
دانشکده مهندسی کامپیوتر

# پروژه پایانی نظریه زبان‌ها و ماشین‌ها - پیوست

**حسین بابازاده - محمدجواد جلیلوند**

بهار ۱۴۰۴

سلام.
ما با تعدادی از بچه‌ها بصورت تصادفی صحبت کردیم که چه ابهاماتی درباره پروژه دارن و توی این داکیومنت سعی کردیم که برطرفشون کنیم. همچنین مثال‌ها، توضیحات بیشتر و دقیق‌تر، یک سری لینک برای مطالعه بیشتر هم هست.

## بخش اول

# سوالات متداول (FAQ)

## ۱ برای انجام پروژه از چه زبان برنامه‌نویسی باید استفاده کنیم؟

استفاده از هر زبانی که بخواید مجازه. توصیه ما اینه که بخاطر راحتی استفاده از python استفاده کنید. اما تصمیم با خودتونه.

## ۲ آیا باید کامپایلر C بنویسیم؟

خیر! نوشتن کامپایلر زبانی مثل C کار ما و شما نیست. (باور کنید، نیست). باید از گرامر یک زبان ساده، که خیلی توانایی های محدودی داره استفاده کنیم، اون زبان رو parse کنیم، و یک کاری با درخت تجزیه تولید شده انجام بدیم.

## ۳ خب چرا از اول گرامر این زبان ساده رو ندادید؟

نکته اینجاست که گرامر «یک» زبان خاص مد نظر نیست. کدی که شما می‌زنید باید بتونه هر زبانی که درحد کافی ساده باشه (برای توضیح این «حد کافی» بیشتر اسکرول کنید) رو بفهمه و روش تغییر ایجاد کنه.

## ۴ ورودی و خروجی پروژه چیه؟

توجه کنید که قالب ورودی و خروجی بیشتر شبیه درس AP هست تا درس الگوریتم. یعنی خیلی روی ورودی استاندارد کنسول مانور نمی‌دیم. همچنین جاج خودکاری وجود نداره. باید این اتفاق ها بیوفته:

- یک فایل متنی که گرامر توش مشخص شده رو بخونید و گرامر رو لود کنید
- کدی که باید تغییر کنه (طبق ادبیات درس، یک کلمه از اون زبان، اما خب یه مقدار بلند تر از چیزیه که معمولا بهش کلمه گفته میشه) رو بگیرید. می‌تونه از فایل خونده بشه، یا ورودی استاندارد.

- بعد از پردازش اون کد، درخت تجزیه نشون داده بشه، می‌تونید با یک مقدار زحمت درخت رو توی کنسول چاپ کنید (که واقعا پسندیده نیست) یا اینکه با یک کتابخونه ای بصورت گرافیکی نشون بدید.
- حالا باید یک نود از درخت انتخاب بشه. اینجا هم دستتون بازه، برای نودها یک چیزی شبیه id بذارید و موقع نیاز از کنسول بخونید، یا بشه روی نود کلیک کرد، یا هر راهی دیگه.
- درآخر باید همه نودهایی از درخت که مربوط به همون موجودیت هستن (یعنی تعریف و همه اشاره ها به همون متغیر) رو مشخص کنید.
- اگر تا اینجا رسیدید و دوست داشتید بیشتر ادامه بدید، اسم جدید اون متغیر رو هم از ورودی بگیرید، و کد اصلاح شده رو هم تولید کنید (چاپ کنید، توی فایل بنویسید، یا حتی روی فایلی که کد توش بود تغییر بدید!)

## ۵ کد ناقص پروژه رو نمیدید؟

این موضوع رو بررسی کردیم. هر گروه دو نفره مثل خودش فکر می‌کنه، کد نصفه دادن باعث می‌شد که شما مجبور بشید شبیه ما فکر کنید، اما می‌خواستیم که نحوه فکر کردن پنجاه و چند تا گروه دیگه رو هم ببینیم. :)

## ۶ از کجا شروع کنیم؟

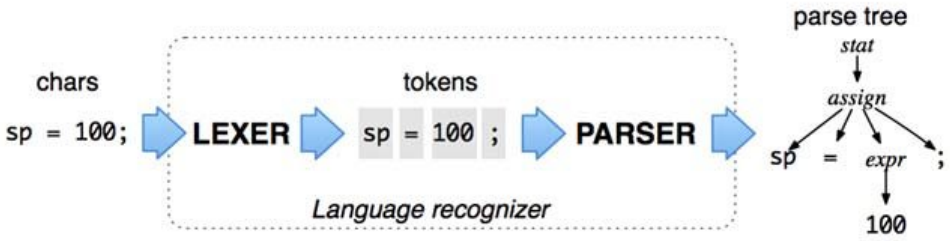
تلاش کردیم فاز بندی پروژه جواب این سوال باشه. هر فاز یک امکان چشم‌گیر اضافه می‌کنه و دو فاز اول تقریبا پیاده‌سازی مطالب کلاس هستن.

# بخش دوم

# ادامه تئوری

## ۱ الفبای ورودی اتوماتای پشته‌ای

توی درس نظریه، بصورت کلاسیک از انواع اتوماتا برای بررسی پذیرش یا عدم پذیرش رشته در زبان استفاده می‌شه. اما این سطح از بررسی رشته، مثل خالی کردن حوض با قاشق می‌مونه. کار می‌کنه، اما سخته، و الکی سخته. برای بررسی رشته بلند، این کار رو توی دو مرحله انجام می‌دیم. این شکل رو ببینید:




قسمت اول، متن رو به دنباله ای از کلمات تجزیه می‌کنه، ما به این کلمات می‌گیم توکن. توی هر مرحله، تشخیص دادن اینکه از اینجا تا چقدر جلوتر میشه یک توکن، کار آسونیه، همچنین هر توکن یه نوع داره، کلمه کلیدی زبان (مثل if)، مقدار ثابت، عملگر، اسم یک چیز و غیره. تشخیص نوع کلمه هم کاری نداره. درحد عبارت‌های منظم. انقدر آسون که وقت شما رو باهاش نمی‌گیریم و می‌سپریم به کتابخونه‌هایی که RegEx رو تحلیل می‌کنن. این **نوع کلمه** میشه ورودی مرحله بعد.

قسمت دوم، یک اتوماتای پشته ایه که نوع کلمه ورودی رو می‌گیره، و بر اساس اون تشخیص می‌ده که ساختار این دنباله از کامات معنی‌دار هست یا نه.

مثلا توی زبان C:

- کلمه `10q4` کلمه معتبری نیست. هیچ کد C همچین توکنی نداره. این ارور از نوع بخش اوله
- عبارت `while if` عبارت معتبری نیست. هر دو تا توکن معتبرن و بخش اول اشکالی نمی‌گیره، اما پشت سر هم معنی ندارن. این ارور از نوع بخش دومه

پس اون اتوماتای پشته‌ای که قراره پیاده‌سازی کنید، خود متن رو بصورت مستقیم ورودی نمی‌گیره، بلکه به ترتیب نوع توکن‌ها رو ورودی می‌گیره. اما خود متن رو هم دور نریزید، شاید بعدا به درد خورد.

## ۲ اتوماتای پشته‌ای با پیش‌بینی (Lookahead PDA)

برخلاف مدل کلاسیک این ماشین می‌تونه یک نماد بعدی ورودی رو نگاه کنه (Lookahead) و براساس اون تصمیم بگیره که چه عملیاتی روی پشته انجام بده، و اینکه ورودی رو مصرف کنه یا نه. این قابلیت، انعطاف بیشتری برای پردازش ورودی بهمون میده، اما از نظر قدرت محاسباتی **معادل** اتوماتای پشته‌ای معمولیه. یعنی هر زبانی که توسط این مدل پذیرفته بشه، توسط یک اتوماتای پشته‌ای کلاسیک نیز قابل تشخیصه.

به بیان دیگه، هر ترنزیشن از این ماشین جدید، علاوه بر **عملیات معمول روی پشته** و **تغییر حالت**، یک آیتم lookahead هم داره که تعیین می‌کنه ماشین باید **نماد بعدی ورودی رو مصرف (consume) کنه** یا اون رو **دست‌نخورده بذاره** این ویژگی به ماشین اجازه می‌ده که بدون تغییر حالت فعلی، بر اساس نماد آیندهٔ ورودی تصمیم‌گیری کنه.

## ۳ گرامر LL1

گرامر LL1 یه فرم از گرامره که تضمین می‌کنه با یه ماشین پشته‌ای قطعی می‌تونیم بدون ابهام (ambiguity) اون زبان رو پارس کنیم. بصورت جزئی تر، این شرایط رو داره ($A, B$ non-terminal و $\alpha, \beta, \gamma$ دنباله‌ای از terminal و

non-terminal ها هستن)

- بازگشت به چپ نداره. یعنی قانونی به فرم $A \to A\alpha$ نداره.
- اگر $A \to \alpha \mid \beta$ اون موقع هیچ حالتی از $\alpha$ و هیچ حالتی از $\beta$ با ترمینال یکسانی شروع نمی‌شن.
- اگر $A \to \epsilon \mid \alpha$ و $B \to \gamma A \beta$ اون موقع هیچ حالتی از $\alpha$ و هیچ حالتی از $\beta$ با ترمینال یکسانی شروع نمی‌شن.

مثلا شرط دوم تضمین می‌کنه که $A$ با ترمینال بعدی مشخص، نمی‌تونه دو تا قانون تولید مختلف داشته باشه. این شرایط رو با مجموعه‌های First و Follow هم توضیح می‌دن که برای پیاده‌سازی و تشریح LL(k) برای kهای بزرگ‌تر کار رو راحت می‌کنه.

## لینک‌هایی برای توضیحات بیشتر

- https://www.youtube.com/watch?v=clkHOgZUGWU
- https://www.geeksforgeeks.org/construction-of-ll1-parsing-table/
- https://www.geeksforgeeks.org/ll1-parsing-algorithm/

### بخش سوم

# نمونه گرامر ورودی

## قبل از اینکه به گرامر برسیم، توجه کنید که این دو تا شرط باعث می‌شن پروژه خیلی آسون تر از پارسر زبان‌های متداول باشه

- شما با یک گرامر LL1 سر و کار دارید، که در یک جدول تجزیه کوچک نمایش داده می‌شود و با lookahead یک توکن پارس می‌شود. زبانی مثل C بدون درنظر گرفتن دستوراتی مانند `typedef`، گرامر LL2 دارد.
- توکن‌های متن ورودی با whitespace جدا شده‌اند. مثلا عبارتی شبیه `3+5` در کد ورودی نیست. به فاصله‌ها در `3 + 5` دقت کنید. همچنین داخل هیچ توکنی whitespace نیست. مثلا `"Hello World!"` یک توکن از این زبان نیست. اما `"HelloWorld!"` می‌تواند باشد.

# ۱ گرامر نمونه اول

```
START = E

NON_TERMINALS = E, E_prime, T, T_prime, F

TERMINALS = IDENTIFIER, LITERAL, PLUS, STAR, LEFT_PAR, RIGHT_PAR

E -> T E_prime
E_prime -> PLUS T E_prime | eps
T -> F T_prime
T_prime -> STAR F T_prime | eps
F -> LEFT_PAR E RIGHT_PAR | IDENTIFIER | LITERAL

IDENTIFIER -> [a-zA-Z_][a-zA-Z0-9_]*
LITERAL -> \d+(\.\d+)?
PLUS -> \+
STAR -> \*
LEFT_PAR -> \(
RIGHT_PAR -> \)
```

- خط START سمبل شروع گرامر رو مشخص می‌کنه.
- خط TERMINALS سمبل‌هایی که متناظر با یک نوع توکن از کد ورودی هستن رو مشخص می‌کنه. هر کدوم از این سمبل‌ها با یک عبارت منظم (RegEx) مشخص شدن
- خط NON_TERMINALS سایر سمبل‌ها رو مشخص می‌کنه که شبیه هر گرامر مستقل از متن دیگه‌ای ساختار کد ورودی رو نشون میدن. هر کدوم از این سمبل‌ها با تعدادی قانون تولید مشخص شدن که توی یک خط با | از هم جدا شدن.
- هر سمبل این فایل توی دقیق یکی از این دو دسته حضور داره
- یکی از سمبل‌های تومینال به اسم IDENTIFIER هست که اسم یک چیز (مثل متغیر) رو نشون میده. **فقط توکن‌هایی از این نوع می‌تونن برای تغییر نام انتخاب بشن.**

## کدهای نمونه این گرامر

```
( a + b ) * ( c + d + ( 123 ) )
```

```
a * b * c + d
```

# ۲ گرامر نمونه دوم

```
START = Program

NON_TERMINALS = Program, Function, Block, Statements, Statement, Expression,
    Expression_pr, Term, Term_pr, Factor

TERMINALS = FUNCTION, ID, NUM, IF, WHILE, RETURN, LEFT_PAR, RIGHT_PAR,
    LEFT_BRACE, RIGHT_BRACE, EQUALS, SEMICOLON, PLUS, MINUS, STAR, SLASH

# Grammar Productions
Program -> Function Program | eps
Function -> FUNCTION ID LEFT_PAR RIGHT_PAR Block
Block -> LEFT_BRACE Statements RIGHT_BRACE
Statements -> Statement Statements | eps
Statement -> ID EQUALS Expression SEMICOLON | IF LEFT_PAR Expression RIGHT_PAR
    Block | WHILE LEFT_PAR Expression RIGHT_PAR Block | RETURN Expression
    SEMICOLON
Expression -> Term Expression_pr
Expression_pr -> PLUS Term Expression_pr | MINUS Term Expression_pr | eps
Term -> Factor Term_pr
Term_pr -> STAR Factor Term_pr | SLASH Factor Term_pr| eps
Factor -> ID | NUM | LEFT_PAR Expression RIGHT_PAR

# Lexical Definitions (RegEx)
FUNCTION    -> /function/
IF          -> /if/
WHILE       -> /while/
RETURN      -> /return/
ID          -> /[a-zA-Z_][a-zA-Z0-9_]*/
NUM         -> /-?\d+(\.\d+)?([eE][+-]?\d+)?/
LEFT_PAR    -> /\(/
RIGHT_PAR   -> /\)/
LEFT_BRACE  -> /\{/
RIGHT_BRACE -> /\}/
EQUALS      -> /=/
SEMICOLON   -> /;/
PLUS        -> /\+/
MINUS       -> /-/
STAR        -> /\*/
SLASH       -> /\//
```

## کد نمونه این گرامر

```
function main ( ) {
    x = 42 ;
    y = 3.14 ;
    z = ( x + y ) * 2 ;

    if ( z ) {
        result = z / 1.5 ;
    }

    while ( x > 0 ) {
        x = x - 1 ;
    }

    return result ;
}
```