نحوه اعمال روش Panic Mode

1. اگر علامت بالای استک یک پایانه باشد که با توکن جاری تطابق ندارد پایانه بالای stack را حذف می‌کنیم.

2. اگر علامت بالای stack یک غیرپایانه باشد همراه توکن جاری ما را به یک خانه خالی ارجاع دهد آنگاه scanner را صدا می‌زنیم تا توکن بعدی را بخواند (انگار آن را نادیده گرفته)

3. اگر علامت بالا stack یک غیرپایانه باشد که همراه توکن جاری ما را به یک خانه حاوی S ارجاع دهد آنگاه غیرپایانه را از stack خارج می‌کنیم (مشروط بر اینکه تنها غیرپایانه مانده در stack نباشد.)

چون اگر این را که تنها غیرپایانه stack است دور بریزیم stack از کار می‌افتد.

مثال >

id+id	$E'T$
	$E'T'F$
	$E'T'id$
	E'
	E'(T) T را می‌بیند
1.A. =+	E'

T, + طبق جدول S دارد T را دور می‌اندازیم 1.A.=+

<= انگار id را ندیده است => در اصل

id+id+id را پارس کردیم.

	$E'T$
id * id	$E'T'F$
	$E'T'id$
	E'
	$E'T$ * حذف شد

در اصل * را حذف کردیم id+id

اگر $A \rightarrow \alpha$ و $Follow(A) \neq \emptyset$ $First(\alpha) \cap$ حتی اگر $\alpha \rightarrow \epsilon$ هم برود

باز مشکل پیش نمی‌آید یعنی مقابل غیرپایانه A تداخل نداریم. اگر α از غیرپایانه

مانند B تشکیل شده باشد

$S \rightarrow Ab$

$A \rightarrow \alpha$ ← B

$B \rightarrow b|\epsilon$

گرامر مشکل دارد ⟸

این قانون با => LL(1) نیست $\alpha \overset{*}{\Rightarrow} \epsilon$

ولی حالت اکی داریم.

جلسه چهارم

Phrase level

میتوان داخل هر خانه خالی یک pointer (در جدول پارس (LL(1)) می گذاریم.

روتین خطای ۱ قرار است هندل کنیم. با از هر پیغام خطا که اول رشته باید ارائه دهیم.

مثلاً در جدول پارس خانه ای پایتر که به خانه خالی است شاید اضافی باشد. (دنبال همین استتاح داده باشیم)

در حالت E ← ε اصلاً نمیتوان وارد شد (هرگز به این انتقال برخورد نمی کنیم.)

خیلی منطقی تر از Panic Mode رفتار می کند

Error Production

در مورد خطاهای مرسوم (خطاهای در کلاس بالا) رخ میدهد. مثلاً جا انداختن سمی کالون در برنامه در مورد همه خطاها بکار نمی رود

مثلاً اگر ایده ی درست ما E ← TE' است و خطا رخ دهد E ← TE'

(جا انداختن سمی کالون) باشد هر دو قانون را در گرامر قرار دهیم و با شماره های مخصوص شماره گذاری می کنیم. اگر LL(1) شود آن را درست می کنیم حال جدول پارس آن را می کشیم.

اما عمل پارس هم متوقف نمی شود

با پارس به دلیل خطاهای مرسوم موجود Commite نمی شود و ما متوجه خطا ها می شویم

با شماره گذاری که داریم میتوان متوجه شد که کدام خطا بوده است.

1) E ← TE' و 1) E ← TE'X ⇒
1) E ← TE' 2-2) X ← ;|ε

چون این خطا زیاد رخ میدهد آن را Interupt به ما برنیابی کند ⇐ لذا از این روش زیاد استفاده می شود.

Global Correction

وقتی 2 choise داریم برای پیغام خطا یک بار یک خطا داده ببینیم کدام پیغام خطا گزارش داده می شود بار دیگر خطای دیگر را داده ببینیم دوباره کدام پیغام خطا گزارش داده می شود

در خطاهای بیش از یک خطا داشتیم این را برای آن خطا چک می کنیم به ازای یک خطا کدام روش خطای کمتر (پیغام های خطای کمتر) را به ما می دهد در مورد کل خطا ها این را انجام میدهیم

و از بین آنها خطای کمینه را انتخاب می کنیم روش در حد تئوری است مثلاً ۱۰ خطا هر کدام ۲ حالت ۲^۱۰ حالت باید بررسی کرد.

قانون 3 :

مثال: $First'(A \to c) . Follow(A) \cap First'(cc) = \phi$

مثال) آیا گرامر زیر $LL(2)$ است؟ LL است؟

$S \to Ab|Ac$

$A \to aA|\varepsilon$

جواب: این گرامر به ازای هیچ k، $LL(k)$ نیست.

چون تا b یا c دیده نشود نمی‌توان گفت از کدام قانون استفاده می‌کند و تا کاملاً آخر رشته دیده نشوند نمی‌توان قانون را مشخص کرد که کدام قانون می‌شود.

هیچ پارسر بالا به پایین نمی‌تواند این کار را انجام دهد اما چون پارسرها پایین به بالا این کار را انجام خواهند داد.

روشهای تجزیه پایین به بالا:

Buttom-up Parsing

از رشته شروع کرده عمل انتقاص را انجام میدهیم عکس RMD هستند

مثال)

1) $S \to aABe$

2,3) $A \to Abc|b$

4) $B \to d$

رشته ورودی: abbcde

$$S \overset{1}{\underset{R}{\Rightarrow}} aABe \overset{4}{\underset{R}{\Rightarrow}} aAde \overset{2}{\underset{R}{\Rightarrow}} aAbcde \overset{3}{\underset{R}{\Rightarrow}} abbcde$$

29

گرامرهای $LL(k)$:

گرامر $LL(2)$:

(دو تای اول)

$A \to a \Rightarrow First'(A) = First\{a\}\ Concat\ Follow(A)$

مثال)

$S \to a|bA$

$A \to c \qquad First'(S \to a) = \{a\$\}$

در صورت مشاهده $a\$$: به جای $S \to a$ را انتخاب می‌کنیم

$S \to a|aAb$

$A \to c|cc \qquad First'(A \to cc) = \{cc\}$

$First'(aAb) = \{cb\}$

$S \to a|aAb$ — برای قانون ε باید دنبال $Follow'$ باشیم:

$A \to \varepsilon|cc$ — $Follow(A)$ آنچه به دنبال A می‌آید + $First(A)$

$Follow'(A) = \{b\$\}$

$Follow'(S) = \{\$\$\}$

در جدول پارس:

به تعداد Non-Terminal ها سطر داشته و ترکیبات دوتایی هم ستونهای ما را

برای کشیدن جدول تشکیل میدهیم.

برای $LL(2)$ همان قوانین برای $First'$ و $Follow'$ بررسی می‌شوند

+ قانون ۳

هر فرم جمله ای یک عبارت به حساب می آید.

$S \Rightarrow aABe \Rightarrow aAde$

عبارت ساده است

Handle:

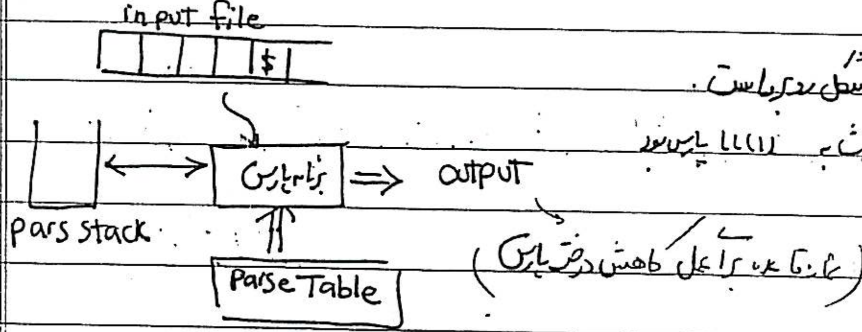
به سمت چپ ترین عبارت ساده که در یک فرم جمله ای راست وجود داشته باشد Handle گفته می شود.
(طبق RMD بوجود آمده باشد.)

همیشه Handle باید سمت راست با قواعد تطابق داشته باشد.

چون عکس RMD، Handle می آید لذا باید سمت چپ ترین عبارت ساده انتخاب شود به Handle

اگر گرامر مبهم باشد علاوه بر سمت راست در انتخاب Handle دچار مشکل می شویم

پارسرهای Shift Reduce (پارسرهای کاهش انتقال)



دارای شکل روبرو است.

ساختارش به LL(1) شبیه بود

(شامل ترتیب اعمال کاهش در حین پارس)

در پارس پایین به بالا. عمل پارسینگ معکوس برعکس انجام میشود. مثلا abbcde

b به عنوان Handle انتخاب می شود و طبق قانون 3 برمی گردد به A و بعد دوباره

Abc به عنوان Handle انتخاب و طبق قانون 2، Reduce می شود به aAde

aABe → S (1)

aAde (2)

aAbcde (3)

abbcde

چون آخرین پایانه ای بود که کاهش یافت

(انتخاب چپ ترین هنوزی که وجود دارد) چون سمت چپ ترین بود این انتخاب شد. انتخاب می شود.

همینطور ادامه میدهیم تا به start برسیم

Phrase (عبارت):

یک عبارت بخشی از یک فرم جمله ای است که دقیقا از یک غیرپایانه بوجود آمده باشد
(همه تا باید از یک شروع کنیم)

مثال

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

γ یک عبارت گفته می شود که از A بوجود آمده

عبارت ساده:

به عبارتی ساده گفته می شود اگر دقیقا در یک مرحله بوجود آمده باشد

$S \Rightarrow aABe \Rightarrow aAde$

عبارت ساده در یک گام بدست آمده

عبارت است اما در یک مرحله بوجود نیامده از ... لذا ساده نیست.

۴ عمل اصلی داریم: 1. shift 2. Reduce 3. Accept 4. Error

Shift: یعنی انتقال lookAhead (توکن جاری) به بالای Pars stack

Reduce: یعنی از بالای Pars stack به عنوان Handle حذف شده و به جای آن یک غیر پایانه وارد stack شود

شرط آغاز کار: یک $ به انتهای رشته ورودی اضافه کرده و یک $ در ته stack قرار میدهیم

شرط پذیرش: از رشته ورودی فقط $ و در stack، $S داشته باشیم

Error: یا به حالتی وارد شویم یا Handle انتخاب شده باشد که با سمت راست قواعد تطابق نداشته باشد.

مثال >

1) E → E+E
2) E → E*E
3) E → (E)
4) E → id

رشته مورد نظر: id+id*id$

ابتدا L.A=id

چیزی در stack نیست (shift)

عمل	stack	L.A
	$	L.A=id
s (shift)	$id	+
R4 (Reduce)	$E	
s	$E+	id
s	$E+id (handle)	*
r4	$E+E	
s	$E+E*	id
s	$E+E*id	$
r4	$E+E*E (Handle)	
r2	$E+E	
r1	$E	
	Accept	

تداخل R و S (اولویت دهی) — S را انتخاب کردیم

اگر تمام stack را بالا بخوانیم و رشته ورودی که پیمایش می‌شود، RMD را خواهیم دید

گرامر مبهم است چون تداخل shift_Reduce داریم (دو جدولی)

سوالات:

1) چطور تشخیص دهیم عمل بعد عمل shift است یا Reduce؟

2) کدام سمت از stack را بررسی کنیم تا Handle شود؟

3) اگر زمانی بفهمیم عمل Reduce است و Handle سمت راست کدام قاعده بود کدام را

یک Function با پارامترهای P
Parameter list

statement

انتخاب کنیم (بدون تداخل کاهش - کاهش)

مثال:

(1,2) ST → id (PL) | E := E → expression

3,4) PL → PL , P | P → parameter

5) P → id   expression list   می‌توان با آرایه عرضاً جدی

6,7) E → id (EL) | id   A:=B(I,J) را داشت. هم می‌تواند آرایه باشد هم تابع

8,9) EL → EL , E | E

آرایه با پارامترهای E $A:=B(I,J) \rightarrow id_1 := id_2 (id_3 , id_4)$

\$id → R_2 → E:=E   \$E:= →(S) \$E:=id

S → \$E:=id_2 (id_3 r   به کاما رسیدیم

SymbolTable (Func / arr / var)

57 با کاما باید first Reduce کدام را انجام داد

که در این حال با استفاده از symbol Table تعیین

تعیین func یا var ... معلوم می‌شود از کدام باید

استفاده کنیم. مثلاً اگر آرایه بود از 7 استفاده می‌کنیم

تقدم عملگر Operator Precedence (OP):

گرامر عملگر:

گرامری عملگر است که هیچ دو غیرپایانه پشت سر هم را دارا نباشد.

1. هیچ قاعده A → ε نباشد.

2. در سمت راست هیچ قاعده‌ای دو غیرپایانه مجاور نداشته باشیم.

مثلاً:
A → EAE | (E) | id   X
A → + | *
مناسب نیست شرط لازم را ندارد.

E → E+E | E*E | (E) | id   X

شرط لازم را دارد نه قاعده ε دارد نه دو غیرپایانه مجاور اما چون مبهم است استفاده نمی‌شود. شرط کافی را ندارد. (در جدول پارس تداخل داریم)

S → \$E\$
E → E+T | T
T → T*F | F   → شرط لازم و کافی را دارا بررسی کنیم
F → (E) | id   بدون تداخل است. جدول پارس را رسم می‌کنیم

	+	*	(	)	id	\$
+	⋗	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋖	⋗	⋖	⋗
(	⋖	⋖	⋖	≐	⋖	
)	⋗	⋗		⋗		⋗
id	⋗	⋗		⋗		⋗
\$	⋖	⋖	⋖		⋖	≐

برای (⋗) باید دید کدام پایانه بعد از غیرپایانه آمده یعنی lastterm NT از T بزرگتر است.

از first term(s) کوچکتر است.

33

در این روش بین عملگرها تقدم برقرار می‌کنیم.

روابط تقدم:

$a \doteq b \not\rightarrow b \doteq a$

$a \cdot> b \not\rightarrow b <\cdot a$

$a <\cdot b \rightarrow b \ ? \ a$ ← روابط تقدم b و a را نمی‌دانیم.

در جدول روابط تقدم را پر می‌کنیم و چون روابط تقدم یکطرفه هستند a را در سطر می‌بینیم b را در ستون. مثلاً اگر + از ضرب جلوتر است + را در سطر می‌بینیم * را در ستون.

$a \doteq b$ iff $\exists U \rightarrow \ldots ab \ldots$ or $\exists U \rightarrow \ldots aEb \ldots$

اگر در سمت راست گرامر دو پایانه مجاور داشته باشیم سمت چپ با سمت راست مساوی است

و اگر یک غیرپایانه هم بین آنها باشد باز سمت چپ با سمت راست غیرپایانه مساوی خواهد بود.

تابع Firstterm(A):

مجموعه‌ای از پایانه‌ها که در صفر گام یا بیشتر به $a\beta$ یا $Ba\beta$ برسد.

$$Firstterm(A) = \{a \mid A \Rightarrow^* a\beta \text{ or } A \Rightarrow^* Ba\beta\}$$

تعریف first

تابع lastterm(A):

$$Lastterm(A) = \{a \mid A \Rightarrow^+ \beta a \text{ or } A \Rightarrow^+ \beta a B\}$$

	firstterm	lastterm
E	{+, *, (, id}	{+, *,), id}
T	{*, (, id}	{*,), id}
F	{(, id}	{), id}

پایانه‌هایی هستند که از آخر شروع می‌کنیم تا آخر باشند یا قبل از یک غیرپایانه باشند. مثلاً E+T → lastterm

Firstterm(E)

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

روابط تقدم روابطی یکطرفه است.

$a <\cdot b$ iff $\exists U \rightarrow \ldots aB \ldots$ and $b \in Firstterm(B)$

$a \cdot> b$ iff $\exists U \rightarrow \ldots Ab \ldots$ and $a \in lastterm(A)$

$\$E\$$ آخرین مرحله‌ای است که باید با این گام اشتقاق رشته گرامر را تطابق داشته باشد. پس $S \rightarrow \$E\$$ را به ترتیب گرامر اضافه می‌کنیم.

خانه‌های خالی جدول نشان دهنده خطاهای نحوی هستند.

چون در جدول تداخل نداریم ⇐ مشکلی از نظر ابهام نداریم.

31

هرجا با $ رابطه داشته باشد از آن بزرگتر است چون $ در انتها می‌آید.

غیرپایانه‌ها ظاهر نمی‌شوند.

نحوه‌ی انجام عمل پارس به روش تقدم عملگر: (تقدم پایانه را مطالعه می‌کنیم)

در هر مرحله از پارس با نگاه کردن به پایانه بالای stack (a) و توکن جاری (b) به سراغ خانه‌ی PT(a,b) رفته و به صورت زیر عمل می‌کنیم. (PT: pars Table)

1. اگر PT(a,b) مساوی علامت کوچکتر باشد آنگاه ابتدا علامت کوچکتر پس توکن جاری یعنی b را وارد stack می‌کنیم. (shift)

2. اگر PT(a,b) مساوی علامت مساوی باشد آنگاه b را وارد stack می‌کنیم (shift)

3. اگر PT(a,b) مساوی نماد بزرگتر باشد آنگاه داخل stack تا رسیدن به اولین نماد کوچکتر پایین می‌رویم رشته‌ی بالای این علامت (و غیرپایانه زیر آن در صورت وجود) Handle است آن را از بالای stack حذف کرده و به جای آن یک غیرپایانه‌ی نمادین وارد stack می‌کنیم (Reduce)

کوچکتر سمت چپ دهانه سمت راست.

4. اگر PT(a,b) خالی باشد آنگاه یک خطای نحوی رخ داده است که باید خطاپرداز فراخوانی شود.

مثال/ id+id*id$

(اندازه جدول: $(|T|+1)^2$ خانه)

$	1.A=id
$<.id (غیرپایانه N)	+
$N<.+ (غیرپایانه عادی (مثلا N))	id
$N<.+<.id	*
$N<.+N	
$N<.+N<.*	id
$N<.+N<.*<.id	$
$N<.+N<.*N	
$N<.+N	
$N (در رشته غیرپایانه عادی نگذاشته)	
Acc.	

برای اثبات ... این روش ... غیرپایانه‌ها عملگرها ... ضعیف است ... ممکن است رشته‌های ... غیر رشته‌های گرامر را بپذیرد.

مثال ۲

S → A+B | C+D

A → a

B → b

C → c

D → d

مثلاً a+c را accept می‌کند.

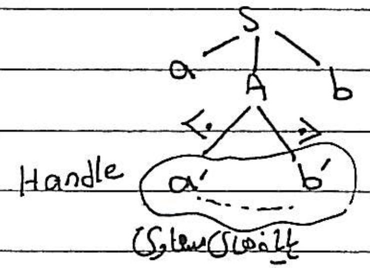
Δ در رشته نیست.

تقدم عملگر مشکل مراحل کاهش ـ کاهش نداریم

با نگذاشتن غیرپایانه‌ها درین حتی ضعیف‌تر خواهد شد.

از لحاظ قدرت کشف خطا LL(1) از تقدم عملگر قوی‌تر است. چون LL(1) هرگز رشته‌های غلط را پارس نمی‌کند.

روابط تقدم خاص این روش هستند و هدف شناسایی Handle ، Reduce بود.



نحوه‌ی برخورد با خطا در روش تقدم عملگر:

پارس جدول LR(1) :

پیکربندی: Configuration

زوج مرتبی است که دلیل اول آن محتوای stack و محتوای دوم آن باقیمانده رشته ورودی است.

Configuration: (stack contents, remaining input)

یکی در میان state داریم بین هر دو state یک علامت است ابتدا و انتهای stack و state است. مبدا configuration

$(S_0 X_1 S_1 X_2 S_2 \ldots X_n S_n , b \ldots \$)$ ⊢

Action(S_n, b) =
- shift j : $(S_0 X_1 S_1 \ldots X_n S_n b S_j , \ldots \$)$
- $|\alpha| = r$: Reduce$_{A \to \alpha}$
- هنگامی که $ در ورودی دیده باشیم و ... : Accept
- فقط در انتهای ورودی دیده شده باشد : Empty

Accept رویت می شود.

Reduce$_{A \to \alpha}$: $(S_0 X_1 S_1 \ldots X_{n-r} S_{n-r} A S_m , b \ldots \$)$

Goto$(S_{n-r}, A) = m$

مثال/ id+id*id $

Stack	
Q	L.A = id
Qid5	+
QF3	
QT2	
QE1	L.A
QE1+6	
QE1+6id5	id
QE1+6F3	*
QE1+6T9	
QE1+6T9*7	id
QE1+6T9*7id5	$
QE1+6T9*7F10	
QE1+6T9	
QE1	
Acc.	

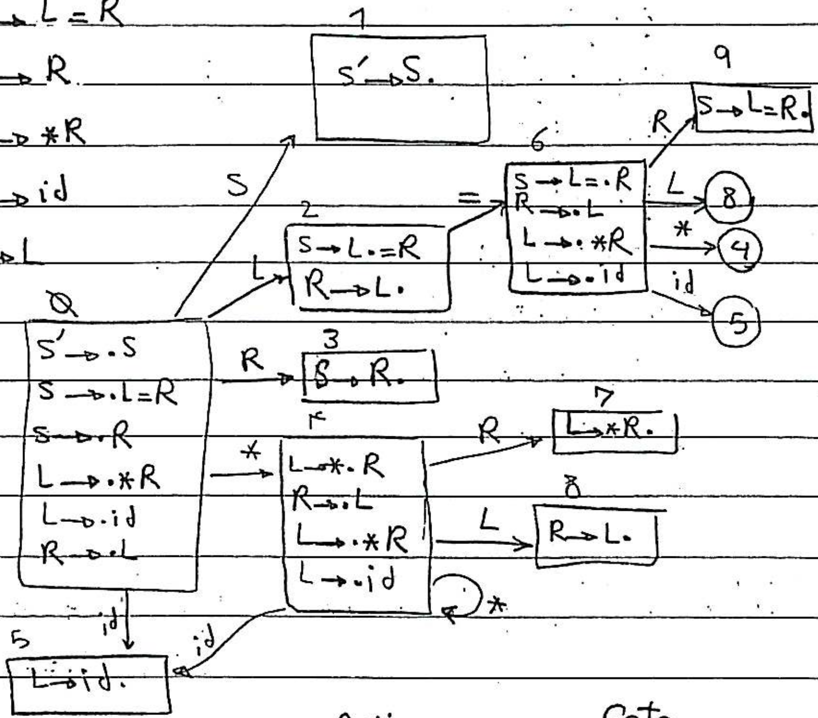
آنچه در بالا علامت خط کشیده ایم یا خود Handle هستند یا پیش وند آن در اصطلاح به آن Viable prefix گویند.

اگر قاعده ای به فرم $A \to \alpha . B\beta$ به ازای آیتم 2 از top بیشتر نداریم
$B \to \varepsilon$
و بعد با قانون $B \to \varepsilon$ ، reduce می کنیم.

اگر گرامر مبهم داشته باشیم به ازای هیچ k ، LR(k), SLR(k), CLR(k) نیست. تداخل کار حتمی است.

اگر SLR نداشتیم حتما نمی توان گفت گرامر مبهم است.

1) S → L = R
2) S → R
3) L → *R
4) L → id
5) R → L



	Follow
S	{$}
L	{=,$}
R	{$,=}

	Action				Goto		
	=	*	id	$	S	L	R
0		s4	s5		1	2	3
1				Acc			
2	S6/r5			r5			
3				r2			
4		s4	s5			8	7
5	r4			r4			
6		s4	s5			8	9
7	r3			r3			
8	r5			r5			
9				r1			

تداخل در نتیجه SLR(1) نیست

باید با تمام شماره قواعد Reduce داشته باشیم و باید دقت به ACTION در بخش

Shift های goto اعداد را نگاه می‌کنیم باید شماره state ها ظاهر شده باشد

در جدول مشکل درست عمل کرده‌ایم.

گرامر مثل SLR(1) نبود اما ممکن است SLR(2) باشد.

LALR(1) : تعداد subset از follow ها lookahead است

در مثل SLR(1) shift ها با reduce ها کم می‌شود. تعداد reduce ها => لازم می‌شود

است.

یعنی اگر فرض کنیم گرامر شامل LALR(1) است ممکن است بی‌شود از بین به کمک آن باید state ها ادغام شوند. چون جدول LALR(1) فشرده‌تر است خطا دیرتر کشف می‌شود.

تعریف یک state مشکوک به تداخل:

دو نوع داریم

I) state های مشکوک به تداخل s/r (shift/Reduce)

هر آیتمی داریم که نقطه آخر نیست و هر آیتمی داریم که نقطه آخر است یعنی هم shift داریم هم Reduce.

گرامر در مرحله اول نیاز به بررسی نداریم زیرا مسلول نیست shift نیز مربوط به goto است

و ما shift در گرامر مرحله قبل نداریم.

زمانی پتانسیل shift داریم که نقطه قبل از یک پایانه قرار گرفته باشد.

LR(0) : (بدون دیدن token بعدی shift و Reduce تعیین می‌گردد.)

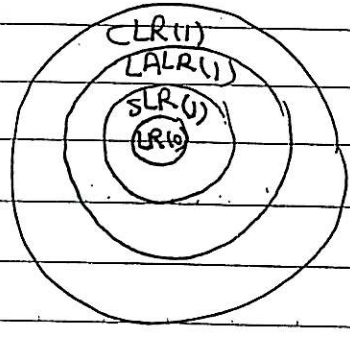
اگر در دیاگرام SLR(1) یک گرامر هیچ state مسلول به تداخل وجود نداشته باشد آن گرامر LR(0) نامیده می‌شود.

بدون دیدن ورودی می‌توان گفت عمل بعدی shift یا Reduce است.

حال اینکه مشخص کنیم shift با کدام شماره قانون است باید ورودی را دید

اما اگر بفهمیم Reduce است می‌توان بدون دیدن ورودی فهمید Reduce

با کدام شماره قانون می‌باشد



در صورتیکه پایانه ثبت در follow آن NT سمت چپ وجود داشت به تداخل

داریم (حتما s/r است)

II) تداخل مسلول به Reduce Reduce

اگر در یک state حداقل 2 Item داشته باشیم که نقطه انتهای آنها رسیده باشد

state مسلول به تداخل r/r است.

زمانی در follow آن NT هم‌قانون اشتراک داشته باشند تداخل محقق می‌شود.

زمانی در دیاگرام رسم می‌کنیم state های غیر مشکوک نیز نیاز به بررسی ندارند تا فقط

state های مسلول را بررسی نمود.

مثال

S → AB

A → a

B → ε

state نیاز به بررسی دارد که نقطه قبل از یک پایانه

قبل از shift قرار گرفته باشد.

S' → S

S' → .S

S → .AB

A → .a

A: [S → A.B , B → .] —B→ [S → AB.]

a: A → a.

تمرین) آیا گرامر زیر $LR(0)$ است؟

$E \rightarrow E+T \mid T$

$T \rightarrow (E) \mid id$

روش $CLR(1)$:

اصطلاح Kernel item (آیتم هسته‌ای):

به یک آیتم هسته‌ای گفته می‌شود اگر نقطه در شروع سمت راست آن قرار نگرفته باشد

(به استثنای قاعده $S' \rightarrow .S$) بقیه آیتم‌ها آیتم‌های غیر هسته‌ای (none kernel item) گفته می‌شود.

مثلاً: $A \rightarrow \alpha.B\beta$ هسته‌ای و $A \rightarrow .\gamma$ و یا $A \rightarrow .$ غیر هسته‌ای هستند

نکته:

در هر state حداقل یک آیتم هسته‌ای را داریم.

ما می‌توانیم آیتم‌های غیر هسته‌ای را از روی هسته‌ای بسازیم (صرفه‌جویی در حافظه مصرفی)

: LR(1) Item

زوج مرتبی است که درایه اول آن یک آیتم $LR(0)$ و درایه دوم آن مجموعه‌ای

از پایانه‌ها یعنی look_ahead است.

مثلاً: $LR(1)$ ✓ مثلا $(A \rightarrow \alpha.B\beta\ ,\ \{a_1,a_2,a_3\})$

($A \rightarrow \alpha.B\beta$: $LR(0)$ ، $\{a_1,a_2,a_3\}$: look_ahead)

حال اگر از $LR(1)$ ، closure بگیریم

$closure(I) = \{I'\}$

$(B \rightarrow .\gamma\ ,\ \{b_1,b_2,\ldots\})$

$b_i \in First(\beta.\{a_1,a_2,a_3\})$

اگر $\beta = \varepsilon$ باشد یعنی a_1,a_2,a_3 باشند

مثال) رسم دیاگرام $CLR(1)$

$S' \rightarrow S$

1) $S \rightarrow CC$

2,3) $C \rightarrow cC \mid d$

با این شروع می‌کنیم

$S' \rightarrow .S\ ,\ \{\$\}$

closure می‌گیریم

$S \rightarrow .C\underset{\beta}{C}\ ,\ \{\$\}$

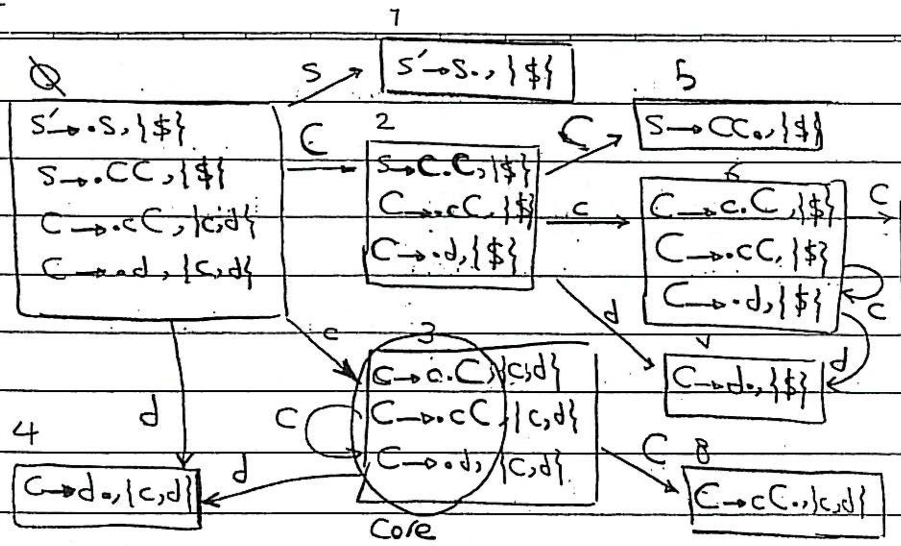
$C \rightarrow .cC\ ,\ \{c,d\}$

$C \rightarrow .d\ ,\ \{c,d\}$

*مهم

اگر نقطه مثلاً $C \rightarrow c.C$ بود از آن C که بعد دوباره باید closure بگیری می‌کردیم

آنقدر باید ادامه دهیم که دیگر lookahead ها بدست آید و اجتماع همه LA ها را در آخر بدست می‌آوریم



در جدول حالت قبل است منظورم جاهایی که به نظر رسیدیم بجای Follow در زیر

آنها reduce. قرار دهیم در زیر lookahead آن Reduce قرار دهیم.

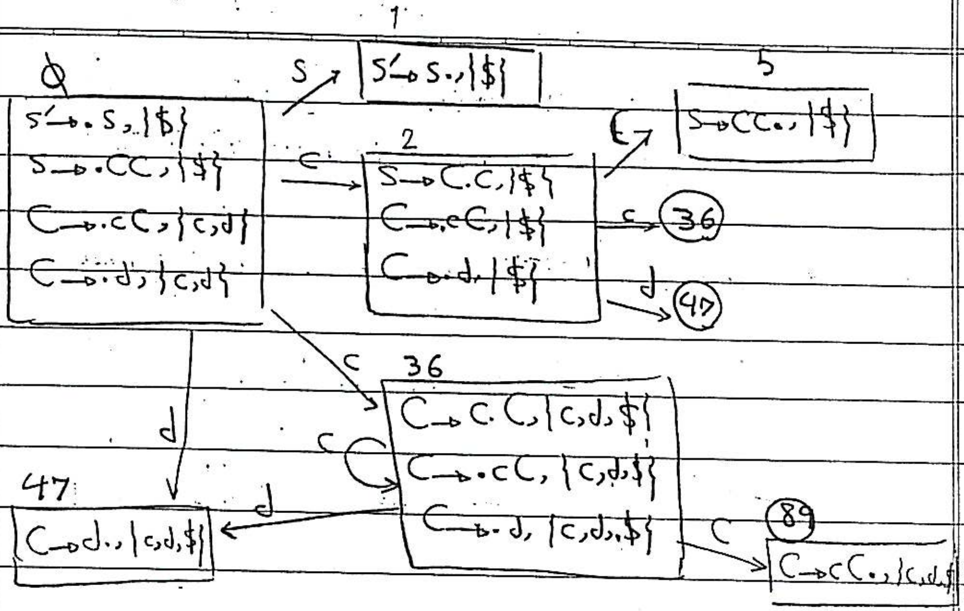
3،6 مشابه هستند فقط در L.A فرق دارند. اگر بخش اول آن قواعد Core

گوییم و آنهایی که Core یکسان دارند را با هم ادغام کنیم و L.A آنها را اجتماع

بگیریم میزان دیاگرام را کوچکتر نمود. و دیاگرامی بین CLR(1) و SLR(1)

داشت

- 3،6
- 4،7
- 8،9

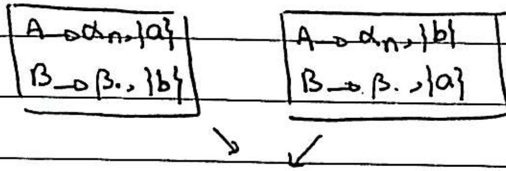


امکان ندارد گرامری CLR(1) باشد ولی بدلیل تداخل shift_Reduce ،

LALR(1) نباشد. (زمانی که در LALR(1) تداخل s/r داریم حتماً در CLR(1) هم این تداخل را داریم)

* مهم: اگر گرامر CLR(1) باشد ممکن است LALR(1) نشود به دلیل آن تداخل Reduce_Reduce

نباشد.



چون تلفیق state ها

گرامر CLR(1) تمام T های تلفیق تداخل rr داریم.

(سطرها) که کل شیفت هستند یکسان اند ← تداخل s/r نداریم اما r/r ممکن است

Reduce.

اگر گرامری (LALR(1 باشد و (SLR(1 نباشد. تعداد R در هر سطر

کمتر مساوی (SLR(1 است ⇐ پتانسیل تداخل پایین می آید. ⇐ حوزه بیشتری از

گرامرها را بتوانند تحت پوشش قرار دهد.

حوزه ی گرامرهایی که با (CLR(1 است تحت پوشش بیشتری نسبت به (LALR(1

خواهد بود. چون تعداد stateها در (CLR(1 زیادی شوند ⇐ تعداد سطر

زیادی شوند ⇐ تعداد R نسبت به سطر (خانه های R در state) کاهش پیدا میکنند

⇐ امکان تداخل کم می شوند ⇐ احتمال اینکه گرامر (CLR(1 باشد تا (LALR(1

بیشتر خواهد بود.

اگر گرامری (SLR(1 باشد در مورد نسبت کل Rهای (LALR(1

(SLR(1 و (CLR(1 :

تعداد Rهای (LALR(1 تو خانه ها مساوی (SLR(1 خواهد بود.

CLR(1) نباشد ⟵	LALR(1) نیست
LALR(1) نباشد ⟵	SLR(1) نیست
SLR(1) باشد ⟵	LALR(1) هست
LALR(1) باشد ⟵	CLR(1) هست

اگر در یک سطر جدول reduce با ۲ شماره داریم ⇐ آن سطر مشکوک است.

-۱۱

مهم * (اگر state مسئول به // بنداریم یا داریم آنگاه آن state تلفیق پذیر نیست آن گرامر (LALR(1 هست. در صورتی که (CLR(1 باشد.

تلفیق پذیری در مورد core بود که stateها تلفیق پذیر که اگر core یکسان

داشته باشند. در جدول اگر shift ها در یک محل و Reduce هاهم شماره

باشند (با اینکه دیگر جابجا کنند) تلفیق پذیر هستند. اگر به state مسئول داشتیم که

تلفیق پذیر نبود (LALR(1 هست. در تلفیق پذیری حتماً باید goto آن دو state

یکسان باشد تنها تفاوت در محل Reduce هاست در آن با توجه به جدول

LA در نظر گرفته میشود. اما Reduce ها هم شماره هستند.

shift Reduce تداخل

چون اگر جدول (LALR(1 یک گرامر را بدهند و گرامر تداخل SR ... (LALR(1 باشد.

در مورد (CLR(1 بود نیز میتوان گفت (CLR(1 نیست. و (SLR(1 هم

قطعاً نخواهد بود چون (LALR(1 نیز نیست)

توجه به گرامری که در جدول (LALR(1 آن تداخل S/R باشد مطمئن باشید که در جدول (CLR(1

آن تداخل S/R داریم

47 state 2 {a,b} reduce ε

state 1,2 {a,b,c} reduce c

مثال ←

Reduce

تداخل های R ها LALR(1) که هیچ مسئله ای CLR(1) خواهد بود. لذا در پارس LALR(1) تعداد R ها را در بین بیشتر دارا خواهد بود.

مثال) فرض کنید جدول زیر SLR(1) باشد.

	Action				Goto		
	a	b	c	$	A	B	C
0	s_4		(r_1)			2	
1	r_1	s_3		Acc		(3)	1
2	s_2/r_1		r_3	r_3	3		
3	r_3/r_2	r_2/r_1				2	
4	s_3/r_2		r_1	r_3			

حتما در مرحله صفر، ε داشتیم که Reduce داشتیم

اگر بود مشکل داشتیم چون نمی شد که در مرحله صفر با r_1، goto با B بوده اما در مرحله (1) چون r_1 با C انجام شده و B بوده باشد

این گرامر بنا به سطر (4)، SLR(1) نیست.

بدون در نظر گرفتن سطر 2 و 3 و با دیدن سطر 4، نه CLR(1) است نه LALR(1)

برای LALR

نمیتوان r_2 را برداشت چون در هر state در سطر 3 با هم ادغام شده باید یک R را داشته باشیم و چون follow را نمیتوانیم داشته باشیم لذا LALR(1) است و با همین توجیه CLR(1) هم نیست.

حال بنا بر سطر 2، نیز نه CLR(1) است نه LALR(1). در مورد سطر 3 نیز LALR(1) نیست چون نمی توان هر دو r_2 را برداشت یکی می ماند ⇐ با r_2 LALR(1) نیست

تداخل r_2 است ممکن است CLR(1) شود. r_1 و r_3 باید در هر جا باشد r_2 باید در دو state شود و در هر جای باشد با r_1 یا r_3 ایجاد تداخل میکند ⇐ CLR(1) نیز نخواهد بود.

(برای CLR(1) بودن ممکن است state ها را جدا کنیم به ۲ state تبدیل کنیم)

مثال

قانون	
3. A → id.	{a}
2. B → +.	{a,b}
1. C → *.	{b}

⇒

A → id.	{a}
B → +.	{a}
C → *.	{b}

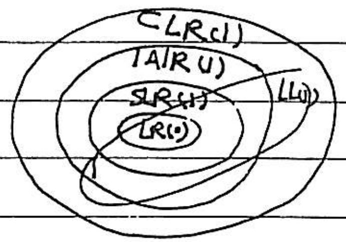
A → id.	{a}
B → +.	{b}
C → *.	{b}

باز در هر دو تا state برخورد داریم.

[illegible]

	a	b	c	$
i	s_1	s_2	r_1	
j	s_1	s_2	r_2	r_1

جدول CLR غلط است چون Core های یکسان دارد
با r های متفاوت ← نمی‌تواند LALR باشد.



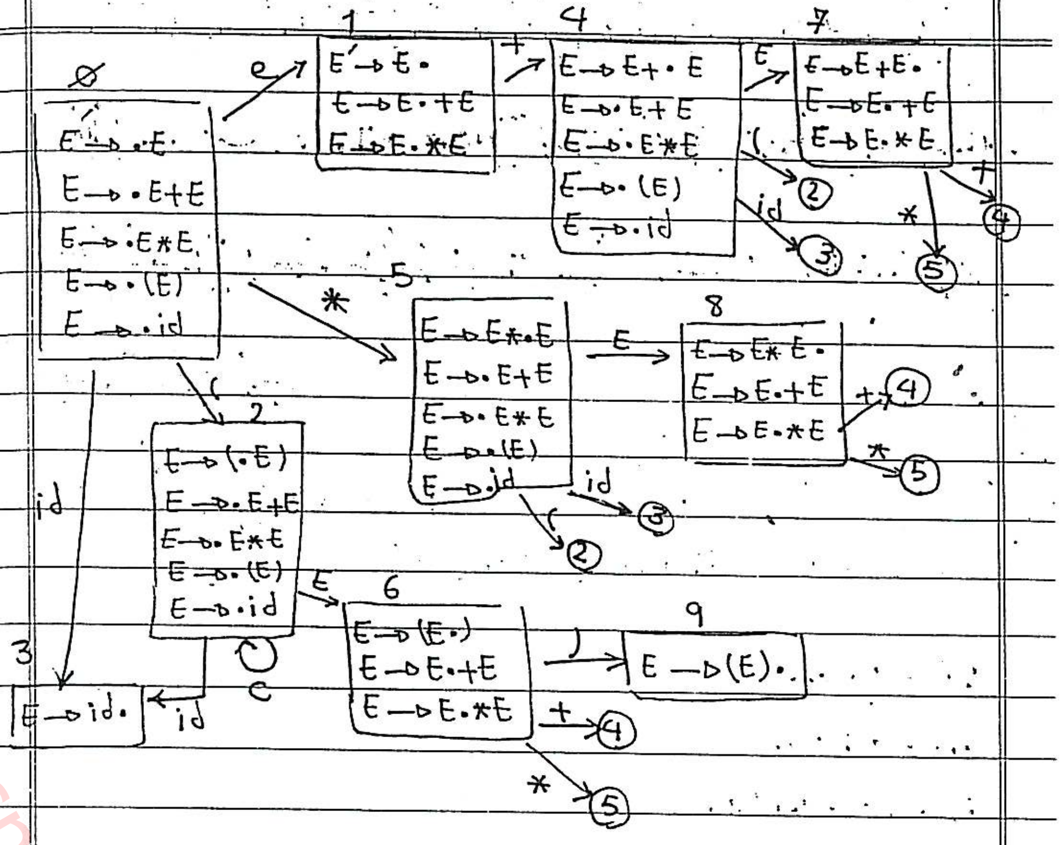
هر گرامری که LL(1) است حتماً
CLR(1) هست.
اما گرامری که CLR(1) است لزوماً
LL(1) نیست.

اگر گرامر LL(1) باشد اما LALR(1) نباشد مسلماً تداخل r/r دارد. *

سوال > آیا گرامری LL(1) باشد و تداخل نداشته باشد در مورد LR(0) بودن آن
می‌توان قضاوت کرد؟

رسم دیاگرام SLR(1) برای گرامر مبهم زیر:

1) E → E+E
2) E → E*E
3) E → (E)
4) E → id



	+	*	(	)	id	$	E
0			s_2		s_3		1
1	s_4	s_5				Acc	
2			s_2		s_3		6
3	r_4	r_4		r_4		r_4	
4			s_2		s_3		7
5			s_2		s_3		8
6	s_4	s_5		s_9			
7	~~s_4~~ ① r_1	~~s_5~~ ④ ~~r_1~~		r_1		r_1	
8	~~s_4~~ ④ r_2	~~s_5~~ ④ r_2		r_2		r_2	
9	r_3	r_3		r_3		r_3	

shift باشد:
id+id+id ①
id+id*id
s ②
id*id+id
③
④
id+id*id

گرامر SLR(1) نیست اما توانستیم برای آن پارسر SLR(1) بسازیم

مثلاً با 12 حالت و 3 goto برای این گرامر مبهم که رفع ابهام شده اش پارسر SLR(1) ساختیم اما در اینجا با صرف هزینه کمتر و یک NT در بخش goto ها و 8 state با پارسر همین سری حل کردیم.

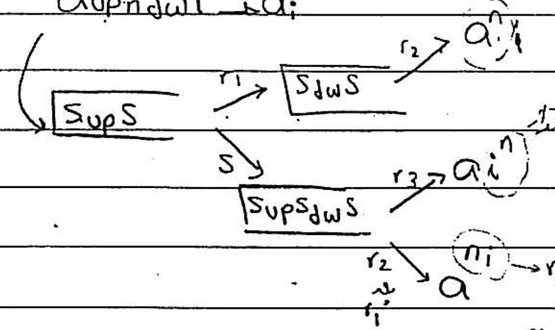
مثالی از گرامر مبهم با تداخل r/r :

گرامری برای یک نرم افزار EQN → Editor

1) $S \rightarrow SupS$ $\quad aupn \rightarrow a^n$

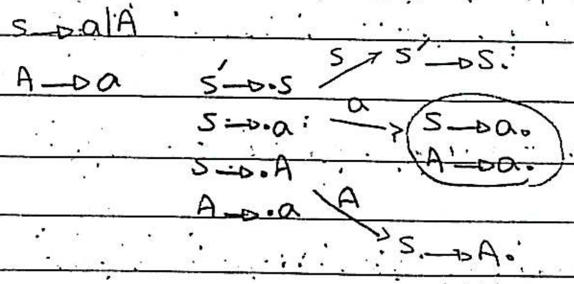
2) $S \rightarrow SdwS$ $\quad adwi \rightarrow a_i$

3) $S \rightarrow SupSdwS$ $\quad aupndwi \rightarrow a_i^n$



که رفع تداخل r/r سخت تر از s/r است

سوال) اگر جدول SLR(1) گرامر رسم شد و تداخل s/r نداشت آیا میتوان گفت گرامر مبهم نیست ؟ جواب خیر است و این گرامر میتواند تداخل r/r داشته باشد ← مبهم باشد.



مثلاً الگوریتمی برای اثبات عدم ابهام گرامر وجود ندارد r/r

اگر گرامر ابهام داشت حتماً نوعی تداخل در جدول داریم.

روشهای خطایابی گرامر :

phrase_level : در خانه های خالی جدول یک Pointer به روتین خطا میگذاریم مثلاً در جدول مثل (انتظار بسته unbalanced ، unexpected eof = a$

اگر در state 1 پرانتز بسته ببینیم آنگاه اولاً با یکی از state های 2 و 9 ، 8 یا 3 r انجام شده . مثلاً اگر id بود میتوان گفت) اضافه است

state 6 و id ← اپراتور جا افتاده

6 و $ ← پرانتز بسته جا افتاده چون حداقل یک پرانتز بسته لازم است follow های E را ببینیم

Error production :

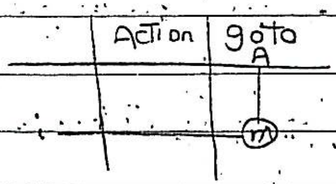
برای خطاهای متداول قواعد با ساختار متنی میگذاریم ← کار پارسر با خطا متوقف نمیشود

41

Panic Mode

بخش زیادی از ورودی حذف می‌شود. اگر به خانه خالی برخوردیم شروع به حذف ورودی می‌کنیم. دچار خطا شویم . باید مصنوعی انجام دهیم.

شروع به حذف عناصر stack می‌کنیم تا برسیم به یک state که در بخش goto زیر یک غیر پایانه عدد داشته باشیم (خالی نباشد بخش goto). آنقدر از ورودی حذف می‌کنیم تا به Follow(A) برسیم. و L.A کار token جدید است و در state هستیم که نقطه بعد از A قرار دارد => انتقال می‌دهیم Follow(A) را داریم.



| ... Si

| ... Si A Sm   goto(Si, A) = m

حال اگر به state برسیم که در بخش goto زیر بیش از یک غیر پایانه پر بود آنگاه دو راه وجود دارد.

I) آنکه Follow بزرگتری دارد آن را انتخاب کنیم چون ممکن بود زودتر به آن برسیم و تعداد کمتری از عناصر ورودی حذف شود.

II) آنقدر از ورودی حذف کنیم تا به Follow یکی از آنها برسیم سپس با goto نظیر آن به بعد از آن انتقال می‌دهیم.

« Semantic Analyzer »

پرسش‌هایی که تحلیل‌گر معنایی باید به آنها پاسخ دهد:

1. اینکه آیا x یک اسکالر یا یک آرایه یا یک تابع است؟
2. اینکه آیا x قبل از استفاده تعریف شده است؟
3. آیا هیچ نامی وجود دارد که Declare شده باشد ولی استفاده نشده باشد؟ (حداکثر یک warning است نه error)

warning هم به عهده Semantic Anal. است.

4. در یک ارجاع به کدام تعریف x مراجعه کنیم؟ (متناظر با scope باشد.)
5. اینکه آیا یک عبارت type Consistent هست یا نه؟ (یعنی سازگاری

(type

Dynamic checks

برخی را در زمان runtime می‌توان انجام داد.

بیشتر بررسی‌های معنایی static هستند.

type checking استاتیک است (البته اگر زبان تبدیل type نداشته باشد.)

nested related Blocks: w

اگر قرار بود زبان طوری باشد (blockname) مثلاً مستقل از متن نبود، فرمان
end (blockname)

چون که متن‌های نوشته شده‌اند جای اینکه زبان مستقل از متن شود در آخر نام بلاک را برعکس می‌نوشتیم مثلاً rof (به جای for) (wcw^R)

Control flow checks

اندیس خارج از دامنه

- a/0 --> پارسر هم می‌تواند تشخیص دهد.
- a/(5-5) تنها با تشخیص ... Semantic Anal.
- const i=0; a/i در زمان کامپایل می‌شود تشخیص داد چون ثابت است با تحلیل معنایی
- int i=0; a/i --> در زمان اجرا Hardware تشخیص می‌دهد که SA دیگه وجود ندارد

41

6. آیا هیچ ارجاع به تعریف مربوط نشان‌گذاری خارج هست؟ (مثلاً در ارجاع به یک آرایه دو بعدی با 3 اندیس یا 1 اندیس.)

7. ... در کجا ذخیره می‌شود؟ (مثلاً heap، stack و ...)

8. آیا p* به نتیجه‌ای تابعی مثل malloc() اشاره می‌کند؟
ابتدا یک حافظه برای یک متغیر در نظر بگیریم بعد اشاره کنیم

9. آیا ارجاع به یک آرایه در محدوده آرایه می‌گنجد؟ (مثلاً آرایه با اندازه 100 با شماره 101 بخواهیم به آن اشاره کنیم.)

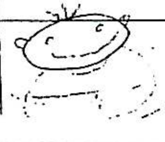
10. آیا تعداد پارامترهای formal و actual در یک تابع با هم همخوانی دارد؟
(formal: در تعریف تابع مشخص می‌شوند / actual: در فراخوانی تابع مشخص می‌شوند)

11. آیا یک تابع f همیشه یک مقدار ثابت برمی‌گرداند؟

Semantic checks:

static checks:

برخی را در زمان Compile می‌توان انجام داد.



حکم می‌کنیم که آیا پارسر می‌تواند تصمیم داده شود یا اینکه بتوان پارسری برای زبانها حساس به متن ساخت. اما معمولاً جوابش منفی است چون فضا مورد نیاز P-space است (یعنی چندجمله‌ای).

در زبانها مستقل از متن، از subset زبانها مستقل از متن قطعی برای O(n) رشته‌ها را شناسایی کرد ولی در زبانها حساس به متن در بدترین حالت برای زبانهای ndcfL $O(n^{2.81})$ است. زبانها حساس به متن حتی بدتر هم هستند ⇐ از دو ایده زیر استفاده می‌کنیم:

I) Ad-hoc methods

II) Formal methods

I) دو ایده طی با موردی بررسی قرار می‌دهیم

A> Symbol Table & code

B> Action routines

خصوصیات مشترک موارد فوق:

- همه آنها به اطلاعات غیر local نیاز داشتند

- پاسخ به بسیاری از پرسش‌ها که به مقدارها وابسته است نه به syntax.

- پاسخ به بسیاری از پرسش‌ها که نیاز به محاسبه دارد. مثلاً در $\%_{5-5}$ ، (5-5) محاسبه می‌خواهد

بسیار مطلب فوق خواصی هستند که با ابزارهای مستقل از متن نمی‌توان به آنها جواب داد.

چون با cf گرامرها نمی‌توان به اطلاعات nonlocal دست یافت یا مقایسه داشت یا محاسبه ⇐ مثلاً برای حصول آرایه اندیس را باید با سقف و کف آرایه مقایسه کرد و اینکار با زبانهای مستقل از متن کافی نیست.

قدرت زبانهای حساس به متن کافی و حتی زیاد هم هست

حال چگونه می‌توان با زبانهای حساس به متن اینکار را انجام داد؟

قواعد Semantic را با زبان حساس به متن می‌سازیم پارسری می‌سازیم و بعد

II) حوزه را انسانه‌ای می‌بینیم. (روش‌های مبتنی گرامر)

مسیر توسط پارسر ← A> Syntax-directed Translation

مشخص می‌شود و جاذبه (attribute grammars)

مراحل پارس عملیاتی انجام بدهیم تا تحلیل معنایی داشته باشیم

B) Type systems and checking Algorithms

جامع‌ترین ایده نه داریم. تبدیل خطاهای معنایی به نوعی Type error ها

Symbol Table & Code

از اطلاعات Symbol Table بیشتری کدهای بعدی استفاده می‌کنند. به عنوان نمونه

یک بلاک

یک scope checking مطرح می‌شود. در هر scope به نزدیک‌ترین

تعریف متغیر ارجاع می‌شود و اگر متغیر در scope جاری نبود در scope

قبلی می‌گردیم. پس از اتمام کار scope دیگر به متغیرهای آن نیازی نیست پس

باید از ST حذف شوند ← متغیرها در ST دائمی نیستند.

اگر هم یک متغیر هم نام ولی با scope متفاوت داشته باشیم ← کنترلر معمولی

باید به کمک scanner آمده و این موضوع را تشخیص دهد و اجازه وارد کردن متغیر را به ST

بدهد چون اسکنر به تنهایی نمی‌تواند identifier ها را در ST وارد کند.

Scope checking

Proc (A) ← چون به ته رسید scope جدید شروع می‌شود

Var x, y : int ;

x := 1 ;

⋮

scope ایجاد شده، قسمت آدرس ← Proc (B)

4 Var (x) : int ;

Z := x + y ; ← 4 به پایین می‌گردیم، پیدا نکنیم از 4 تا 1، باز می‌گردیم پیدا کردیم

⋮

end B

proc C

⋮

endc

scope Global end A

اگر پیدا نشد scope

X: پیدا نکرد نشد خطا بود ولی الان نه

Z = X + (y)

(4) از 4 به پایین می‌گردیم از 1 تا scope B شروع است تا می‌گردیم در ST پیدا نمی‌کنیم

	Name	type	func or var	...	Add
1	A		proc	...	100
2	x	int	var	...	101
3	y	int	var		102
4	B		proc		103
5 (پس از else)	~~X~~, C	~~int~~	~~var~~ func		64
	⋮				

Scope stack:

~~4~~ 5
~~1~~

رفته اگر پیدا نشد پیام error میدهیم

end B

scope عوض می‌شود، B از ST حذف نمی‌شود چون در scope A تعریف شده

ممکن است بعداً call B داشته باشیم

end A ← همه چیز غیر از خود A برداشته می‌شود شماره 1 هم از scope stack برداشته می‌شود

در حقیقت (Semantic Analyser) SA در دستور var x : int ;

معین میکند که اظهار داده است پس است تصمیم نمیگیرد که x را برابر دارند

اگر متغیر استفاده شده در برنامه در scope محلی نباشد تا جایی که scope Global نسبت

به scope محلی وجود دارد به زبان متغیری گردیم اگر متغیر نبود پیام error مبنی بر

undeclared ID را میدهد و کار را ادامه میدهد و فرض می کند (مثلاً % در مثال قبل)

در scope محلی تعریف شده و نوع آن را هم استفاده ای که از آن شده تعریف می کند

Attribute grammers :

تعمیمی روی گرامرهای مستقل از متن هستند برای یک غیرپایانه تعدادی

attribute در نظر می‌گیریم. attribute ها طبق قواعدی مقداری گیرند

که به این قواعد evaluation Rule (قواعد ارزیابی) گوییم. مقدار

صفات attribute ها از صفات گره پدر، گره برادر، گره فرزند و یا از ثابت

میباشد.

اگر صفات از ثابت ها، گره پدر یا گره برادر گرفته شده باشد A . attr1

.attr2

به آن Inherited attribute گوییم

اگر صفات از ثابت ها یا گره فرزند گرفته شوند اصطلاحاً به آن صفت synthesized attribute

گویند.

AT grammer یا Attribute grammer

مجموعه ای از قواعد مستقل از متن مورد استفاده هستند که برای N.T آنها

صفات تعریف می کنیم. گرامرهای جدید را با توجه به صفات تعریف می کنیم.

اگر در گرامری هم صفات Inherited AT باشند گرامر را L attributed grammer

گوییم و اگر در گرامری هم صفات synthesized at باشند گرامر را S_attribute gr

گوییم.

S_attribute grammer برای مقداردهی گره در عمل پارس پایین به بالا مناسب است مثل LR

L_attribute grammer " " " " " بالا به پایین "

مثال :

<NUM> ::= <sign> <list>

<sign> ::= + | -

<list> ::= <Bit> | <list> <Bit>

<Bit> ::= 0 | 1

گرامر تولید کننده اعداد باینری علامت دار است

Production Rules	Evaluation Rules
<NUM> ::= <Sign> <list>	<list>.pos ← 0. <NUM>.val ← if <sign>.neg then -<list>.val else <list>.val
<Sign> ::= -	<sign>.neg = True
<Sign> ::= +	<sign>.neg = False
<list> ::= <Bit>	<Bit>.pos ← <list>.pos <list>.val ← <Bit>.val
$<list_0>$ ::= $<list_1>$ <Bit>	$<list_1>$.pos ← $<list_0>$.pos + 1 <Bit>.pos ← $<list_0>$.pos $<list_0>$.val ← $<list_1>$.val + <Bit>.val
<Bit> ::= 0	<Bit>.val ← 0
<Bit> ::= 1	<Bit>.val ← $2^{<Bit>.pos}$

۱ صفت را از صفری گیرد

صفر صفت را از ۱ می گیرد

صفت list.pos ، Inherited است. (هر جا هم باید pos از فرزند ترشه بیاد)

صفت list.val ، synthesized است

یک صفت sign.neg است که از ثابت مقدار ترشه ۰ یا ۱ پاسخ دیگری ندارد.

۱۵۱- ایستوان با گراف درخت ساخت. <NUM>.val = -5

<sign>.neg = True

<list>.pos = 0

<list>.val = 5

<list>.pos = 1

<Bit>.pos = 0

<list>.val = 4

<Bit>.val = 2^0 = 1

<list>.pos = 2

<Bit>.pos = 1

<Bit>.pos = 2

<list>.val = 4

<Bit>.val = 1

<Bit>.val = 4

-1

pass ۱۲۱ مقایسه

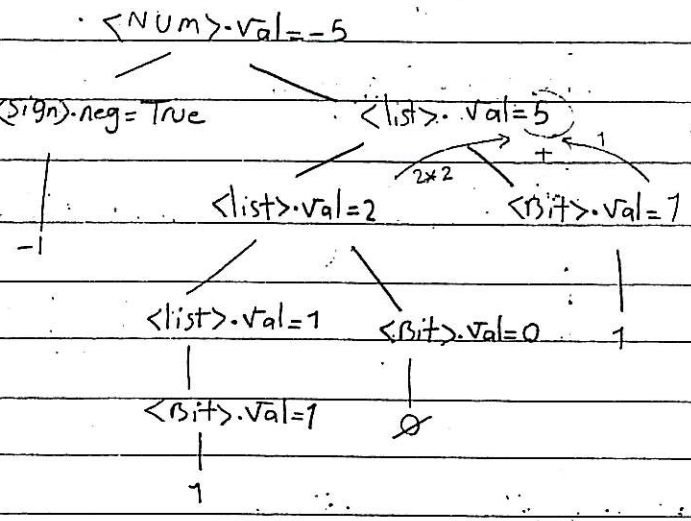
باید جایش ۲ بیت پایین

یک جایش بیت بالا مقدار val

را توانستم محاسبه کنیم سپس امنت انرس value برای مقایسه با Domain آرایه ی خود

<NUM> ::= <sign> <list>	<list>.pos ← 0 <NUM>.val ← if <sign>.neg then -<list>.val else <list>.val
<Sign> ::= -	<sign>.neg = True
<Sign> ::= +	<sign>.neg = False
<list> ::= <Bit>	<list>.val ← <Bit>.val

$\langle list_0 \rangle ::= \langle list_1 \rangle \langle Bit \rangle$	$\langle list_0 \rangle.val \leftarrow \langle list_1 \rangle.val * 2 + \langle Bit \rangle.val$
$\langle Bit \rangle ::= 0$	$\langle Bit \rangle.val \leftarrow 0$
$\langle Bit \rangle ::= 1$	$\langle Bit \rangle.val \leftarrow 1$



باید Pass در من Syntax A ساختاری سپری شود.