



# *The Two Pass Assembler* <sup>a</sup>

Teodor Rus

`rus@cs.uiowa.edu`

The University of Iowa, Department of Computer Science

---

<sup>a</sup>These slides have been developed by Teodor Rus. They are copyrighted materials and may not be used in other course settings outside of the University of Iowa in their current form or modified form without the express written permission of the copyright holder. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of the copyright holder.

# The assembler

*Assembler* :  $AL \rightarrow ML$  where:

- *AL* is specified on three levels of structuring:
  1. Generators: mnemonics (operations, directives, immediate values) and user defined symbols (labels, operands, expressions)
  2. Statements: [Label:] Mnemonic [Operands] [Comment]
  3. Program: Sequence of statements.
- *ML* is specified on three levels of structuring:
  1. Machine codes: operations, registers, immediate values, address expressions.
  2. Instruction and data patterns.;
  3. Machine language program: sequence of instruction and data.

# *Assembler implementation*

A three readings of assembly language program that perform:

1. Recognize statement generators, map them into machine codes and store them into tables (SYMTAB).
2. Recognize statements, map them into internal forms, (IFAS), and store them into the File of Internal Form, (FIF).
3. Recognize program in terms of its statements, map each IFAS in FIF into the actual machine representation using its generator translations, and map FIF into the File of Object Generated (FOG).

# Optimization

The readings (2) and (3) can be combined thus leading to a two pass assembler:

- Pass 1 that performs generator translation.
- Pass 2 that performs program translation.

**Note:** the notion of pass actually refers to the reading of the assembly language program by the assembler.

# Specification

- $Pass_1$  reads the assembly source program statement by statement. For each statement translates its generators, saves their translations in appropriate table (called SYMTAB), and generates an intermediate form of the statement.

**Note:** For optimization reason the source language program may be transformed into an intermediate language program which is written into a file called the File of Internal Form, FIF.

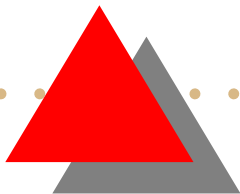
- $Pass_2$  examines FIF statement by statement. For each statement  $S$  in FIF,  $Pass_2$  searches for its generator translations in SYMTAB, use them to generate the machine language representation of  $S$ ,  $M(S)$ , and then assemble  $M(S)$  into the machine language program.



# Data structures

Assembler implementation is based on two major data structures: Operation Table (`OPTAB`) and Symbol Table (`SYMTAB`).

- For each mnemonic  $N$  the `OPTAB` contains:
  1. Mnemonic type and its Machine language expression;
  2. The pattern  $M(N)$  generated by  $Pass_2$  when a statement with mnemonic  $N$  is encountered;
  3. The function called by  $Pass_1$  to translate the assembly language statements whose mnemonic is  $N$ ;
  4. The function called by  $Pass_2$  to instantiate the pattern  $M(N)$  when a statement with mnemonic  $N$  is encountered.
- `SYMTAB`, that holds the translation of user defined symbols.





# Facts

For optimization and generality purposes the following supplementary data structures may also be supported by the assembler:

1. Internal Form of Assembly Statement ( *IFAS* ) that allow the assembler to perform only one reading of the source;
2. File of Internal Form ( *FIF* ) generated by  $Pass_1$  to holds the internal representation of the source and to be processed by  $Pass_2$ ;
3. File of Object Generated ( *FOG* ), that holds machine language form of an assembled module.

# OPTAB design

**OPTAB is the heart of the assembler.**

OPTAB structuring results from the following analysis:

- Since mnemonics are given OPTAB is predefined. There are four types of mnemonics in a general assembly language:
  1. Machine operations, `ADD`, `SUB`, `DIV`, `etc.`. Their type in OPTAB is **O**.
  2. Data definitions called pseudo-operations or directives, such as `x DV.integer 20; x EQU y`, `etc..`. Their type in OPTAB is **P**.
  3. Macro-operation definitions. Their type in OPTAB is **P**.
  4. Macro operation call. Their type in OPTAB is **C**.



# Example macros

- Macro definition:

```
PUSH DMACRO #1, #2;  
    Load      #1;  
    Store     Stack[#2];  
    Increment #2;  
EMACRO
```

**Note:** parameters are identified by prefixing them with the symbol #.

- Macro call:

```
MACCALL PUSH ALPHA, BETA;
```



# *Oolong mnemonics*

In Oolong there are only two kinds of mnemonics:

- `Directives`, (pseudo-operations) that represent instructions to the assembler. Their type in `OPTAB` is `D`;
- `Byte-codes` that represent machine operations. Their type in `OPTAB` is `B`.

# Assumptions

- Internal Form of Assembly Statement, ( *IFAS* ), is the same for all AL and ML.
- Each mnemonic is associated with a *translation pattern* the binary pattern that is used by Pass 2 to translate AL statements having this mnemonic. it.
- Each mnemonic is associated with the function  $StmtMap : Statement \rightarrow IFAS$  called by Pass 1 when a statement having this mnemonic is discovered.
- Each mnemonic is associated with the function  $GenMap : IFAS \rightarrow Instruction/Data\ Words$  called by Pass 2 when a statement having this mnemonic is discovered.

# OPTAB entry

The OPTAB entry is shown in Figure 1

Mnemonic	OpCode	$O P M C$
translation pattern		
$GenMap : IFAS \rightarrow translation\ patter$		
$StmtMap : Statement \rightarrow IFAS$		

Figure 1: OPTAB entry

**Note:** for an Oolong assembler the fields  $O|P|M|C$  should be  $ByteCode|Directive$

# Example 1

OPTAB entry for a Mixal assembler is in Figure 2

Mnemonic	OpCode	$O P M C$	
***	***	**	*
$GenMap : IFAS \rightarrow Addr\ Index\ Ext\ OpCode$			
$StmtMap : Statement \rightarrow IFAS$			

Figure 2: OPTAB entry

**Note:** the parameters of the [translation pattern](#) in this example are: [address](#) denoted by `***`, [index](#) denoted by `***`, [extension](#) denoted by `**`, and [OpCode](#) denoted by `*`, respectively.



# Notations

Here we use the machine language structure of Mix machine, where:

- \* \* \*\* is the address field
- \* \* \* is the index or indirection field
- \*\* is the field specifier, OpCode extension, or I/O device
- \* is the opcode field

## Example 2

OPTAB entry for Oolong assembler is in Figure 3.

Mnemonic	OpCode	B   D
ByteCode Pattern		
<i>GenMap : IFAS <math>\rightarrow</math> ByteCode</i>		
<i>StmtMap : Statement <math>\rightarrow</math> IFAS</i>		

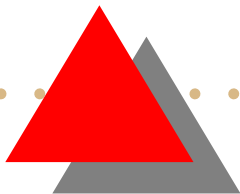
Figure 3: OPTAB entry

**Note:** *B* stands for Byte Code and *D* stands for Directive.




# *Structure of the OPTAB*

- OPTAB is constructed by the assembler designer;
- OPTAB is used by the assembler: for each statement in the assembly program the assembler searches the mnemonic in OPTAB;
- OPTAB is never updated. Efficient search is required.





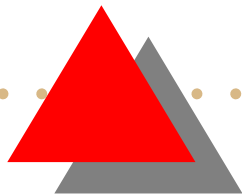


# *OPTAB* implementation

The following are the choices for **OPTAB** implementation:

- array and binary search; complexity:  $\mathcal{O}(\log(OPTAB - length))$
- linked list and linear search; complexity  $\mathcal{O}(OPTAB - length)$
- hash table and direct access.

**Suggested:** array or hash table



# *Symbol table, SYMTAB*

- SYMTAB stores the symbols defined by the user in the assembly program, such as identifiers, constants, labels, etc.
- SYMTAB is dynamic and its length cannot be predicted.
- Hence, choices for SYMTAB implementation are:
  1. array,
  2. linked list,
  3. hash table.

# *SYMTAB implementations*

## Choosing a data structure for SYMTAB implementation:

- Array of records and linear search. This implies estimating a maximum size. Not advisable.
- Linked list and linear search: This is too expensive.
- Hash table and direct access implemented by two tables:
  1. a fixed size table on which hash function operate, and
  2. a variable size table implemented as a linked list starting from each entry of the fixed part.

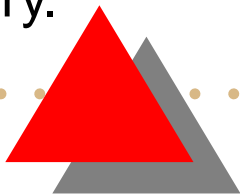


# *Our choice*

Hash table, where hash-function is the symbol type! This means that we will use:

- A fixed size table whose entries are the types of the symbols used in the assembly program. This is called further the Type Definition Table (TDT).
- A variable size table called Object Definition Table (ODT). ODT will be structured as a collection of linked lists, each of which containing all the symbols of a given type.

**Note:** the header of an ODT linked list is the TDT entry of the type that ODT list accumulate. Hence, an ODT list appears as the overflow of a TDT entry.



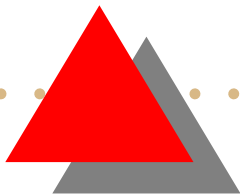


# Fact

TDT contain the `undefined` type.

## Management:

- When a symbol is read by the assembler and its type is determined to be  $t$  it is entered in the ODT list whose header is `TDT[ $t$ ]`.
- When a symbol is read by the assembler and its type cannot be determined, the symbol is stored in the ODT list whose header is `TDT[undefined]`.
- When the type of a symbol in the ODT list whose header is `TDT[undefined]` is determined to be  $t$  the symbol is migrated in the list whose header is `TDT[ $t$ ]`.





# *Rationale*

The rationale for (TDT, ODT) implementations of the SYMTAB are:

- Types are given and define completely the assembly language.
- Collecting all objects of the same type on the same linked list allows us to optimize assembler's space and speed.
- Collecting all objects of the same type on the same linked list allows us to optimize target program's space and speed (think at CFF representing JVM programs).

# Hash table

The hash table implementation of SYMTAB is shown in Figure 4.

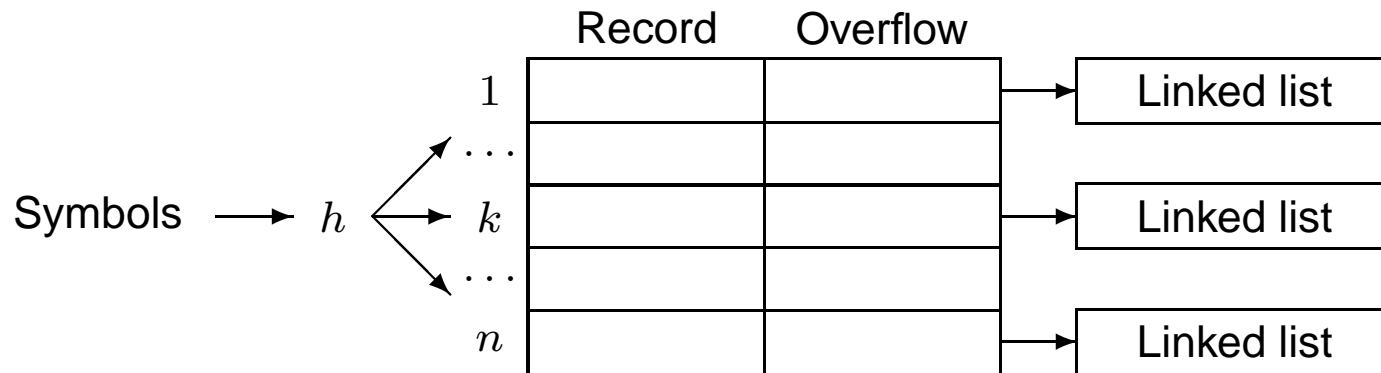


Figure 4: Hash table implementation of SYMTAB

# Customizing the hash table

For every computer platform (including JVM) the hash-table implementations of the SYMTAB is obtained by:

- TDT is filled-out by the assembler constructor using the set of types supported by computer architecture completed with the `undefined` type.
- ODT entry is a standardized structure, characteristic to the ODT entry, that is defined by the assembler constructor and is manipulated by the assembler. That is, the `Variable Part` of the SYMTAB a linked list of symbols of the same type.
- Hash function maps each symbol *Symb* into a tuple  $(TDT(Symb), ODT(Symb))$  where  $TDT(Symb)$  is the TDT entry of the type of *Symb* in TDT and  $ODT(Symb)$  is the place of *Symb* in the linked list of symbols of type  $TDT(Symb)$ .



# *TDT entry*

The structure of TDT entry is shown in Figure 5.

That is, each entry in TDT should have the following fields:

1. Type name
2. Type representation in the target language;
3. The list of operations defined on the type;
4. Type Location Counter, (**TLC**) which shows number of symbols of this type discovered so far;
5. A pointer to the ODT-list holding the first symbol of this type.

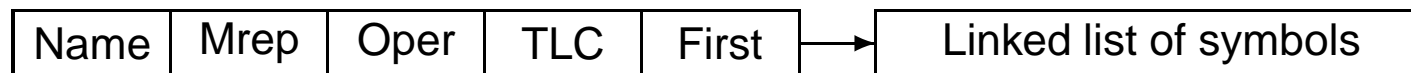


Figure 5: TDT entry

# Components of TDT entry

- Type *name*, such as integer, real, instruction, macro, address, operator, undefined, etc.
- Type representation in machine language, *Mrep*, shows the structure and the number of bytes occupied by a symbol of this type.
- Operations, *Oper*, shows the operations available in assembly language on symbols of this type.
- Type location counter, *TLC*, shows the number of symbols of this type encountered so far in the assembly language program.
- Pointer to ODT showing the first symbol of this type in ODT.



# Facts

1. TDT entry can be customized to the kind of constructs a type supports.
2. **Example Oolong customization:**
  - `TDT[method].Oper` may show the collection of keywords a method can have. `TDT[method].Mrep` may show the structure  $\langle \text{ByteCode}, \text{ArrayOfLocals}[] \rangle$  of the method.
  - Similarly, `TDT[class].Oper` may show the keywords and `TDT[class].Mrep` may show the structure of the Class File Format (CFF).
3. Other customizations may also be defined.

# *Object Declaration Table*

Each symbol in ODT is specified by its assembly language form and its machine language translation.

- Assembly language form of a symbol is specified by its name (a string), its type (an index in TDT), and its value (binary translation of the value)
- Machine language translation of a symbol is specified by: address allocated, size of the object, object usage called mode.

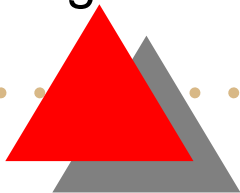


# ODT Customization

ODT can also be customized by the assembler implementer.

**Example Oolong customization:**

- A method may be represented by tuple  $\langle \text{ByteCode}, \text{ArrayOfLocals} \rangle$  where `ByteCode` is a pointer to a stream of characters representing the method byte code and `ArrayOfLocals` is a pointer to the tuple  $(\text{Length}, \text{Locals}[\text{Length}])$  where `Locals` represent arguments and variables as appropriate.
- A class may be represented by a pointer to the CFF containing that class.
- Oolong constants can be represented as specified





# *Usual modes are*

- IM, immediate
- RV, relocatable value
- AV, absolute value
- UO, undefined object,
- XR, externally referenced
- GD, globally defined
- SD, section definition

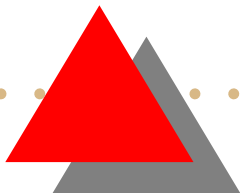


# Relationships

Some symbols may be used to inter-relate various sections of code.

Inter-relationships symbols are usually identified by their modes:

- **GD**, globally defined (or exported), **GR**, globally referenced (or imported);
- **SD**, section definition (the symbol denoting a section of assembly language code is by definition globally defined);
- For Oolong, the keywords representing properties of classes and methods should be interpreted as modes.



# Example

An example TDT and ODT representing the SYMTAB of a conventional assembler is in Figure 6

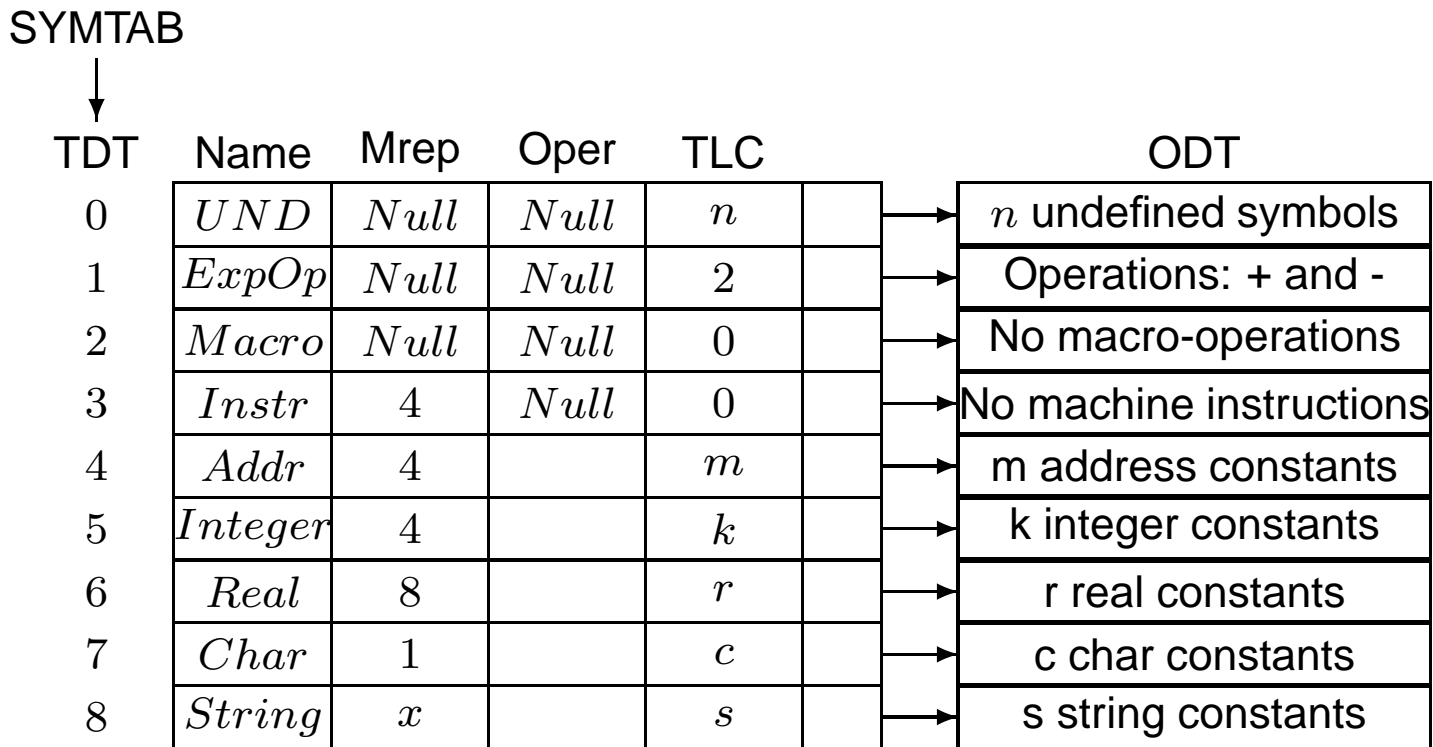


Figure 6: Symbol table implementation





# *Operations on SYMTAB*

The following are the operations performed by the assembler of SYMTAB:

1. Search type: SearchTDT(type);
2. Search symbol: SearchODT(type,symbol), SearchODT(Symbol)
3. Delete symbol: DeleteODT(type,symbol)
4. Enter symbol: EnterODT(type,symbol)
5. Append symbol: AppendODT(type,symbol),  
PrependODT(type,symbol)



# *C expressions*

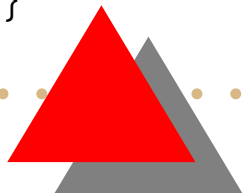
Potential C expressions used in SYMTAB implementation are:

- Assembly language symbol representation:

```
struct Symbol
{
    int kind;
    char AssemblyForm[MaxLength];
};
```

- SYMTAB symbol representation:

```
struct SymtabSymbol
{
    struct TDT *symbolType;
    struct ODT *symbolAddr;
}
```





# IFAS

## Internal Form of Assembly Statement (IFAS)

The following C structure can be used as implementation design for

*FIFentry*:

```
struct FIFentry
{
    struct FIFhead header;
    struct FIFbody body[MaxBodyLength];
};
```

# Structure of FIF entry

- **FIFhead:**

```
struct FIFhead
{
    struct SymtabSymbol *SYMB; /* Label translation, if any */
    struct MNEMONIC *MNEM; /* Mnemonic translation */
    int BodyLength; /* Number of operands */
};
```

- **FIF body:**

```
struct FIFbody
{
    int ExpLength; /* Specifies the length */
    struct ExpElement expression[MaxExpLength];
} DummyExpression;
```



# *More on FIF structure*

Symbols in FIF are represented by pointers to their definition in SYMTAB.

- A symbol in `FIFheadi` has type `SYMTAB-entry` and represents the translations of the label;
- A mnemonic in `FIFhead` has type `OPTAB-entry` and represents the mnemonic translation;
- The body of the `FIFentry` is the postfix form of the expression operand.

# Postfix representation

The postfix form of an expression is an array of `MaxBodyLength` expression elements.

- Each expression element is described by:

```
struct ExpElement
{
    char type; /* Distinguishes the union element */
    union element ExpressionElement;
};
union element
{
    struct OBJSymbol *ToObject;
    struct TDTSymbol *ToType;
    char OPERATOR [MaxOpLength];
};
```

# FIFentry

The graphic picture of FIF entry is shown in Figure 7.

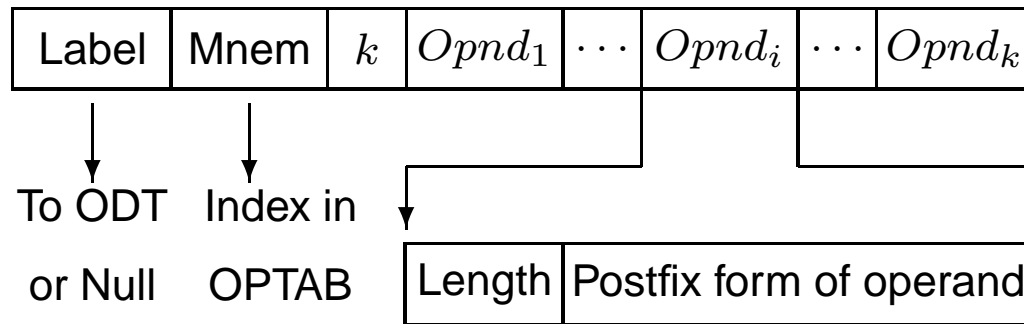
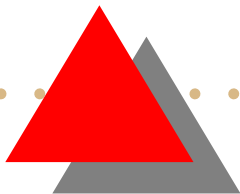


Figure 7: FIFentry



# *Pass<sub>1</sub>*

- Read assembly program statement by statement;
- For each statement perform:
  1. Constructs the `FIFhead` with a `BodyLength` zero and write it in `FIFentry`.
  2. Search for the next operand specifier. If found, translate the expression specifying the operand into postfix form, write the postfix form of the expression into the `DummyExpression`, and update `BodyLength` in `FIFentry`.
  3. If end of the statement is encountered writes the `FIFentry` into the FIF.







# Observations

- `DummyExpression` is constructed by translating the actual symbols, storing them in the appropriate tables, and writing in `DummyExpression` pointers to the symbol translation.
- When  $Pass_1$  reaches the end of the source program all generators are translated.
- The  $Pass_1$  call a memory allocation function before passing the control to the  $Pass_2$ .

# *C expression of Pass<sub>1</sub>*

```
#define Symbols /* Maximum number of user defined symbols */
AssemblerPass_1 (FILE *source, FILE *target)
{
    struct AssemblyStatement ALS, *APC;
    struct FIFentry IFS, *FPC = &IFS;
    struct SymTabEntry SymTab[Symbols];
    APC = read (ALS, source);
    while (APC->Opcode != End)
        {
            map_1 (APC, IFS, SymTab);
            update(FPC);
            write (FPC, target);
            APC = read (ALS, source);
        }
}
```



# *Pass<sub>1</sub>, continuation*

```
map_1 (APC, IFS, SymTab);  
update(FPC);  
write (FPC, target);  
Memory allocation;  
Construct ESD;  
}
```

**Note:** ESD collects information about locally referenced and globally defined symbols.

For Oolong such symbols are method and class names, and thus ESD is the Method Table, MT.

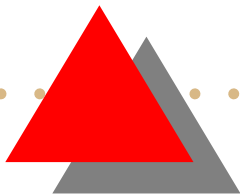


# *Pass<sub>2</sub>*

*Pass<sub>2</sub>* is called by the *Pass<sub>1</sub>*.

- Read FIF statement by statement;
- For each statement read from FIF, *Pass<sub>2</sub>* maps it into machine language code.
- *Pass<sub>2</sub>* completes the translation by generating the Machine Object Module,  $MOM = \langle ESD, Text, RLD \rangle$

**Note:** RLD is a directory of relocatable constants encountered in the program.



# *Oolong MOM*

External symbol dictionary, ESD, in Oolong is Method Table, MT.

- MT must be constructed such that it support inheritance and method overriding
- There are no relocatable constants in Oolong.
- Text is the Class File Format, CFF

Hence,  $OolongMOM = \langle MT, CFF, \emptyset \rangle$



# *C expression of Pass<sub>2</sub>*

```
AssemblerPass_2 (FILE *Target, FILE *MachineObject)
{
    struct FIFentry  IFS, *FPC;
    struct MachineInstruction  MLS, *MPC = &MLS;
    struct SymTabEntry *SymTab = &SYMTAB;
    write (ESD, MachineObject);
    FPC = read (IFS, Target);
    while (FPC->FIFhead->MNEM != End)
        {
            map_2  (FPC, MLS, SymTab);
            update (MPC);
            write  (MLS, MachineObject);
            Construct RLDentry if any;
            FPC = read (IFS, Target);
        }
```

# *Pass<sub>2</sub>, continuation*

```
map_2 (FPC, MLS, SymTab);  
update(MPC);  
write (MPC, MachineObject);  
write (RLD, MachineObject);  
}
```

# Machine Object Module

$MOM = \langle ESD, Text, RLD \rangle$

- The ESD is the external symbol dictionary  
A symbol in the assembly program is written in the ESD if it is exported or imported; ESD is used by the memory allocation routine and by the loader/linker
- Text is the binary form of the instruction and data words generated by the assembler
- Relocatable code is the machine language code that depends upon the memory area where program is loaded for execution; it is recorded in the Relocation and Linking Directory, RLD.