اصول طراحی کامپایلر

حسین کارشناس

دانشکده مهندسی کامپیوتر

ترم اول ۹۸ – ۹۷

معرفی

- کامپیوترها
  - پیچیده‌ترین ابزار در دست بشر
  - بکارگیری در امور مختلف زندگی
- مهم‌ترین نرم‌افزارهای مورد استفاده
  - سیستم‌های عامل
  - مرورگرهای شبکه و نرم‌افزارهای ارتباطی و چندرسانه‌ای
  - واژه‌پردازها، صفحات گسترده و سایر نرم‌افزارهای دفتری
  - ...
  - نرم‌افزارها و محیط‌های برنامه‌نویسی

معرفی

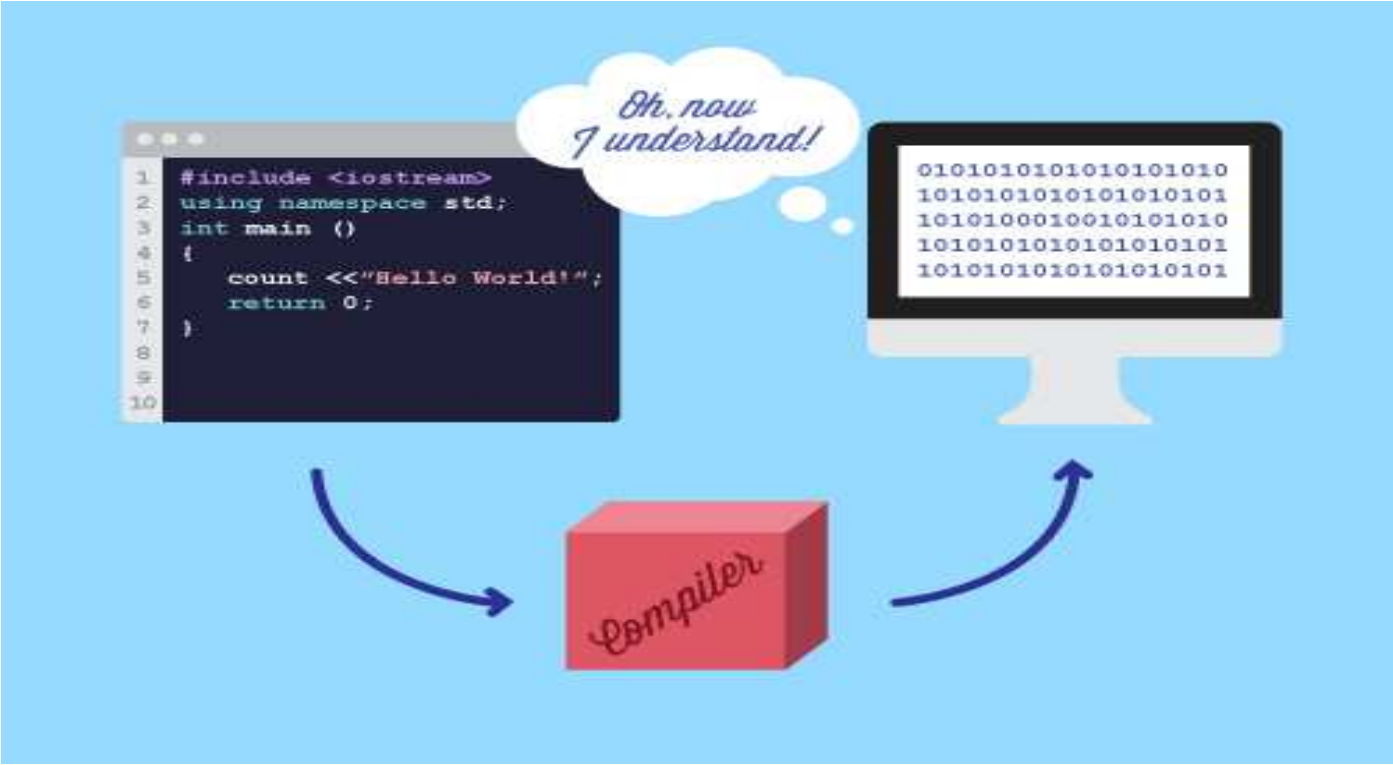
- زبان‌های برنامه‌نویسی (programming languages)
  - مجموعه‌ای از نمادگذاری‌ها
    - جهت توصیف محاسبات برای انسان‌ها و ماشین‌ها
  - برای دادن دستورات و برنامه‌ها به کامپیوترها
  - تمام نرم‌افزارهای با زبان‌های برنامه‌نویسی نوشته شده‌اند

- اهمیت آشنایی با نحوه ایجاد برنامه‌ها
  - امکان ارزیابی دقیق‌تر
  - برنامه‌نویسی مؤثرتر (effective)

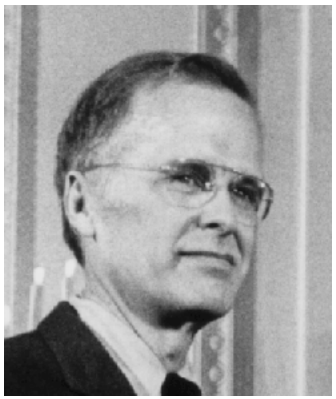
معرفی

- پیاده‌سازی زبان‌های برنامه‌نویسی

کامپایلرها

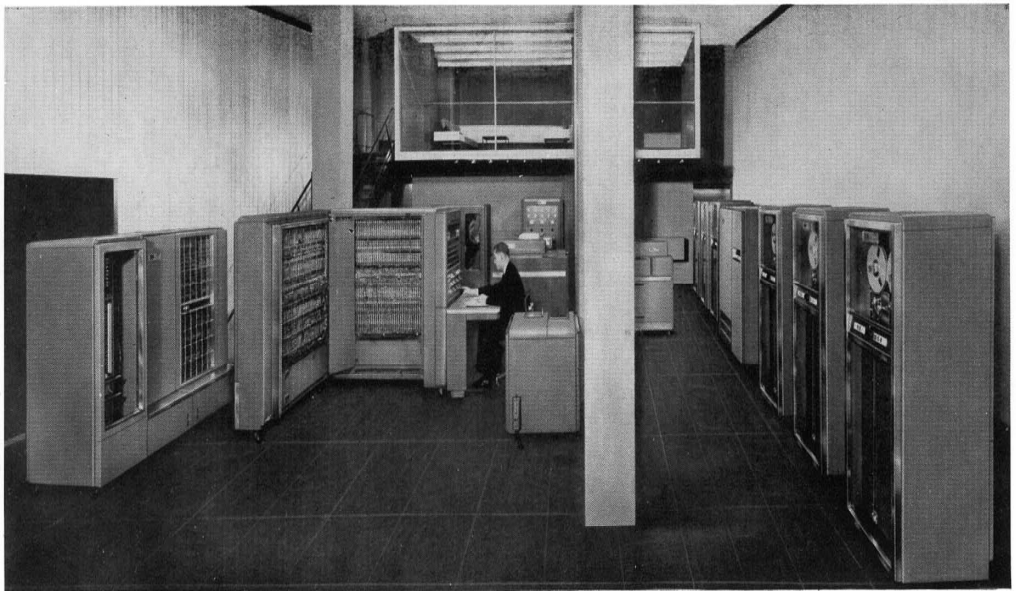


تاریخچه



- کدزنی سریع (Speedcoding)
  - در سال ۱۹۵۳ توسط John Backus معرفی شد
  - برای ماشین IBM 701
  - اولین تلاش‌ها در جهت افزایش بازدهی برنامه‌نویسی
  - یک نمونه اولیه از مفسر (interpreter)
  - مزیت: افزایش سرعت تولید برنامه‌ها
  - معایب
    - برنامه‌های تولید شده ۱۰ تا ۲۰ برابر کندتر از برنامه‌های نوشته شده با دست بودند
    - برنامه مفسر تقریباً ۳۰ درصد حافظه کامپیوتر را اشغال می‌کرد

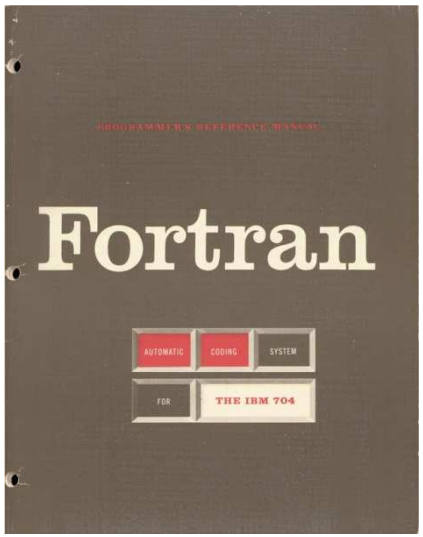
تاریخچه



- ۱۹۵۴: ماشین IBM 704
  - اولین نوع دارای تولید انبوه
  - دارای محاسبات اعشاری
  - قیمت بسیار بالا

- هزینه های نرم افزاری **بیشتر** از هزینه سخت‌افزاری
- نیاز مبرم به افزایش بازدهی برنامه‌نویسی و تسهیل تولید نرم‌افزار

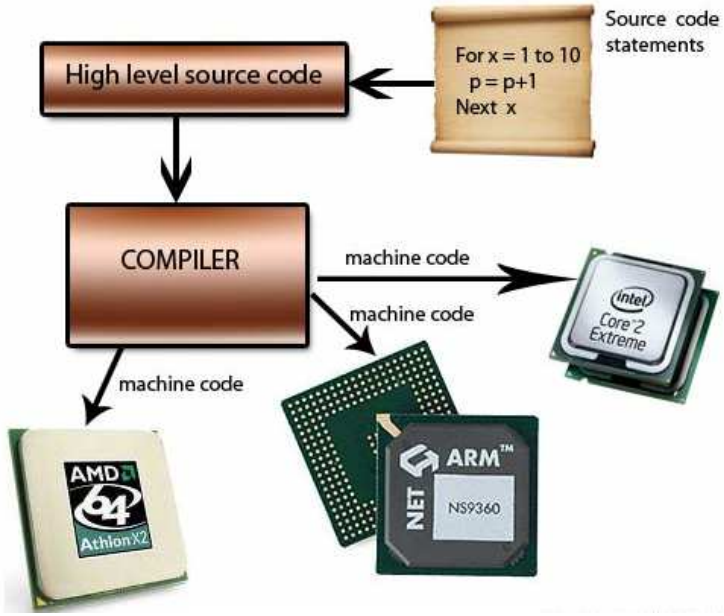
تاریخچه



- Fortran 1
- ادامه کار Backus برای بهبود برنامه‌نویسی
- هدف: نوشتن فرمول‌های علمی جهت اجرا روی ماشین
- تغییر رویکرد: ترجمه کردن فرمول‌ها بجای تفسیر
  - Formulae Translated
- پروژه از سال ۱۹۵۴ تا ۱۹۵۷ طول کشید
- موفقیت: تا سال ۱۹۵۸، ۵۰ درصد برنامه‌ها به این زبان نوشته شده بودند
  - منجر به کارهای تئوری وسیعی در حوزه زبان‌های برنامه‌نویسی شد
- آخرین نسخه: Fortran 2008

تاریخچه

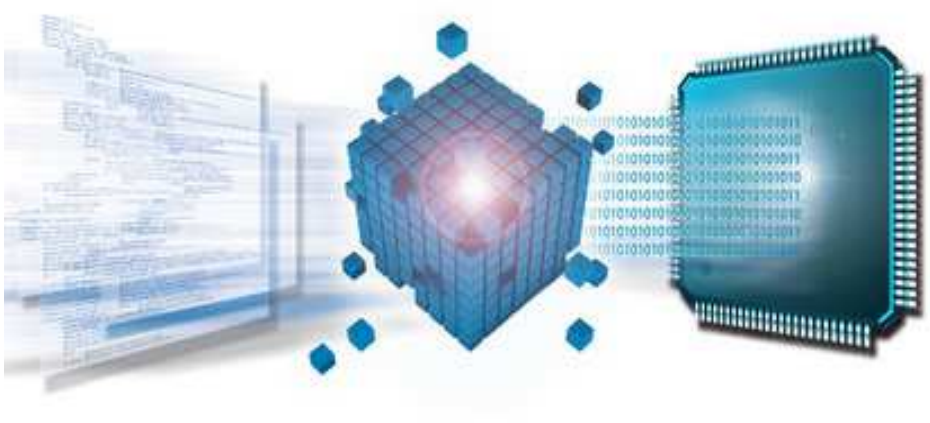
- مزایای مهم Fortran
  - افزایش سطح انتزاع (abstraction) و بهبود بازدهی (productivity) در برنامه‌نویسی
  - استفاده بهتر از ماشین‌های موجود



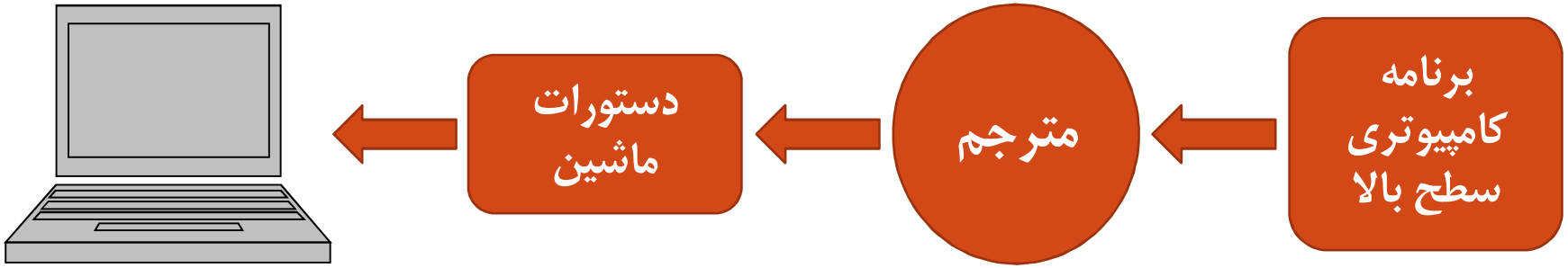
- کامپایلرهای جدید همچنان طرح کلی ساختار این کامپایلر را حفظ کرده‌اند

مقدمه

- اهداف درس
  - معرفی مفاهیم اولیه کامپایلر
  - معرفی ساختار کامپایلرها
  - پیاده‌سازی یک کامپایلر ساده



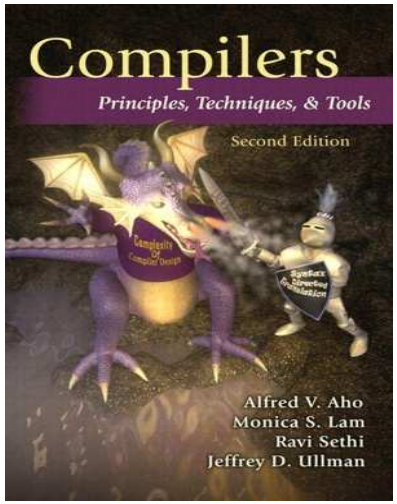
- آشنایی با کامپایلر به عنوان یک نرم‌افزار ترجمه



مقدمه

- منبع اصلی درس

- A. Aho, M. Lam, R. Sethi and J. Ulman, Compilers: Principles, Techniques & Tools, 2nd edition, Addison-Wesley, 2007.

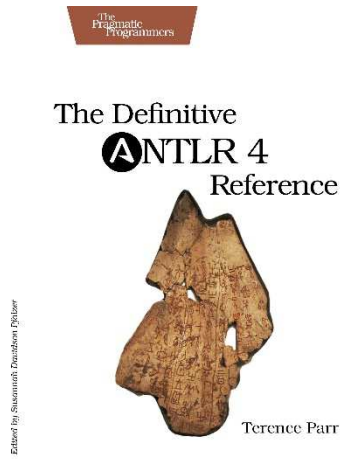


- روح‌اله آل‌شیخ، اصول طراحی و ساخت کامپایلرها، پوران پژوهش، چاپ پنجم، ۱۳۹۲.
- حسین ابراهیم‌زاده، اصول طراحی کامپایلر، سیمای دانش، چاپ پنجم، ۱۳۹۰.
- زارع سلطانی و چگینی، اصول طراحی کامپایلرها، ناقوس، چاپ اول، ۱۳۹۳.

مقدمه

- سایر منابع

- T Parr, The Definitive ANTLR 4 Reference, The Pragmatic Programmers, 2012.



- K. D. Cooper and L. Torczon, Engineering a Compiler, 2nd edition, Morgan Kaufmann, 2012.
- A. W. Appel and J. Palsberg, Modern Compiler Implementation in Java, 2nd edition, Cambridge University Press, 2004.

مقدمه

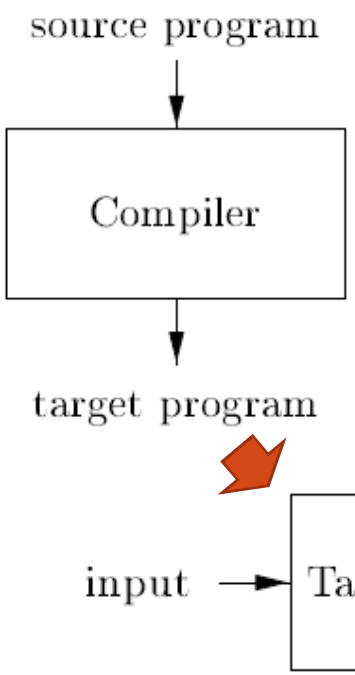
- شیوه ارزیابی
  - امتحان میان‌ترم و پایان‌ترم
  - آزمونچه (quiz)
  - تمرین و فعالیت کلاسی
  - پروژه

- سیاست برخورد با تقلب در کلاس

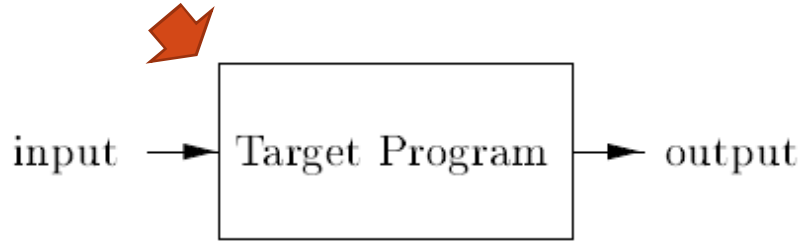
معرفی کامپایلر

پردازشگرهای زبان

- کامپایلر
  - برنامه‌ای که می‌تواند برنامه‌ای به یک زبان (زبان مبدأ) را بخواند و آن را به برنامه‌ای معادل (از نظر معنایی) در زبان دیگر (زبان هدف) ترجمه کند
    - کشف خطاهای برنامه ورودی در فرآیند ترجمه

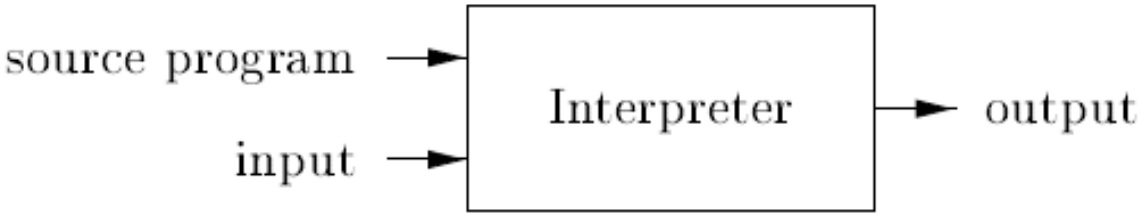


- اجرای برنامه هدف: پردازش ورودی و تولید خروجی



پردازشگرهای زبان

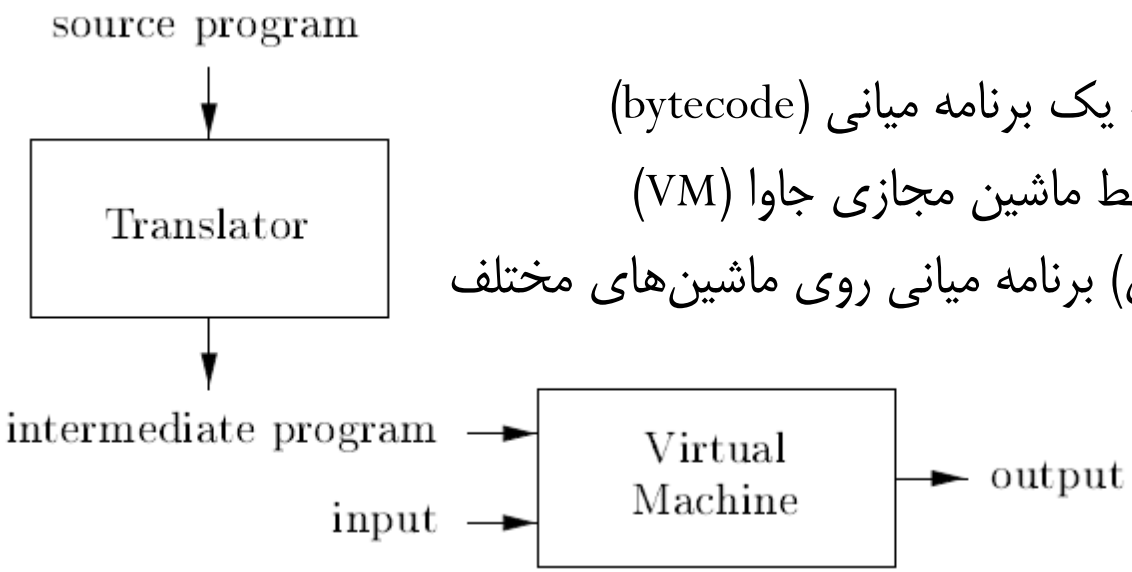
- مفسر
  - مستقیماً عملیات مشخص شده در برنامه ورودی را روی ورودی‌ها اجرا می‌کند



  - ترجمه به برنامه هدف صورت نمی‌گیرد
  - اجرای دستورات برنامه ورودی به صورت پشت سر هم
    - statement by statement
  - امکان تشخیص و بررسی بهتر خطا در برنامه ورودی
  - سرعت اجرای کمتر در پردازش ورودی‌ها و تولید خروجی

پردازشگرهای زبان

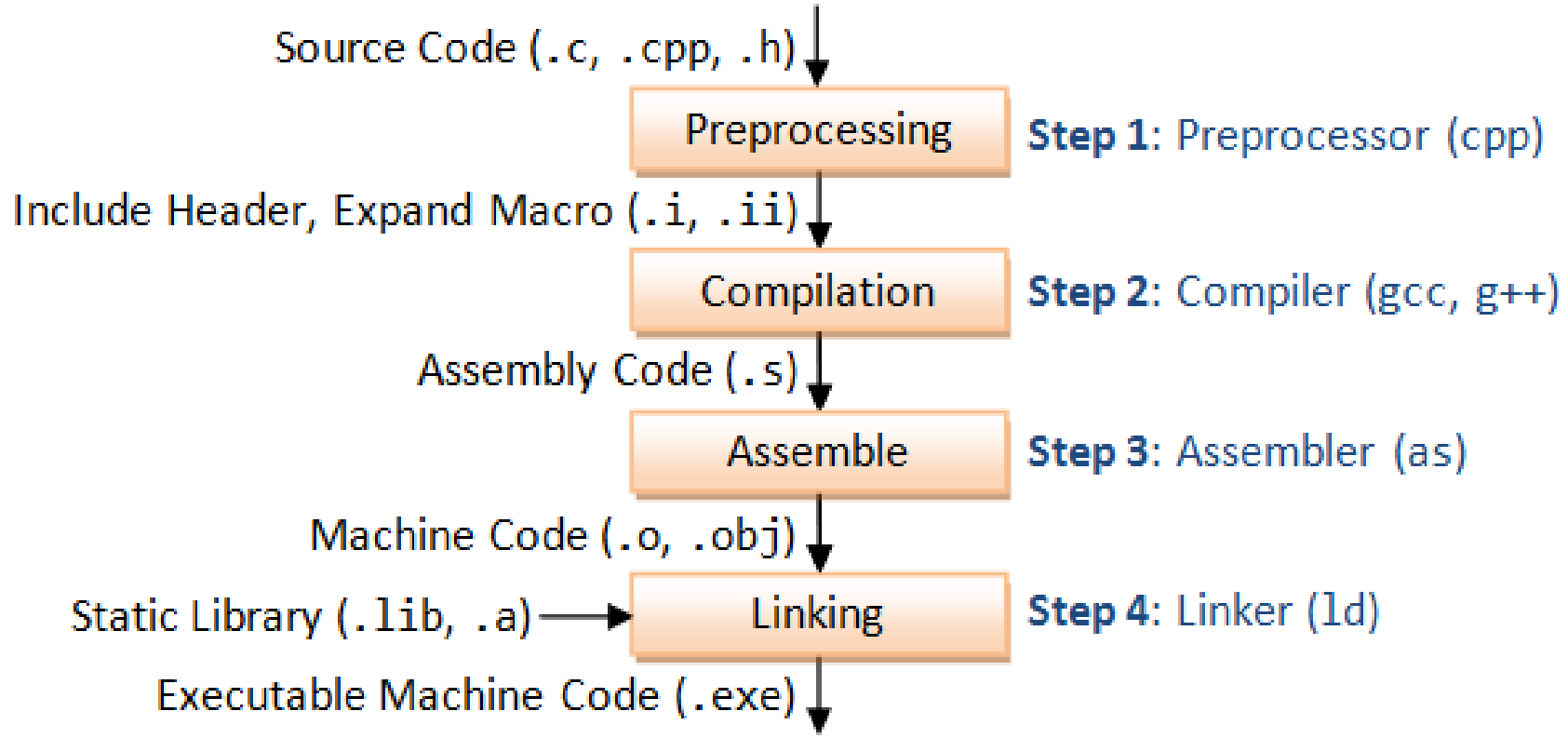
- مثال: زبان جاوا (Java)
  - ترکیب ترجمه و تفسیر
    - ترجمه برنامه ورودی به یک برنامه میانی (bytecode)
    - تفسیر برنامه میانی توسط ماشین مجازی جاوا (VM)
    - امکان تفسیر (اجرای) برنامه میانی روی ماشین‌های مختلف



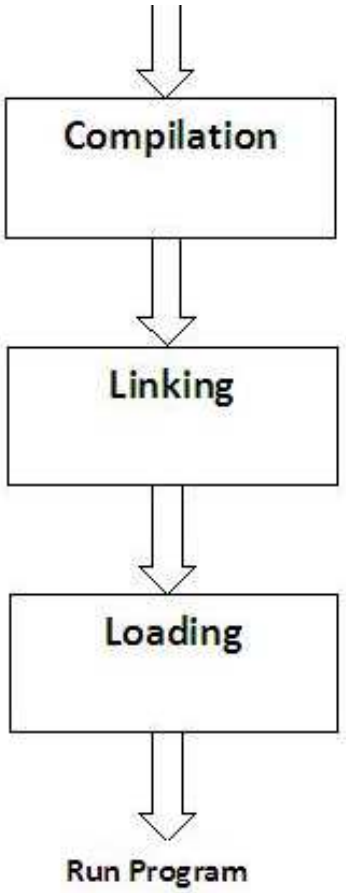
- کامپایلرهای just-in-time
  - برنامه میانی نیز قبل از اجرا به دستورات ماشین هدف ترجمه می‌شود

پردازشگرهای زبان

- مراحل اجرای یک برنامه



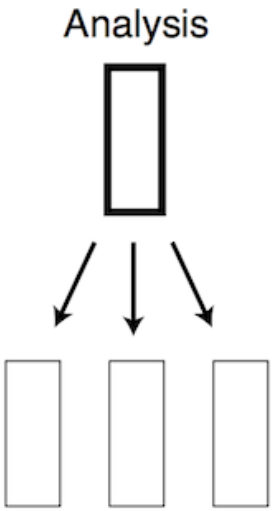
پردازشگرهای زبان



- مراحل اجرای یک برنامه (ادامه)
  - پیش‌پردازش (Preprocess)
    - تجمیع قسمت‌های مختلف برنامه ورودی
  - کامپایل و همگذاری (assemble)
    - تولید کد اجرایی قابل جابجایی (relocatable)
  - پیوند (link)
    - ترکیب کد اجرایی با سایر تکه کدها و کتابخانه‌ها
    - تنظیم آدرس‌های مورد ارجاع در کل برنامه
  - بارگذاری (load)
    - بارگذاری برنامه نهایی در حافظه برای اجرا

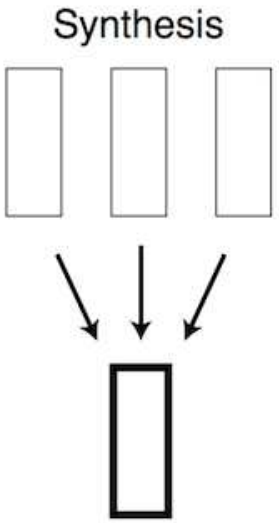
ساختار کامپایلرها

- دو بخش اصلی کامپایلرها



- تجزیه و تحلیل (analysis) برنامه ورودی
  - قسمت پیشین (front-end)
  - شناسایی اجزا تشکیل دهنده و ساختار دستوری برنامه ورودی
  - بررسی صحت معنایی برنامه و تولید نمایش میانی
  - ساختن جدول نشانه‌ها (symbol table)

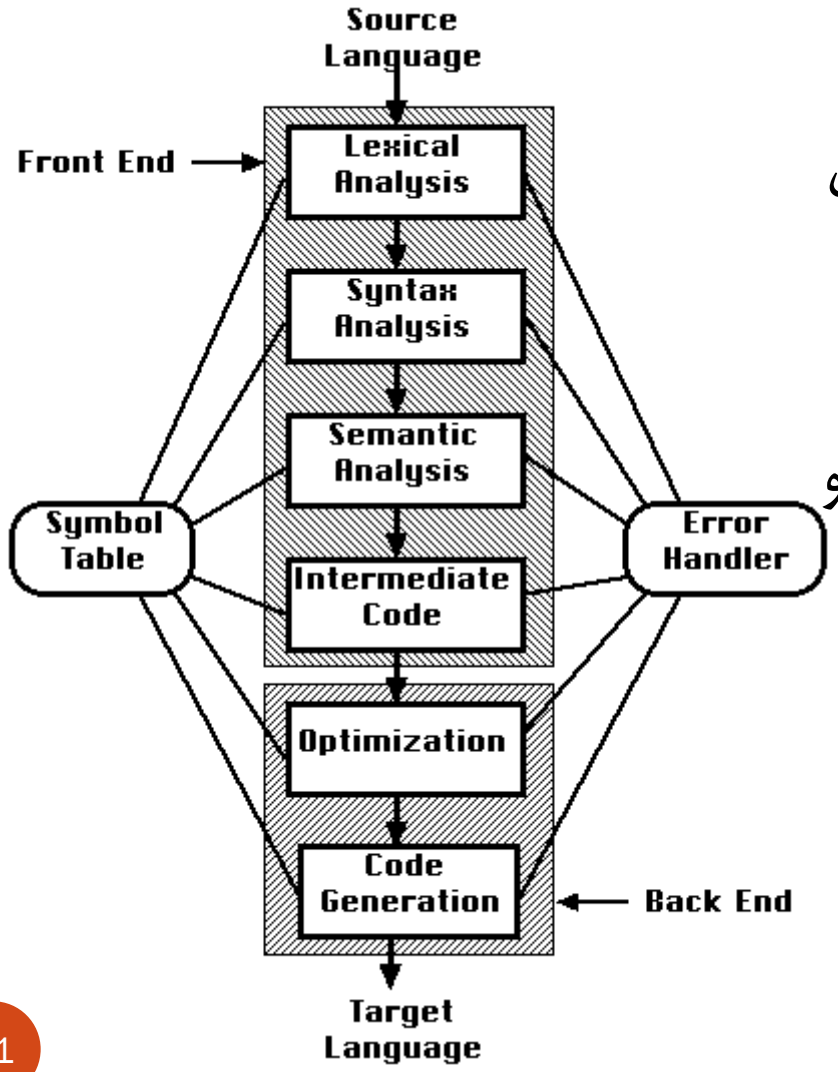
ساختار کامپایلرها



- ترکیب و ساخت (synthesis) برنامه خروجی
  - قسمت پسین (back-end)
  - استفاده از نمایش میانی و جدول نشانه‌ها
  - تولید برنامه به زبان هدف

- مجموعه‌های کامپایلری (compiler collections)
  - استفاده از یک زبان میانی مشترک
  - استفاده از قسمت‌های پیشین مختلف برای پشتیبانی از چندین زبان ورودی
  - بکارگیری قسمت‌های پسین مختلف جهت تولید کد برای ماشین‌های متفاوت

ساختار کامپایلرها



- کامپایل فرآیندی مرحله به مرحله
  - هر مرحله یک نمایش از برنامه ورودی را به نمایش دیگر تبدیل می‌کند
- جدول نشانه‌ها
  - حاوی اسامی شناسایی شده در برنامه و ویژگی‌های آنها

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

مراحل کامپایل کردن

- تحلیل واژه‌ای (lexical) یا پویش (scanning)
  - تفکیک جمله ورودی به اجزای تشکیل دهنده (بعد از حروف)
  - مثال در زبان طبیعی:

This is a sentence.

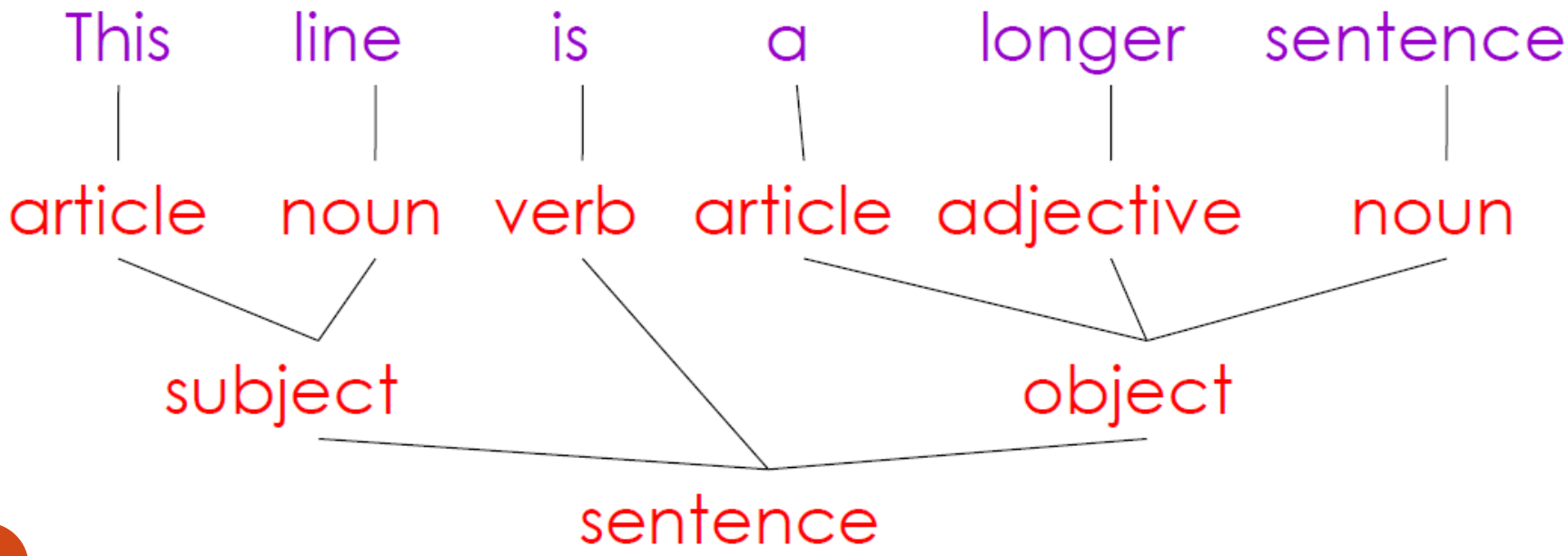
ist his ase nte nce

- خواندن دنباله حروف برنامه ورودی و جدا کردن واژه‌ها (lexemes)
  - شناسایی نمادها (tokens): tokenization

```
if x == y then z = 1; else z = 2;
```

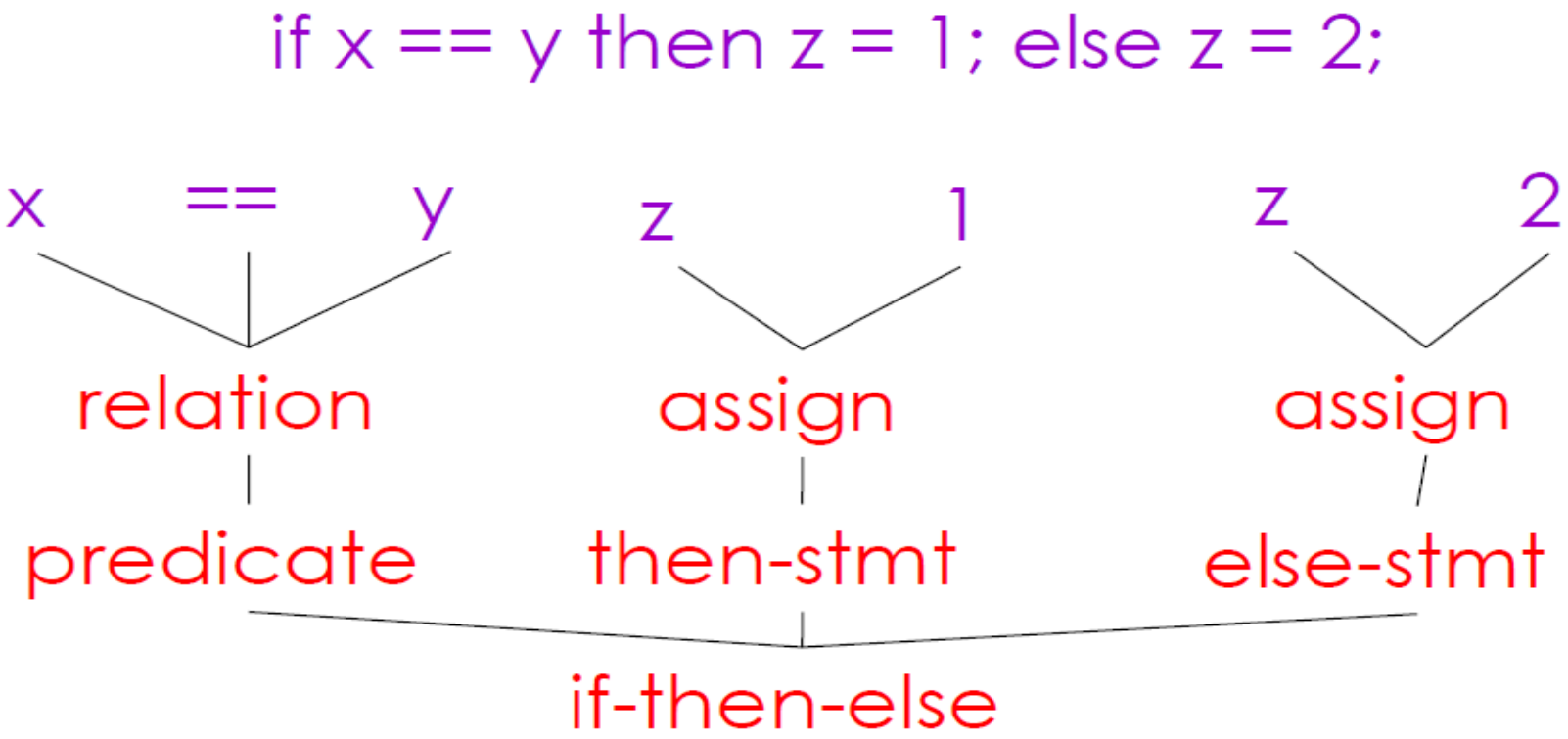
مراحل کامپایل کردن

- تحلیل نحوی (syntactic) یا تجزیه کردن (parsing)
  - شناسایی ساختار جملات
    - معمولاً توسط درخت نشان داده می‌شود
  - مثال در زبان طبیعی:



مراحل کامپایل کردن

- تحلیل نحوی (ادامه)
  - بدست آوردن درخت نحو (parse tree)



مراحل کامپایل کردن

- تحلیل معنایی (semantic)
  - درک معنای جملات
    - فرآیندی پیچیده (در مورد انسان هنوز ناشناخته است)
    - وجود ابهام (ambiguity) در تفسیر جملات

- مثال در زبان طبیعی:

Jack said Jerry left his assignment at home.

یا

Jack said Jack left his assignment at home?

مراحل کامپایل کردن

- تحلیل معنایی (ادامه)
  - استفاده از قواعد بخصوص در زبان‌های برنامه‌نویسی برای رفع ابهام
    - قواعد رفع ابهام مانند شرکت‌پذیری و اولویت

- مثال: پیوند متغیرها (variable binding)
  با توجه به حوزه (scope) آنها

```
{
   int Jack = 3;
   {
        int Jack = 4;
        cout << Jack;
   }
}
```

مراحل کامپایل کردن

- تحلیل معنایی (ادامه)
  - تحلیل معنایی محدود در کامپایلرها برای شناسایی عدم سازگاری‌ها در برنامه
    - بررسی سازگاری نوع
    - شناسایی عدم تطبیق نوع (type mismatch)
  - مثال در زبان طبیعی:

Jack left her homework at home.

  - بکارگیری تبدیل نوع

```
int a = 2;
float b = 3;
bool c = 4;
char d = 300;
```

مراحل کامپایل کردن

- تولید کد میانی (intermediate)
  - یک کد انتزاعی یا مفهومی
    - معمولاً نزدیک به کد ماشین (یک زبان سطح پایین)
  - هدف: ساده‌سازی فرآیند ترجمه و خطایابی (debugging)

- مثال: کدهای سه آدرسه (three-address code)

```
position = initial + rate * 60
```

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

مراحل کامپایل کردن

- بهینه‌سازی کد
  - تلاش برای بهبود کد تولید شده
  - مثال در زبان طبیعی: ویرایش کردن متون

But a little bit like editing ➡ But akin to editing

  - تغییر خودکار برنامه به نحوی که از منابع (resource) کمتری استفاده کند
    - سرعت اجرای بالاتر، مصرف حافظه کمتر، دستورات کمتر، مصرف انرژی کمتر
  - امکان اعمال تغییرات ساده (بدون صرف وقت زیاد) برای کاهش زمان اجرا

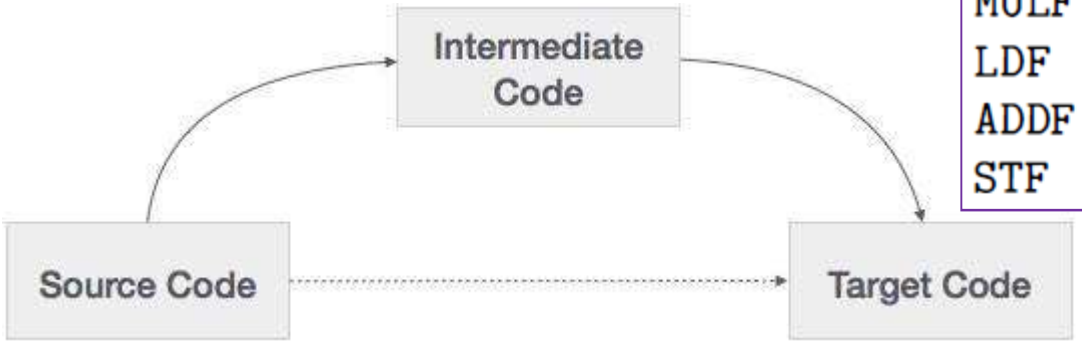
X = Y * 0 ➡ X = 0

  - اطمینان از صحت تغییرات!

مراحل کامپایل کردن

- تولید کد
  - تولید برنامه به زبان هدف (معمولاً زبان اسمبلی)
    - مانند ترجمه یک زبان طبیعی به زبان دیگر
  - نیاز به مدیریت نحوه اختصاص فضای ذخیره‌سازی به شناسه‌ها
    - تصمیمات مربوطه در زمان تولید کد میانی یا کد نهایی گرفته می‌شود

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




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

```
position = initial + rate * 60
```

کامپایلرها و زبان‌های برنامه‌نویسی

تکامل زبان‌های برنامه‌نویسی

- کامپیوترهای اولیه در دهه ۱۹۴۰
  - برنامه‌ها به صورت دنباله‌ای از 0 و 1 (زبان‌های نسل اول)
- زبان‌های اسمبلی ابتدایی در اوایل دهه ۱۹۵۰ (زبان‌های نسل دوم)
- آغاز زبان‌های سطح بالا در نیمه دوم دهه ۱۹۵۰ (زبان‌ها نسل سوم)
  - Fortran، Cobol، Lisp
  - C، C++، Java، C#
- زبان‌های متنی (script) (زبان‌های نسل چهارم)
  - NOMAD، SQL، Postscript، ...
- زبان‌های مبتنی بر منطق و محدودیت (زبان‌های نسل پنجم)
  - Prolog، OPS5

تکامل زبان‌های برنامه‌نویسی

- زبان‌های دستوری (imperative)
  - چگونه (how) محاسبات باید انجام شود
  - اکثر زبان‌های متداول برنامه‌نویسی
- زبان‌های اعلانی (declarative)
  - چه (what) محاسباتی باید انجام شود
  - زبان‌های تابعی (functional) و زبان‌های منطق (logic)

- زبان‌های شیء گرا (object-oriented)
  - برنامه‌ها شامل مجموعه‌ای از اشیاء تعامل کننده با هم هستند

تکامل زبان‌های برنامه‌نویسی

- چرا تعداد زیادی زبان‌های برنامه‌نویسی داریم؟
  - کاربردهای مختلف نیازهای متفاوت (بعضاً متضاد) دارند
    - مثال: محاسبات علمی، برنامه‌های تجاری، برنامه‌نویسی سیستمی

- چرا زبان‌های برنامه‌نویسی جدیدی ابداع می‌شوند؟
  - آموزش برنامه‌نویس هزینه غالب در بکارگیری یک زبان برنامه‌نویسی است
    - زبان‌های متداول برنامه‌نویسی به کندی تغییر می‌کنند
    - ابداع یک زبان جدید برای نیازهای جدید راحت‌تر است
    - زبان‌های جدید برای پر کردن یک کمبود بکار گرفته می‌شوند
    - زبان‌های جدید شباهت زیادی به زبان‌های قبلی دارند

تکامل زبان‌های برنامه‌نویسی

- کدام زبان برنامه‌نویسی خوب است؟
  - نیاز به یک معیار برای تعیین خوبی زبان
  - معیار همه پسندی برای طراحی زبان‌های برنامه‌نویسی وجود ندارد
    - زبانی خوب است که بیشتر از آن استفاده شود (شهرت)
      - Visual Basic بهترین زبان است؟