Compiler Design

Fatemeh Deldar

Isfahan University of Technology

1402-1403

References

1. **Compilers: Principles, Techniques, and Tools** (Second Edition), Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Addison-Wesley, 2007.

2. **Modern Compiler Design** (Second Edition), D. Grune, H. Bal, C. Jacobs, and K. Langendoen, John Wiley, 2012.

Syllabus

- **Introduction**
- **Lexical Analysis**
- **Syntax Analysis**
- **Semantic Analysis**
- **Intermediate-Code Generation**
- **Run-Time Environments**
- **Code Generation**
- **Machine-Independent Optimizations**

Introduction

Introduction

- Programming languages are notations for describing computations to people and to machines
- Before a program can be run, it first must be **translated** into a form in which it can be executed by a computer
  - **The software systems that do this translation are called compilers**
- **This course is about how to design and implement compilers**

-
کاری که کامپایلر انجام میده اینه که این ترجمه رو انجام بده
این ترجمه این که از چی به چی ترجمه بشه حالت های مختلف پیش میاد براش

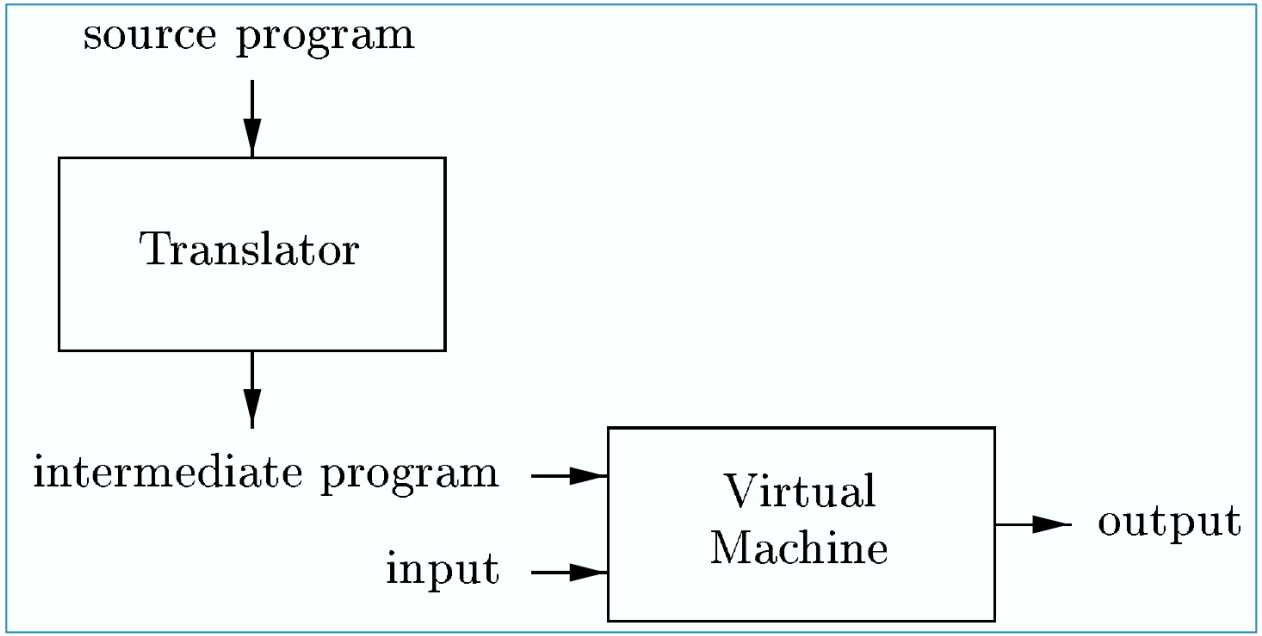
Compiler vs. Interpreter

- **A compiler** is a program that can read a program in one language (**the source language**) and translate it into an equivalent program in another language (**the target language**)
- **An interpreter** directly executes the operations specified in the source program on inputs supplied by the user
- ***The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs***
- ***An interpreter can usually give better error diagnostics than a compiler, because it executes the source program statement by statement***

-
کامپایلر نقطه مقابل مفسر است

Compiler vs. Interpreter

- **Example**
  - Java language processors combine compilation and interpretation
    - A Java source program **first be compiled** into an intermediate form called bytecodes
    - The bytecodes are **then interpreted** by a virtual machine



-
کامپایلر یک زبانی می خواد بگیره و یک زبان مقصد به ما تحویل بده حالا اگر این زبان مقصد اگر زبان ماشین باشه کل فرایند ترجمه انجام شده و کد می تونه بره اجرا بشه توی رم
اگر یک زبان میانی باشه این خودش باید دوباره ترجمه بشه به زبان ماشین

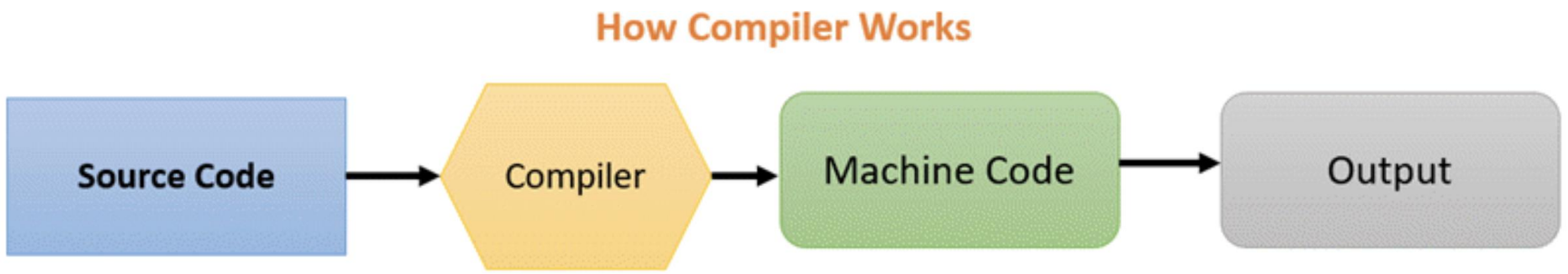
ولی مفسر به صورت خط به خط پیش میره: جمله به جمله ترجمه کردن و اجرا کردن

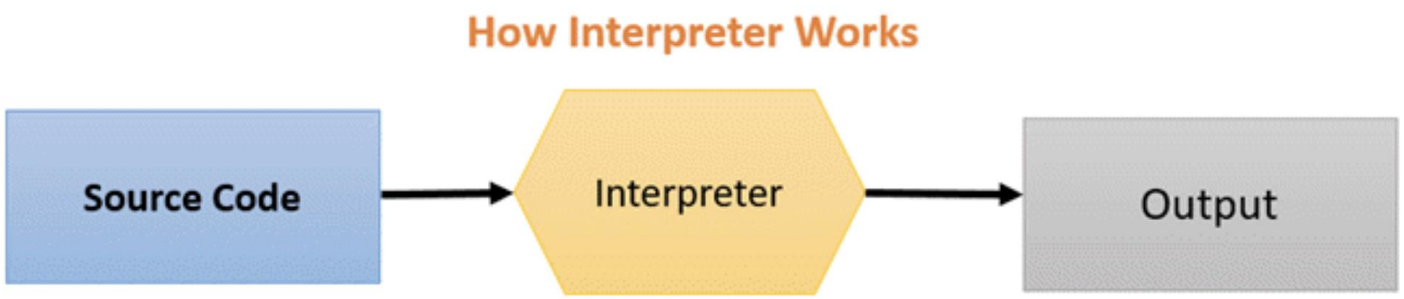
کامپایلر کاری که انجام میده از این جهت که کد رو کامل ترجمه می کنه ب یک زبانی ینی زبان نزدیک سخت افزار و بعد اجرا میکنه پس سریعتر است موقع اجرا نسبت به مفسر

مفسر وقتی که داره خط به خط برنامه رو ترجمه می کنه و داره اجرا میکنه می تونه بهتر خطاهای مربوط به زمان اجرا رو نشون بده

کامپایلر بیشتر استفاده میشه نسبت به مفسر

Compiler vs. Interpreter





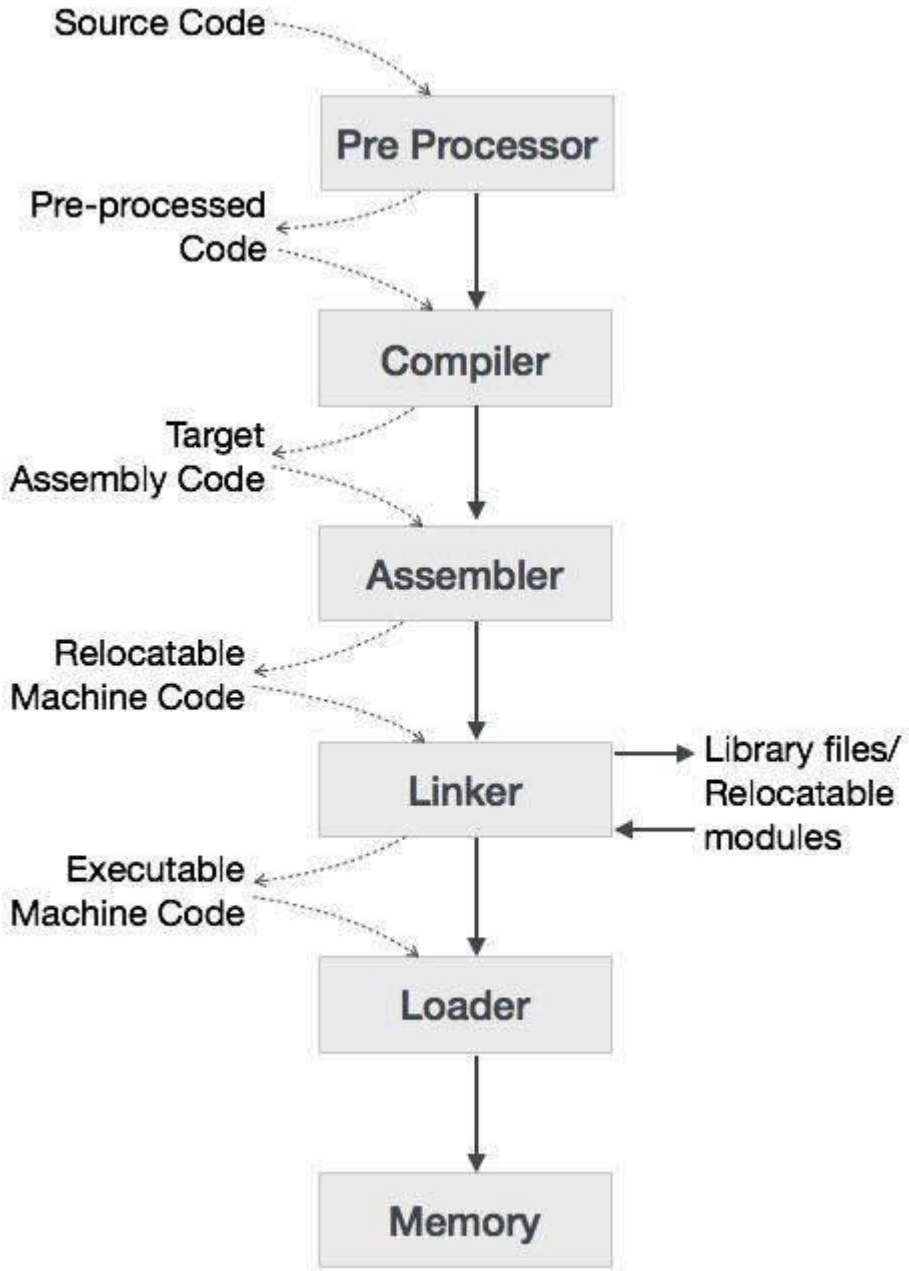
مفسر یک کامپایلری است که داره خط به خط کار میکنه ینی خط به خط به زبان ماشین تبدیل میشه

از یک لحاظی هم مفسر سرعتش تندتر از کامپایلر است مثلا اگر ما خط 10000 رو بخوایم از یک برنامه 1 میلیون خط مفسر میاد خط به خط می ره جلو تا به این خط برسه ولی توی کامپایلر تا 1 میلیون خط رو اجرا نکنه و کامپایلش نکنه نمی تونیم به اون خط 10000 برسیم پس اینجا کامپایلر کندتر است (توی این حالت)

کامپایلر بهینه تر است و توی مفسر خیلی از اون بهینه سازی هایی که توی کامپایلر داریم نمی تونه اینجا انجام بشه

A Language-Processing System

- **Preprocessor**
  - Collecting the source program
- **Compiler**
  - Producing an assembly-language program
- **Assembler**
  - Producing relocatable machine code
- **Linker**
  - Resolving external memory addresses, where the code in one file may refer to a location in another file
- **Loader**
  - Putting together all of the executable object files into memory for execution



-
جایگاه کامپایلر توی یک سیستم پردازش زبان برنامه نویسی:
Preprocessor: یک سری پردازش های اولیه اینجا انجام میشه و یک کد مرتب تری تحویل کامپایلر داده بشه مثلا توی زبان سی میشه همون بحث اضافه کردن کتابخونه ها میشه --> در واقع میاد جمع میکنه اون سورس کد ها رو و یک جا جمع میکنه و یک سورس کد یکدستی رو به وجود میاره و تحویل کامپایلر میده چون کامپایلر کارش اینه که از خط اول تا خط اخر بخونه و ترجمه کنه
Compiler: کد میانی تولید میشه
نکته: کامپایلرها از Assembler ها جدا هستن بخاطر این است که فرایند طراحی راحتر باشه و هم اینکه تعداد کمتری کامپایلر لازم باشه --> اگر کامپایلر نمی خواست از Assembler جدا باشه ما باید روی سخت افزارهای مختلف باید برای هر زبانی کامپایلرهای مختلف می داشتیم که اون دقیقا بره به زبان اون سخت افزار ترجمه کنه و علاوه بر اینکه تعداد کامپایلرها رو خیلی بیشتر می کرد پیچیدگیش هم خیلی بیشتر می کرد و به این راحتی هم نمیشد کامپایلر طراحی کرد

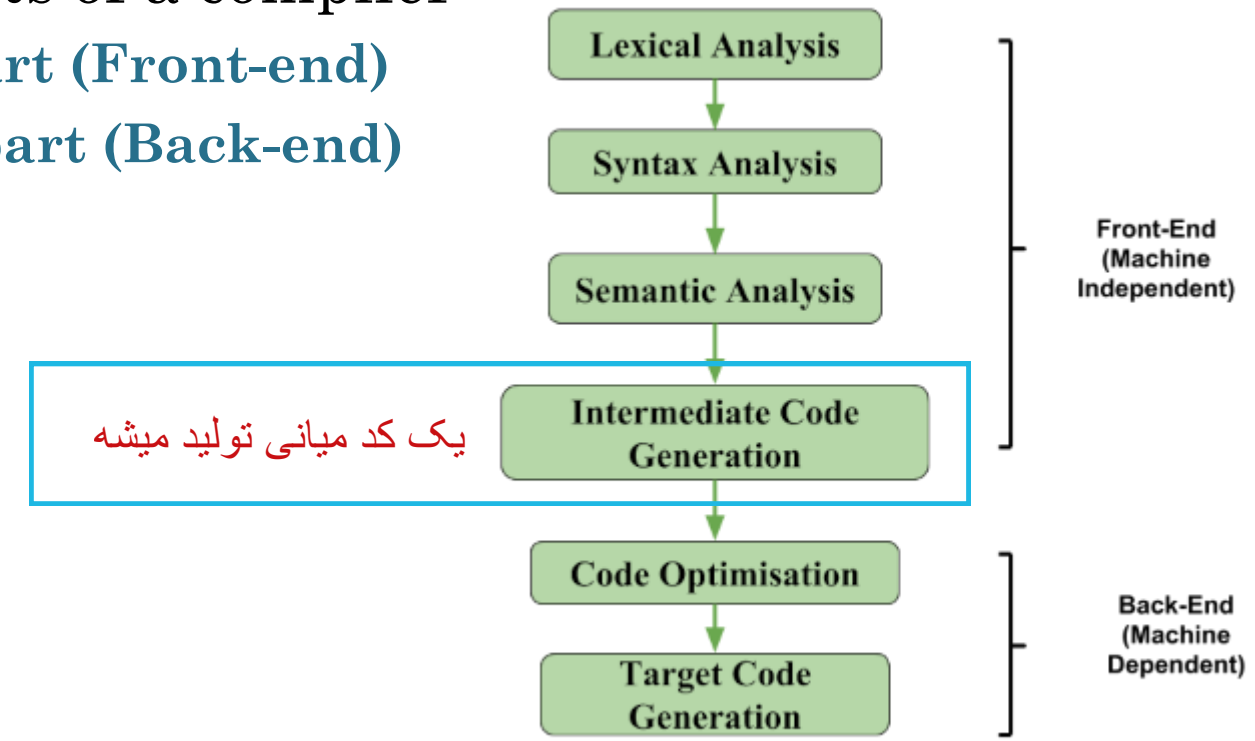
Assembler: زبان خود ماشین رو تولید میکنه
کارهای بعد از این مربوط به این میشه که این بره برای اجرا

Linker: مواردی که لازم است لینک بشه به برنامه مثلا کتابخونه های زمان اجرا و ماژول های مورد نیازو.... اینارو لینک میکنه به برنامه

Loader: لود میکنه و میاره توی رم قرار میده برنامه رو

The Structure of a Compiler

- Two general parts of a compiler
  - **The analysis part (Front-end)**
  - **The synthesis part (Back-end)**



این 6 مرحله مراحل اصلی کامپایلر است :
که خود این 6 مرحله به دو بخش اصلی فرانت اند و بک اند شکسته میشه
6 مرحله:
1- تحلیل لغوی
2- تحلیل نحوی
3- تحلیل معنایی
اکثر خطاهای کامپایلری مربوط میشه به این سه تا فاز
این سه تا مرحله که رد بشه بعد از این سه تا کد میانی اولیه که مد نظر است تولید میشه
نکته: این کد میانی که اینجا تولید میشه با اون کد میانی که توی پردازش زبان داشتیم ینی بعد از کامپایلر داشتیم، این با اون فرق داره و این کد میانی اینجا منظورش داخل خود این کامپایلر است قبل از اینکه اصلا کد اسمبلی تولید بشه --> پس این کد میانی که فاز 4 ام است این منظور اینجا اسمبلی نیست چون اسمبلی فاز اخر میشه اینجا --> این منظور اینه که خود کامپایلر بعد از اون تحلیل های اولیه که انجام داد ینی اون سه تا فاز می تونه برای خودش یک کد میانی تولید کنه
نکته: با این کد میانی که اینجا تولید شده می تونه بهینه سازی رو انجام بده ینی اگر یک تیکه از کد استفاده نمیشه اونو حذف کنه و یا اگر یک تیکه از کد رو میشه ساده تر کرد ساده ترش میکنه

The Structure of a Compiler

- Two general parts of a compiler
  - **The analysis part**
    - This part **breaks up the source program** into constituent pieces and **imposes a grammatical structure** on them
    - It then uses this structure to **create an intermediate representation of the source program**
    - The analysis part also collects information about the source program and stores it in a data structure called a **symbol table**
  - **The synthesis part**
    - The synthesis part **constructs the desired target program** from the intermediate representation and the information in the symbol table

-
بخش اول میخواد برنامه رو بگیره و تا حد امکان همه خطاهاش رو دربیاره و اونو بهبود بده از نظر خطاهای کامپایلری
بعد از اون کامپایلر از اون خطاهایی که سمت کاربر بوده و می تونسته رفعشون کنه ینی داده به کاربر خطا رو و کاربر اونا رو رفع کرده
و توی پارت دوم داره بهینه سازی رو انجام میده

symbol table: این جدول نماد یک دیتا استراکچر اصلی طراحی کامپایلرها است و همه کامپایلرها هم اینو دارن و توی همه فازها هم استفاده میشه --> symbol table بین همه فازها مشترک است
از فاز تحلیل لغوی که شروع میشه جدول نماد شروع میشه به ساخته شدن و از روی لغت به لغت برنامه جدول نماد ساخته میشه و توی فازهای بعد این جدول نماد یا استفاده میشه یا کامل میشه

The Structure of a Compiler

- **Lexical Analysis**
  - The first phase of a compiler is called lexical analysis or scanning
  - 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>***

An abstract symbol that is used during syntax analysis

Points to an entry in the symbol table for this token

-
اولین کاری که کامپایلر انجام میده اینه که یک برنامه می گیره که از نظر اون، یک رشته ای از کاراکترهاست --> هر کاراکتری: یا اسپیس است یا حرف است یا ... --> یکسری کاراکتر رو پشت سر هم می بینه --> اول از همه کامپایلر باید بتونه اینارو از هم تفکیک کنه ینی کلمه اول رو از کلمه دوم و متوجه بشه که اینا چی هستن
به هر لغتی که استخراج میشه lexeme گفته میشه مثلا int میشه خودش یک lexeme و پرانتز باز هم میشه یه دونه lexeme و...
مثلا اسم متغییر با عدد نمیتونه شروع بشه و اگر یک متغییر تعریف کردیم که اولش عدد است اون خطایی که کامپایلر به ما میده توی این فاز تولید میشه
این فاز فقط میخواد کلمه به کلمه رو بررسی کنه
توی این فاز جدول نماد ساخته میشه ینی هر کلمه رو که می خونه بیاد یه دونه سطر رو به اون جدول اضافه کنه البته همیشه این اتفاق نمی افته

توی هر سطری از جدول نماد تقریبا یک زوجی به صورت * نگهداری میشه --> یه دونه token نگهداری میشه توی هر سطرش که اون token یک اسم داره و یکسری attribute داره که این attribute ممکنه یه دونه باشه یا چندتا باشه
اسم token به عنوان اولین مولفه ذخیره میشه

The Structure of a Compiler

- **Lexical Analysis**
  - **Example**

position = initial + rate * 60

$\langle \mathbf{id}, 1 \rangle \langle = \rangle \langle \mathbf{id}, 2 \rangle \langle + \rangle \langle \mathbf{id}, 3 \rangle \langle * \rangle \langle 60 \rangle$

خروجی تحلیل لغوی همچین چیزی میشه و بعد وارد فاز بعدی میشه

  - **Example: Lexical errors**
    - int 5temp;
    - a = 123.23.45;
    - char s[10] = “ali;

معمولا اسم متغییرها و اسم ثابت و همه چیزهایی که کاربر تعریف می کنه به اسم همون id توی کامپایلرها است

این دستور الان شده 7 تا token
و token می تونه فقط یک بخش داشته باشه مثل مساوی یا می تونه دوبخشی باشه
توی دوبخشی ها ارگومان دوم میشه شماره سطر:
مثلا برای متغییر position میشه سطر اول جدول نماد ینی سطر اولش مربوط به متغییر position است و اطلاعات مربوط به position توی سطر اول قرار داره

خروجی این فاز میشه یکسری token

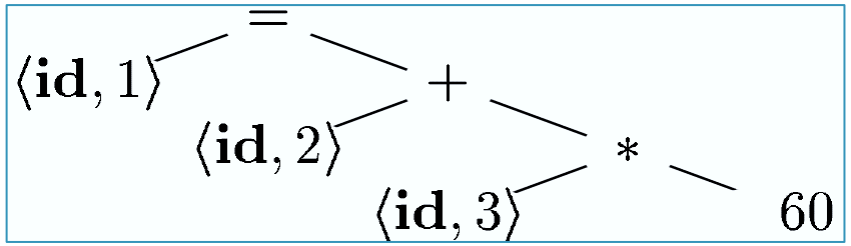
نکته: لازم نیست برای همه tokenها اطلاعات ذخیره بشه مثل مساوی یا اپراتورها و اعداد ثابت

نکته: position به اسم id شناسایی میکنه و کلا توی این فاز id رو اسم همه چیزهایی می ذاریم که کاربر می تونه تعریف کنه مثل متغییر یا... و الان اینجا از نظر تحلیلگر لغوی این سه تا متغییر یک تایپ دارند واسمشون به نام id است ولی بقیه اطلاعاتشون که شاید باهام فرق کنه مثلا شاید این اعشاری باشه یا شاید اون صحیح باشه و.. اینا توی جدول نماد ذخیره میشه و این فقط می گه که اینا توی کدوم سطر از جدول نماد قرار داره (ارگومان دومش میگه)

نکته: جدول نماد فقط یکی است

The Structure of a Compiler

- **Syntax Analysis**
  - The second phase of the compiler is syntax analysis or parsing
  - The parser uses the first components of the tokens produced by the lexical analyzer **to create a tree-like intermediate representation that depicts the grammatical structure of the token stream**
  - A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation



-
توی این فاز قراره سینتکس برنامه چک بشه --> این فاز واضح ترین فاز است که کامپایلر انجام میده
خیلی از خطاهای کامپایل مربوط به این فاز است
این فاز برای اینکه بتونه این خطاها رو بگیره و باگ های برنامه رو در بیاره نیاز به این داره که بتونه سینتکس رو درست بسازه که معمولا این رو از طریق یک syntax tree می سازه و این درخت رو تحلیل میکنه و اگر توی ساخت این درخت به مشکل بخوره خطاش رو به کاربر نشون میده

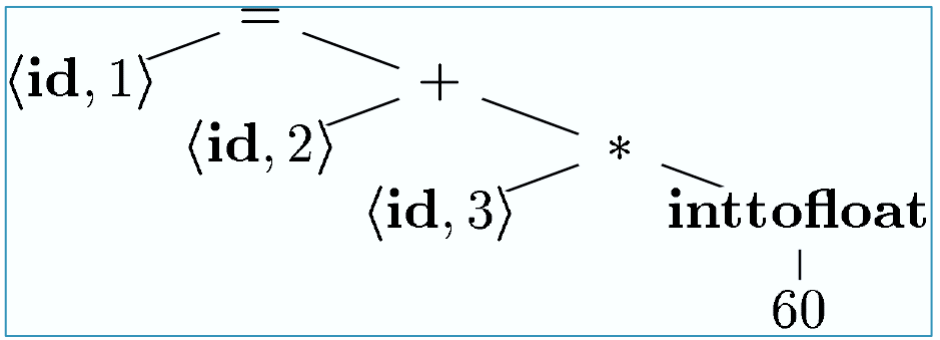
The Structure of a Compiler

- **Syntax Analysis**
  - **Example: Syntax errors**
    - a b =;
    - int q = 6
    - if (a > 1

The Structure of a Compiler

- **Semantic Analysis**
  - The semantic analyzer uses the syntax tree and the information in the symbol table to **check the source program for semantic consistency with the language definition**
  - It also **gathers type information** and saves it in either the syntax tree or the symbol table

ضرب اولویتش بیشتره پس توی درخت پایین تر است پس اونایی که اولویت بیشتری دارند توی درخت پایین ترند



-
فاز سوم:
تحلیل معنایی --> خودش دیتا استراکچر جدیدی رو اضافه نمیکنه --> اون درختی که توی مرحله قبل ساخته شد اون درخت به این داده میشه و این میاد روی این درخت هر اطلاعات دیگه ای که نیاز باشه رو جمع اوری میکنه یا اگر نیاز باشه اضافه میکنه یا...

اینجا یک چیزی که این اضافه کرده به این درخته این است که می خواسته این 60 رو به float تبدیل کنه

توی این فاز اطلاعات مربوط به تایپ رو جمع اوری میکنه که توی فازهای قبلی این کار انجام نمیشد که این یکی از کارهای این فاز است

The Structure of a Compiler

- **Semantic Analysis**
  1. **Type checking**
     - **Example**
       - The compiler must report an error if a floating-point number is used to index an array
  2. **Type casting**
     - **Example**
       - The compiler may convert the integer into a floating-point number
  3. **Redefine variable**
  4. **Check function parameters**

-
کارهایی که توی این فاز انجام میشه عبارتند از:
1- چک کردن تایپ هاست که ببینه معتبر است یا نه مثلا اندیس ارایه نمی تونه عدد اعشاری باشه و میاد اینو چک میکنه که اگر اعشاری بود خطا بده
2- کست کردنشون --> مثلا 60 رو چک کرد که باید اعشاری بشه و مشکلی هم توش نیست پس تبدیلش میکنه به اعشاری و خطایی هم نمیده ولی مثلا اگر اندیس ارایه رو اعشاری بذاریم میاد چک میکنه و اینجا خطا میده و دیگه نمیاد این کست رو انجام بده
3- تعریف های دوباره متغییر --> که این فاز تشخیصشون میده مثلا int x و توی خط بعد گفتیم float x
4- پارامترهای ورودی تابع مثلا تابع رو با یک پارامتر ورودی تعریف کردیم ولی با سه تا پارامتر فراخوانیش کردیم

The Structure of a Compiler

- **Intermediate Code Generation**
  - Many compilers **generate an explicit low-level or machine-like intermediate representation**
  - This intermediate representation should have two important properties
    - It should be easy to produce
    - It should be easy to translate into the target machine
  - Three-address code consists of a sequence of assembly-like instructions with three operands per instruction

این یک شکل استاندارد برای کد میانی که بهش Three-address گفته میشه ینی کد سه ادرسه چون بیشتر از سه تا متغییر توی یک خط نداشته باشه و حتی ممکنه کمتر از 3 تا متغییر توی یک خط باشه ولی حداکثرش سه تا است

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

-

فاز 4:

اغلب کامپایلرها یک کد سطح پایین تقریبا شبیه به اسمبلی رو تولید می کنن ولی اسمبلی نیست برای اینکه بتونن اون بهینه سازی لازم رو انجام بدن و اسون هم باشه

این اخرین فاز قسمت فرانت اند است

The Structure of a Compiler

- **Code Optimization**
  - The machine-independent code-optimization phase attempts to **improve the intermediate code** so that better target code will result
  - There is a great variation in the amount of code optimization different compilers perform

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

-
فاز 5:
این بهینه سازی که انجام میشه روی همین کد میانی است --> ینی کد میانی رو اومده کوتاه ترش کرده
بهینه سازی هایی که انجام میشه صرفا اینا نیست مثلا خیلی هاش از جهت اینه که یک چیزی رو بیاد از کد ما حذف کنه مثلا یک if که هیچ وقت اجرا نمیشه حذف میشه

The Structure of a Compiler

- **Code Generation**
  - The code generator takes as input an intermediate representation of the source program and maps it into the target language

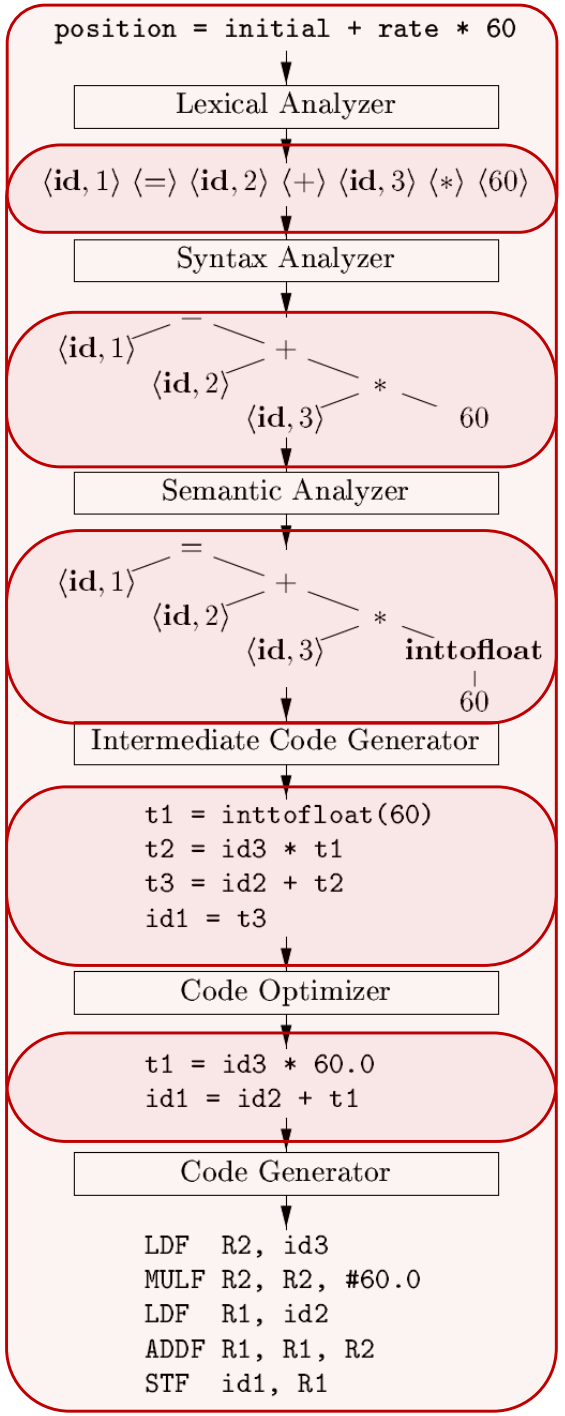
```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2
ADDF R1,  R1, R2
STF  id1, R1
```

-
فاز 6:
تولید کد به اسمبلی اینجا انجام میشه

The Structure of a Compiler

1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE



تنها چیزی که بین همه فازها مشترک است جدول نماد است و این جدول نماد مخصوص فاز خاصی نیست

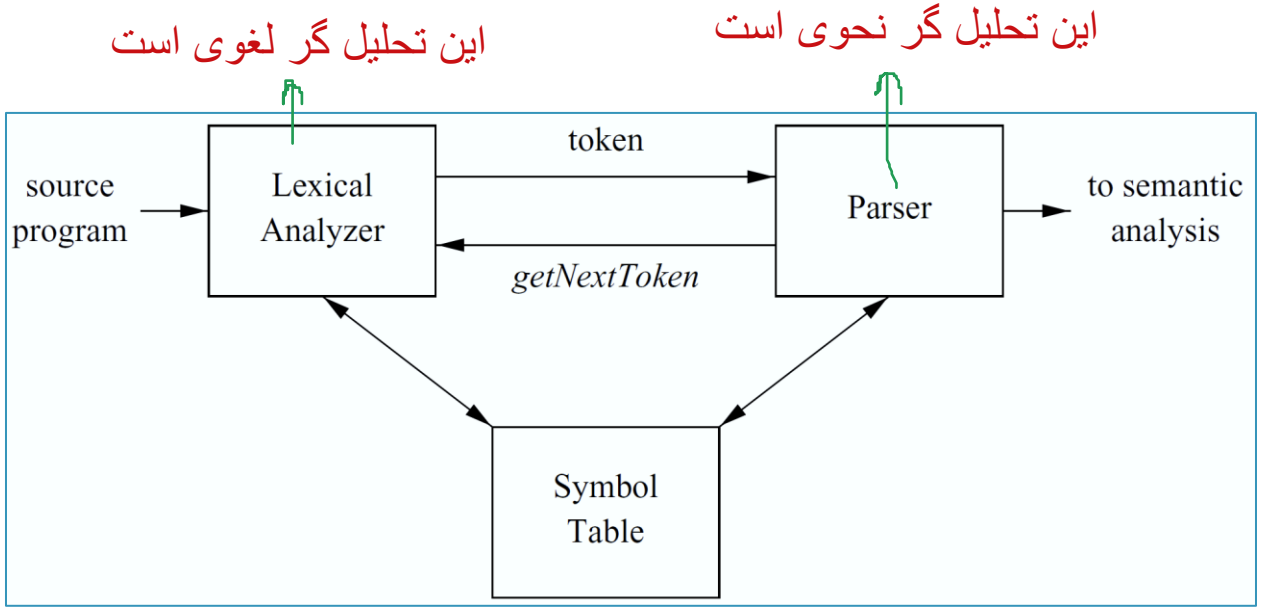
Lexical Analysis

Lexical Analysis

- The main task of the lexical analyzer:
  1. **Read the input characters of the source program**
  2. **Group them into lexemes**
  3. **Produce as output a sequence of tokens**

- **Lexical errors**
  - It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error
  - **Example**

    fi (a == f(x)) ...



-
کار اصلی که تحلیلگر لغوی انجام میده:
1- بخونه کل برنامه رو به صورت یک رشته ای از کاراکترها
2- گروه بندی کنه اونها رو به لغات مختلف که به این لغات lexeme گفته میشه
3- دنباله ای از token ها رو تولید میکنه ینی اون دنباله ای از کاراکترها رو که گرفت تبدیل کنه به دنباله ای از tokenها
غیر از این ها هم مثلا اسپیس های اضافی یا کامنت ها یا ... رو هم حذف میکنه توی این فاز
مثال:
تحلیل گر لغوی بعد رو فقط به ازای یک کاراکتر می بینه مثلا f رو می خونه و i هم می خونه و بعد اسپیس هم می خونه و حالا که می بینه این اسپیس است پس یکی برمیگرده عقب و fi رو یک لغت تشخیص میده و از این به بعد رو نمی خونه و این fi در قالب یک token میده به تحلیل گر نحوی ینی خودش الان به تنهایی نتونست تشخیص بده که این یک خطا است و توی اون رفت و برگشتی که با تحلیل گر نحوی انجام میشه و دراخر که این جمله کامل شد می تونه اینو متوجه بشه که این یک خطا است و اسم متغییر نبوده پس دو تا فاز لغوی و نحوی کاملا با هم در ارتباط هستند

نکته: پرانتز و مساوی و عملگرها هم به عنوان اون جدا کننده می شناسه مثل همون اسپیسی که توی صفحه قبل گفتیم و تا اونجا فقط می خونه ولی اگه اینا رو نداشت داخلش و یه مشت حرف و عدد پشت سر هم بود نه دیگه اینو تا اخرش می خونه

این که f(x) یک تابع است رو تحلیل گر نحوی می فهمه توی این فاز مشخص نمیشه

مثلا الان اینجا تحلیل گر لغوی f رو یک لغت می گیره و پرانتز باز هم همینطور و x هم همینطور و پرانتز بسته هم همینطور پس در کل الان 4 تا لغت برای f(x) داریم --> در کل اینطوری است که f رو می خونه و پرانتز باز هم می خونه و یکی برمیگرده عقب که میشه روی f پس f الان یک لغت است برای خوندن بعدی از پرانتز باز شروع میشه و اینو می خونه و اینم میشه یک لغت و به همین صورت می ره جلو
