Subject:
Year. ۹۰ Month: ۸ Date. ۱۸ ( )

اصول طراحی کامپایلر — چهارشنبه

جلسه پنجم — سلیمانی تبار

واضح است که یکی از تولیدات A می‌تواند ebbb باشد. در source program فقط token جاری را می‌بینیم و در این نمونه مثال نمی‌دانیم چند بار $A \to Ab$ باید استفاده شود.

حذف بازگشتی از چپ:

بیاییم تولید $\begin{cases} A \to A\alpha \\ A \to \beta \end{cases}$ را بررسی کنیم.

$$A \xrightarrow{*} \beta\alpha^*$$

پس کار این است این تولید $\beta\alpha^*$ را به روش دیگری توسط تعدادی قانون دیگر تولید کنیم. که این قوانین بازگشتی از چپ نباشد.

$$A \to \beta K$$
$$K \to \alpha K \mid \epsilon$$

مثال:

$$\begin{matrix} A \to Ab \\ A \to e \end{matrix} \xrightarrow{\text{حذف بازگشتی از چپ}} \begin{matrix} A \to eK \\ K \to bK \mid \epsilon \end{matrix}$$

Source Program: ebbbbb$ → end of file

برای ورودی w: ebbbbb توکن جاری e می‌باشد که باید قانون $A \to eK$ را استفاده کند، توکن جاری e صحیح است و بقیه ورودی یعنی bbbbb توسط K از $K \to bK$ تولید می‌شود و به همین ترتیب تمام b ها توسط $K \to bK$ تولید می‌شود و به همین ترتیب تمام b ها توسط $K \to bK$ بررسی می‌شود تا وقتی که به $ برسیم که می‌فهمیم باید K به ϵ برود.

فرم کلی noun terminal های بازگشتی از چپ:

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_n$$
$$A \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \cdots \mid \beta_m$$

اصول طراحی کامپایلر — جلسه چهارشنبه

مثال:

$A \to AbD \underset{\alpha_1}{} \mid AcFe\ \alpha_2$

$A \to \underset{\beta_1}{mD} \mid \underset{\beta_2}{nP} \mid \underset{\beta_3}{aB}$

تولید A برابر است با

$$A \overset{*}{\Rightarrow} (\beta_1 + \beta_2 + \beta_3 + \cdots + \beta_m)(\alpha_1 + \alpha_2 + \alpha_3 + \cdots + \alpha_n)^*$$

حذف بازگشتی از چپ:

$A \to mDK \mid npK \mid aBK$

$K \to bDK \mid cfeK \mid \epsilon$

فرم‌های دارای فاکتور مشترک چپ:

یک noun terminal دارای فاکتور مشترک چپ به فرم کلی زیر است:

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n$$

به طوریکه ابتدای هیچ کدام از β_i ها با α یکسان نیست.

برای حذف از قانون تولید زیر استفاده می‌کنیم:

$$A \overset{*}{\Rightarrow} \alpha(\beta_1 + \beta_2 + \cdots + \beta_n)$$

قوانین جدید: (حذف فاکتور مشترک چپ) (فاکتورگیری از چپ) left factoring

$$\begin{cases} A \to \alpha K \\ K \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \cdots \mid \beta_n \end{cases}$$

مثال:

$A \to aBD \mid aBC$
$B \to bD$
$D \to d$
$C \to c$

left factoring ⇒

$A \to aBK$
$K \to D \mid C$
$B \to bD$
$D \to d$
$C \to c$

Subject:
Year. ۹۰ Month. ۸ Date. ۱۸ ( ) چهارشنبه اصول طراحی کامپایلر

Source Program : abdd

با دیدن اولین Token نمی توانیم تشخیص بدهیم اگر کدام مسیر را برویم از voice

واضح است که با a ، اگر A→aBC را انتخاب کنیم کامپایلر جاری Token اعلام خطا خواهد کرد ، یعنی رشته عضو زبان نیست !!!

مثال : A→ abcDB | abFD | abcL

در این مثال دو مرحله فاکتورگیری داریم

left factoring ↘
A→ abcK | abFD
K→ DB | L

left factoring ↘
A→ abH
H→ cK | FD
K→ DB | L

مبحث مهم follow , first

توجه : این مبحث در تمامی درس ها در این به بعد کاربرد دارد .

ضرورت بحث :

S→ ADF | BLM
A→ DaD
B→ bLf
D→ dD | ε
F→ fIe
L→ m | n
M→ aA | ε

مثال : Source Program: adddde

برای رشته ی addde بایستی تصمیم بگیریم a ی token بایستی به کدام یک از قوانین $S \to ADF$ و یا $S \to BLM$ ربط دهیم. با داشتن token ابتدای ADF و ابتدای BLM می توانیم تصمیم به انتخاب بگیریم زیرا a Token بعنوان ابتدای تولید ADF خواهد بود. پس باید ی نون $S \to ADF$ را ربط دهیم.

به این مفهوم first می گوییم.

تعریف first(α) برای α که یک | token | noun terminal | یک سمت راست یک ی نون

می باشد. first(α) عبارتست از مجموعه ی token هایی از ترمینال که می توانند در ابتدای تمامی تولیدات α بیایند. هم چنین اگر α بتواند ϵ تولید کند یعنی ($\alpha \overset{*}{\Rightarrow} \epsilon$) آنگاه ϵ نیز عضو first(α) خواهد بود.

مثال: برای ترمینال زیر first تمامی قوانین و first تمامی noun terminal ها را محاسبه کنید.

$S \to AaB$

$A \to dBD \mid eB \mid BD$

$B \to bB \mid \epsilon$

$D \to dD \mid m \mid \epsilon$

حل:

$first(D \to dD) = \{d\}$

$first(D \to m) = \{m\}$ $\Rightarrow$ $first(D) = \{d, m, \epsilon\}$

$first(D \to \epsilon) = \{\epsilon\}$

$first(B \to bB) = \{b\}$

$first(B \to \epsilon) = \{\epsilon\}$ $\Rightarrow$ $first(B) = \{b, \epsilon\}$

first(A→dBD) = {d}

first(A→BD) = {b,d,m,ε}

توجه: اگر B، b را تولید کند اول این رشته b و در صورت تولید ε توسط B، اول رشته‌ی بعدی یا اول D خواهد بود.

first(A→eB) = {e}

⇒ first(A) = {b,d,m,e,ε}

first(S→AaB) = {b,d,m,e,a}

⇒ first(S) = {b,d,m,e,a}

مجموعه قدمهایی برای محاسبه‌ی $first(A)$ به طوریکه $A \in V_N$

قدم صفر: مجموعه‌ی first(A) را به صورت تهی فرض می‌کنیم. first(A) = {}

قدم ۱: اگر برای A در گرامر قانونی به فرم $A \to Y_1 Y_2 Y_3 \dots Y_{i-1} Y_i Y_{i+1} \dots Y_k$ وجود داشته باشد و اگر به ازای $j = 1, 2, \dots, i-1$ داشته باشد $\varepsilon \in first(Y_j)$ و $Y_p \in V_N \cup V_T$ *

آنگاه first(Y_i) را به غیر از ε به first(A) اضافه می‌کنیم.

قدم ۲: اگر در قانون قدم ۱، برای $j = 1, 2, \dots, k$، $\varepsilon \in first(Y_j)$ باشد، آنگاه ε را به first(A) اضافه می‌کنیم.

قدم ۳: قدمهای فوق را (۱ و ۲) آنقدر تکرار می‌کنیم تا هیچ token (یا ε) دیگری به first(A) اضافه نگردد.

مثال:

S → ADB | BFE

A → aB | FD

* $Y_p \in V_N \cup V_T$ یعنی Y_p یا یک token است یا یک Noun Terminal

$B \to bD \mid BEe$ $\quad first(E) = \{m, \epsilon\}$

$D \to cFE \mid FE$ $\quad first(F) = \{e, \epsilon\}$

$F \to eFe \mid \epsilon$ $\quad first(D) = \{c, e, m, \epsilon\}$

$E \to m \mid \epsilon$ $\quad first(B) = \{b\} \cup first(BEe)$ ← first(B)

$first(B) = \{b\} \cup first(B) = \{b\}$ چون B هیچگاه ϵ نمی‌شود.

$first(A) = \{a, e, c, m, \epsilon\}$

$first(S) = \{a, b, c, m, b\}$

S سمبل شروع هر گرامری است، پس هر Source Program ی را داشته باشیم با یکی از عضوهای مجموعه‌ی first(S) شروع خواهد شد.

توجه: چون b عضو $first(S \to ADB)$ و $first(S \to BFE)$ هست پس Source Program ما با b شروع شود ما نمی‌توانیم تشخیص دهیم از کدام گرامر استفاده کنیم.

Subject:
Year. ۹۰ Month. ۸ Date. ۲۵ ( )

جلسه ششم — چهارشنبه

اصول طراحی کامپایلر — سلیمانی تبار

بحث follow ها :

تعریف follow(A) برای $A \in V_N$

مجموعه‌ای از token هایی که بعد از یک token ویژه به نام $ یا اصطلاحا (end of source program) می‌توانند دقیقا بعد از تولیدات A بیایند. درواقع خود Noun Terminal، follow را تولید نمی‌کند.

چند حالت داریم:

① if S is the start symbol then $S \xRightarrow{*} Source\ Program\$$

$\rightarrow \$ \in follow(S)$

① $A \rightarrow \alpha B \beta$ اگر قانونی به فرم در گرامر داشته باشیم follow(B) را تعیین کنید.

$?\in follow(B)$:

then $first(\beta) - \{\epsilon\} \subseteq follow(B)$

یعنی ابتدای β، followB می‌شود. یعنی هرچه اول β باشد جزو follow(B) می‌شود.

② if $A \rightarrow \alpha B$ then $follow(A) \subseteq follow(B)$

یعنی هرچه بعد از follow(A) باشد جزو بعد از follow(B) می‌شود.

مثال

$L \rightarrow A\delta$

$A \rightarrow \alpha B$

می‌دانیم در قانون $L \rightarrow A\delta$ هرچه که $first(\delta)$ باشد بعد از follow A خواهد بود.

می‌توان A را با استفاده از قانون $A \rightarrow \alpha B$ با αB جایگذاری کرد بنابراین هرچه که بعد از follow(A) بوده است حال به عنوان follow(B) نیز خواهد بود.

Subject:
Year. Month. Date. ( )

مجموعه قدمهای محاسبه‌ی follow(B) برای $B \in V_N$

قدم صفر: اگر S سمبل شروع گرامر باشد آنگاه $ را بعنوان مقدار اولیه به follow(S) می‌افزاییم و همچنین برای هر $B \neq S$، follow(B) را تهی فرض می‌کنیم.

قدم ۱: اگر قانونی به فرم $A \rightarrow \alpha B \beta$ در گرامر باشد آنگاه first(β) − {ε} را به follow(B) می‌افزاییم. همچنین اگر $\varepsilon \in$ first(β) باشد، آنگاه follow(A) را به follow(B) می‌افزاییم.

قدم ۲ − (حالت خاص قدم ۱): اگر قانونی به فرم $A \rightarrow \alpha B$ در گرامر باشد، آنگاه follow(A) را به follow(B) می‌افزاییم.

قدم ۳: قدمهای فوق را آنقدر تکرار می‌کنیم تا هیچ token دیگری به follow(B) اضافه نشود.

مثال:

$S \rightarrow ABCd$

$A \rightarrow mAn \mid CB$

$B \rightarrow Bfs \mid CC$

$C \rightarrow ec \mid \varepsilon$

ابتدا first و سپس follow را محاسبه می‌کنیم.

$first(C) = \{e, \varepsilon\}$

$first(B) = \{e, f, \varepsilon\}$

$first(A) = \{m, e, f, \varepsilon\}$

$first(S) = \{m, e, f, d\}$

follow(S) = {$} ∪ follow(B)

follow(A) = first(BCd) ∪ {n}

↓ جزء اول ↓ جزء دوم

follow(A) = {e, f, d} ∪ {n} = {e, f, d, n}

follow(B) = first(Cd) ∪ follow(A) ∪ first(fS) = {e, d} ∪ {e, f, d, n} ∪ {f}

= {e, f, d, n}

follow(C) = {d} ∪ {e, f} ∪ follow(A) ∪ first(C) ∪ follow(B) = {d, e, f, n}

مثال:

S → AbBa

A → dBc | BSA

B → BbA | DBD | D

D → dDdSB | ϵ

first(D) = {d, ϵ}

first(B) = {b, d, ϵ}

first(A) = {d, b} ∪ first(SA)

first(S) = {d, b}

⇒ first(A) = {d, b}

follow(S) = {$} ∪ first(B) ∪ follow(D) − {ϵ} ∪ first(A) − {ϵ} =

{$, d, b} ∪ follow(D)

follow(A) = first(bBa) ∪ follow(A) ∪ follow(B)

↓ که آن را اضافه نمی‌کنیم

یادآوری { A = AUA
A = AΠA

Subject:
Year. Month. Date. ( )

follow(B) = {a} ∪ {b} ∪ first(SA) ∪ first(bA) ∪ first(D) ∪ follow(D)
- {ϵ} - {ϵ}

follow(D) = {d, b} ∪ follow(B)

---

follow(B) = {a, c, d, b}

follow(D) = {a, c, d, b}

} باهم برابرند.

first(S) = {$, d, b, a, c}

* مرتبه‌ی زمانی الگوریتم polynomial می‌باشد.

طراحی تک تک قابلیت‌های کامپایلر به طور جمع

تحلیل و طراحی قابل تحلیل لغوی (Lexical A.) Lexical Analyzer

پس از آنکه طراح زبان گرامر Context free مربوط به زبان را طراحی کرد و token های زبان مشخص شدند، token زبان دسته‌بندی شده و برای هر دسته از token های یک زبان یک NFA با یک حالت شروع طراحی می‌کنیم.

در یک NFA از حالت شروع با دیدن به یک حالت پایانی یک token خاص را از آن دسته token بررسی می‌کنیم. در حالت پایانی که token مربوط را برمی‌گردانیم تمام NFA ها را پس از طراحی به صورت یک NFA با چندین حالت شروع در نظر می‌گیریم که در مجموع تمامی token های زبان را شامل می‌شوند.

بعد از آن به پیاده‌سازی NFA کلی با چندین حالت شروع (به تعداد دسته‌های token) می‌پردازیم.

Subject:
Year. Month. Date. ( )

به طور کلی در یک زبان دسته token های زیر وجود دارند:

الف) دسته tokenهای مربوط به عملگرهای رابطه‌ای (مقایسه‌ای) (relop)

خانواده‌ی پاسکال: = ، <> ، <= ، >= ، < ، >

خانواده‌ی C: == ، != ، <= ، >= ، < ، >

ب) دسته tokenهای مربوط به عملگرهای حسابی، منطقی، shift

خانواده‌ی پاسکال: + ، - ، * ، / ، AND ، OR ، NOT ، ...

خانواده‌ی C: + ، - ، * ، / ، ++ ، -- ، += ، -= ، && ، & ، | ، ! ، || ، ...

ج) دسته tokenها کلمات کلیدی، کلمات رزرو، idهای توقعی برنامه‌نویس

پاسکال: if ، begin ، end ، SUM ، SQRT ، ...

C: if ، while ، ...

د) دسته tokenهای عددی مثل اعداد صحیح، اعشاری، حقیقی و غیره ...

ه) دسته tokenهای جداساز مثل ; ، (،) ، { ، } ، [،] ، : ، ...

Subject :

Year . Month . Date . ( )

Subject:
Year. ۹۰ Month. ۹ Date. ۲ ( ) چهارشنبه

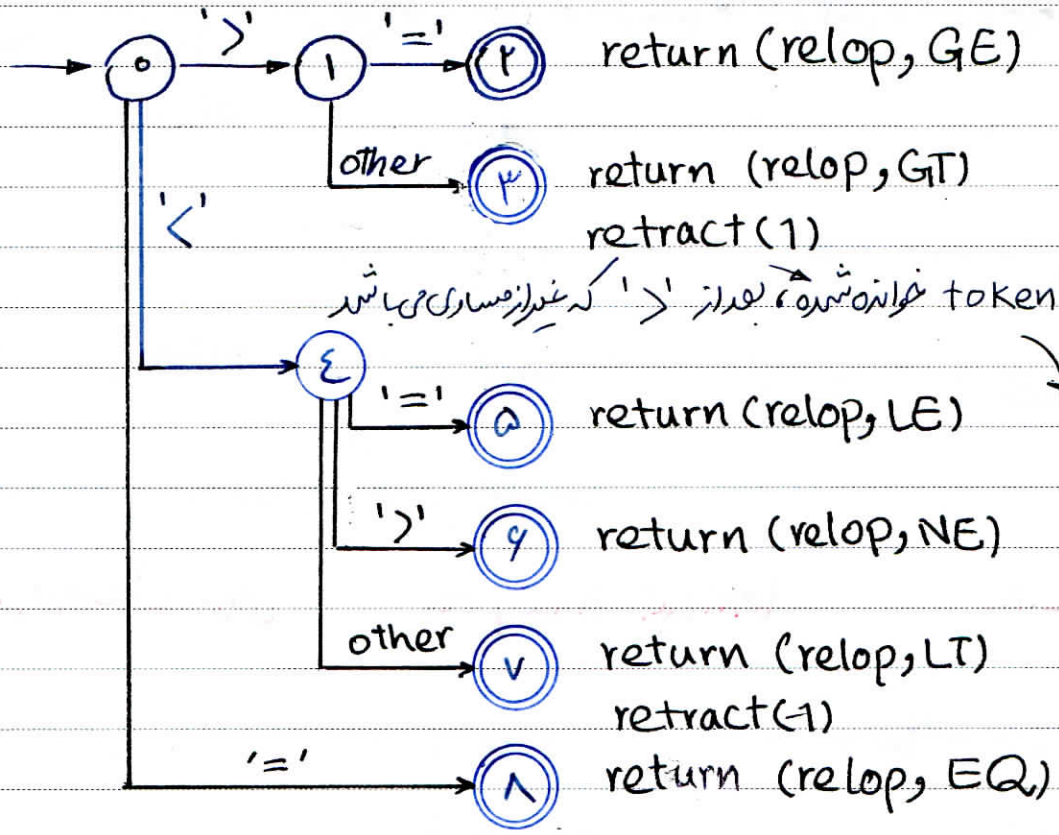
اصول طراحی کامپایلر

جلسه‌ی هفتم.

استاد سلیمانی تبار

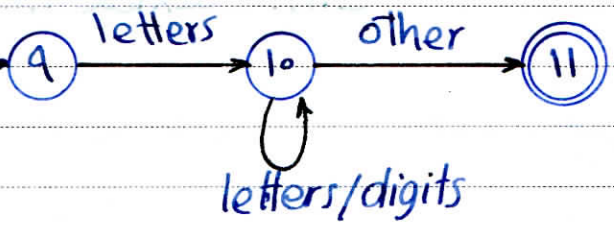
دسته‌ی اول: NFA برای دسته‌ی اول relop : relation operator

pascal : > , >= , < , <= , <> , =



دسته‌ی دوم: tokenid برای NFA token هایی از جنس id

شامل کلمات کلیدی، reserve words، id های تعریف شده کاربر

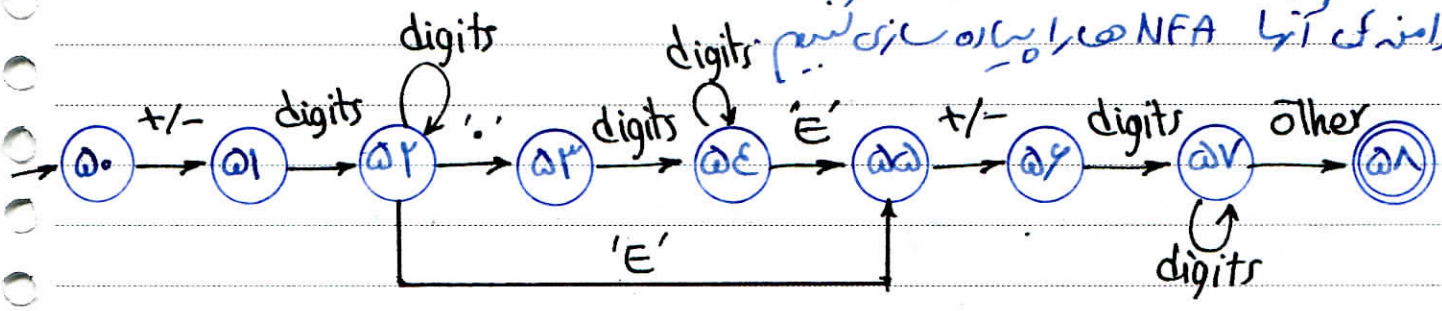
(حرف)(حرف|رقم)* 

توضیح: در این NFA با رسیدن به state شماره‌ی ۱۱، رشته‌ی بازشده را بعنوان یک token، ابتدا با کلمات رزرو سپس با کلمات کلیدی مقایسه می‌کنیم

Subject:
Year. ۹۰ Month. ۹ Date. ۲ ( ) چهارشنبه

چنانچه هیچ یک از این دو مورد نبود رشته‌ی مربوط به یک id تعریف خواهد بود.

token های عددی برای رشته token های عددی نیز می‌توانیم با اولویت بندی بزرگی دامنه‌ی آنها NFA ها را پیاده سازی کنیم



* به همین ترتیب برای تمامی رشته token های مشابه NFA ها را رسم می‌کنیم و پس از رسم همه NFA ها را به صورت یک NFA با چندین حالت شروع در نظر می‌گیریم.

(نحوه‌ی پیاده سازی قبلا توضیح داده شد.)

ادامه‌ی مبحث طراحی lexical Analyzer L.A.

برنامه‌ی زیر به پیاده سازی بخشی از NFA های فوق می‌پردازد و قابلیت تکمیل و تعمیم دارد. برنامه‌ی زیر شامل NextToken ، token جاری را Source Program جدا کرده و کد آنرا برمی‌گرداند. در حقیقت این برنامه همان Lexical Analyzer است.

طراحی L.A. lexical Analyzer

برای پیاده سازی و نوشتن برنامه‌ای برای L.A. متغیری بنام state در نظر می‌گیریم. و با شروع از حالت شروع اولین NFA در NFA مربوطه، آن قدر جلو می‌رویم (تغییر حالت می‌دهیم) تا به یک پایانی برسیم و یا در آن شکست بخوریم. NFA (fail کنیم)

Subject:
Year. ۹۰ Month. ۹ Date. ۲ ( ) چهارشنبه

اگر به شکست خوردیم کار را با شروع از حالت شروع NFA بعدی آغاز می‌کنیم.

اگر token جاری معتبر باشد حتما دریکی از NFA ها شناسایی خواهد شد.

برای مثال: if a>b

برنامه‌ی زیر به پیاده سازی بخشی از NFA های فوق می‌پردازد و قابلیت تکمیل و تعمیم دارد. برنامه‌ی زیر بنام NextToken ، token جاری را از SourceProgram جدا کرده و کد آنرا برمی‌گرداند. در حقیقت این برنامه همان LexicalAnalyzer است.

```
token NextToken() {

    lexicalbegining = forward;

    while (1) {
        switch (state) {

            case ø:

                C = nextchar();

                if (C == '>') state = 1;
                else if (C == '<') state = 4;
                else if (C == '=') state = 8;

                else state = fail();

                break;

            case 1:

                C = nextchar();
                if (C == '=') state = 2;
                else state = 3;
```

```
            break;
        case 2:
            return (relop, GE);
            break;
        case 3:
            retract(1);
            return (relop, GT);
            break;
            ⋮
        case 9:
            C= nextchar();
            if (isletter(C)) state = 10;
            else state = fail();
            ⋮
    } /* end of switch */
} /*end of while*/
} /*end of next token*/
```

مثال: برخورد تابع NextToken با عبارت SUM)=AVE چیست؟

ما با تابع fail() بجایی برخواهیم گشت که قبل از شکست می باشد.

تابع fail()

توجه: forward ، pointer ای است که در source program جلو می رود. lexical begining جایی است که ما از آنجا به بعد دنبال جداکردن token

Subject:
Year. ۹۰ Month. ۹ Date. ۲ ( )

چهارشنبه

```
state fail() {
    forward = lexicalbegining;
    switch (start) {
        case 0:
            start=9;
            break;
        case 9:
            start=50;
            break;
            :
    }
    return start;
}
```

* با قرار دادن forward=lexicalbegining; در ابتدای تابع fail ما به جایی برخواهیم گشت که به شکست خورده‌ایم. دوباره با دیدار همان مسیر NFA ها را برای استخراج token جدید بکار برم.

* نوشتن این برنامه با استفاده از خواص NFA ها به راحتی انجام می‌شود.

طراحی کامل فاز تحلیلگر لغوی (Parser) (SyntaxA.) Syntax Analyzer

انواع پارسرها:

پارسرها
- پارسرهای از بالا به پایین topdown
  - پارسرهای پیشرو ← iterative
    - Recursive Decsend
    - LL(K)
- پارسرهای از پایین به بالا: bottom up ← Shift reduce
  - بدون اپراتور
    - LR(0)
    - SLR
    - CLR
    - LALR
    - Optimal_LR
  - با اپراتور
    - LR(K)

پارسرهای RD: Recursive Decsent

در پارسرهای RD: Recursive Decsent برای هر Noun Terminal از گرامر یک زیر برنامه بازگشتی (مستقیم یا غیر مستقیم) می‌نویسیم و در بدنه‌ی آن زیر برنامه

Subject:
Year. Month. Date. ( )

با استفاده از if-then-else های تودرتو تک تک سمت راست های قوانین مربوط به آن Noun Terminal را پیاده سازی می کنیم.

برای پیاده سازی یک سمت راست یک قانون، ؟ first جاری را با token آن قانون مقایسه می کنیم. درصورت برابری اجازه ی ورود به سمت راست را داریم و در پیاده سازی آن سمت راست دیدن هر token را با عبارت match(token) جلو می بریم. و دیدن هر Noun Terminal را برعهده ی زیر برنامه ی هم نام آن Noun Terminal می گذاریم.

در برنامه ی اصلی (main program) زیر برنامه ی مربوط به سمبل شروع را فراخوانی می کنیم. کد جاری همواره در متغیری بنام lookahead خواهد بود.

```
main() {
  :
  lookahead = nexttoken();
  S(); ← سمبل شروع گرامر ما S می باشد.
  if (lookahead == '$') accept();
  :
}
token match(token t)
{
  if (lookahead == t)
      lookahead = nexttoken();
  else error_recovery();
}
```

* عملیات match چه کاری انجام می دهد؟ token مورد انتظار

تابع match یک token را می گیرد و با token جاری برابری می کند و اگر برابر بود یک token به جلو می رود.

Subject:
Year. ٩٠ Month. ٩ Date. ٢ ( )

مثال: گرامر Context Free زیر را از نظر گرفته و پارسر Recursive Decsent مربوط به آن را بنویسید:

S → aBd | FE

B → bB | d

F → mEn | dn

E → SB

زیربرنامه‌ی SO هنی باید چیزی را که S تولید می‌کند درست آنالیز کند.

زیربرنامه‌ی S

```
S() {
  if (lookahead == 'a')
  {
    match ('a');
    B();
    match('d');
  }
  else if (lookahead in {m,d})
  {
    F();
    E();
  }
  else Error_Recovery();
}
```

S → aBd

S → EF

زیربرنامه‌ی B

```
B() {
    if (lookahead == 'b')
    {
      match('b');
      B();
    }
```

```
    else if (lookahead == 'd')
        match('d');
    else
        Error_Recovery();
}
```

زیربرنامه‌ی F

```
F() {
  if (lookahead == 'm') {
      match('m'); E(); match('n');}
  else if (lookahead == 'd') {
      match('d'); match('n');}
  else Error_Recovery();
}
```

زیربرنامه‌ی E

```
E() {
  if (lookahead in {a,m,d})
  {
    S();
    B();
  }
  else Error_Recovery();
}
```

مثال:

Source Program: abbdd

Subject:
Year. ٩٠ Month. ٩ Date. ٩ ( )

اصول طراحی کامپایلر — چهارشنبه

جلسه‌ی هشتم — استاد سلیمی نیکبار

معایب و مزایای پارسرهای R.D. (Recursive Decsent)

برای گرامرهای ساده با عمق بازگشتی کم به راحتی می‌توان در کمترین زمان یک پارسر R.D. طراحی کرد.

اگر گرامر دارای عمق بازگشتی زیاد باشد ممکن است Stack Overflow رخ دهد.

همچنین این پارسرها برای گرامرهای بازگشتی لزجب و گرامرهای دارای فاکتور مشترک چپ قابل استفاده نیست.

فرم کلی گرامرهای بازگشتی لزجب $A \to A\alpha | \beta$

$first(\beta) \subseteq first(A\alpha)$

```
A() {
  if (lookahead ∈ first(Aα)) {
    A();
    process(α);
  }
  else if (lookahead ∈ first(β))
    process(β);
  else Error_Recovery();
}
```

در این الگوریتم اگر یک گرامر بازگشتی لزجب داشته باشیم یک حلقه‌ی بینهایت رخ می‌دهد و به طور قطع Stack Overflow رخ می‌دهد.

مشکل دوم گرامرهای دارای فاکتور مشترک چپ:

$A \to \alpha\beta_1 | \alpha\beta_2$

```
A() {
```

Subject:
Year. Month. Date. ( )

```
if (lookahead ∈ first(αβ1))
    process(αβ1)
else if (lookahead ∈ first(αβ2))
    process(αβ2)
else error_recovery();
}
```

$first(\alpha) = first(\alpha\beta_1) \cup first(\alpha\beta_2)$

در اینحالت ممکن است برنامه‌ی عاری از خطا، آلوده به خطا تشخیص داده شود، زیرا میپریم به اشتباه می‌رویم.

ضمن: دقت کنید در گرامرهای دارای فاکتور مشترک از چپ چون $first(\alpha\beta_1)$ با $first(\alpha\beta_2)$ اشتراک دارد در اینحالت ممکن است ما با خواندن token جاری از Source Program token ی را بخوانیم که عضو $first(\alpha\beta_1) \cap first(\alpha\beta_2)$ باشد.

در اینصورت چه خبر؟ در خواندن token جاری از source program، token مربوط به $first(\alpha\beta_2)$ را بخوانیم چون در if اول ما با خواندن این token (token جاری) وارد این if شویم. چون $first(\alpha\beta_1)$ نیز با $first(\alpha\beta_2)$ اشتراک دارد و ما با خواندن token جاری از source program که باید قانون $A \to \alpha\beta_2$ را استفاده می‌کردیم به اشتباه وارد $\alpha\beta_1$ می‌شویم و در ادامه جای اشتباه تشخیص داده می‌شود و توسط L.A پذیرفته نمی‌شود.

Subject :
Year . Month . Date . ( )

پارسرهای LL(K)

این مدل از پارسرها فراخوانی بازگشتی ندارند (هر چند از stack استفاده می‌کنند.)

در این پارسرها Source Program از چپ به راست (left to right) token بندی می‌شود. (scan می‌شود) و براساس اشتقاق‌های سمت چپ ترین (left most derivation) با استفاده از K تا token متوالی عملیات parssing را انجام می‌دهند.

LL(K)

Left to Right scan : token بندی چپ به راست

Left most derivation : اشتقاق سمت چپ ترین

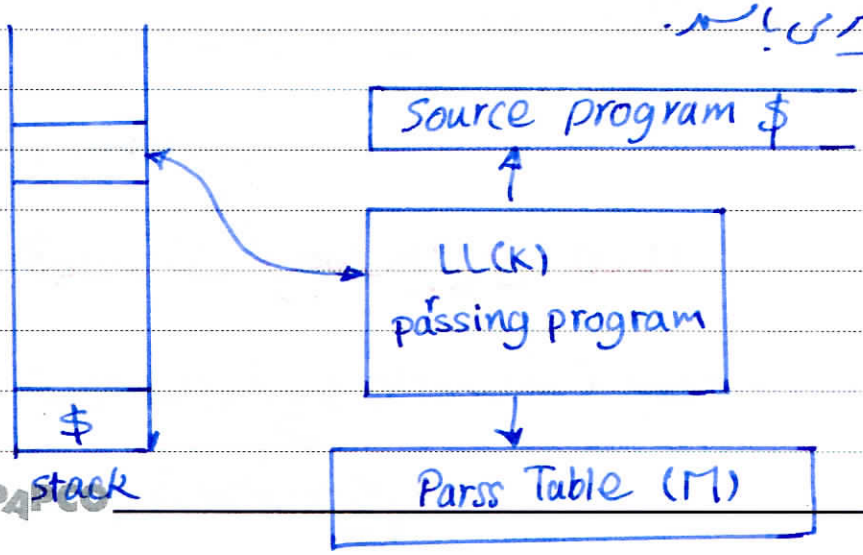
With K Symbols lookahead یا with k Token lookahead

از یک به کجا می‌رویم / نزدیک‌ترین نقاط به زبان

مثال: این گرامر به ظاهر فاکتور مشترک LL(1) نیست ولی LL(2) می‌باشد.

A → abBC | aCEF

معماری این پارسرها به صورت زیر می‌باشد.



K محدود و ثابت یعنی نمی‌تواند در داخل برنامه تغییر کند.

Subject:
Year. Month. Date. ( )

اگر بخواهیم برای یک گرامر context free برای یک K داده شده (که در واقع ثابت finite است) یک پارسر LL(K) داشته باشیم ابتدا اصول تجزیه LL(1) آن را به دست آورده و در LL(K) برنامه parssing پارسرهای LL(K) قرار داده که از آن به بعد یک پارسر LL(K) برای زبان مربوطه خواهیم داشت.

فرض کنید ما یک گرامر داریم:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

ما چطور تشخیص بدهیم از کدام قانون استفاده کنیم؟

جدول تجزیه‌ی LL(1)

جدولی است دوبعدی که سطرهای آن برچسب نام یکی Noun Terminal گرامر را دارد و ستونهای آن برچسب یکی Token گرامر به همراه ویژه‌ی $ (end of file/source program) و درخانه‌ای از جدول پایینی هر کدام یک قانون قرار گیرد که تلاقی یک سطر و ستون شرایط ورود به تولید مربوط به یک Noun Terminal را براساس Token‌های آنشان می‌دهد، مثال:

$S \rightarrow aBD \mid eF$

$B \rightarrow dF \mid \in$

نحوه‌ی ساخت یک جدول تجزیه‌ی LL(1)

مجموع قدمهای زیر را برای ساخت جدول LL(1) برای گرامر Context Free بنام G در نظر می‌گیریم:

Subject:
Year. Month. Date. ( )

قدم صفر: یک جدول خالی با $|V_N|$ سطر و $|V_T|+1$ ستون با نام‌های مربوط تعریف می‌کنیم یا در نظر می‌گیریم.

قدم ۱: اگر در گرامر قانونی به فرم $A \to \alpha$ وجود داشته باشد، برای تمامی $a \in first(\alpha)$ بصورت $M[A,a] = "A \to \alpha"$ قرار می‌دهیم:

قدم ۲: اگر در گرامر قانونی به فرم $A \to \alpha$ وجود داشته باشد و $\epsilon \in first(\alpha)$ باشد آنگاه برای تمامی $b \in follow(A)$ قرار می‌دهیم: $M[A,b] = "A \to \epsilon"$

قدم ۳: قدم‌های ۱ و ۲ را آنقدر تکرار می‌کنیم تا هیچ قانون دیگری اضافه نگردد.

قدم ۴: خانه‌های خالی جدول به معنای Syntax Error می‌باشند و باید به روتین‌های error_recovery بپردازیم.

مثال: گرامر Context free زیر را در نظر گرفته و جدول LL(1) آن را تشکیل دهید.

$S \to abA \mid aB$ $\quad first(S) = \{a\}$ $\quad follow(S) = \{\$, f\}$

$A \to dB \mid \epsilon$ $\quad first(A) = \{d, \epsilon\}$ $\quad follow(A) = \{\$, f, a\}$

$B \to eASf \mid \epsilon$ $\quad first(B) = \{e, \epsilon\}$ $\quad follow(B) = \{\$, f, a\}$

	a	b	d	e	f	$
S	* $S \to abA$ $S \to aB$	e_1	e_2	e_3	e_4	e_5
A	$A \to \epsilon$	e_6	$A \to dB$	e_7	$A \to \epsilon$	$A \to \epsilon$
B	$B \to \epsilon$	e_8	e_9	$B \to eASf$	$B \to \epsilon$	$B \to \epsilon$

Subject:
Year. Month. Date. ( )

* دو قانون در یک خانه داریم، شرط LL(1) نقض شد.

تعریف گرامر LL(1): گرامر Context Free بنام G را LL(1) گوییم ...

اگر و تنها اگر در هر خانه از جدول حداکثر یک قانون از گرامر قرار گیرد.

با توجه به این تعریف گرامر مثال قبل LL(1) نیست.

* این مثال LL(2)، LL(3) هم نمی باشد.

مثال: گرامر Context free زیر را در نظر گرفته و جدول LL(1) آنرا تشکیل دهید:

$E \to TE'$	$first(E) = \{(, id\}$	$follow(E) = \{\$,)\}$
$E' \to +TE' \mid \epsilon$	$first(E') = \{+, \varepsilon\}$	$follow(E') = \{\$,)\}$
$T \to FT'$	$first(T) = \{(, id\}$	$follow(T) = \{+, \$,)\}$
$T' \to *FT' \mid \epsilon$	$first(T') = \{*, \varepsilon\}$	$follow(T') = \{+, \$,)\}$
$F \to (E) \mid id$	$first(F) = \{(, id\}$	$follow(F) = \{*, +, \$,)\}$

* id یک token می باشد.

	+	*	(	)	id	$
E	e_1	e_2	$E \to TE'$	e_3	$E \to TE'$	e_4
E'	$E' \to +TE'$	e_5	e_6	$E' \to \varepsilon$	e_7	$E' \to \varepsilon$
T	e_8	e_9	$T \to FT'$	e_{10}	$T \to FT'$	e_{11}
T'	$T' \to \varepsilon$	$T' \to *FT'$	e_{12}	$T' \to \varepsilon$	e_{13}	$T' \to \varepsilon$
F	e_{14}	e_{15}	$F \to (E)$	e_{16}	$F \to id$	e_{17}

Subject:
Year. Month. Date. ( )

برنامه‌ی parssing و پارسرهای LL(1)

مجموع قدمهای زیر را درنظر می‌گیریم:

قدم ۰: $ را در ته stack خالی قرار می‌دهیم و بر روی آن سمبل شروع گرامر را می‌گذاریم

و سپس a را بعنوان token ی در حال lookahead درنظر گرفته که با هر بار Next Token یک token جلو می‌رود. هم‌چنین X را بعنوان سمبل روی stack درنظر می‌گیریم.

قدم ۱: زیرقدمهای زیر را آنقدر تکرار می‌کنیم تا X برابر $ شود. (یعنی stack خالی گردد)

قدم ۱-۱: اگر X برابر a باشد آنگاه X را از روی stack پاک کرده و دور می‌اندازیم و در source program یک a = next token() انجام می‌دهیم.

قدم ۱-۲: اگر X یک Noun Terminal برابر A باشد

قدم ۱-۲-۱: اگر $M[A,a] = "A \to Y_1 Y_2 \cdots Y_k"$ باشد آنگاه A را از stack pop کرده و به جای آن Y_k ، Y_{k-1} ، Y_{k-2} ، ... ، Y_2 ، Y_1 قرار می‌دهیم بطوریکه

قدم ۱-۲-۲: اگر $M[A,a]$ خالی باشد به error-recovery می‌رویم.

قدم ۲: اگر در ورودی source program به $ رسیده‌ایم و stack نیز به $ رسیده برنامه خطا ندارد.

Subject:
Year. Month. Date ( )

مثال: گرامر Context Free و جدول مثال قبل را در نظر گرفته و برای LL(1) parsing داده شده با استفاده از برنامه‌ی Source Program فوق مرحله به مرحله مراحل parsing را با استفاده از stack برای $id_1 + id_2 * id_3$ داده شده source program بررسی کنید:

	stack ($S)	source program $	action (outputs)
۰	$E → stack کن	$id_1 + id_2 * id_3$	$E \to TE'$
۱	$E'T	$id_1 + id_2 * id_3$	$T \to FT'$
۲	$E'T'F	$id_1 + id_2 * id_3$	$F \to id$
۳	$E'T'id	$id_1 + id_2 * id_3$	pop(); NextToken();
۴	$E'T'	$+ id_2 * id_3$	$T' \to \varepsilon$
۵	$E'	$+ id_2 * id_3$	$E' \to +TE'$
۶	$E'T+	$+ id_2 * id_3$	pop(); NextToken();
۷	$E'T	$id_2 * id_3$	$T \to FT'$
۸	$E'T'F	$id_2 * id_3$	$F \to id$
۹	$E'T'id	$id_2 * id_3$	pop(); NextToken();
۱۰	$E'T'	$* id_3$	$T' \to *FT'$
۱۱	$E'T'F*	$* id_3$	pop(); NextToken();
۱۲	$E'T'F	id_3	$F \to id$
۱۳	$E'T'id	id_3	pop(); NextToken();
۱۴	$E'T'	$	$T' \to \varepsilon$
۱۵	$E'	$	$E' \to \varepsilon$
۱۶	$	$	accept();

Subject:
Year. Month. Date. ( )

از مشکلات پارسرهای LL(K) آن است که خیلی از زبانها LL(1) نیستند و بازگشتی از چپ و دارای فاکتور مشترک از چپ هستند و نمی توان این روش را برای این زبانها با گرامرهای بازگشتی از چپ و دارای فاکتور مشترک از چپ به کار برد.