به نام خدا

طراحی و ساخت کامپایلرها

1

Shapourifardin@gmail.com

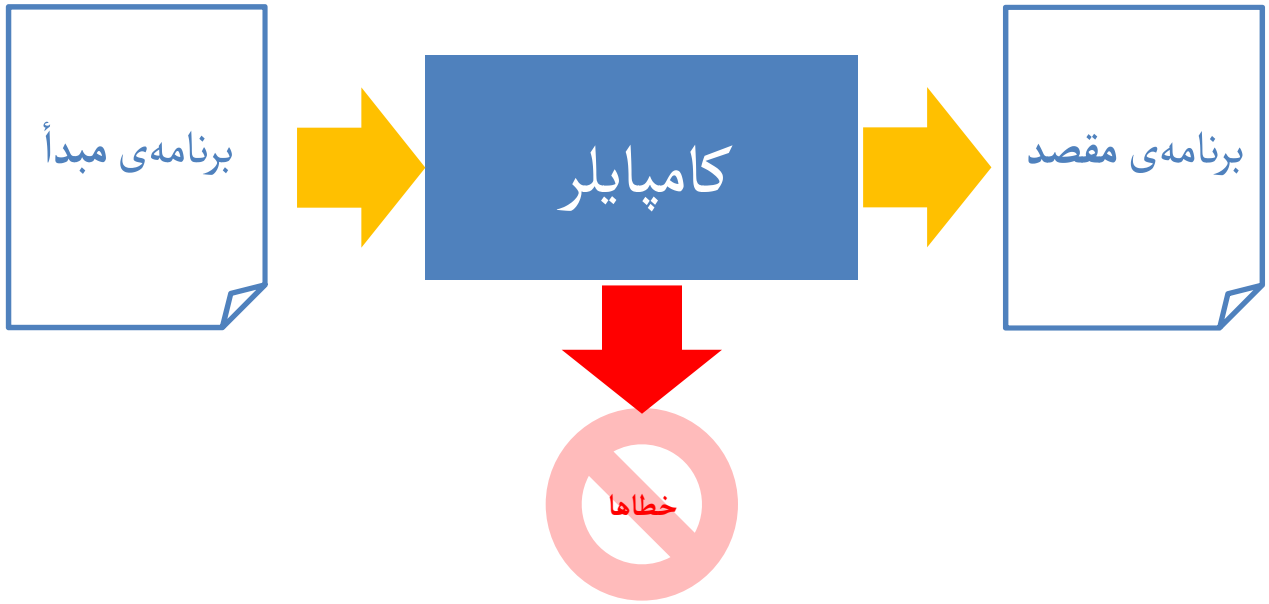
آموزش درس کامپایلر

- آیا موضوعی قدیمی نیست ؟
  - بله، و به خوبی تبیین شده است
  - الگوریتم‌ها، روش‌ها، و تکنیک‌های آن در مراحل اولیه‌ی شکل‌گیری علوم کامپیوتر طراحی شده‌اند
  - در حال حاضر تعداد زیادی کامپایلر وجود دارد، و
  - تعداد زیادی ابزار که به صورت خودکار کامپایلر تولید می‌کنند
- پس چرا باید این درس را بخوانیم ؟
  - با وجودی که احتمالاً هیچ‌وقت یک کامپایلر کامل را خودتان نمی‌سازید، اما
  - تکنیک‌هایی که در این درس می‌آموزید کاربردهای زیادی دارند؛ مثلاً:
    - ساخت مفسر برای زبان‌های اسکریپتی
    - اعتبارسنجی (validation) فرم‌ها
    - و غیره

اصطلاحات

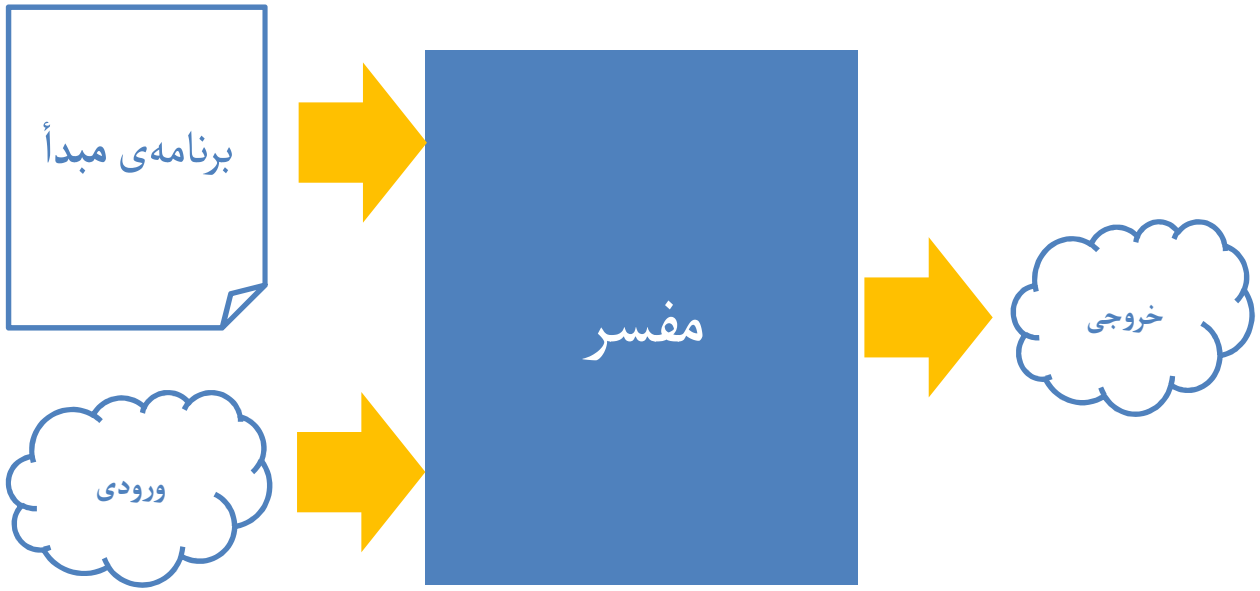
- **کامپایلر**
  - برنامه‌ای که یک برنامه‌ی «قابل اجرا» به **زبان مبدأ** (معمولاً زبانی سطح بالا) را به برنامه‌ی «قابل اجرای» **معادل** به **زبان مقصد** (معمولاً زبانی سطح پایین) ترجمه می‌کند
- **مفسر (Interpreter)**
  - برنامه‌ای که یک برنامه‌ی قابل اجرا را می‌خواند و نتایج حاصل از اجرای آن را تولید می‌کند
  - این کار معمولاً به معنی «اجرای» برنامه است
- **این درس عمدتاً درباره‌ی کامپایلرهاست، ولی در بسیاری از موارد، همین مباحث برای مفسرها هم مطرح می‌شود**

یک کامپایلر


برنامه‌ی مبدأ
کامپایلر
برنامه‌ی مقصد
خطاها


ورودی
برنامه‌ی مقصد
خروجی

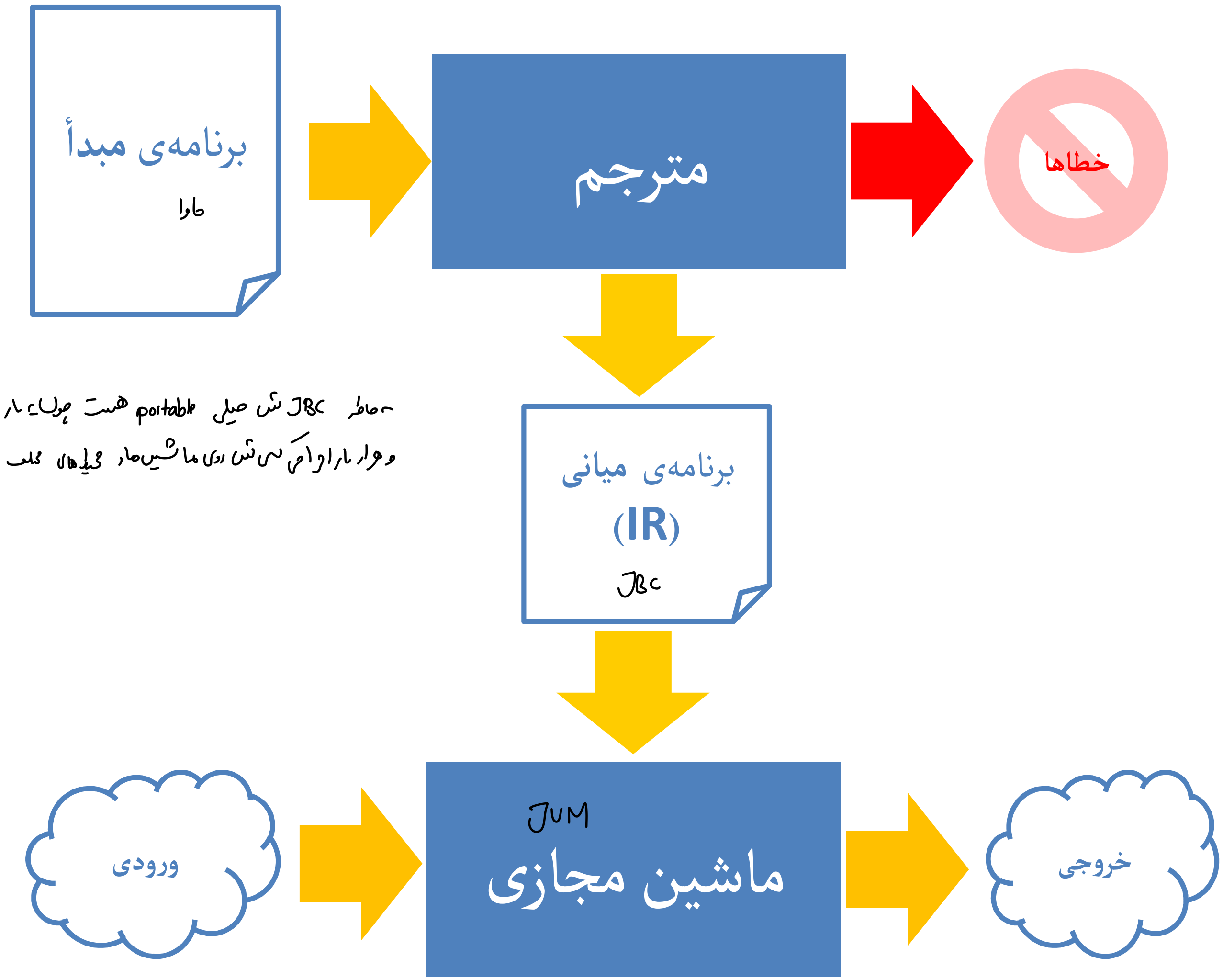
یک مفسر



- ترجمه‌ی خط به خط برنامه
- اجرای بلافاصله‌ی هر خط ترجمه‌شده
- اجرا کندتر است زیرا ترجمه تکرار می‌شود
- ولی معمولاً خطاها را نسبت به کامپایلر بهتر تشخیص می‌دهد

کد میانی جاوا ← Jbc که برای تفسیرش از JVM استفاده می‌شه
Java byte Code

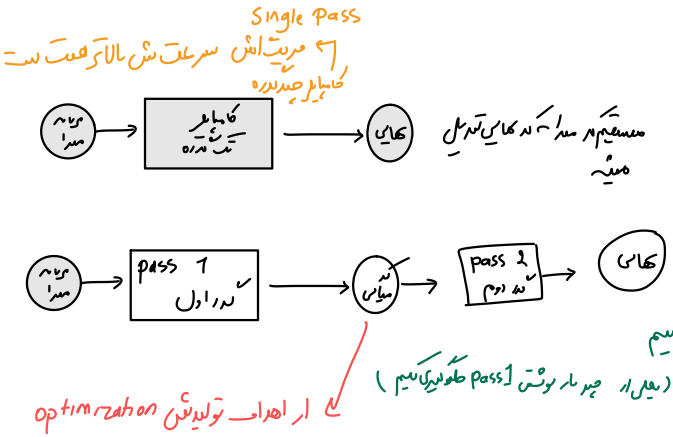
یک کامپایلر ترکیبی (Hybrid)



به خاطر JBC ش خیلی portable هست چون یه بار کامپایل می‌کنی
و هر بار اجرا می‌کنی ش روی ماشین‌های محیط‌های مختلف

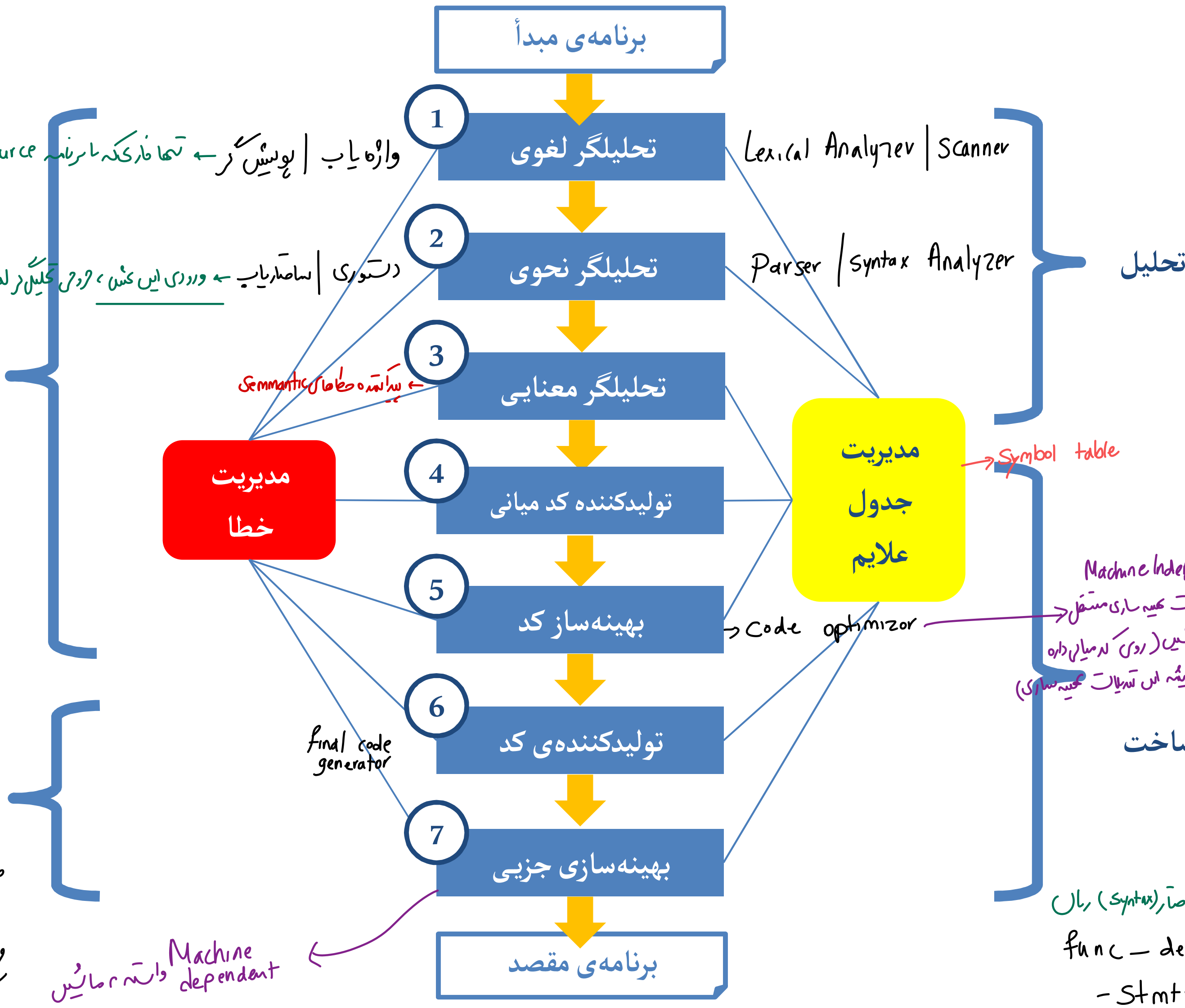
دسته‌بندی کامپایلرها

- از لحاظ **ساخت**:
  - تک‌گذره Single Pass
  - چندگذره Multi Pass
- از لحاظ **نوع کد تولیدشده**:
  - مطلق (مثل **.com**)
  - قابل جابه‌جایی (مثل **.exe**)



مراحل کامپایل

error recovery → ترمیم خطا
داخل مدیریت خطا انجام میشه



واژه‌یاب | پویشگر ← تنها فازی که با برنامه source سروکار داره این فاز هست

Lexical Analyzer | Scanner

دستوری | ساختاریاب ← ورودی این بخش، خروجی تحلیلگر لغوی است ← خطای Syntax پیدا میکنه

Parser | Syntax Analyzer

Semantic ← پیدا کردن خطاهای

Front-End ← پیشین / ابتدائی

→ Symbol table

اطلاعات بخش اعلانی رو یه جایی نگهداری میکنه (به نام Symbol table) ذخیره میکنیم
که هر دفعه کامپایلر هر جا مراجعه کنه به بخش اعلان
(اگر بخواد مدام به بخش اعلان بره) چیدمانش
سرعت بخش خیلی میاد پایین تحت این شرایط)
و برای مثال از تعریف دوباره‌ی متغیر (شناسه) جلوگیری میکنه

Machine Independence
→ code optimizer → عملیات بهینه‌سازی مستقل از ماشین (روی کد میانی داره) انجام میشه این عملیات بهینه‌سازی

final code generator

Machine dependent وابسته به ماشین

فاز ۶ و ۷ به ماشین مقصد وابسته هستند
مراحل با وجود کد میانی میشه کامپایلر رو
وابسته کرد تقسیم کرد توسعه و نگهداری
کامپایلر سون آسون‌تر این مدل

اگر ما مفهوم داخل بخش اعلان هستیم؟ ورودی ساختار (Syntax) زبان

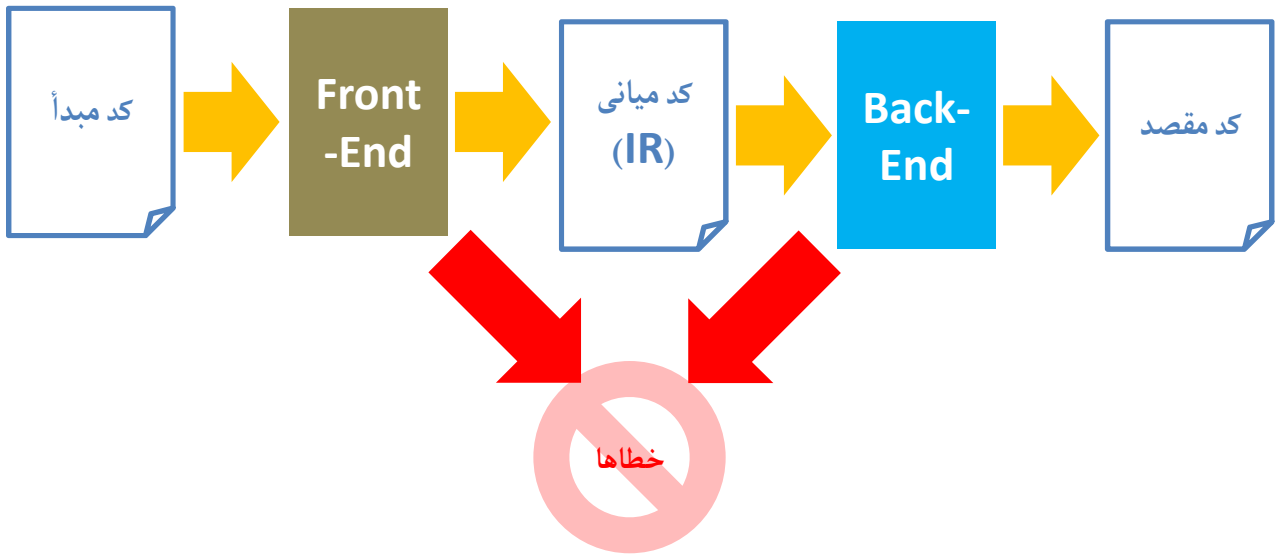
func — del → type — func_name — ( ) — { — decl

— stmts

قسمت دستورات اجرای

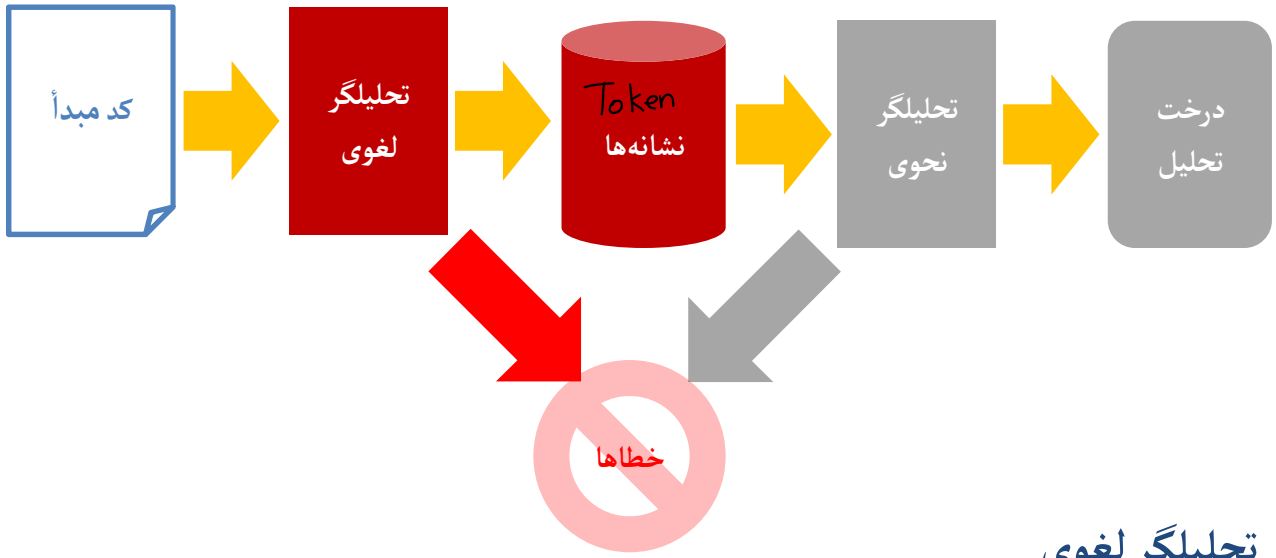
اینجا هستیم یعنی در کامپایلر دستورات اجرای هستیم

تقسیم به Front-End و Back-End



- نگاشت کد مبدأ به کد میانی در **Front-End**
- نگاشت کد میانی به کد مقصد در **Back-End**
- ساده‌سازی پرداخت کد (**optimization**)
- امکان وجود چند **Front-End**

(1) Front-End



تحلیلگر لغوی

- نگاشت نویسه‌ها به مجموعه‌ای از «نشانه‌ها» -(Tokens)- واحد پایه‌ی نحو

Tokenizer
ورودی رو به دنباله‌ای از توکن ها تبدیل کنه

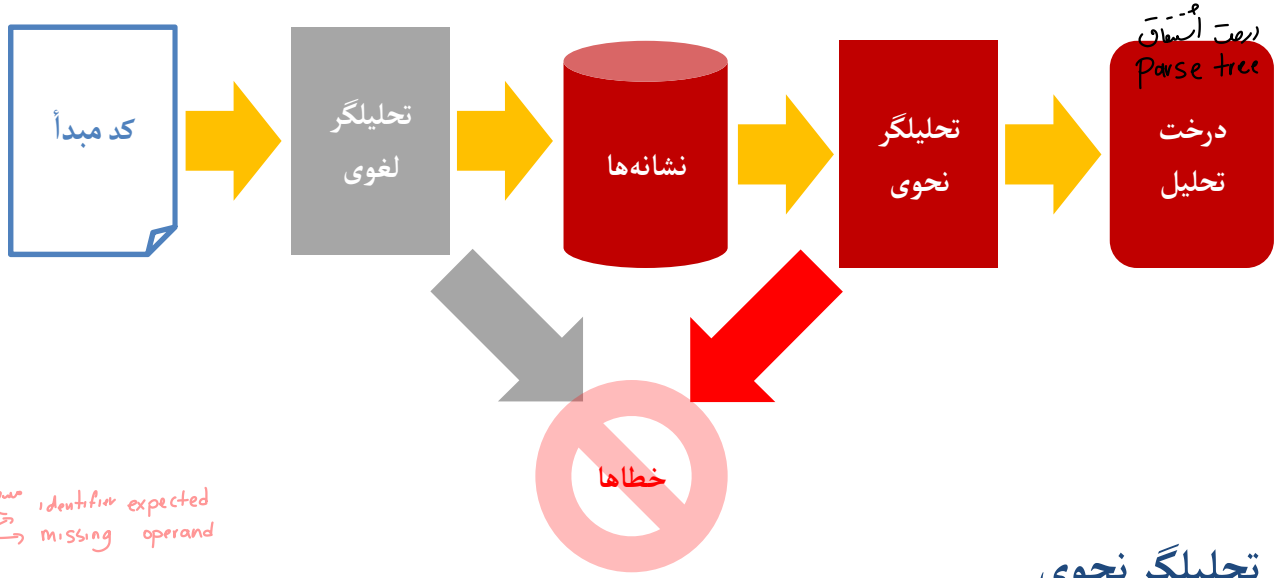
- مثالی از نشانه‌ها:

num | id | + | - | * | / | do | end

- حذف فاصله‌ها (tab ها، فضاهای خالی، و توضیحات)
- ساخت دستی تحلیلگر لغوی به جای استفاده از ابزارهای خودکار (مثل LEX) ممکن است برای افزایش سرعت لازم باشد

مثل → FLEX, LEX, ALEX, JLEX

(2) Front-End



تحلیلگر نحوی

- تشخیص نحو «مستقل از متن»
- هدایت تحلیل وابسته به متن
- ساخت کد میانی (IR)
- تولید پیغام های خطاهای با معنا
- تلاش برای اصلاح خطا ← error recovery
- ابزارهایی مثل YACC می‌توانند فرآیند ساخت آن را خودکار کنند

left hand side → right hand side

lhs → rhs

سمت چپ هر قاعده ۱ متغیر بود

$A \rightarrow \alpha, A$

$\alpha \in (V \cup T)^*$

(3) Front-End

- برای توصیف نحو زبان‌های برنامه‌سازی از **گرامرهای مستقل از متن** استفاده می‌شود

CFG

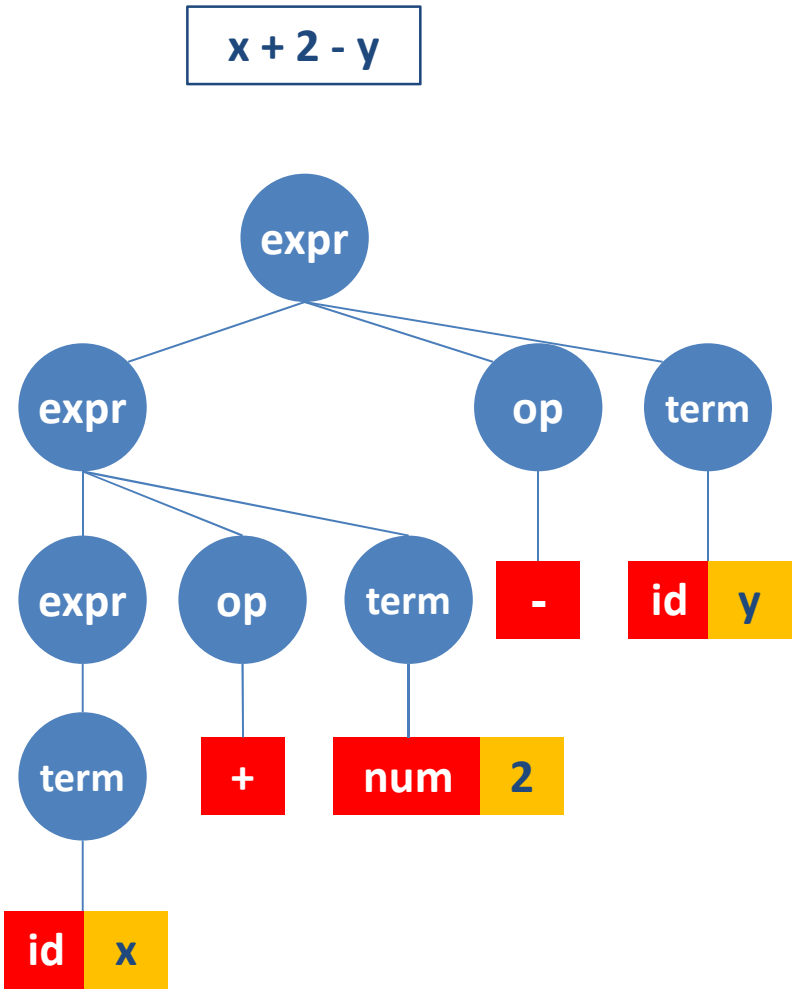
BNF → در اینجا متغیرها داخل براکت می‌داریم اونی که داخل براکت نیست ترمینال یا همون پایانه است

<expr> ::= <expr> <op> <term> | <term>

<term> ::= <number> | <id>

<op> ::= + | -

(4) Front-End

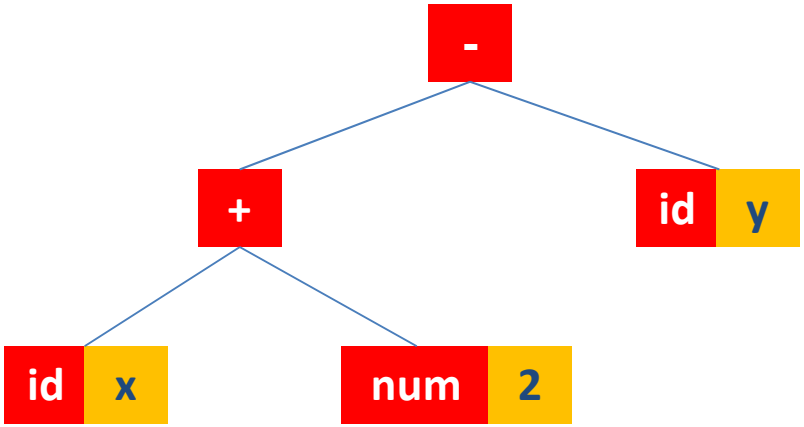


- تحلیلگر نحوی سعی می‌کند یک برنامه را به اجزای نحوی تعریف‌شده در گرامر تقسیم کند
- تحلیل نحوی یا پارس را می‌توان در قالب یک درخت پارس (Parse Tree) و یا درخت نحو (Syntax Tree) نمایش داد

Start symbol ؟ parse tree ریشه ی این

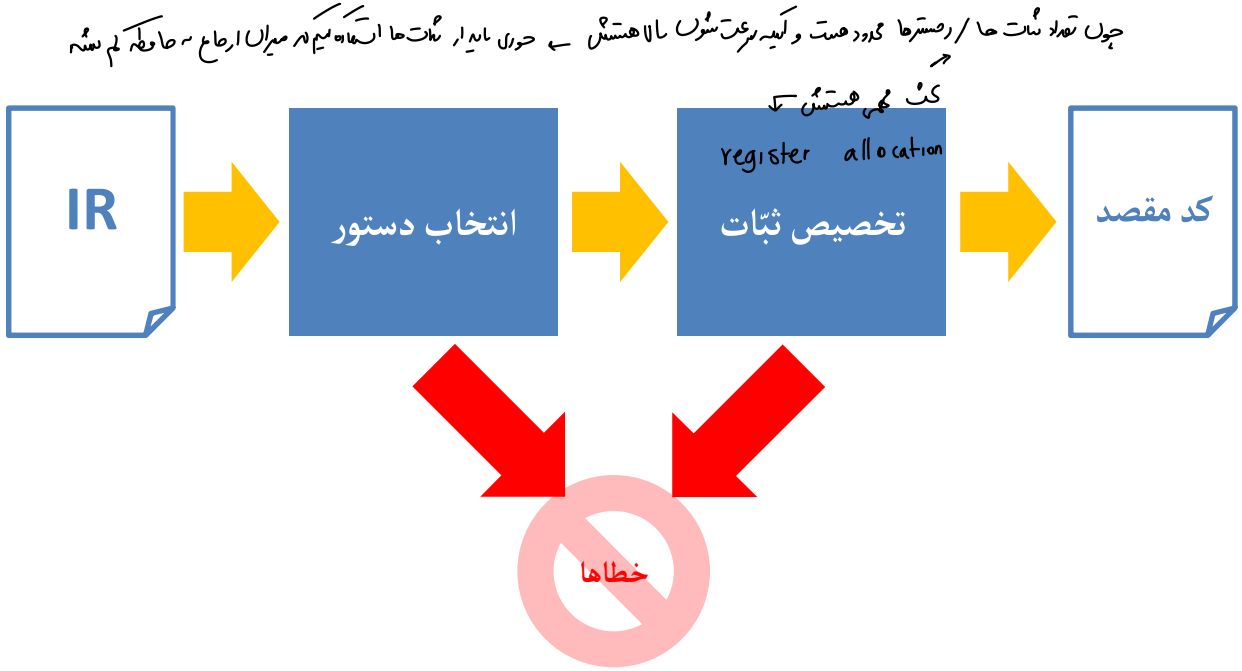
(5) Front-End

- درخت پارس را می‌توان به صورت فشرده‌تر در قالب درخت انتزاعی نحو (Abstract Syntax Tree) نمایش داد
- AST می‌تواند به عنوان IR بین Front-End و Back-End به کار رود



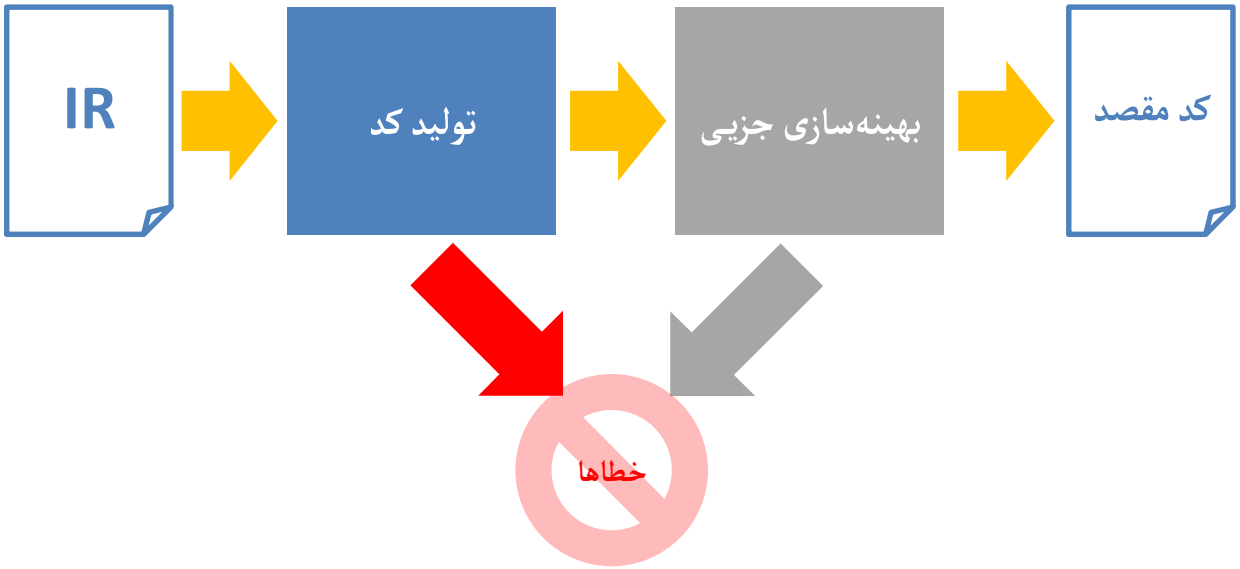
x + 2 - y

(1) Back-End



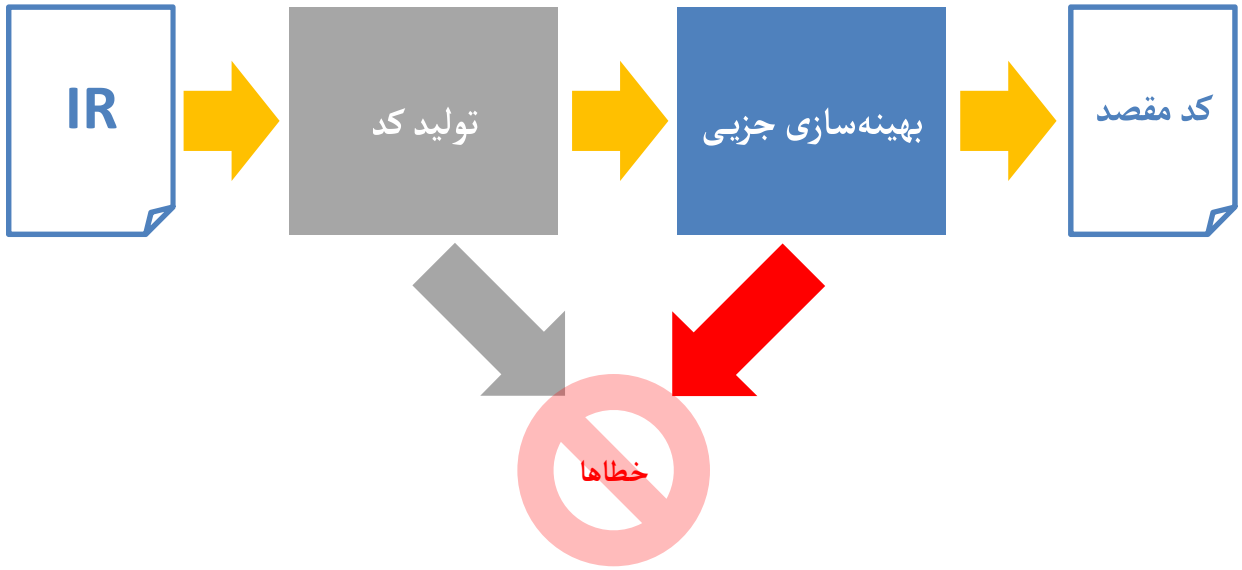
- ترجمه‌ی کد میانی (IR) به کد مقصد
- انتخاب دستور مناسب برای هر عمل در IR
- تصمیم‌گیری در مورد آن‌چه در هر لحظه باید در ثبّات‌ها ذخیره شود

(2) Back-End



- تولید کد فشرده و سریع
- استفاده از انواع آدرس‌دهی های موجود

(3) Back-End



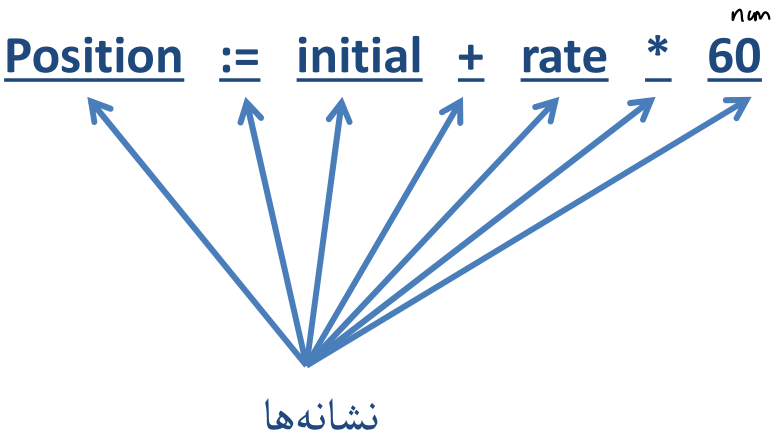
- منابع محدود
- دشواری تخصیص بهینه‌ی منابع

بخش تحلیل در فرآیند کامپایل

- شامل سه مرحله:
  - تحلیل لغوی
    - پیمایش چپ به راست کد برنامه برای شناسایی نشانه‌ها (یک نشانه، توالی‌ای از نویسه‌هاست که در کنار هم معنا دارند)
  - تحلیل نحوی
    - گروه‌بندی نشانه‌ها در دسته‌های بامعنا
  - تحلیل معنایی
    - بررسی صحت اجزای به‌دست‌آمده

مرحله اول: تحلیل لغوی

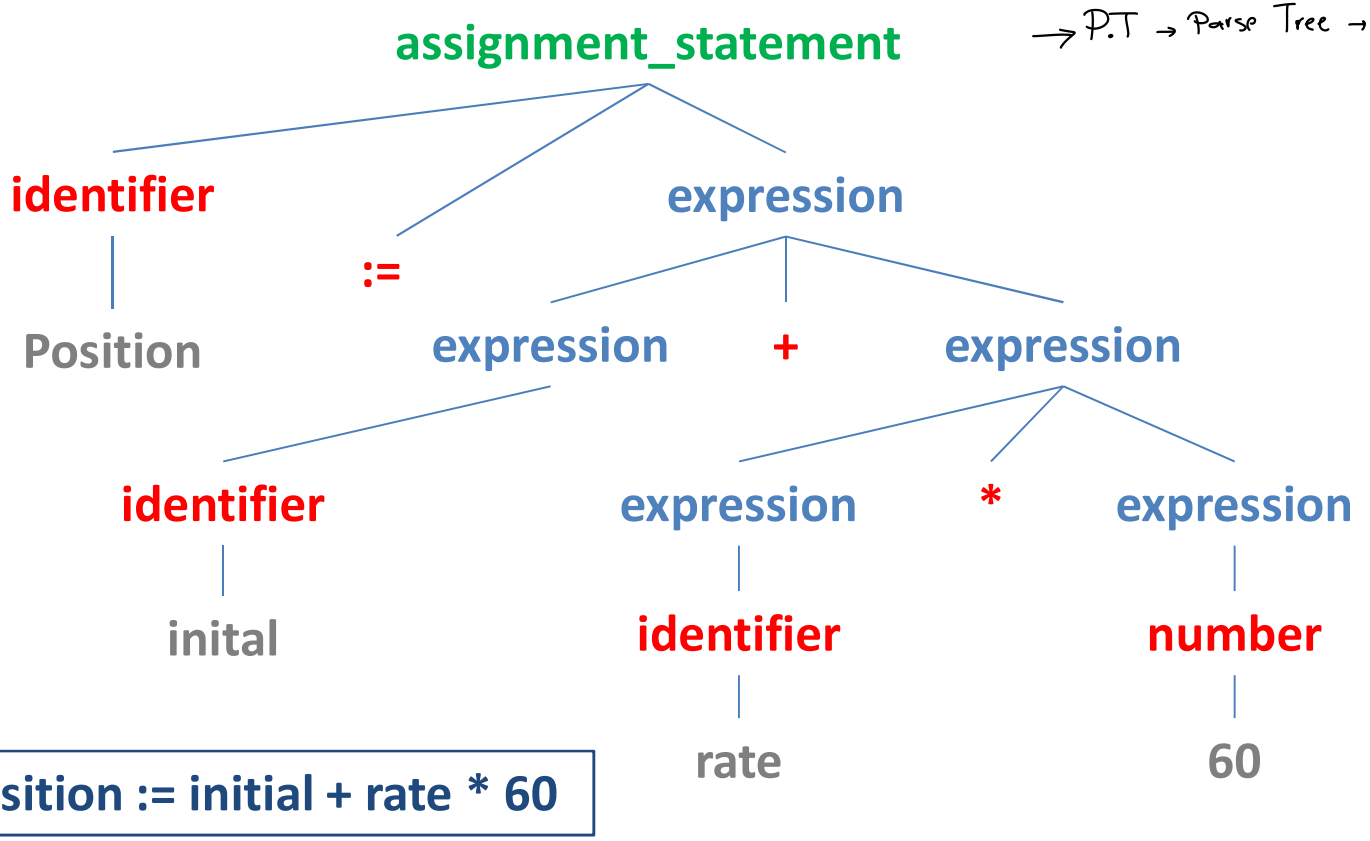
- ساده‌ترین تحلیل
- شناسایی **نشانه‌ها** که پایه‌ای‌ترین واحدهای سازنده هستند



- فاصله‌ها، نویسه‌های رفتن به ابتدای خط جدید، و غیره نادیده گرفته می‌شوند

مرحله دوم: تحلیل نحوی یا پارس

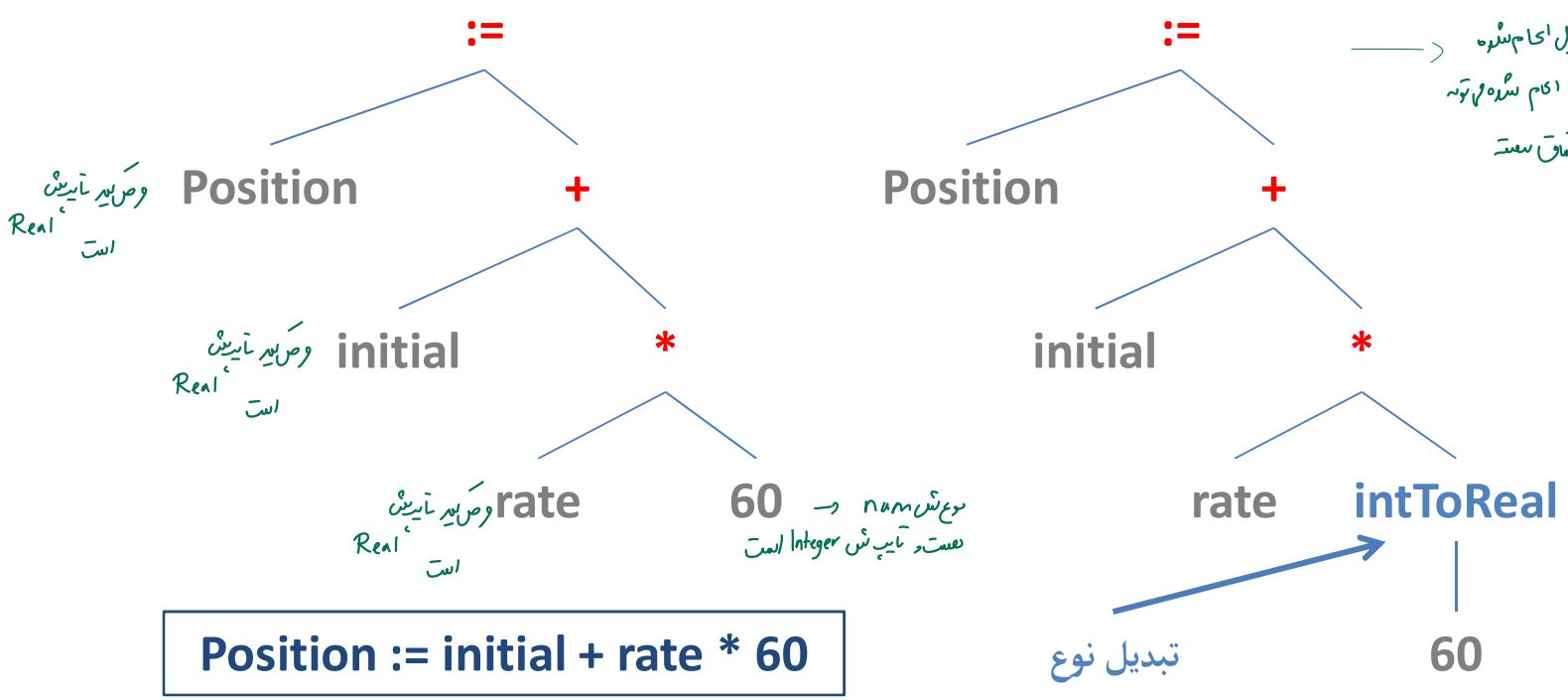
- گره‌های درخت پارس به کمک گرامر زبان برنامه‌سازی ساخته می‌شوند



Position := initial + rate * 60

مرحله سوم: تحلیل معنایی

- یافتن خطاهای معنایی



Position := initial + rate * 60

- وارسی نوع‌ها، و مجازبودن عملوندها؛ دو تحلیل مهم در این مرحله

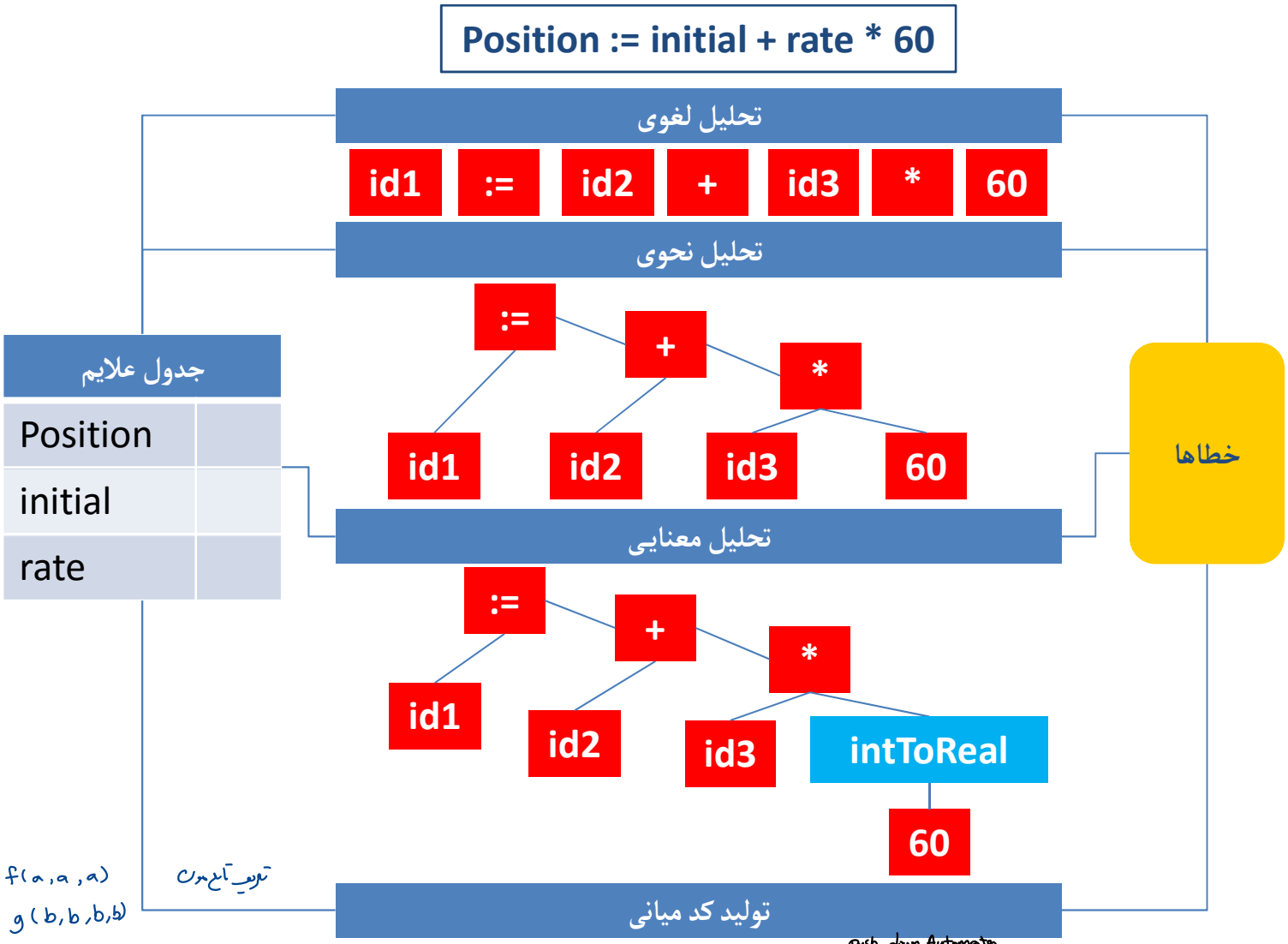
مراحل و فعالیت‌های مکمل تحلیل

- ایجاد و نگه‌داری جدول علایم
  - حاوی اطلاعاتی (در مورد ذخیره‌سازی، نوع، محدوده، و آرگومان‌ها) برای هر نشانه‌ی «بامعنا» (معمولاً شناسه‌ها)
  - داده‌ساختار مناسب، در مرحله تحلیل لغوی، ایجاد، و مقداردهی اولیه می‌شود
  - جدول علایم در مراحل بعدی تحلیل و ساخت، استفاده، یا به‌روز می‌شود
- خطا پردازی
  - شناسایی خطاهای گوناگون مرتبط با تمام مراحل کامپایل
  - چگونگی برخورد با خطاها بعد از شناسایی

بخش ساخت در فرآیند کامپایل

- تولید کد میانی
  - کد «انتزاعی» مقصد که مستقل از ماشین است
  - تسهیل فرآیند تولید کد «نهایی» مقصد که وابسته به ماشین است
- بهینه‌سازی (پرداخت) کد
  - یافتن راه‌های بهتر برای اجرای کد
  - جای‌گزین کردن کدهای تولیدشده با کدهای بهینه
- تولید کد نهایی
  - تولید کد قابل جابه‌جایی وابسته به ماشین
- بهینه‌سازی جزیی
  - با دیدی محدود، کد نهایی تولیدشده را بهبود می‌دهد

مروری بر کلیت فرآیند کامپایل (1)



```
int func(int , float x){
    ⋮
}

k = func(100, y, p);
⋮
```

آیا خطای لغوی داره؟ → نه هیچ کلمه‌ای نیست که املاش غلط باشه

آیا خطای Syntax داره؟ → نه

k = func(100, y, p); → خطا داریم اینجا → Semantic error (معنایی)

انواع خطاها

چه زمانی شناسایی میشن؟

① لغوی	(Lexical)	← compile
② نحوی	(Syntax)	← compile
③ معنایی	(Semantic)	← compile , runtime
④ منطقی	(Logical)	← runtime

مثلاً تابع بازگشتی که پایان نداشته باشه

مثال ← null pointer

یه مثال از خطای معنایی در زمان اجرا:

y = A [i];

→ در زمان کامپایل (قبل اجرا) خطا نداره در یه حالت خاص البته می‌شد گفت اگر i یک آرایه تعریف نمی‌شد یا

(در زمان کامپایل معلوم باشه مقدار i constant باشه و داخل range کار A باشه و مقدار i در زمان اجرا مشخص میشه } runtime error

در زمان اجرا این خطای معنایی رو اعلام می‌دیم ←

مروری بر کلیت فرآیند کامپایل (2)