The compiler

By: Yehia M. Abu Eita

Outlines

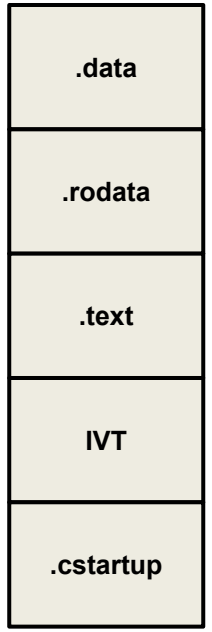
- Introduction
- Memory sections
- Compiler design
- Front end analysis
- Back end synthesis
- Object files

Introduction

- The Compiler is a **program** that **converts your code** into **machine language** code after **doing some checks**.
- **Input**: **.i** file.
- **Output**: **.o** file, **object** file.
- **Operations**:
  - **Makes lexical analysis**
  - **Makes syntax analysis**
  - **Makes semantic analysis**
  - **Code generation**
  - **Code optimization**
- **Syntax errors** and **warning** are generated by the compiler.

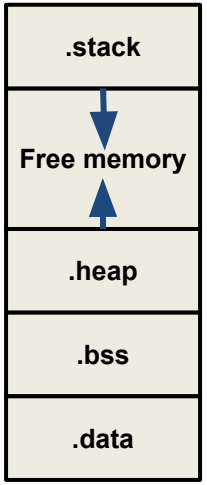
Memory sections

- **Program memory**:
  - This memory is used to store the program code, global, static, constant variables and literals.
- **Program memory is divided into**:
  - **.data segment**: This segment is used to store initialized global and static variables.
  - **.rodata segment**: This segment is used to store constant variables and literals.
  - **.text segment**: This segment is used to store the binary converted code.
  - **IVT**: This segment stores the interrupt vector table.
  - **.cstartup**: This segment stores the startup code.



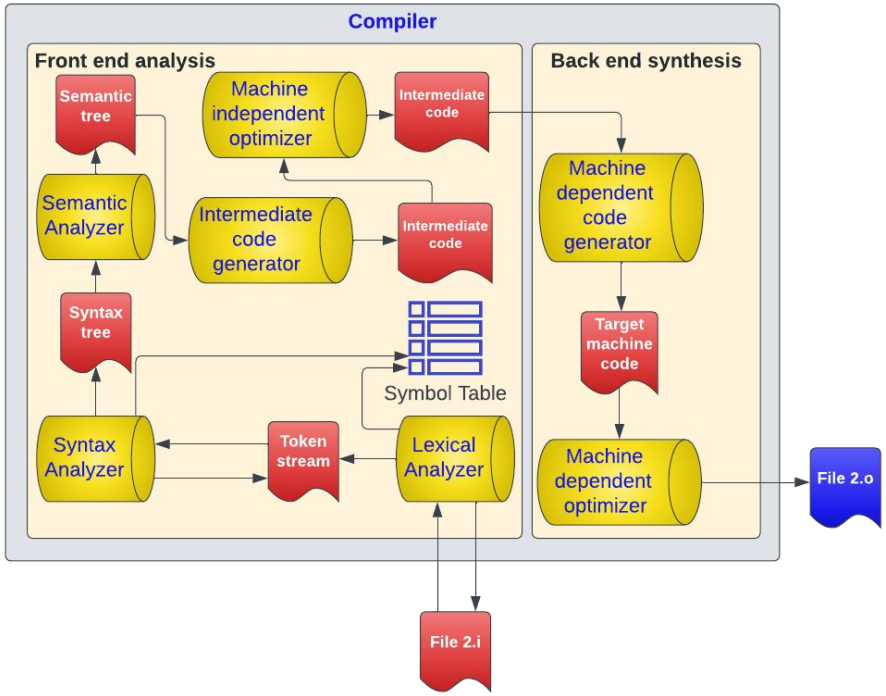
Memory sections

- **Data memory**:
  - This memory is used to store **local**, **dynamic variables** and has a **copy of global and static** section from the program memory.
- **Data memory is divided into**:
  - **.stack**: This segment is used to store the local variables and it expands in the free memory.
  - **Free memory segment**.
  - **.heap**: This segment is used to store dynamic variables and it expands in the free memory.
  - **.bss**: This segment is used to store uninitialized global and static variables.
  - **.data**: This segment is used to store initialized global and static variables (copy of .data in program memory).



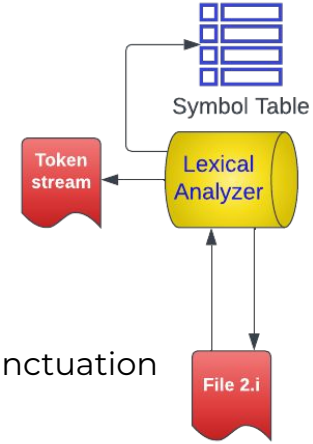
Compiler design

- Compiler consists of two stages:
- **Front end analysis**:
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis
  - Intermediate code generation
  - Intermediate code optimization
- **Back end synthesis**:
  - Machine dependent code generation
  - Machine dependent code optimization



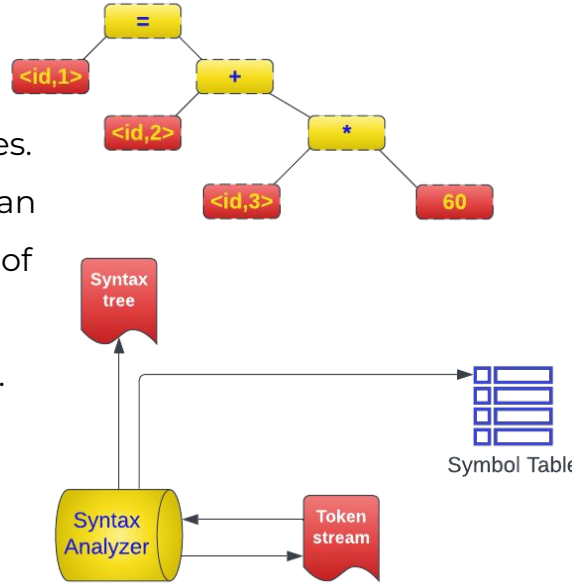
Front end analysis

- **The lexical analyzer**:
  - The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.
  - For each lexeme, the lexical analyzer produces as output a token of the form:
    - `<token-name, attribute-value>`
- **Example:**
  - `position = initial + rate * 60;`
  - `<id,1> <=> <id,2> <+> <id,3> <*> <number,60> <;>`
  - `<id,pointer>` **represents the id of the lexeme into the symbol table**
  - **Types of tokens: Keywords**, **operators**, **identifiers**, **constants**, **literal strings**, punctuation **symbols**(such as commas,semicolons)



Front end analysis

- **The syntax analyzer**:
  - It **parses** and **validate** tokens according to **compiler grammar** rules.
  - It generates a **syntax tree** in which each **interior node** represents an **operation** and the **children** of the node represent the **arguments** of the operation.
  - **Storing** data related to **variables** and **functions** into **symbol table**.
- **Example:**
  - **Input token stream**:`<id,1> <=> <id,2> <+> <id,3> <*> <number,60> <;>`



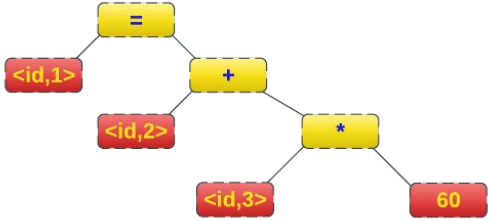
Front end analysis

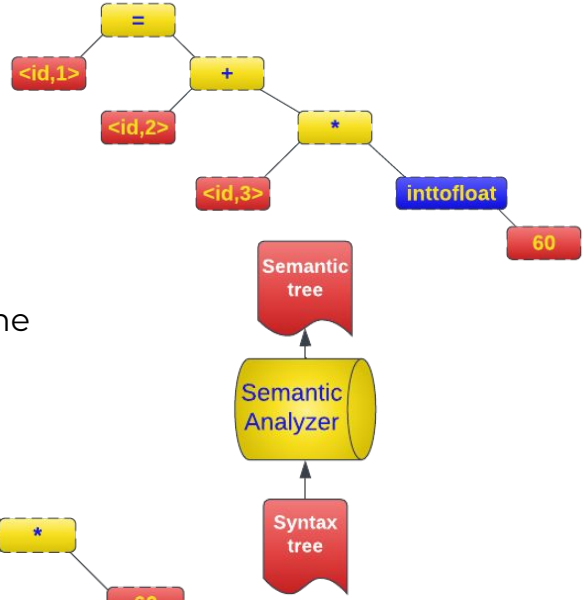
- **The symbol table**:
  - The symbol table is a **data structure** containing a record for each variable name, with fields for the **attributes of the name**.
  - **Variable attributes** may provide information about the **storage** allocated for a name, its **type**, its **scope** (where in the program its value may be used).
  - **Function attributes** provide **names**, **number** and **types** of **arguments**, the method of **passing** each argument (for example, **by value** or **by reference**), and the **type returned**.
- **Example:**
  - `float position, initial, rate;`
  - `<keyword,float> <id,1> <,> <id,2> <,> <id,3> <;>`

id	symbol	type	Scope
1	position	float	local
2	initial	float	local
3	rate	float	local

Front end analysis

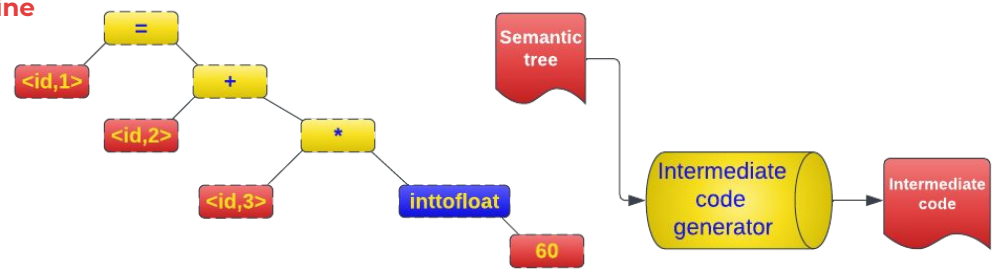
- **The semantic analyzer**:
  - It uses the syntax tree and the information in the symbol table to **check** the source program for **semantic consistency** with the language definition.
  - **An important** part of semantic analysis is **type checking**, where the compiler checks that each operator has matching operands.





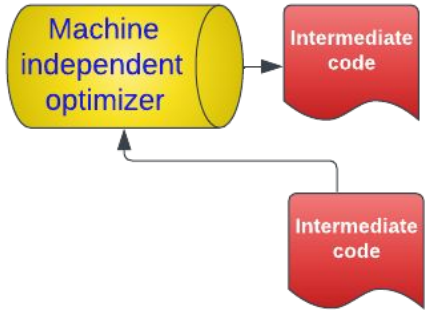
Front end analysis

- **Intermediate code generator**:
  - In the process of **translating a source program into target code**, a compiler may **construct** one or more **intermediate representations**, which can have a variety of forms.
  - **Syntax trees are a form of intermediate representation**.
  - This intermediate representation should have **two important properties**:
    - **Easy to produce.**
    - **Easy to translate into the target machine**
- **Example**:
  - `t1 = inttofloat(60)`
  - `t2 = id3 * t1`
  - `t3 = id2 + t2`
  - `id1 = t3`



Front end analysis

- **Machine independent optimizer**:
  - The machine-independent code-optimization phase attempts to **improve** the **intermediate code** so that better target code will result.
  - In this stage optimization may be for **speed**, **code reduction**, or for **memory saving**.
- **Optimize means**:
  - **Remove unused variables**
  - **Remove dead code**
  - **Loop unrolling**
  - **Reduce unnecessary operations**

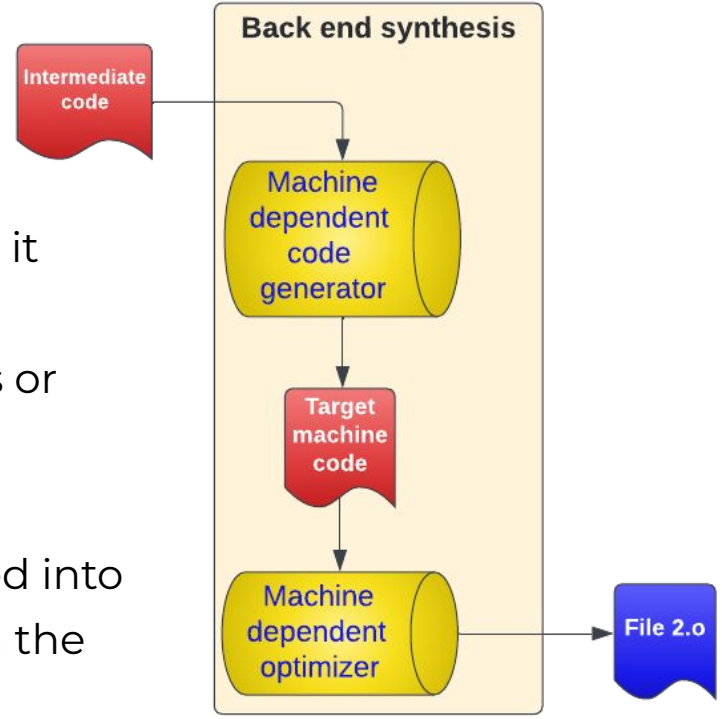



```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Back end synthesis

- The code generator takes the intermediate representation of the source program and maps it into the target language.
- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.



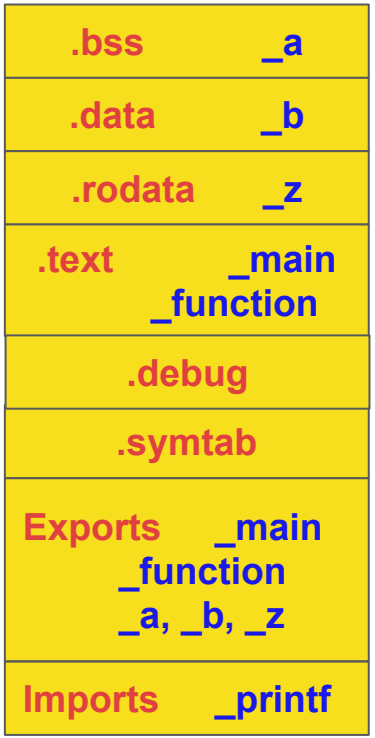
Object files

- It is a **machine language** representation of the compiled .c file.
- The compiler gives a **logical addresses** for all variables.

```
int a;
int b = 20;
const int z = 30;

int function(int *ptr, int x)
{
        int y = 5;
        static int m = 10;
        .........
        return d;
}

int main()
{
        int x = 5, *ptr = &x;
        function(ptr, x);
        printf("Hello");
}
```



Summary

- Now you have good understanding about the C compiler.
- You have learned how compiler front end works, starting from lexical analysis to code generation.
- Remember that syntax errors are generated by the compiler in syntax and semantic analysis stages..
- Remember that warnings are generated from the semantic analysis stage.