کامپایلر

مراجع

فصل ۱ 1. Compilers Principles Techniqus & Tools ✓
By Sethi and Ulhum (2007) EBook
فصل ۱و ۲ را بخوانید

فصل ۱و۲ 2.* Crafting a Compiler تمرینات
By Fischer (2007)

3. The Theory and Pratice of Compiler writing By Thrembley and Soreson (1985)

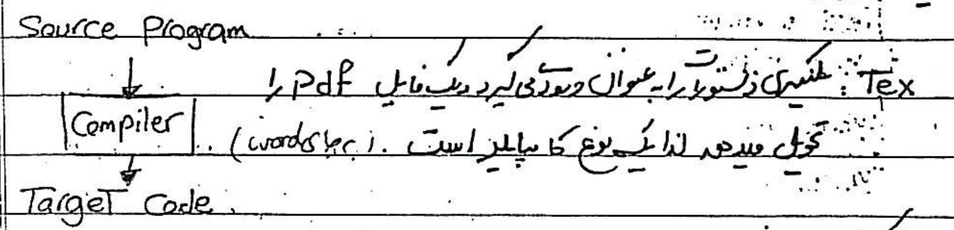
نظریه : انتزاعی : تئوری، ریاضی ← تمرین زیاد

کامپایلر : دانش : کاربردی ← حتمی مسئله تئوری را بازیابی را حل کنید
چون برای تست مراحل را بنویسید

کار تحقیقی

مراحل

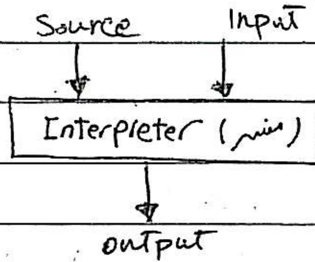
تعریف کامپایلر: یک Source Program را به یک Target Code تبدیل می‌کند.



Tex : تکنیکی دستورات را به عنوان ورودی می‌گیرد و یک فایل pdf را تحویل میدهد لذا یک نوع کامپایلر است. (بجای words)

تفاوت کامپایلر و مفسر:

کامپایلر برنامه را یک جا اجرا می‌کند اما مفسر خط به خط برنامه را اجرا می‌کند.



در کامپایلر input نداریم
و Target مستقل (از ورودی)
بدست می‌آید

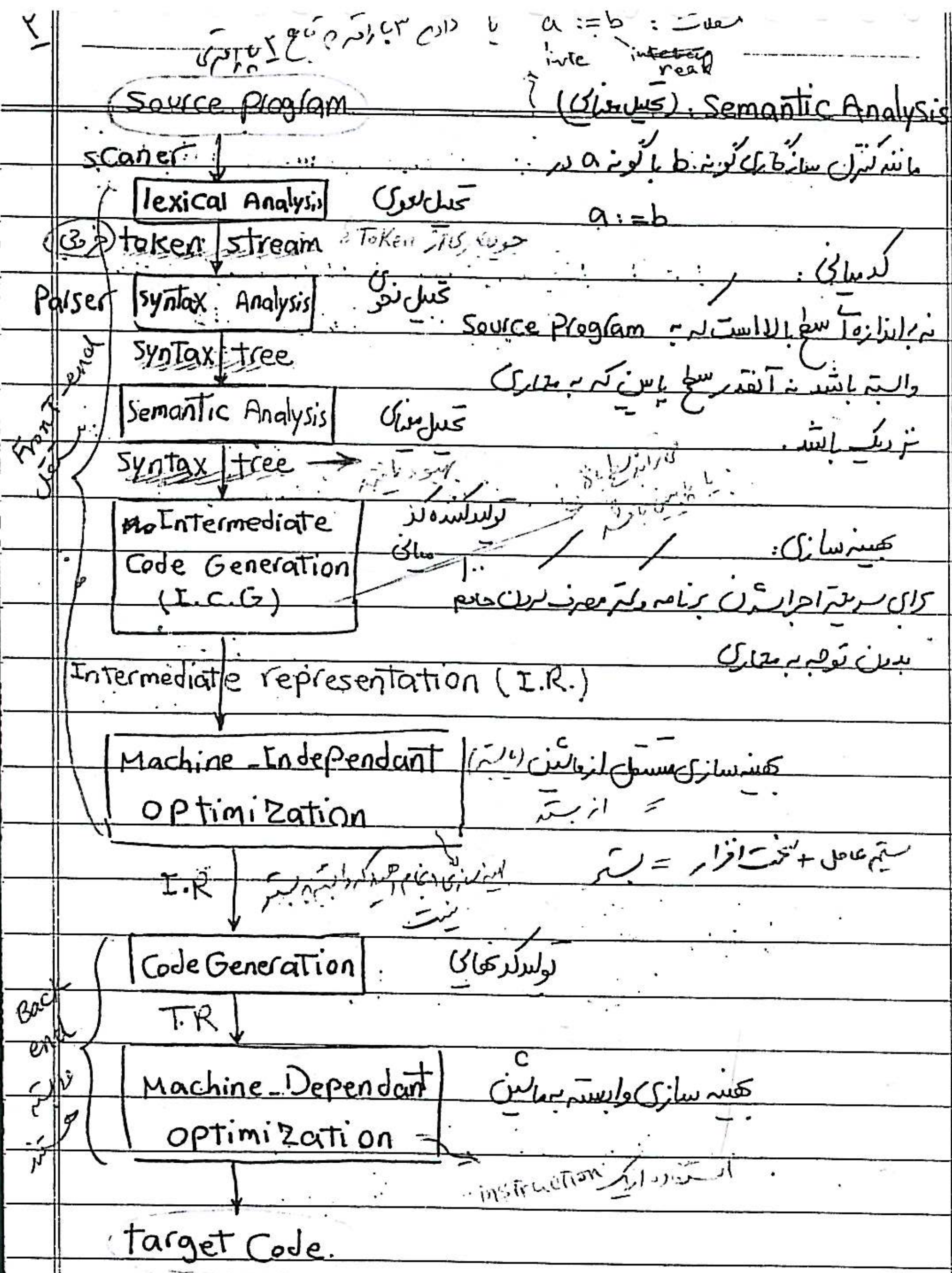
مراحل کامپایل:

lexical Analysis : (تحلیل لغوی)

کارش تفکیک لغات برنامه است مانند عملگرها، پرانتزها، سمی کالون و if
(لغت هر چیزی که به تنهایی نقشی در برنامه ما دارد)

Syntax Analysis : (تحلیل نحوی)

ترتیب کنار هم قرار گرفتن لغات را مورد بررسی قرار میدهد نظم کنار هم قرار گرفتن لغات توسط گرامر است. (آیا لغات در ترتیب صحیح کنار هم قرار گرفته اند)
چه کسی مشخص می‌کند ترتیب قرار گرفتن ؟



از این مراحل 5 مرحله اول مستقل از ماشین هستند که به آنها Front_end گویند. و به 2 مرحله آخر Back_end گویند و البته به ماشین هستند.

در بین این 7 مرحله 2 کامپوننت داریم که با تمام مراحل کامپایلر در ارتباط هستند

1\ Symbol Table جدول علائم

✓ برای ذخیره اطلاعات مربوط به Identifier ها استفاده می‌شود.

2\ Error_Handler خطایاب

Token : به جای اصطلاح لغت از Token به معنی نشانه استفاده می‌کنیم.

انواع نشانه‌ها (Token ها):

- Keywords کلمات کلیدی
- Identifiers شناسه‌ها
- Operators عملگرها
- Delimiters جداکننده‌ها
- Literals ثابت‌ها (Constants)

مثال) مراحل مختلف کامپایلر

$a := b + c * 60$

۱) اولین مرحله کامپایلر Tokenize کردن رشته‌های ورودی است: (attribute)

<id,1> <op, :=> <id,2> <op,+> <id,3> <op,*> <num,#60>

جدول سمبل؟ (علائم)

	Name	type	Dimension (ابعاد آرایه)	Var (func. / array)		Address
1	a	float	1	var		100
2	b	float	1	"		101
3	c	float	1	"		102
	⋮					

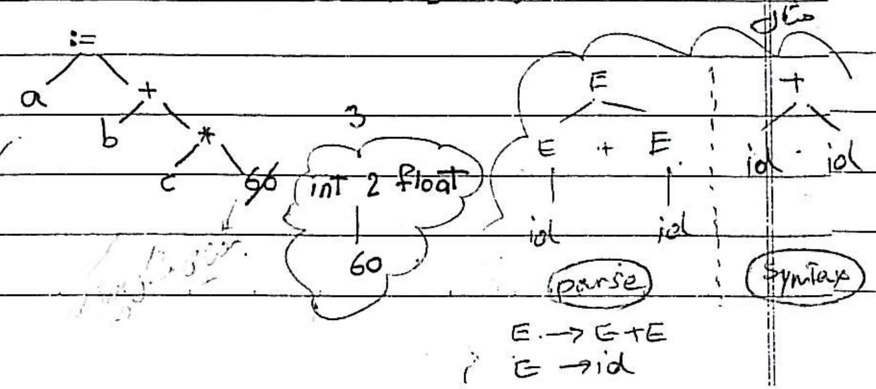
symbol Table

۲) Pars کردن رشته‌ها با استفاده از گرامر. اگر رشته‌ای با گرامر پارس نشد لذا syntax error داریم. جواب: Yes / No ⇐ با گرامر هم‌خوانی دارد یا نه؟

Parse Tree: ساختن یک عبارت با گرامر (شامل تمام مراحل میانی و غیر ترمینال)

حاصل: syntax Tree: نودهای مربوط به غیر ترمینال‌ها حذف شده‌اند و ساده شده‌اند.

syntax Tree: ساده شده درخت Pars است.



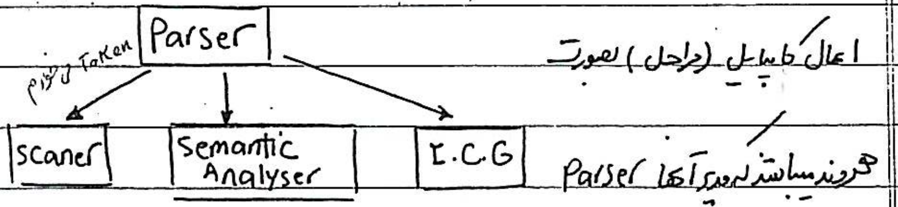
3. آیا تعداد پارامترهای فرمال (اظهار شده در تعریف تابع) و پارامترهای Actual (اظهار شده در زمان اجرا) با هم برابرند یا خیر.

در syntax tree چون 60 از نوع int است و c از نوع float است Semantic A.

60 را از syntax tree برداشته و یک int 2 float به جای آن می‌گذارد. در حقیقت خروجی Semantic یک syntax tree عبور داده شده است. Semantic Analyser کار

مواردی که مربوط به مشکل معنایی است را می‌تواند تشخیص بدهد.

اعمال کامپایلر (مراحل) به صورت



هم‌روند می‌باشند که مدیر آنها Parser است.

موازی بودن مراحل کار:

در یک گام Parser را فعال کرده و آن به Scaner می‌گوید که Token بعدی را بدهد و خود

Parser درخت Parse را گسترش انجام می‌دهد و بعد تحلیل معنایی را انجام داده و بعد کد تولید

کرده که توسط I.C.G تولید می‌شود و در آخر نیز دوباره درخواست Token می‌دهد.

Sematic: اگر به یک تابع دو پارامتری یک تابع ۳ پارامتری بفرستید ...

در همه جا نیاز نیست که ۷ مرحله انجام شود. (3 و 4 و 5 الزامی نیست)

در برخی کامپایلرها وجود Semantic Analyser الزامی نیست.

در بعضی ها وجود ندارد مانند مثلاً زبانهایی بدون آرایه ... optimizer را نیز می‌توان حذف نمود. تولید کننده کد میانی را نیز می‌توان حذف نمود.

کامپایلر MINIMUM:

اگر کامپایلر شامل بخش های scaner ، Parser و تولید کد نهایی باشد.

معمولاً برای scaner زبانهای سطح پایین، برای Parser زبان های مستقل از متن و برای کد میانی حساس به متن کافی است.

نکته: ممکن است یک زبان برنامه نویسی آنقدر ساده باشد که به قدرت زبان های حساس به متن نیازی نداشته باشد در اینجا نیاز Semantic Analyser نداریم.

تولید کد میانی: از قالب دستورالعمل های سه آدرسه استفاده می شود.

(OP, A1, A2, A3)

Portability، optimize ← کاربرد کد میانی

تولید کد میانی

مثال/ (+, 100, 101, 500)   500 := 100 + 101

(:=, 100, 500)   500 := 100   خانه 100 را در خانه 500 بریز

(:=, #100, 500) ← Immediate

از کد میانی استفاده می شود به ۲ دلیل:

1. راحت تر بودن Optimization به دلیل ساده بودن دستورها

2. Portable شدن برنامه (چون هنوز به معماری وابسته است.)

Back-end به ۲ عامل وابسته است: به معماری ماشین و سیستم عامل.

چگونگی طراحی کد میانی:

کد میانی ممکن است بسیار سطح بالا تعریف شود (Portable) یا در پایین ترین سطح (optimization) نزدیک معماری باشد؛ یک trade off است. بهتر است سطح بالا نوشته شود تا قابل حمل باشد.

مثال کد میانی (۶)

(int 2 float, #60, 500)

(*, 102, 500, 501)

(+, 101, 501, 502) ← سطح پایین

(:=, 502, 100,)

(2) اگر کد را به Optimizer بدهیم

(*, 102, #60.0, 500)

(+, 101, 500, 100)

از نظر حجم و سرعت کد بهینه شده است.

pass = خواندن یک بار برنامه

3) تولید کدهای

```
MOV    #60.0 , R1
LD     102 , R2 (c)
MULT   R2 , R1      c*60 -> R1
LD     101 , R2     (101) -> R2   b
ADD    R2 , R1      R2+R1 -> R1   b c*60
STORE  R1 , 100     R1 -> 100 (a)
```

Peephole optimization:

2 مورد است
- تا جایی که میتوان از جمله دوا استفاده کنید. به جای حافظه
- در مصرف رجیسترها صرفه جویی کنند

مثلاً در اینکه ما استفاده کردیم مثلاً ۲ رجیستر

برای بهبود کارایی که حاصل آن زبان ماشین می باشد.

$a*b + a*c$

a را در یک رجیستر و b را در رجیستر دیگر بگذاریم و حاصل ضرب را در رجیستر a بریزیم اشتباه است چون دوباره نا چار به load a خواهیم بود لذا بهتر حاصل را در رجیستر b ریخته می شود تا در سرعت کمتر شود.

ارتباط symbol Table و Scanner:

اسم Identifier ها در Symbol Table توسط Scaner انجام می شود

ارتباط symbol Table با Parser

ارتباط symbol Table با Semantic:

type متغیرها را بررسی کند کنترل type ها هم Semantic انجام میدهد.

int a ;

a:= 2 ;

اگر خط int a نبود error می دهیم scaner a را می گیرد یک خط بعد آن اشکال سیستم دارد ولی خط syntax مشکل ندارد

بعد به Semantic A می رسد به int در type نوشته می شود و بعد خط بعدی که a:= 2 است می رود می بیند آیا type آنها یکی است یا نه اگر مشکل نبود ادامه داده می شود.

در عمل نیاز به ۲ pass نداریم اگر اولین بار متغیر را ببینیم (مثلاً در a*b) که اشتباه است متغیر اظهار نشده (مراجعه می کند به گونه a دی گفت گونه اظهار نشده)

نکته: لزومی ندارد اول کار syntax تمام شده باشد تا کار Semantic شروع شود

مانند بررسی type در for(i:=...) قبل از بسته شدن الولار

ارتباط I.C.G:

برای تولید کد میانی به آدرس متغیرها است که نیاز به مراجعه به Symbol Table

داریم البته بگیریم میتوان آدرس را ICG هم میتواند انجام بدهد.

آیا شکل قبلی تناقض بود؟

one pass : موازی بودن مراحل کار او 2 و 3

* در optimize از نقاط one pass نداریم چون باید چند بار بخوانیم تا بهینه شود.

متوقف نکنیم و پیغام خطا را به کاربر گزارش دهد.

Syntax Error:

مثلاً پرانتزها باز بسته شود تعداد باز و بسته شده پرانتزها با هم یکی نباشد یا

سعی کامپایلر جایی برای handle کردن خطای کمبود پرانتز بسته می‌توان

فرض کرد که اگر... و خطا را گزارش دهد خطاها باید حتماً گزارش داده شوند و

سعی شود کامپایلر متوقف نشود و خطاها handle شوند.

Semantic Analyser:

warning ها توسط Semantic Analyser گزارش داده می‌شوند مثلاً

اخطار تخصیص float به int یا مثلاً اگر متغیر تعریف نشده باشد یک type

صورت پیش فرض به آن تخصیص داده می‌شود و اگر a := 2.5 را دید خطای

type می‌گیرد روش دیگر برای انتظار اولین جایی که عدد مورد استفاده قرار

بگیرد گونه را تعیین کند ممکن است اشتباه تنها در اولین مرحله باشد و در مراحل

ارتباط Optimizer با Symbol Table:

مثلاً حذف متغیرهای بی‌استفاده از حافظه و تغییر آدرس؟

ارتباط Code Generator با Symbol Table:

بسته به اینکه سطح کد میانی پایین است یا بالا ممکن است نیاز به تعیین این

باشد که ضرب float در float لازم باشد یا int در int در این صورت باید

به ST مراجعه شود و گونه متغیرها را استخراج کند (ممکن است opcode های

سخت‌افزاری لازم برای این دو عمل متفاوت باشد.)

ارتباط Error Handler با مراحل کامپایلر

scaner

lexical error:

در زبان c $ را کاراکتر مجاز نمی‌دانیم اگر بیاید ابتدا مستقل کامپایلر یا

Identifier ها نباید با digit شروع شوند اینها error های

لغوی هستند Error Handle در مقابله با این مشکلات در error های

لغوی، خطاها را Handle می‌کند و سعی می‌کنیم جوری کامپایلر را

ممکن است در زبانی برای اظهار توابع آرایه از نحوه مقابل A(I,J) استفاده می شود آیا A قبلاً اظهار شده و نوع آن چیست؟ توسط Semantic تعیین شده

Pass. یکبار خواندن کد ورودی از ابتدا تا انتها.

کامپایلرها ممکن است یا One-pass یا Multi-pass باشند.

پاسخ در تولید کردن کامپایلرها One-pass هستیم اما در Optimization بیش از یک Pass نیاز داریم تا کد میانی را مرتباً جستجو کنند.

تفاوت Compiler با Interpreter ها:

کامپایلر یک بار برنامه را خوانده ترجمه می کند و بارها آن را اجرا می کند اما Interpreter بارها کد را اجرا می کند اما هر دفعه برنامه را می خواند.

زبانهای مفسری: Basic، Matlab، Pearl، Prolog، ...

بعدی مدام error میدهد (Cascade error)

روش بهتر این است که یک گونه رکورد برای این متغیر تعریف شده در نظر گرفت و برحسب استفاده نوع آن را تعیین کرد چون تنها یادآور خطای undeclared برنامه نویس دیگری را ملزم به اظهار نوع آن نیز می کند.

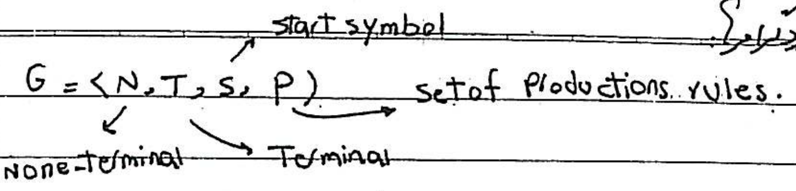
تولید کد میانی:

با handle کردن خطاها کد میانی تولید می شود اما اگر خطا وجود داشت کد commit نمی شود در این صورت کد را delete می کند. رابطه ICG با error handle پیام هایی در مورد کامپایل شدن حاصل برای ذخیره کد میانی یا خطاهای این چنینی داشت.

بحث در مورد رابطه Parser با Symbol Table:

پارسر برای گسترش درخت پارس از Symbol Table بهره می برد

یادآوری درمورد گرامر؟



$A, B, C \rightarrow$ غیرپایانه‌ها

$a, b, c \rightarrow$ پایانه‌ها

$X, Y, Z \rightarrow$ نماینده یک علامت (علامت چه پایانه چه غیرپایانه)

$x, y, z \geq 1 \rightarrow$ برای رشته‌ها (طول رشته ≥ 1)

$\alpha, \beta, \gamma \geq 0$ نماینده‌ای از رشته‌ای از علائم

ε رشته‌ای به طول صفر

(غیرپایانه‌ها)

$G = \langle N, T, S, P \rangle \equiv \langle \{E\}, \{id, +, *, (,)\}, E, P \rangle$

$P = \{$ 1) $E \rightarrow E+E$

2) $E \rightarrow E*E$

3) $E \rightarrow (E)$

4) $E \rightarrow id \}$

vacabulary

$V = N \cup T$

انواع گرامرها: $\alpha \rightarrow \beta$ (قانون گرامری)

1. گرامرهای نوع صفر (بدون محدودیت) (Unrestricted $\alpha \neq \Lambda$ (Null)

2. " " اول (حساس به متن) (context-sensitive $|\alpha| \leq |\beta|$

کاهش طول نداریم

مزایا: در زبان‌های مفسری تشخیص خطا خیلی ساده است زیرا جهت بتوان سعی و خطا را انجام داد.

معایب: بدون مفسر برنامه اجرا نمی‌شود در زبان‌ها مفسری ممکن است syntax error داشته باشد (هرگز به یک else نرسیم) اما در زبان‌های کامپایلری در زمان اجرا هم syntax گرفته شده

بهینه‌سازی در زبان مفسری معنایی ندارد.

در زبان‌های کامپایلری نیاز به داشتن source برنامه نداریم اما

در زبان‌های مفسری source برنامه نیاز است.

3. گرامرهای نوع دوم (مستقل از متن (Context free) $|\alpha|=1$, $\alpha \in N$

4. « « سوم (منظم (Regular)

افزایش محدودیت

خطی از راست: $|\beta|=1, \beta \in T$ ، $|\beta|=2, \beta=aA$

خطی از چپ: $|\beta|=1, \beta \in T$ ، $|\beta|=2, \beta=Aa$

BNF . (Backus-Naur form)

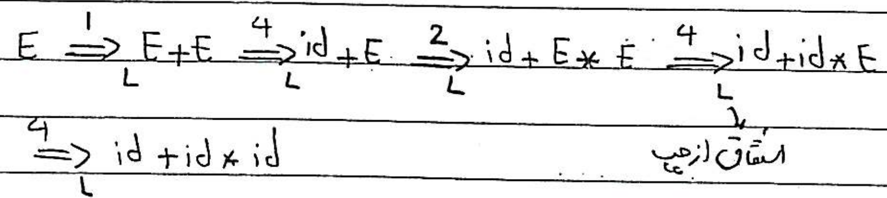
غیرپایانه‌ها را بین دو علامت < > قرار دهید بجای فلش از علامت =:: استفاده کرده.

<E> ::= <E> + <E>

برای رفع محدودیت تکرار N با قرار دادن هر رشته در < > به عنوان N

BNF فقط برای گرامرهای مستقل از متن به کار می‌رود

مثال / مراحل تولید رشته id+id*id



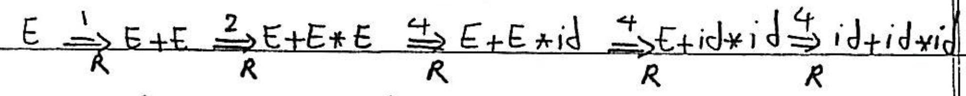
$E \underset{L}{\overset{1}{\Rightarrow}} E+E \underset{L}{\overset{4}{\Rightarrow}} id+E \underset{L}{\overset{2}{\Rightarrow}} id+E*E \underset{L}{\overset{4}{\Rightarrow}} id+id*E \underset{L}{\overset{4}{\Rightarrow}} id+id*id$

اشتقاق از چپ: LMD: left-Most Derivation

اگر در تمام مراحل اشتقاق چپ‌ترین غیرپایانه جایگزین شود.

اشتقاق از راست RMD (Right-Most Derivation)

برعکس اشتقاق از چپ می‌باشد

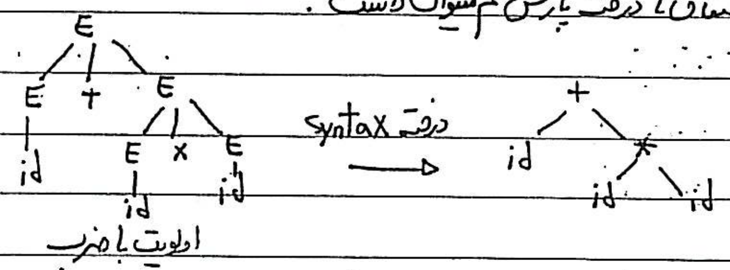


$E \underset{R}{\overset{1}{\Rightarrow}} E+E \underset{R}{\overset{2}{\Rightarrow}} E+E*E \underset{R}{\overset{4}{\Rightarrow}} E+E*id \underset{R}{\overset{4}{\Rightarrow}} E+id*id \underset{R}{\overset{4}{\Rightarrow}} id+id*id$

اگر برای اشتقاق از روش LMD یا RMD استفاده کنیم اصطلاحاً گفته

می‌شود که Canonical (قانون‌مند) است اشتقاق قانون‌مند است.

متناظر با اشتقاق یک درخت پارس هم می‌توان داشت.



درخت اشتقاق هیچ ربطی به LMD یا RMD ندارد

گرامر مبهم: Ambiguous grammer

به یک گرامر مستقل از متن مبهم گفته می‌شود اگر در زبان آن گرامر حداقل یک

از start با یک گام یا بیشتر به α برسید (فقط از ترمینال‌ها)

$L(G) = \{\alpha \mid \alpha \in T^* , S \overset{+}{\Rightarrow} \alpha\}$

SF(G) فرم جمله‌ای Sentential form

$SF(G) = \{\alpha \mid \alpha \in V^* , S \overset{*}{\Rightarrow} \alpha\}$ — $V^* = (N \cup T)^*$ — خود S هم یک فرم جمله‌ای است.

اگر فرم جمله‌ای در اشتقاق چپ به دست آمده باشد ← فرم جمله‌ای چپ است

〃 〃 〃 راست 〃 〃 〃 ← فرم جمله‌ای راست است

در غیر این صورت فقط به آن فرم جمله‌ای گوییم.

رشته‌های از پایانه و غیر پایانه را می‌شود از start گرامر به آن رسید SF(G)

L(G) همیشه زیرمجموعه‌ای از SF(G) است.

Reduced grammer:

باید ۳ خاصیت داشته باشد

(قابلیت پیش‌بینی در مراحل رسیدن به اشتقاق را نتوان گفت)

(گرامر را مبهم می‌کند به شرطی که غیرترمینال معادل هم نباشند)

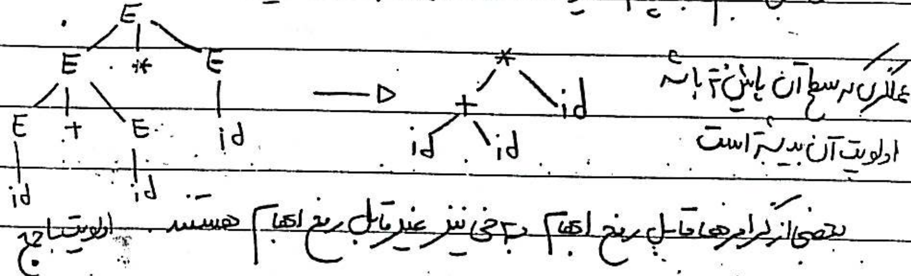
۱. قاعده‌ای به فرم A → A نباشد

۲. تمام غیرپایانه‌های آن active یا فعال باشند.

۱- اگر در یک گام یا بیشتر به ε یا λ نرسد

رشته وجود داشته باشد که برای آن چند تا LMD یا چند تا RMD یا چند تا درخت پارس داشته باشیم.

مثال: مثل مبهم بودن جمع و ضرب می‌توان 2 درخت پارس برای آن کشید



عملگری در سطح آن پایین‌تر باشد اولویت آن بیشتر است

بعضی از گرامرها قابل رفع ابهام و برخی نیز غیرقابل رفع ابهام هستند. اولویت با جمع

در مثال مثل جمع و ضرب اولویت بندی بین عملوندها نشود لذا ابهام ایجاد کرد برای رفع

ابهام مثال قبل:

E → E+T | T (کمتر چپ)

T → T*F | F

F → (E) | id (اولویت ضرب سمت چپ)

ضرب‌ها نسبت به جمع یک سطح پایین‌تر قرار گرفتند لذا اولویت پیدا می‌کند. اولویت جمع - ضرب است (کمتر)

ممکن است جمع‌ها هم سطح داشته باشیم a+b+c اول a+b انجام می‌شود بعد a+b+c

برای اینکه اولویت جمع سمت چپ رعایت شود (چپ گرد) لذا اولویت اول را به E دادیم E+T شد

ابتدا E آمد. اگر جای E و T عوض شود اولویت جمع سمت راست داده می‌شد

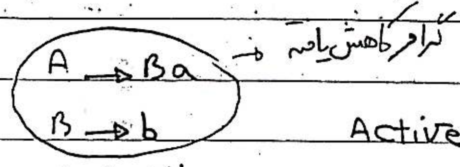
$A \overset{+}{\Rightarrow} \varepsilon \qquad A \overset{+}{\Rightarrow} x$

3. تمام غیرپایانه های آن reachable یا دسترس پذیر باشند. از start بتوان به آن رسید

$S \overset{*}{\Rightarrow} \alpha A \beta$

اگر خاصیت 2 و 3 را نداشته باشد آن گرامر useless می شود.

گرامر کاهش یافته ←



A → Ba

B → b

~~B → Cb~~

~~C → Cc~~

~~C → Ddc~~

~~D → ε~~

Active{B, A, D}

آنهایی که به رشته ای می رسند Active هستند B و A و D به رشته ای رسند اما C نمی رسد لذا C حذف شد

Reachable = {A, B}

D حذف شد چون هیچی به آن نمی رود.

ترتیب هم است ابتدا باید Active ها پیدا شوند

غیر Active حذف بعد Reachable پیدا شود Unreachable ها حذف شود

lexical Analyser

Scanner

- تمایز بین keyword و identifier

- پیدا کردن مثلا name در identifier ها در Symbol Table

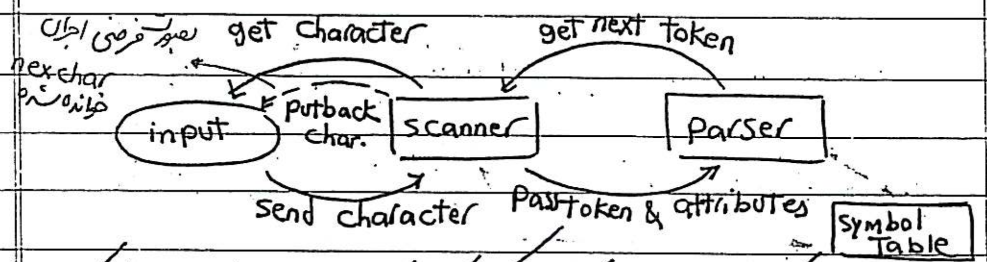
- حذف کردن comment ها (شناسایی و حذف آنها)

- Back Up گرفتن از source code.

بهترین قسمت compiler، optimize کردن opcode است

زمانگیرترین قسمت compiler مربوط به Symbol Table است.

بعد از optimize کردن بهترین سمت تبدیل مربوط به scanner است.



هر Token یک Pattern (الگو) دارد به این اگر در الگو هستیم باید ادامه به خواندن کنیم

و اگر از الگو بیرون آمدیم دیگر ادامه نمی دهیم. مثلا pattern عملگرها مقایسه ای < ... <= <

بعضی اوقات lexem همان Pattern است. مثل abc12

برای تشخیص token، scanner آن را با attribute آنها به Parser

اولین Token زمانی که =: برسیم شناسایی می‌شود $ad := x+y$

دومین Token همان =: است سومین Token و بیشتر که + را خواندیم ...

مثال / در زبان فرترن space اهمیت ندارد لذا جمله‌های زیر مجاز هستند.

DO5I = 1.5 → assignment

DO5I = 1,5 → for حلقه

با خواندن DO5I نمی‌توان Tokenize کرد باید به نقطه یا , برسیم لذا بعضی اوقات برای Tokenize کردن باید زمان بیشتری بگذارد.

در بعضی زبانها مانند PLI، می‌توان از keyword ها به عنوان یک نام مجاز استفاده کرد مانند

If then then Else := then ; else then := Else ;

در اینجا scanner قادر نیست تفاوت identifier را با keyword تشخیص دهد اما

Parser است که تشخیص می‌دهد کدام identifier است و کدام keyword با توجه به Parser

محل Tokenها تشخیص می‌دهد. (براساس گرامر تشخیص می‌دهد.)

این روش هم برای کامپایلر هم برای برنامه‌نویس سخت است.

می‌نویسند مثلاً سطر و ستون یا نوع خود عملگر.

اگر بخواهیم scanner را حذف کنیم Parser به جای دریافت Token

کاراکتر تحویل می‌گیرد و اگر مثلاً در مثال قبل E ← id باید Token را از scaner()

لذا گرامر به صورت جزئی‌تر براساس کاراکتر باید نوشته می‌شد کارهایی که با گرامر منظم

انجام می‌دادیم در Scaner باید با یک گرامر مستقل از متن انجام دهیم و این یک

هزینه اضافی و عدم Optimize بودن گرامر را در بر دارد. در تحلیل LL(1)

آن نیاز به مصرف کردن حافظه بیشتر خواهد بود چون تعداد سطرها بیشتر خواهد شد.

(سمت راست گرامرها) و ستونها (پایانه‌ها) در نتیجه LL(1) بزرگ خواهد شد.

در مورد زمان نمی‌توان قضاوت درستی داشت چون هم زبانهای منظم و هم

زبانهای مستقل از متن هر دو از $O(n)$ هستند.

در نتیجه اما کارایی پایین‌تر می‌آید و اصلاح کردن کامپایلر سخت‌تر خواهد بود.

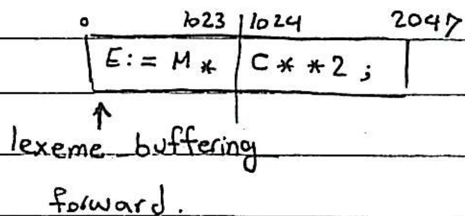
مبحث Double buffering :

واحد خواندن ما حداقل یک بلاک است که چون از حافظه خوانده می شود.

دوتا pointer معرفی می کنیم

Lexeme_begining که به ابتدای یک Token جدیدی است که شناسایی کند اشاره می کند.

forward ← یکی یکی جلو می رود تا از Pattern خارج شود.



دلیل استفاده اصلی از double buffering :

اگر یک buffer داشته باشیم ممکن است ابتدای token در بافر باشد که با reload کردن آن از بین می رود ولی در این سیستم بقیه Token در بافر بعدی است باقی می ماند

حداکثر سایز Token ، 1024 بایت است.

تعریف برای حرکت forward :

```
if forward = = 1023 then | load 2nd half of buf
                         | forward ++ ;

else if forward = = 2047 then | load 1st half of buf
                              | forward := 0
else
    forward ++ ;
```

برای جدا کردن رساندن این مقایسه ها از کاراکتری به نام کاراکتر نگهبان یا Sentinal استفاده کرده و وقتی به انتهای بافر رسیدیم کار مورد نظر را انجام دهد. چک می کنیم که به انتهای فایل رسیدیم یا نه.

```
forward ++ ;
if forward ↑ = 'eof' then
    if forward == 1023 ...
    elseif forward == 2047 ...
    else
        TERMINATE ;
```

عبارت منظم Regular Expression (REs)

ابزاری است برای توصیف یک زبان منظم یا نمایش یک زبان منظم

گرامر: ابزاری برای تولید رشته‌ای زبان.

1/ ϕ is a RE ; $L(\phi) = \{\}$

2/ λ is a RE ; $L(\lambda) = \{\lambda\}$

3/ $a \in \Sigma$ is a RE ; $L(a) = \{a\}$

4/ if r_1 and r_2 are two REs

$r_1 | r_2$ $\quad L(r_1|r_2) = L(r_1) \cup L(r_2)$

$r_1 . r_2$ $\quad L(r_1 . r_2) = L(r_1) . L(r_2)$

(r_1) $\quad L((r_1)) = L(r_1)$

r_1^* $\quad L(r_1^*) = (L(r_1))^*$

are REs, too

مثال)

$a^* = \{\epsilon, a, aa, \ldots\}$

$(a|b)^* = \{\epsilon, a, b, aa, \ldots\}$

$(a|ba|b) = \{a, ba, b\}$

$r.\epsilon = \epsilon.r = r$

$r.(s|t) = rs|rt$

$(r^*)^* = r^*$ $\quad a? = a|\epsilon$

$r.r^* = r^+$

$(a|b)^* = (a^*b^*)^* = (a?b?)^*$

رشته‌ای به طول 2 متشکل از a و b : $(a|b)^* (a|b)(a|b) (a|b)^*$

$L = \{a, \ldots, z\} = [a-z]$

$D = \{0, \ldots, 9\} = [0-9]$

مثال id : $L.(L|D)^*$

مثال) عبارت منظمی بنویسید که نماینده زبان زیر باشد؟

تمام رشته‌های متشکل از a و b که بیشتر از 2 a پشت سر هم نداشته باشد.

$(b|ab|aab)^* \underline{a?a?}$

$\epsilon|a|aa$

تمرین) عبارت منظمی بنویسید که نماینده زبان زیر باشد؟

تمام identifier های یک زبان مرسوم مانند C و پاسکال متشکل از letter

و digit و underscore با محدودیت زیر:

'_' underscore باهم نیاید

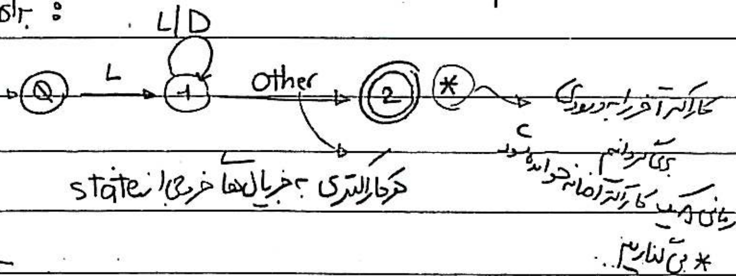
'__' دو underscore پشت سر هم نیاید

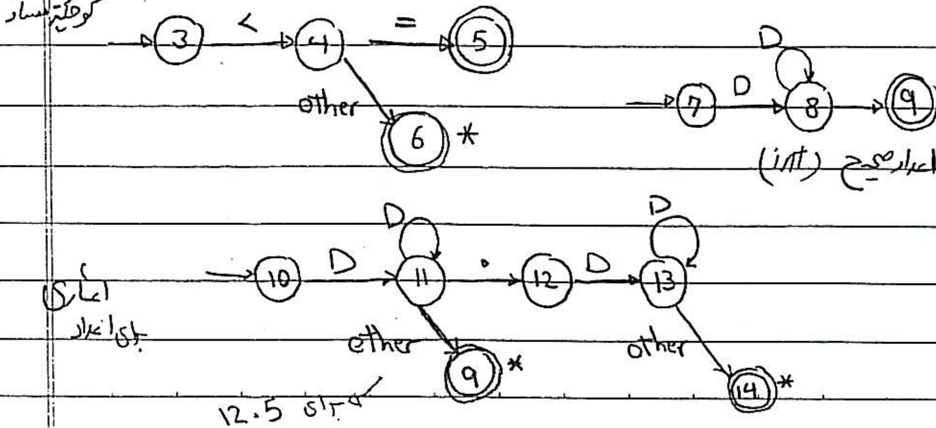
lex یک Scaner Generator است

معمولاً Scanner ها در حوزه زبان منظم می‌گنجند لذا Scanner را می‌توان با زبان منظم برای lex توصیف کرد.

lex : یک زبان منظم را به یک ماشین DFA یا NFA تبدیل می‌کند.

DFA اگر رشته را بخواند در O(n) پاسخ می‌دهد اما NFA چنین کاری را وابسته است چون تمام حالات بررسی باید شود. لذا مطلوب نیست از DFA اکثراً استفاده می‌کنیم





برای پیاده‌سازی DFA از switch case استفاده می‌کنیم

```
switch <ch> of
'L' : ____  →  مسیر ماشین
'D' : ____
' ' : ____
'<' : ____
 :
end
```

ساده‌ترین بخش کامپایلر Scaner است.

در استفاده از switch case ترتیب قرارگیری case ها مهم نیست

اگر از if else استفاده کنیم آنهایی که احتمال تکرار بیشتری دارند زودتر نوشته می‌شوند

اگر Scaner خطا تشخیص نداد Error handler داریم.

روش‌های برخورد با خطاهای لغوی :

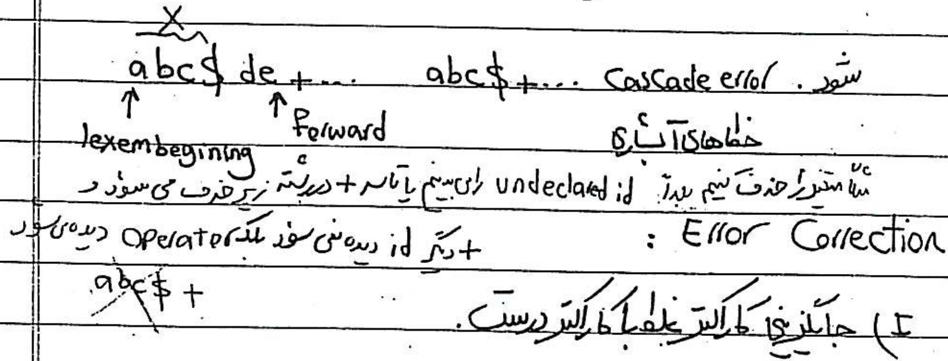
1) Error Recovery (پیش‌بینی پوشش بازیابی از خطا)
خود را از شر خطا خلاص کنیم

2) Error Correction (تصحیح خطا)
خطا را به صورت بهینه تصحیح کنیم

Pannic Mode

بخش زیادی از ورودی را ممکن است دور بریزیم (از ابتدای token

تا کل رخداد خطا را دور بریزیم) ممکن است موجب cascade errors



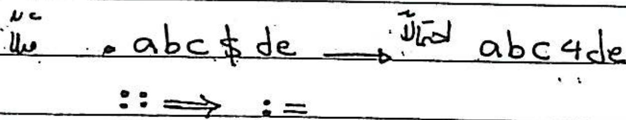
abc$ de + ...   abc$ + ...   cascade error شود.

lexembegining   Forward

خطاهای آبشاری

مثلاً متغیر را حذف کنیم پیدا undeclared id این سیستم یا تابع + دیگر زیر حذف می‌شود و + دیگر id دیده نمی‌شود بلکه operator دیده می‌شود

Error Correction :

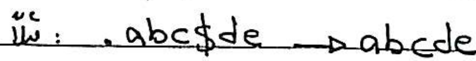


abc$ +

I) جایگزینی کاراکتر غلط با کاراکتر درست.

مثلاً abc$de → احتمالاً abc4de

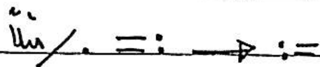
:: ⟹ :=



II) حذف کاراکتر نادرست.

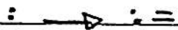
مثلاً: abc$de → abcde

III) جابجا کردن دو کاراکتر.



مثلاً =: → :=

IV) اضافه کردن کاراکتر گمشده



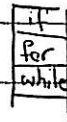
: → :=

Minimal distance :

برای هر تغییر یک distance تعریف کنیم مثلاً اگر کاراکتری را جابجا کنیم distance آن 2 است. آنگاه با کمترین فاصله خطا را تصحیح کنیم.

وظایف Scanner :

تمایز قائل شدن بین identifier ها و keyword ها. یک کاربرد این است که برای keyword ها ماشین در نظر بگیریم. با این روش با ورود هر ماشین keyword تشخیص داده می‌شود راه کار دیگر داشتن یک جدول از keyword است. تمام identifier ها را با این جدول چک کنیم. این جدول می‌تواند در قسمتی از حافظه باشد و یا قسمتی از Symbol Table را هم به این جدول اختصاص دهیم.



مثلاً دو تابع gettoken() و installid() ... gettoken. parser فراخوانی می‌کند. بعد install_id فراخوانی می‌شود. اگر id

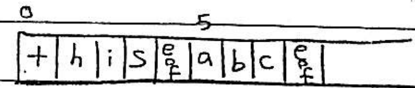
حداکثر طول identifier ها = 256

قبلاً نصب شده بود شماره‌ای برمی‌گرداند که آن را به Parser برمی‌گردانیم

اما اگر عددی از (1-32) برگرداند یعنی keyword است و شماره

id یا keyword مورد نظر آن است را باید در Parser متوجه می‌شود

کدام keyword است.



Name Symbol Table

0
5
⋮

keyword ها طول بسیار کمتری نسبت به شناسه‌ها دارند => ذخیره آنها

در st بسیار هزینه‌بر است (از لحاظ حافظه) => در بخشی از حافظه ثابت

کلیدی را به صورت فوق ذخیره می‌کنیم (و شناسه‌ها)

قدرت زبان‌های منظم برای scanner ها کافیست.

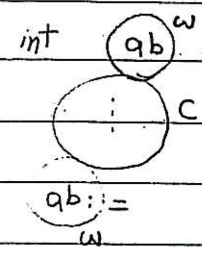
در Syntax Analyzer نمی‌توان از قاعده زبان‌های منظم استفاده کرد چون تعداد پرانتزها

باز و بسته باید یکی باشد که زبان $a^n b^n$ را نشان می‌دهد این منظم نیست. (((...))) a^n b^n

قدرت زبان‌های مستقل از متن برای Syntax Analyzer کافی است. اما برای

Semantic Analyzer کافی نیست. مثلاً در مورد تعریف شده بودن متغیرها در ALGOL

می‌خواهیم استفاده کنیم



باید دوباره به متن رجوع کرد ← wcw

جلسه سوم ←

تحلیل نحوی (Parsing (syntax Analyser

مثال $S \rightarrow cAd$

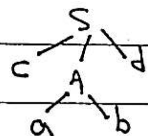
$A \rightarrow ab \mid a$

گرامر روبه‌رو دو رشته تولید می‌کند: cabd و cad

مثلاً cad را فرض می‌کنیم رشته گرفته شده از گرامر باشد.

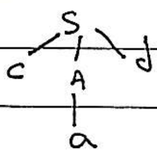
token گرفته شده از رشته look_Ahead (L.A.) ، current token

L.A = c



L.A بعدی a است. می‌توان a را match کرد.

۲) اما از b، حاصل نمی‌شود چون token بعد d است لذا نمی‌توان از درخت دیگر این خطای نحوی را برطرف نمود. کافیست:



→ Back Track داریم.

هدف: بیان روش‌های Parse (بیان خطاهای نحوی)

ما دنبال Parser هایی هستیم که تعداد مراحل آن قابل پیش‌بینی و محدود باشد

اینکه هر بار Back tracking داریم بد است. هرچه تعداد Back track (برگشت به عقب) کمتر باشد بهتر است. ما دنبال پارسرهایی هستیم که برگشت به عقب نداشته باشیم.

اگر گرامر مستقل از متن قطعی نباشد اصطلاحاً نمی‌توان پارسرهایی بدون برگشت به عقب ساخت، امکان‌پذیر نیست.

ما دنبال گرامرهای مستقل از متن غیرقطعی نیستیم چون اگر غیرقطعی باشد از این روش‌های پارس نمی‌توان استفاده کرد و برگشت به عقب نیاز می‌شود.

روش‌های Parse:

به دو دسته کلی تقسیم می‌شوند

1. Top-Down (بالا به پایین) → معادل LMD

از start گرامر شروع می‌کنیم و سعی می‌کنیم آن رشته را ایجاد کنیم

2. Bottom-up → معادل عکس RMD (پایین به بالا)

در رشته عمل انتقال انجام می‌دهیم تا به start گرامر برسیم

روش‌های TopDown دو روش اصلی را استفاده می‌کنیم.

1. Recursive Descent

با در آمدن k توکن به گره قضاوت می کنند
با خواندن ۱ توکن قضاوت می توانند بکنند

2. LL(1) ⊆ LL(k)

رشته ورودی را از چپ می خوانیم. رشته ورودی k توکن k توکن

می خوانیم هرچه k بزرگتر باشد گرامر قوی تر است و با آن پارسر

می توان حوزه ی بیشتری را تحت پوشش قرار داد.

درخت های پایین به بالا همه را زیر پوشش می دهد.

1. Operator Precendence (OP) تقدم عملگر

2. Simple Precedence (S.P) تقدم ساده

3. LR(1) ⊆ LR(K)

L ← رشته ورودی از چپ به راست خوانده می شود.

R ← از عکس رشته ورودی می خوانیم

K ← k توکن k توکن می خوانیم

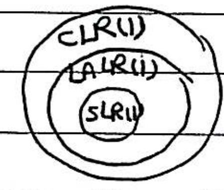
دارای موارد زیر است.

1. SLR(1) : Simple LR
2. LALR(1) : lookAhead LR
3. CLR(1) : Canonical LR

ممکن است گرامری داشته باشیم که SLR(1) باشد حتما LALR(1) و CLR(1) هم

هست.

از لحاظ قدرت



Top-Down:

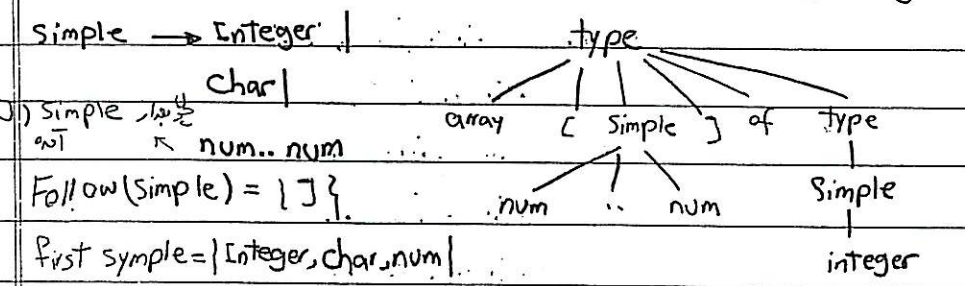
Recursive Descent

ابتدا با یک مثال گرامری شروع می کنیم.

Type → array [Simple] of type |
↑ id |
Simple

Simple → Integer |
Char |
num.. num (آن) Simple (جداکننده) ↖

array [num..num] of integer



Follow(Simple) = {] }

first symple = { Integer, Char, num }

first (array) = { array, first (symple) } [ ]

match() :

match توکن می گیرد چک می کند اگر آن توکن همان توکن .LA است یا نه

اگر نبود error می دهد

به ازاء هر غیر پایانه یک زیر برنامه باید بنویسیم.

```
Proc match(t)
begin
   if t == L.A then L.A := Scanner ;
   else
       error ;
end.
```

پروسیجر مربوط به گرامر:

```
Proc Type
begin
```

(برای هر غیرپایانه یک فراخوانی دارد.)

```
if L.A == 'array' then
        begin  match('array');
               match('[');
               Call Simple ;
               match(']');
               match('of');
               Call Type ;
               end.
else if L.A == '↑' then
               begin
               match('↑');
               match(id);
               end
else if L.A. ∈ {integer, char, num} then
                                   Call Simple ;
else   error ;
end Type ;
```

دستور match() کردن برای پایانه‌ها است.

کارایی این روش به علت صدا کردن تابع پایین است و Recursive است [illegible] نیست

گرامر بازگشتی دارد لذا Recursive function call نیاز است.

وقتی در مورد یک گرامر سؤال شده هم گرامر است قرار نیست تبدیل کنیم.

نوع گرامر مورد استفاده ما مستقل از متن است اگر قوانین ناشی از یک none Terminal

ابتدای مشترک داشته باشند دیگر نمی‌توان از این روش رفت

```
STL → begin STS end
 STS → ε | other
```

```
Proc STL
  begin   if L.A. == 'begin' then
                  begin
                  match('begin');
                  Call STS ;
```

```
        match('end');
        end
else    error;
end
```

$Follow(STS) = \{end\}$

```
proc STS
begin
if L.A. == 'other' then
        match('other');
else if L.A == 'end' then
return;
else error;
end;
```

ε را match نمی‌خواهد چون پایانه نیست

توابع:

$first(\alpha) = \{$ پایانه‌ها و ε $\} = \{a \mid \alpha \overset{*}{\Rightarrow} a\beta\}$

پایانه‌هایی که از وقتی که از یک رشته شروع می‌کنیم می‌توانند در ابتدای رشته واقع شوند.

نکته:

اگر قاعده‌ای به شکل $A \to \alpha \mid \beta$ داشته باشیم

شرط اول

① $First(\alpha) \cap First(\beta) = \phi$

اگر شرط فوق نقض شود از روش Recursive decent نمی‌توان استفاده کرد.

$Follow(A) = \{$ پایانه‌ها $\} = \{b \mid S \overset{*}{\Rightarrow} \alpha A b \beta\}$

پایانه‌هایی که به دنبال غیرپایانه‌ها می‌توانند ظاهر شوند اما فقط در فرم‌های جملهای

هرگز ε جزء Follow قرار نمی‌گیرد هرگز Follow تهی نمی‌شود.

Recursive Descent

② شرط دوم RD

if $\alpha \overset{*}{\Rightarrow} \varepsilon$ then $first(\beta) \cap Follow(A) = \phi$

اگر گرامری هر دو شرط ① و ② را دارا بود اصطلاحاً به آن گرامر، گرامر LL(1) گوییم.

```
Proc A
begin if L.A ∈ First(α) then use A → α
   else if LA ∈ First(β) then use A → β
 * else if L.A ∈ Follow(A) then use A → ε
else
   error;
end
```

* (بشرطی که A بتواند به ε برود اگر نتواند به ε برود این دستور را نمی‌نویسیم (این یکی خط که را نمی‌نویسیم))

همه زبانها معمولاً LL(1) هستند چون همه زبانها معمولاً DNA دارند. اما هر گرامری که LL(1) باشد ممکن است مبهم نباشد.

$$\overbrace{}^{\alpha}\quad\overbrace{}^{\beta}$$

1,2) $E \to E+T \mid T$
3,4) $T \to T*F \mid F$
5,6) $F \to (E) \mid id$

```
proc E
begin
    if L.A ∈ First(E+T) then
        Call E;
        match('+');
        Call T;
    end
    elseif L.A ∈ First(T) then
        Call T;
    else
        error;
end E.
```

$E+T \Rightarrow T+T \Rightarrow F*T \Rightarrow (E)+T$
$\Rightarrow id+T$

گرامر فوق بازگشتی چپ دارد. E، گرامری بود که به جز E شروع می‌شود یا T.

گرامری بود که با جز T شروع می‌شود در نتیجه چپ‌گردی دارد. چون

$First(T) \cap First(E) = \{(, id\} \neq \phi$

گرامری که چپ‌گرد باشد LL(1) نیست.

راه‌حل: حذف چپ‌گردی. به ازای EZT گرامرها مستقل از متن می‌توان اینکار کرد. به ازای هر گرامر مستقل از متن

۱) برای حذف چپ‌گردی: ا) چپ‌گردهای غیرمستقیم را از بین می‌بریم. می‌توان یک گرامر به فرم نرمال گریباخ داشته باشیم که این فرم دوار یک پایانه شروع می‌شود.

$A \to Bx$
$B \to Ay$ $\quad \to \quad A \to Ayx$

تبدیل چپ‌گرد با واسطه به بی‌واسطه

2) سپس گرامرهای چپ‌گرد را از غیرچپ‌گرد جدا می‌کنیم.

$A \to A\alpha_1 \mid \dots \mid A\alpha_n$ چپ‌گرد

$A \to \beta_1 \mid \dots \mid \beta_m$ غیرچپ‌گرد

$\hookrightarrow (\beta_1 \mid \dots \mid \beta_m)(\alpha_1 \mid \dots \mid \alpha_n)^*$

$A \to \beta_1 A' \mid \dots \mid \beta_m A'$
$A' \to \epsilon \mid \alpha_1 A' \mid \dots \mid \alpha_n A'$ چپ‌گردی از بین رفت

نکته:

برای LL(1) کردن گرامری که LL(1) نیست همیشه جواب نمی‌دهد.

I) حذف چپ‌گردی

II) فاکتورگیری از چپ

مثال) گرامر قبل را که دارای چپ‌گردی بود با حذف چپ‌گردی انجام دهید؟

1) $E \to TE'$
2,3) $E' \to \epsilon \mid +TE'$
4) $T \to FT'$
5,6) $T' \to \epsilon \mid *FT'$
7,8) $F \to (E) \mid id$

نحوه محاسبه First(X):

1. اگر x یک پایانه باشد آنگاه $First(x) = \{x\}$

(x نماینده علامت گرامر است)

2. اگر قاعده‌ای بفرم $x \to \varepsilon$ داشته باشیم آنگاه ε را به $First(x)$ اضافه می‌کنیم.

3. اگر قاعده‌ای بفرم $X \to y_1 y_2 \ldots y_k$ داشته باشیم آنگاه:
$First(y_1) - \{\varepsilon\}$ را به $First(x)$ اضافه می‌کنیم حال اگر $y_1 \Rightarrow^* \varepsilon$ آنگاه $First(y_2) - \{\varepsilon\}$ را به $First(x)$ اضافه می‌کنیم.
و همین روند را ادامه می‌دهیم تا جایی که $y_{k-1} \Rightarrow^* \varepsilon$ آنگاه $First(y_k)$ را به $First(X)$ اضافه می‌کنیم.

$$\begin{cases} A \to Bc \\ B \to \varepsilon \end{cases}$$

$\underbrace{First(B)}_{\{\varepsilon\}} - \{\varepsilon\} = \phi$ هیچی

$\Rightarrow First(A) = \{c\}$

↳ $First(c) = c$

$A \to Bb \mid e$ $\quad First(A) = \{e, b\}$

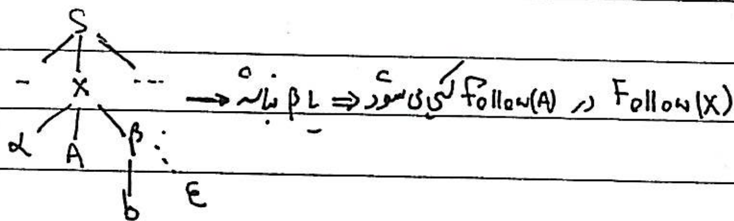
$B \to Ac \mid b$ $\quad First(B) = \{e, b\}$

نحوه محاسبه Follow(A):

1. اگر A علامت شروع گرامر باشد آنگاه \$ را به $Follow(A)$ اضافه می‌کنیم.

2. اگر قاعده‌ای بفرم $x \to \alpha A \beta$ داشته باشیم آنگاه $First(\beta) - \{\varepsilon\}$ را به $Follow(A)$ اضافه می‌کنیم (تاکید: ε هرگز جزء $Follow(A)$ نیست)

3. اگر قاعده‌ای بفرم $x \to \alpha A$ داشته باشیم یا قاعده‌ای بفرم $X \to \alpha A \beta$ داشته باشیم و $\beta \Rightarrow^* \varepsilon$ آنگاه $Follow(x)$ را به $Follow(A)$ اضافه می‌کنیم.



مثال (I):

1) $E \to TE'$ $\quad First(1) = \{(, id\}$

2,3) $E' \to \varepsilon \mid +TE'$ $\quad First(2) = \{\varepsilon\} \cap First(3) = \{+\} = \phi$

4) $T \to FT'$ $\quad First(4) = \{(, id\}$

5,6) $T' \to \varepsilon \mid *FT'$ $\quad First(5) = \{\varepsilon\} \cap First(6) = \{*\} = \phi$

7,8) $F \to (E) \mid id$ $\quad First(7) = \{(\} \cap First(8) = \{id\} = \phi$

$Follow(E) = \{\$,)\}$ — آخرالسلام Follow سمت راست قواعد نگاه می‌کنیم.

$Follow(E') = \{\$,)\}$

قواعدی که E سمت چپ آنهاست 1, 3 است

$Follow(T) = \underbrace{First(E')}_{\{+\}} + \underbrace{Follow(T)}_{Follow\ E\ +\ Follow\ E'}$

$= \{+, \$,)\}$

$Follow(T') = Follow(T)$

$Follow(F) = \underbrace{First(T')}_{*} + \{\$,), +\} = \{*, +, \$,)\}$

در 6 ظاهر شده از قانون دوم تبعیت می‌کند

یعنی T' می‌تواند در قاعده 6 به ε برود پس Follow(T') در Follow(F) کپی می‌شود

مثلاً قانون دوم بررسی LL(1) را امتحان کنیم می‌بینیم که

در 2,3 $First(P) = First(+TE') = \{+\}$
$Follow(E') = \{\$,)\}$ $\cap$ $\} = \phi$

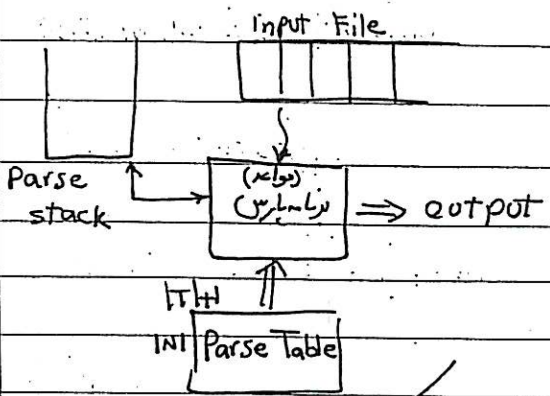
5,6 $First(*FT') = \{*\}$
$Follow(T') = \{*, \$,)\}$ $\cap \} \Rightarrow \phi$

درسته LL(1) است چون در جاهایی که $\alpha \to \varepsilon$ رفته قانون دوم برقرار است

$LL(1) \subseteq LL(K)$

LL(1):

ساختار یک پارسر LL(1):



در جدول Parse سطرها؟ اسامی None_Terminal ها ستون‌ها T + ترمینال‌ها هستند.

باید گرامر را LL(K) با سنتا بتوان از این گرامر استفاده کرد.

Parse Table:

	*	+	(	)	id	$
E			1	S	1	S
E'		3		2		2
T		S	4	S	4	S
T'	6	5		5		5
F	S	S	7	S	8	S

S برای sync در خانه‌هایی قرار می‌گیرد

خانه‌های Parse Table یا خالی هستند یا بوسیله شماره گرامر پر می‌شوند.

نحوه‌ی اجرا عمل Parse به روش (LL(1)):

مرحله آغازی:

در ابتدای کار یک علامت $ به انتهای رشته ورودی اضافه کرده و $S را بصورت برعکس وارد stack می‌کنیم.

در هر مرحله از Parse با نگاه کردن به علامت بالای stack و توکن جاری (L.A.)
(PS (Top) = X)

به صورت زیر عمل می‌کنیم

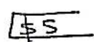
1. اگر X = L.A = $ باشد خاتمه‌ی عمل Parse اعلام می‌شود
2. اگر X = L.A ≠ $ (X یعنی پایانه بوده) آنگاه اسکنر صدا زده می‌شود که توکن بعدی را بخواند و X از بالای stack حذف می‌شود. (اگر X یک پایانه باشد با L.A تطبیق نکند آنگاه یک خطای نحوی رخ داده است.) قوانین را برعکس داخل stack می‌گذاریم
3. اگر علامت بالای stack یک غیرپایانه باشد به همراه توکن جاری ما را به خانه‌ی PT(X, L.A) ارجاع دهد به صورت زیر عمل می‌کنیم.

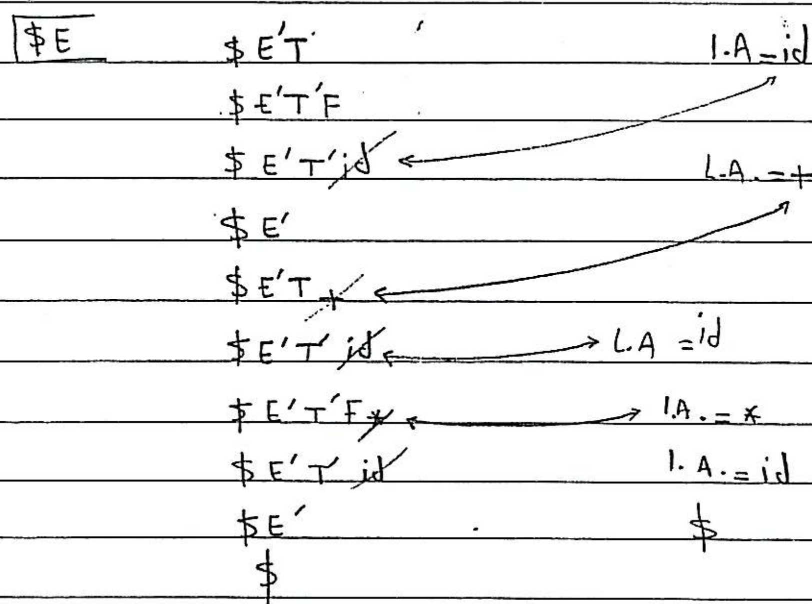
الف) اگر PT(X, L.A.) = X → α باشد آنگاه X را از بالای stack حذف کرده و α را به صورت برعکس وارد stack می‌کنیم. (اول سمت چپ‌ترین غیرپایانه گسترش می‌یابد) LMA

ب) اگر PT(X, L.A.) خالی باشد آنگاه یک خطای نحوی رخ داده است و باید خطا پرداز فراخوانده شود.

مثال) رشته‌ی مقابل را پارس کنید؟ (با توجه به درس مثال I) id + id * id

ابتدا گذاشتن $ در انتهای رشته id + id * id $





نحوه پرکردن جدول پارس LL(1):

به ازاء هر $A \rightarrow \alpha$ به صورت زیر عمل می کنیم

1. مقابل غیر پایانه A و زیر عناصر First(α) شماره ی قاعده ی $A \rightarrow \alpha$ را قرار میدهیم.

2. اگر α در صفر مرحله یا بیشتر به ε برود آنگاه مقابل غیر پایانه A و زیر عناصر Follow(A) شماره ی قاعده ی $A \rightarrow \varepsilon$ را قرار میدهیم

تمرین) آیا با اضافه کردن قاعده ی $E \rightarrow \varepsilon$ گرامر (1) LL(1) می ماند؟ و اگر می ماند جدول LL(1) آن را پر کنید؟ (بله)

مثال)

1,2) $A \rightarrow Bd \mid Ce$ $first(A \rightarrow Bd)=\{b,d\}$

3,4) $B \rightarrow bB \mid \varepsilon$

5,6) $C \rightarrow cC \mid \varepsilon$

$first(C \rightarrow \varepsilon) \rightarrow Follow(C) = e$

اگر LL(1) نبود در خانه ها تداخل داشتیم.

	b	c	d	e	$
A	1	2	1	2	
B	3		4		
C		5		6	

Follow(c)= e در خانه 6 را می گذاریم

چون در هیچ خانه ای تداخل نداریم گرامر LL(1) است.

معنای اینکه ستون $ خالی است یعنی اینکه هیچ انتظاری نداریم که none-terminal دیده شود یعنی انتظار نداریم بالای stack یک none-Terminal باشد تا ما $ ببینیم.

نحوه ی برخورد با خطا در روش پارس LL(1):

1. Panic Mode
2. phase level
3. Error productions
4. Global Correction

Panic Mode:

بخشی از ورودی را ممکن است حذف کند. (1. با پایانه با LA با stack جور نداشته باشد یا 2) با LA و NT (None Terminal) در stack خانه خالی برسیم شرایط error است.)

Synchronizing (هماهنگ کننده):

به ازاء غیر پایانه تعیین می شوند. با یک غیر پایانه بخشی از رشته ورودی را تولید می کنیم اگر حین کار تولید رشته غیر پایانه به خطا رسیدیم کل کار آن را نادیده بگیریم سپس تا جایی که تصور است و جور موفق را برای follow آن NT پیدا کنیم را دور بریزیم