>ls
driver.cc   icode.cc   Makefile     run-smp.sh   symtab.cc
grammar.y   icode.hh   run-all.sh   scanner.yy   symtab.hh
```

```
program : stmt_list T_SEMICOLON
  ;

stmt_list : stmt_list T_SEMICOLON stmt
    | stmt
    ;

stmt : assignment
    | read
    | write
    | declaration
    ;
```

```
assignment : varref T_ASSIGN a_expr
     {
       itab_instruction_add (itab, OP_STORE, $1->addr, $1->datatype, $3->addr);
       $$ = $1;
     }
    ;

declaration: datatype T_ID {
      assert (symtab);
      assert (itab);
      symbol_t * sym = symbol_create (symtab, $2, $1);
      assert (sym);
      symbol_add (symtab, sym);
    }
    ;

datatype : T_DT_INT { $$ = DTYPE_INT; }
    | T_DT_FLOAT { $$ = DTYPE_FLOAT; }
    ;
```

The Grammar File (grammar.yy)

```
program : stmt_list T_SEMICOLON
  ;

stmt_list : stmt_list T_SEMICOLON stmt
    | stmt
    ;

stmt : assignment
    | read
    | write
    | declaration
    ;
```

```
int a;
int b;
int c;
c:= a + b;
write c;
```

The Grammar File (grammar.yy)

```
program : stmt_list T_SEMICOLON
  ;

stmt_list : stmt_list T_SEMICOLON stmt
    | stmt
    ;

stmt : assignment
    | read
    | write
    | declaration
    ;
```

```
int a;
int b;
int c;
c:= a + b;
write c;
```

The Grammar File (grammar.yy)

```
program : stmt_list T_SEMICOLON
  ;

stmt_list : stmt_list T_SEMICOLON stmt
    | stmt
    ;

stmt : assignment
    | read
    | write
    | declaration
    ;
```

```
int a;
int b;
int c;
c:= a + b;
write c;
```

The Grammar File (grammar.yy)

```
program : stmt_list T_SEMICOLON
  ;

stmt_list : stmt_list T_SEMICOLON stmt
    | stmt
    ;

stmt : assignment
    | read
    | write
    | declaration
    ;
```

```
int a;
int b;
int c;
c:= a + b;
write c;
```

The Grammar File (grammar.yy)

```
program : stmt_list T_SEMICOLON
  ;

stmt_list : stmt_list T_SEMICOLON stmt
    | stmt
    ;

stmt : assignment
    | read
    | write
    | declaration
    ;
```

```
int a;
int b;
int c;
c:= a + b;
write c;
```

The Grammar File (grammar.yy)

```
assignment : varref T_ASSIGN a_expr
      {
        itab_instruction_add (itab, OP_STORE, $1->addr, $1->datatype, $3->addr);
        $$ = $1;
      }
    ;

declaration: datatype T_ID {
      assert (symtab);
      assert (itab);
      symbol_t * sym = symbol_create (symtab, $2, $1);
      assert (sym);
      symbol_add (symtab, sym);
    }
    ;

datatype : T_DT_INT { $$ = DTYPE_INT; }
    | T_DT_FLOAT { $$ = DTYPE_FLOAT; }
    ;
```

```
int a;
```

Symbol Table

Key (name)	Value (symbol_t)
"a"	{0,INT}

The Grammar File (grammar.yy)

```
assignment : varref T_ASSIGN a_expr
      {
        itab_instruction_add (itab, OP_STORE, $1->addr, $1->datatype, $3->addr);
        $$ = $1;
      }
    ;

declaration: datatype T_ID {
      assert (symtab);
      assert (itab);
      symbol_t * sym = symbol_create (symtab, $2, $1);
      assert (sym);
      symbol_add (symtab, sym);
    }
    ;

datatype : T_DT_INT { $$ = DTYPE_INT; }
    | T_DT_FLOAT { $$ = DTYPE_FLOAT; }
    ;
```

```
int a;
```

Symbol Table

Key (name)	Value (symbol_t)
"a"	{0,INT}

The Grammar File (grammar.yy)

```
assignment : varref T_ASSIGN a_expr
      {
        itab_instruction_add (itab, OP_STORE, $1->addr, $1->datatype, $3->addr);
        $$ = $1;
      }
    ;

declaration: datatype T_ID {
      assert (symtab);
      assert (itab);
      symbol_t * sym = symbol_create (symtab, $2, $1);
      assert (sym);
      symbol_add (symtab, sym);
    }
    ;

datatype : T_DT_INT { $$ = DTYPE_INT; }
    | T_DT_FLOAT { $$ = DTYPE_FLOAT; }
    ;
```

```
int a;
```

Symbol Table

Key (name)	Value (symbol_t)
"a"	{0 INT}

The Grammar File (grammar.yy)

```
assignment : varref T_ASSIGN a_expr
      {
        itab_instruction_add (itab, OP_STORE, $1->addr, $1->datatype, $3->addr);
        $$ = $1;
      }
    ;

declaration: datatype T_ID {
      assert (symtab);
      assert (itab);
      symbol_t * sym = symbol_create (symtab, $2, $1);
      assert (sym);
      symbol_add (symtab, sym);
    }
    ;

datatype : T_DT_INT { $$ = DTYPE_INT; }
    | T_DT_FLOAT { $$ = DTYPE_FLOAT; }
    ;
```

```
int a;
```

Symbol Table

Key (name)	Value (symbol_t)
"a"	{0,INT}

The Grammar File (grammar.yy)

```
assignment : varref T_ASSIGN a_expr
      {
        itab_instruction_add (itab, OP_STORE, $1->addr, $1->datatype, $3->addr);
        $$ = $1;
      }
    ;

declaration: datatype T_ID {
      assert (symtab);
      assert (itab);
      symbol_t * sym = symbol_create (symtab, $2, $1);
      assert (sym);
      symbol_add (symtab, sym);
    }
    ;

datatype : T_DT_INT { $$ = DTYPE_INT; }
    | T_DT_FLOAT { $$ = DTYPE_FLOAT; }
    ;
```

```
c:= a;
```

Instruction table

index	instruction
0	stack[2] = stack[0];
1	
...	
100	

Stack

Offset (1 Byte)	Value (1B)
0 ("a")	0xBE
1 ("a")	0xEF
2 ("c")	
3 ("c")	
...	
100	

The Grammar File (grammar.yy)

```
assignment : varref T_ASSIGN a_expr
      {
        itab_instruction_add (itab, OP_STORE, $1->addr, $1->datatype, $3->addr);
        $$ = $1;
      }
    ;

declaration: datatype T_ID {
      assert (symtab);
      assert (itab);
      symbol_t * sym = symbol_create (symtab, $2, $1);
      assert (sym);
      symbol_add (symtab, sym);
    }
    ;

datatype : T_DT_INT { $$ = DTYPE_INT; }
    | T_DT_FLOAT { $$ = DTYPE_FLOAT; }
    ;
```

```
c:= a;
```

Instruction table

index	instruction
0	stack[2] = stack[0];
1	
…	
100	

Stack

Offset (1 Byte)	Value (1B)
0 ("a")	0xBE
1 ("a")	0xEF
2 ("c")	
3 ("c")	
…	
100	

The Grammar File (grammar.yy)

```
assignment : varref T_ASSIGN a_expr
      {
        itab_instruction_add (itab, OP_STORE, $1->addr, $1->datatype, $3->addr);
        $$ = $1;
      }
    ;

declaration: datatype T_ID {
      assert (symtab);
      assert (itab);
      symbol_t * sym = symbol_create (symtab, $2, $1);
      assert (sym);
      symbol_add (symtab, sym);
    }
    ;

datatype : T_DT_INT { $$ = DTYPE_INT; }
    | T_DT_FLOAT { $$ = DTYPE_FLOAT; }
    ;
```

c := a;

Instruction table

index	instruction
0	stack[2] = stack[0]
1	
...	
100	

Stack

Offset (1 Byte)	Value (1B)
0 ("a")	0xBE
1 ("a")	0xEF
2 ("c")	
3 ("c")	
...	
100	

The Grammar File (grammar.yy)

```
assignment : varref T_ASSIGN a_expr
      {
        itab_instruction_add (itab, OP_STORE, $1->addr, $1->datatype, $3->addr);
        $$ = $1;
      }
    ;

declaration: datatype T_ID {
      assert (symtab);
      assert (itab);
      symbol_t * sym = symbol_create (symtab, $2, $1);
      assert (sym);
      symbol_add (symtab, sym);
    }
    ;

datatype : T_DT_INT { $$ = DTYPE_INT; }
    | T_DT_FLOAT { $$ = DTYPE_FLOAT; }
    ;
```

```
c:= a;
```

Instruction table

index	instruction
0	stack[2] = stack[0]
1	
…	
100	

Stack

Offset (1 Byte)	Value (1B)
0 ("a")	0xBE
1 ("a")	0xEF
2 ("c")	0xBE
3 ("c")	0xEF
…	
100	

The Grammar File (grammar.yy)

```
a_expr : a_expr T_ADD a_term
       {
         if ($1->datatype != $3->datatype)
         {
           cout << "Incompatible datatypes\n";
           exit (1);
         }
         if ($1->datatype == DTYPE_INT)
         {
           res = make_temp (symtab, $1->datatype);
           itab_instruction_add (itab, OP_ADD, res->addr, $1->addr, $3->addr);
         }
         {
           // TASK: Modify this semantic action to support both DTYPE_INT and DTYPE_FLOAT.
           // For DTYPE_FLOAT you should generate an OP_FADD instruction.
         }
         $$ = res;
         #ifdef _SMP_DEBUG_
         cout << "On a_expr (1)\n";
         #endif
       }
```

```
c:= a+b;
```

Instruction table

idx	instruction
0	stack[6] = stack[0]+stack[4]
1	stack[2] = stack[6]
...	
100	

Stack

Offset (1 Byte)	Value (1B)
0 ("a")	
1 ("a")	
2 ("c")	
3 ("c")	
4 ("b")	
5 ("b")	
6 ("temp0")	
7 ("temp0")	

The Grammar File (grammar.yy)

```
a_expr : a_expr T_ADD a_term
      {
        if ($1->datatype != $3->datatype)
        {
          cout << "Incompatible datatypes\n";
          exit (1);
        }
        symbol_t * res;
        if ($1->datatype == DTYPE_INT)
        {
          res = make_temp (symtab, $1->datatype);
          rtab_instruction_add (rtab, OP_ADD, res->addr, $1->addr, $3->addr);
        }
        if ($1->datatype == DTYPE_FLOAT)
        {
          // TASK: Modify this semantic action to support both DTYPE_INT and DTYPE_FLOAT.
          // For DTYPE_FLOAT you should generate an OP_FADD instruction.
        }
        $$ = res;
        #ifdef _SMP_DEBUG_
        cout << "On a_expr (1)\n";
        #endif
      }
```

c := a+b;

Instruction table

idx	instruction
0	stack[6] = stack[0]+stack[4]
1	stack[2] = stack[6]
...	
100	

Stack

Offset (1 Byte)	Value (1B)
0 ("a")	
1 ("a")	
2 ("c")	
3 ("c")	
4 ("b")	
5 ("b")	
6 ("temp0")	
7 ("temp0")	

The Grammar File (grammar.yy)

```
a_expr : a_expr T_ADD a_term
      {
        if ($1->datatype != $3->datatype)
        {
          cout << "Incompatible datatypes\n";
          exit (1);
        }
        symbol_t * res;
        if ($1->datatype == DTYPE_INT)
        {
          res = make_temp (symtab, $1->datatype);
          itab_instruction_add (itab, OP_ADD, res->addr, $1->addr, $3->addr);
        }
        if ($1->datatype == DTYPE_FLOAT)
        {
          // TASK: Modify this semantic action to support both DTYPE_INT and DTYPE_FLOAT.
          // For DTYPE_FLOAT you should generate an OP_FADD instruction.
        }
        $$ = res;
        #ifdef _SMP_DEBUG_
        cout << "On a_expr (1)\n";
        #endif
      }
```

Instruction table

idx	instruction
0	stack[6] = stack[0]+stack[4]
1	stack[2] = stack[6]
...	
100	

Stack

Offset (1 Byte)	Value (1B)
0 ("a")	
1 ("a")	
2 ("c")	
3 ("c")	
4 ("b")	
5 ("b")	
6 ("temp0")	
7 ("temp0")	

c := a+b;

The Grammar File (grammar.yy)

```
a_expr : a_expr T_ADD a_term
      {
        if ($1->datatype != $3->datatype)
        {
          cout << "Incompatible datatypes\n";
          exit (1);
        }
        symbol_t * res;
        if ($1->datatype == DTYPE_INT)
        {
          res = make_temp (symtab, $1->datatype);
          itab_instruction_add (itab, OP_ADD, res->addr, $1->addr, $3->addr);
        }
        if ($1->datatype == DTYPE_FLOAT)
        {
          // TASK: Modify this semantic action to support both DTYPE_INT and DTYPE_FLOAT.
          // For DTYPE_FLOAT you should generate an OP_FADD instruction.
        }
        $$ = res;
        #ifdef _SMP_DEBUG_
        cout << "On a_expr (1)\n";
        #endif
      }
```

c:= a+b;

Instruction table

idx	instruction
0	stack[6] = stack[0]+stack[4]
1	stack[2] = stack[6]
...	
100	

Stack

Offset (1 Byte)	Value (1B)
0 ("a")	
1 ("a")	
2 ("c")	
3 ("c")	
4 ("b")	
5 ("b")	
6 ("temp0")	
7 ("temp0")	

TA Office Hours

Ega at DEH 115

or on Zoom:
https://oklahoma.zoom.us/j/94320587521?pwd=L2RCVVRCVDFDSW9FSUR1dUlhaUpoZz09
Meeting ID: 943 2058 7521
Passcode: 39931089

- Monday (10:00 - 11:00 am)
- Tuesday (12:00 noon - 1:00 pm)
- Wednesday (9:00 - 10:00 am)
- Thursday (9:00 - 10:00 am)

James on Zoom

https://oklahoma.zoom.us/j/4808825257?pwd=dDAvRTlwSVh2S0FtUkVtTWVzMlBuUT09

Meeting ID: 480 882 5257
Passcode: 7$*%?4CC

- Tuesday (2:00-3:00 pm)
- Wednesday (1:00-2:00 pm)
- Thursday (3:00- 4:00 pm)
- Friday (2:00-3:00 pm)