Subject:
Year. ۹۰ Month. ۷ Date. ۶ ( )

اصول طراحی کامپایلر — چهارشنبه

جلسه اول — ۱۵:۴۵ - ۱۸:۰۰ — سلیمانی تبار

تئوریه زبانها و ماشینها ← کامپایلر ← طراحی پیاده سازی زبانها، برنامه سازی

اصول طراحی کامپایلر

مولف: پورامین

Computer + هوش { هوش مصنوعی، سیستمهای خبره، شبکه های عصبی، منطق فازی، ... } dead lock

محاسبه

تئوریه ی محاسبات: مباحث خیلی پیشرفته تری از تئوریه زبانها

تورینگ: Turing ← تورینگ: تورینگ بیان می کند که من تئوری ای را بیان می کنم که اگر روی ماشین آن ساخته شود ماشین آن یک ماشین الکترومکانیکی است ولی این ماشین قادر خواهد بود تمام مسایل را حل کند. از جمله آن مثلا ما برنامه ای بنویسیم که بتواند برای ما برنامه ای بنویسد، یا مثلا برنامه ای بنویسیم که قضایای ریاضی را برای ما اثبات کند. نمی شود برنامه ای نوشت که خودش یک ماشین تورینگ دیگر را هک کند، نمی شود برنامه ای نوشت که این برنامه ضد هک شود. اینها هم بدان معناست که آخر یک ماشین تورینگ یک dead lock است. یعنی هیچگاه کامپیوتر نمی تواند به طور صد درصد مسایل هوش مصنوعی را detect کند. و کامپیوترها هیچگاه قوی تر از ماشین الکترومکانیکی ماشین تورینگ نخواهند بود و چون turing machine هیچگاه هوشمند نمی شود پس کامپیوترها امروز هیچگاه هوشمند نخواهند شد.

Subject:
Year. Month. Date. ( )

مسائلی که انسان با آن سر و کار دارد دو دسته اند

مسائل { Problem: قبلا حل شده اند و یا حل شده اند یا اصلا حل نمی شوند
Question: اصلا حل نشده اند و هنوز نمی دانیم که حل می شود یا حل نمی شود.

Problem { decidable, undecidable

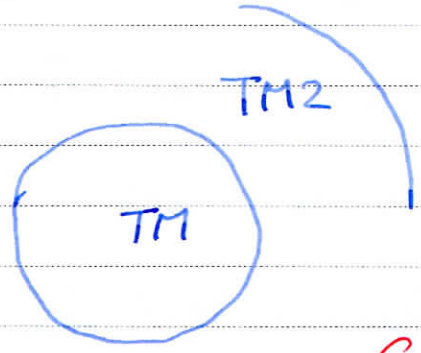
Problem قبلا حل شده اند و یکبار دیگر می خواهیم آنرا حل کنیم. یا می دانیم حل نمی شوند یا حل می شود.

اصول مسایلی که انسان حل می کند دو دسته اند (راه حل دو دسته اند)

Turing Machine ← دسته ۱: راه حلهایی که قدم پذیرند: ۱، ۲، ۳، ...

دسته ۲: راه حلهایی که ترتیب پذیر نیستند. ماشین تورینگ کارهایی که ترتیب پذیر نباشند اصلا انجام نمی دهد. مانند تجربه ای که یک متخصص به دست می آورد و قابل انتقال نیست.

Turing Machine می تواند مسایلی را که قدم پذیر هستند را حل کند. اگر ماشین دومی بتواند مسایل خارج از فضای TM را حل کند آنگاه باز مسایلی خواهد بود که این ماشین دوم نیز نمی تواند آنرا حل کند و الی آخر



* نظریه پیچیدگی: ارشد علوم کامپیوتر

بررسی زبانها و ماشینها

Computer Technology

Finite Automata | DPDA | NPDA | LBA | Turing Machine | Undecidable / Question

Finite Auto mata

NFA ≡ DFA

این ماشین محدود است ولی زبانی که این ماشین تولید/قبول

Subject :
Year . Month . Date . ( )

می کند نامحدود است . دلیل آنکه ماشین محدود است آنست که حافظه در این ماشین محدود است

DPDA : Deterministic Pushdown Automata

یک پشته بعنوان حافظه برای پذیرش زبان دارد که نامحدود است ولی برای دستیابی به عنصری که در پشته قرار می گیرد باید تمام عناصر بالای آنرا بخوانیم .

NPDA: Non deterministic Push Down Automata

می توان NPDA را توسط چند پشته با هم ایجاد کرد که قدرت آن از DPDA بیشتر است .

NPDA ≠ DPDA

LBA : Linear Bounded Automata

از دو طرف باز است ولی محدود است .

TM: Turing Machine

محدودیت ندارد

Undecideable

تصمیم ناپذیرها : می دانیم که نمی توانیم آنرا حل کنیم

Question

اصلا نمی دانیم می توانیم آنرا حل کنیم یا نه !

* مسایلی که می توانیم با کامپیوتر حل کنیم deterministic هستند بعضی از مسایل nondeterministic هستند .

Subject:
Year. Month. Date. ( )

* کامپیوترها از نظر تکنولوژی DPDA هستند دلیل آن این است که حافظه کامپیوتر محدود است ولی از نظر تئوری تا حد ماشین تورینگ است.

تعریف کامپایلر: کامپایلر یک DPDA غیرمبهم است. برای مثال:

int a, b[10]; این جمله به تنهایی درست است.

a = a+b; این جمله نیز به تنهایی درست است.

ولی هر دو با هم غلط هستند.

* کامپایلر نمی‌تواند کلیه error های یک برنامه را نشان دهد.

تحلیل زبانها: هم خطاهایی که مربوط به تحلیل زبانها هستند را می‌تواند تشخیص دهد.

طراحی و پیاده سازی زبانها: هم خطاهایی که مربوط به طراحی و پیاده سازی هستند را می‌تواند تشخیص دهد.

تعریف کامپایلر:

تعریف عام پسند: یک مترجم است که یک برنامه به زبان سطح بالا را به یک برنامه به زبان سطح پایین (ماشین) ترجمه می‌کند.

h.l.l → [Compiler] → l.l.l

C++	زبان اسمبلی 8086	مخصوصیت ماشین (معماری)
java	زبان اسمبلی 8085	
delphi	زبان اسمبلی IBM 370	
pascal	زبان اسمبلی Sun	
	زبان اسمبلی RISC	

Subject :
Year . Month . Date . ( )

تعریف علمی تر: کامپایلر یک Source Program را به یک Target Program تبدیل می کند

Source Program → Compiler → Target Program

فارسی ∉ مگر زبانگ زرا بچمک

فارسی ∈ محدوده پاک کن را بخور

if a>b then a:=a+b; ∉ C ∈ pascal

if if> then then else := else+if; ∉ pascal ∈ pl/1

```
int a, cin;
cin =1 ; a=1;
cin =cin +a;
cout << cin;
```
∈ C++

h.l.l → [Compiler] → l.l.l

frontend | backend

وابسته به زبان مبدا | وابسته به زبان مقصد

اگر m تا زبان و n تا معماری متفاوت داشته باشیم به mn تا کامپایلر نیاز است.

تقسیم f.e , b.e کمک می کند به جای هزینه O(mn) با هزینه O(m+n) به mn تا کامپایلر برسیم.

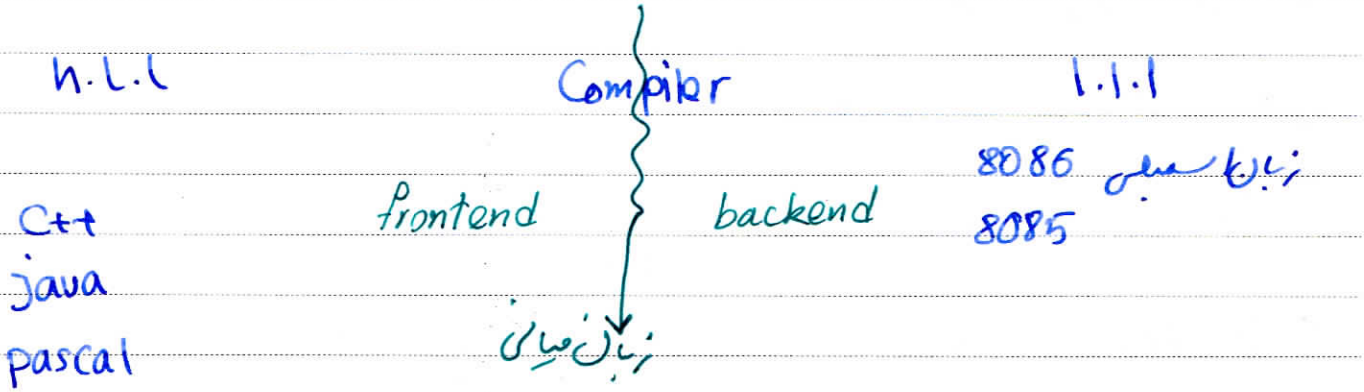
Subject:
Year. ۹۰ Month. ۷ Date. ۱۳ ( )

چهارشنبه

اصول طراحی کامپایلر

جلسه دوم ۱۵:۴۵ - ۱۸:۰۰ استاد سلیمانی تبار

frontend : فازهایی از یک کامپایلر است که وابسته به زبان مبدا (درایتی زبان سطح بالا) می باشد. در حقیقت خواص زبان مبدا بررسی می گردد.



مزیت تقسیم بندی frontend و backend در چیست؟

اگر m تا زبان و n تا ماشین (معماری) داشته باشیم واضح است که به mn تا کامپایلر خاص نیاز است با توجه به عدم وابستگی طراحی frontend و backend به یکدیگر می توان برای هر زبان یک frontend طراحی و برای هر ماشین برای داشتن یک کامپایلر خاص کافی است frontend زبان مربوطه را متوالی با backend معماری مربوطه لینک کنیم. با این تغییر هزینه ساخت mn تا کامپایلر برابر $O(m+n)$ به جای $O(m \cdot n)$ خواهد بود.

if sum > avg then sum := sum+1

token ها شناسه ها علامتهایی هستند که برای

لغت (شناسه) token : کوچکترین جز معنی دار در یک برنامه نوشته شده به یک زبان را گویند. برای مثال:

```
if  sum > AVE  then  SUM:=SUM+1;
```

token ۱۱ در زبان پاسکال

```
if (sum > ave)  sum += sum+1;
```

token ۱۲ در C

token ۱۳ در pascal

```
if  a=>)=<==b  then  a:=a+);
```

token ۱۵ در پاسکال

```
>  a>=b
```

توجه: برای token بندی source program کافی است کاراکتر به کاراکتر از سمت چپ در دری کاراکترها را باز کنیم تا به یک جدا کننده برسیم. رشته ی بازشده یک token خواهد بود.

جداکننده ها می توانند هریک از موارد زیر باشند.

; , [ ] ( ) + * - /b \t \n ...

blank tab newline

مثال: کاراکترهایی که نقش جداکننده را دارند در جمله زیر به زبان پاسکال را پیدا کنید:

```
if a=>)=== a sum then a:=a+b;
```

با استفاده از مفهوم جداکننده ها در یک فوق می توانیم بگوییم یک token به یکی از دو طریق زیر مشخص می شود:

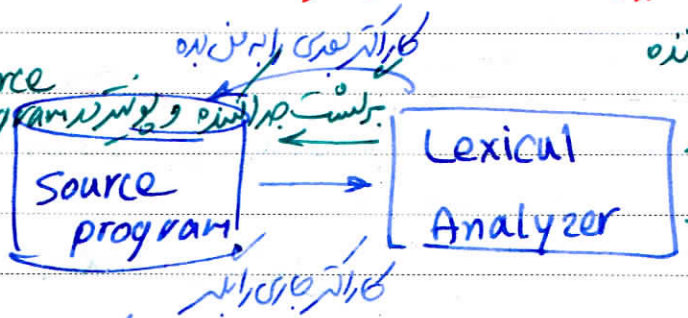
Subject:
Year. Month. Date. ( )

الف) توسط جداکننده ها

if (a <= x) === a + sum then a := a+b; به صورت token ها در

ب) دنباله‌ای از کاراکترها به ذات خود یک token است.

اولین فاز یک کامپایلر: فاز تحلیل لغوی (تحلیلگر لغوی) (Lexical Analyzer



هر بار یک کاراکتر را از source program می‌خواند این چرخه ادامه پیدا می‌کند تا به یک جداکننده L.A

برسد در این صورت یا Token جاری شناسایی شده یا یک lexical error

صادر می‌شود. و کاراکتر جداکننده به source program برگشت داده می‌شود.

توجه: در هر بار فراخوانی Lexical Analyzer ابتدا کلیه کاراکترهای blank tab و enter را رد می‌کند یعنی خوانده و دور می‌ریزد (blank هیچی نه خانواده) new line (صفحه جدید) و

* وظیفه این تحلیلگر Lexical Error تبدیل token به source program می‌باشد.

چنانچه یک کاراکتر غیرمجاز، کاراکتری که جزء الفبای زبان نیست در source program به کار رود Lexical Analyzer باید آن وقوع یک خطای Lexical را اعلام می‌کند.

if sum = b@ +a;

Lexical Error

Subject:
Year. Month. Date. ( )

فاز دوم کامپایلر: تحلیل نحوی تحلیلگر نحوی (Syntax Analyzer)

نحو : عربی
انگلیسی : grammar

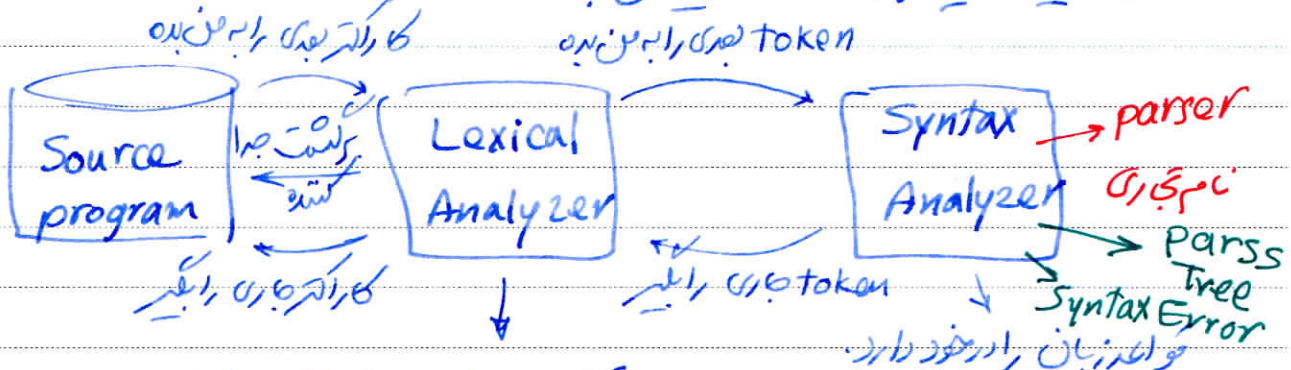
مثال: `if a>= then a:= a+*;`

۱۰ token در پاسکال داریم.

وظیفه این تحلیلگر بررسی درستی ترتیب قرارگرفتن token ها در کنار یکدیگر براساس قواعد زبان مربوطه می باشد. مثلا الان براساس قاعده‌های زبان پاسکال قبل از Then یک identifier یا افتاده و * اضافی است. و

گرامر زبان به ما می گوید که جای یک token درست است یا نه یعنی الان این token اضافی است یا جا افتاده است برای همین 50٪ می تواند خطا را نشان دهد.

معماری این تحلیلگر به صورت زیر می باشد.



قواعد مربوط به زبان نداره، قواعد token را دارد.
یک Finite Automata برای آن کافی است.

در یک کامپایلر Syntax Analyzer است.

Subject:
Year. Month. Date. ( )

توکن‌ها از Lexical Analyzer به Syntax Analyzer فرستاده می‌شود و خوانده می‌شود.

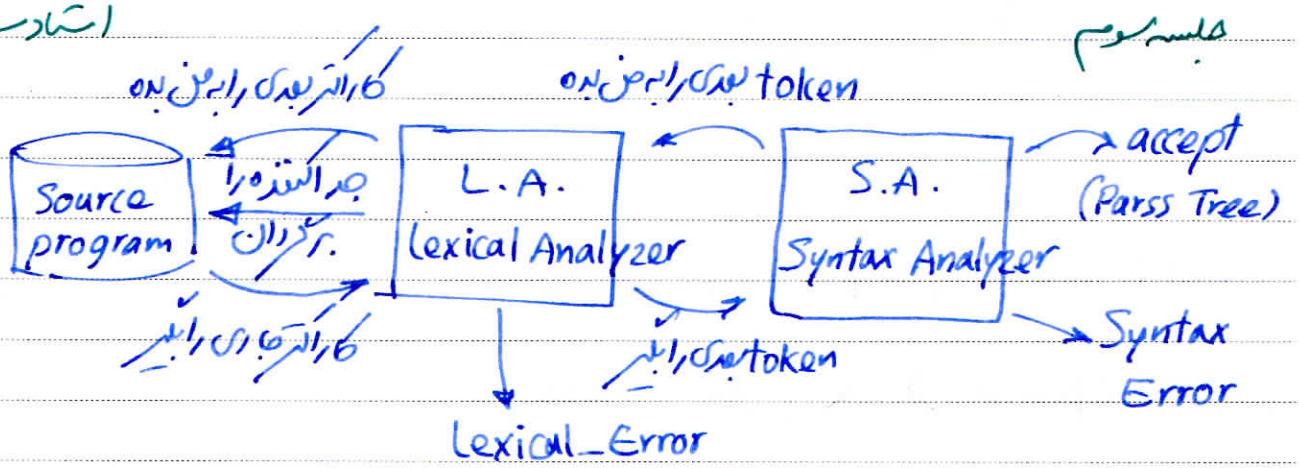
خروجی Syntax Analyzer یا Parss Tree است یا Syntax Error.

Parss Tree به معنای آن است که از توالی گرامری می‌باشد و در صورت خروجی accept می‌شود.

اشکالی نداشته و اصطلاحاً گفته می‌شود که

این Parss Tree مشابه درخت اشتقاق می‌باشد.

* نام دیگری parser (یا تجزیه‌کننده) Syntax Analyzer می‌باشد



توجه: انتهای Source Program ، token ویژه‌ای بنام end of file

وجود دارد که در این درس آنرا بنام $ معرفی می‌کنیم.

هر زبان برنامه‌نویسی یک سری قواعد تولید برای Source Program های خود دارد.

مثال: می‌خواهیم با استفاده از مجموعه قواعد زیر که مربوط به زبان پاسکال می‌باشد

جمله‌ای از یک برنامه به زبان پاسکال را توسط Syntax Analyzer تشخیص کرده و درخت

تجزیه‌ی آنرا رسم کنیم.

⋮

block → begin stmts end

⋮

stmts → if_stmts | while_stmts | ... | assignment_stmts

if_stmts → if bool_expr then stmts

if_stmts → if bool_expr then stmts else stmts

bool_expr → bool_expr boolOp bool_expr

bool_expr → bool_op bool_expr ⟶ not مثل

bool_expr → (bool_expr)

Subject:
Year. Month. Date. ( )

bool_expr → expr

expr → expr arithmatic_op expr | (expr) | id

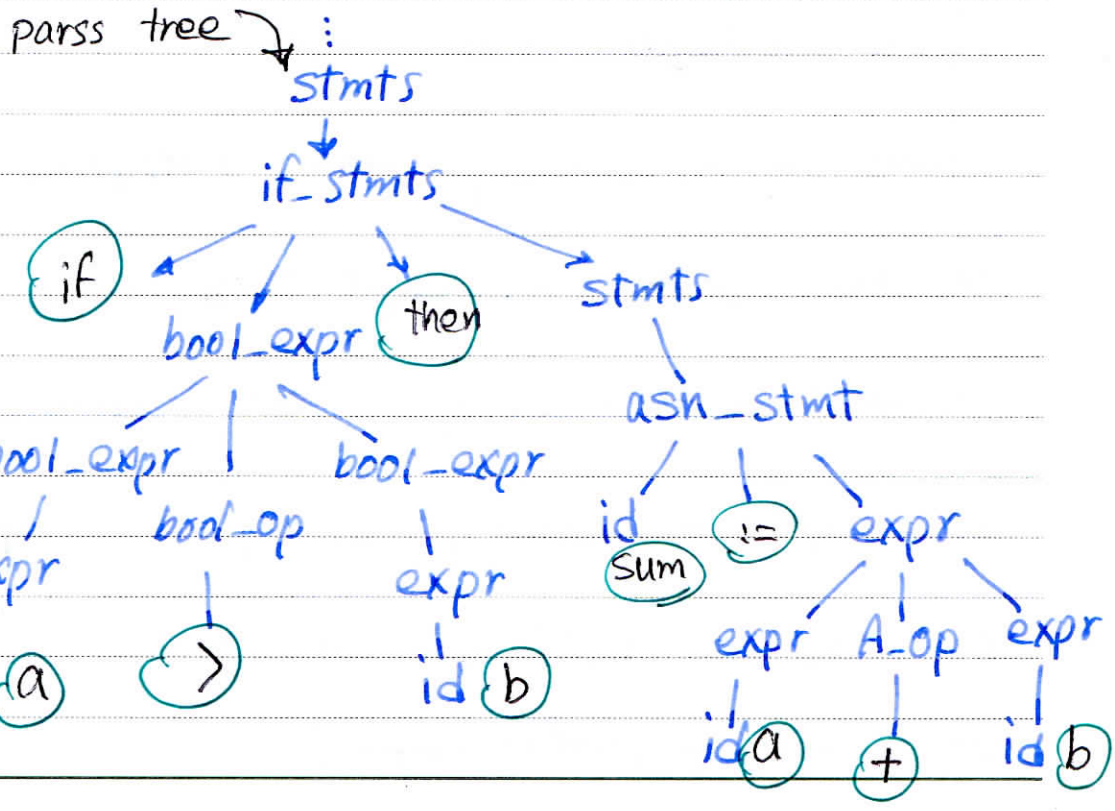
A_op → + | - | * | / | ... مثل‌ها token هستند

bool_op → > | < | >= | AND | ...

assignment_stmt → id := expr

مثال: می‌خواهیم ببینیم جمله زیر را که در واقع source program ما را تولید کننده شده در بالا می‌تواند تولید کند یا خیر؟ اگر بتواند تولید کند یعنی SyntaxAnalyzer توانسته یک parss tree بسازد اگر بتوانیم parss tree آن را بسازیم یعنی جمله‌ی ما این خطا را دارد.

Source Program: if a>b then sum:=a+b



Subject:
Year. Month. Date. ( )

اگر ما برگهای این درخت را از چپ به راست ببینیم در واقع داریم این source program را می‌بینیم. به این درخت Parse tree گفته می‌شود. بعبارت دیگر پیمایش inorder این درخت همین source program به ما خواهد داد.

ما توانستیم این source program را توسط قواعد این زبان pars کنیم. اینکه ما بتوانیم کدام قاعده را استفاده کنیم ما از کامپایلر، جدول را خواهیم ساخت که در مبحث تحلیلگر نحوی بحث خواهد شد. این جدول به کامپایلر خواهد گفت کدام قانون را استفاده کند.

استاد: Source Program توسط کامپایلر مربوط تجزیه شد و درخت تجزیه بدون خطا ساخته شد پس Source Program داده شده خطای نحوی ندارد.

فاز سوم: تحلیلگر معنایی Se.A Sematic Analyzer

در فاز تحلیل نحوی ترتیب قرار گرفتن Token های source program با استفاده از قوانین تولید کامپایلر بررسی می‌شود و تنها نوع Token بخاطر می‌شود.

ولی هر token علاوه بر نوع و نام خود دارای ویژگیهای بسیاری می‌باشد که می‌تواند در صحت جملات نقش داشته باشد.

مثال: دو جمله‌ی زیر هر کدام به تنهایی صحیح می‌باشند اما این دو جمله در کنار یکدیگر یک خطای type mismatch (عدم تطابق type) را صادر می‌کند.

```
int a, b[10];      جمله‌ی اول
a = a + b;         جمله‌ی دوم.
```

Subject:
Year. Month. Date. ( )

صفات و ویژگیهای token

نام token	نوع token	type	scope	...
a	id	scaller integer		
b	id	array integer		

در Syntax Analyzer فقط جمله از نظر نوع و نام token بررسی می‌شود ولی در Sematic Analyzer از نظر Type و scope و ... نیز بررسی می‌شود.

در فاز سمانتیک کجاها به طراح و پیاده‌ساز زبانهای برنامه‌نویسی برمی‌گردد و در یک Syntax کاری به مباحث نظیر زبانها برمی‌گردد. مباحثی مانند ارث بری، scope ها، type mismatch ها، type checking ها به Sematic Analyzer برمی‌گردد.

توجه: خطای لغوی (lexical) برای یک token رخ می‌دهد، خطای syntax برای یک جمله رخ می‌دهد و خطای معنایی در نظر گرفتن ویژگیها و معناهای تعریف شده چندین جمله را بررسی می‌کند. اگر خطا از منتج شدن معنای جمات دیگر باشد به آن خطای معنایی* نقدی می‌شود

سه فاز تحلیل لغوی، تحلیل نحوی و تحلیل معنایی به همراه فاز بعدی (تولید کننده میانی) یک کامپایلر frontend را تشکیل می‌دهند که این سه فاز خواص و ویژگیهای زبان را در سطح لغت (token)، جمله و کل برنامه بررسی می‌کند.

فاز چهارم: تولید کننده میانی Intermediate Code Generator ICG

از روی درخت تجزیه معنا طار مرحله قبل برنامه‌ای به زبان میانی تولید می‌شود که جهت

* خطای معنایی مانند صدا زدن تابع با پارامترهای متفاوت، type mismatch، خارج شدن از گستره آرایه

آن ویژگی زیر را دارند:

الف) جملات ساده‌ی انتساب می‌باشند ب) جملات شرطی و غیرشرطی پرش و فراخوانی توابع یا زیر برنامه‌ها

مثال: می‌خواهیم جملات سطح بالای زیر را که از لحاظ semantic , syntax صحیح می‌باشد به جملات زبان میانی تبدیل کنیم:

sum := a + b * c for pascal

sum = a+b * c for C

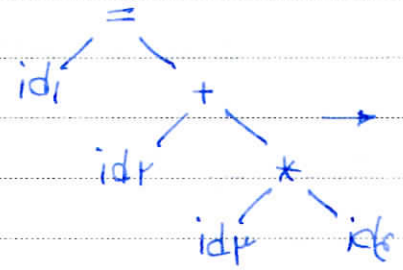
$id_1 := id_2 + id_3 * id_5$

بعد از lexical تبدیل token ها →

$id_1 = id_2 + id_3 * id_5$

→ S.A.
Syntax Analyzer →

pars tree



→ I.C.G
Intermediate Code Generator

↓

یک کدمیانی که مستقل از زبان است.

$t_1 \leftarrow id_4$
$t_2 \leftarrow id_3$
$t_3 \leftarrow id_2$
$t_4 \leftarrow t_1 * t_2$
$t_5 \leftarrow t_3 + t_4$
$id_1 \leftarrow t_5$

Subject:
Year. Month. Date. ( )

مثال: می‌خواهیم سینتکس source code زیر، intermediate code به چه شکل ساخته می‌شود.

```
if a>b then sum:= a+b;
```

```
if id1>id2 then id3 := id1 + id2;
```

```
t1 ← id1
t2 ← id2
t3 ← t1-t2
JEL t3, L1 → Label
                شرط برای پرش
t4 ← t1+t2
id3 ← t4
L1 : .....
```

* نتیجه‌ی کمپایلر intermediate code generator برای C نیز همین خواهد بود.

فاز پنجم: بهینه‌سازی کد میانی ICO: Intermediate Code Optimizer

در این فاز برنامه به زبان کد میانی از لحاظ تعداد جهت و تعداد متغیرهای کلی بهینه می‌شود (minimum)

مثال:

```
sum := a+b;
```

```
t1 ← id4
t2 ← id3
t3 ← id2
t4 ← t1*t2
t5 ← t3*t4
id1 ← t5
```

تعداد متغیرهای کلی

Intermediate Code Generator →

```
t1 ← id4 * id3
id1 ← id2 + t1
```

توجه: بخش‌های تحلیل نحوی و تحلیل معنایی و تحلیل لغوی و تولید کد میانی در حوزه‌ی کامپایلر چندان جای تحقیق و پژوهش ندارد اما بحث بهینه‌سازی کد یکی از مباحث به روزی است.

فاز ششم: تولید کد ماشین M.C.G Machine Code Generator

Subject:
Year. Month. Date. ( )

براساس فرمت دستورات اسمبلی معماری ماشین از روی کد میانی بهینه شده جملات اسمبلی مربوطه به دست می آید.

```
sum := a+b*c;
```

... → Intermediate Code Optimizer → $t_1 \leftarrow id_4 * id_3$; $id_1 \leftarrow id_2 + t_1$ → M.C.G 80X86 →

```
MOV AX, id3
MUL AX, id4
ADD AX, id2
MOV id1, AX
```

* این چهار دستوراتی هستند که در اسمبلی 8086 قابل اجرا هستند.

ادامه در پایین...

فاز هفتم: M.C.O Machine Code Optimizer

فاز هشتم: تولید کننده کد باینری B.C.G Binary Code Generator

ادامه از بالا

Machine Code Optimizer → Binary Code Generator →

البته این کد بهینه است و از این بهینه تر نمی شود

کد عمل	آدرس AX	آدرس id3
mov		
0101	1110	111011
:		

توضیح linker و loader

می دانیم برنامه های سطح بالا برای اجرا به فایلهای کمکی زیادی برای اجرا نیاز مندند.

از جمله فایلهای dll سیستم عامل و دستگاههای جانبی. به واحدی که عمل پیوند کد فایلهای کمکی را برای تکمیل شدن کد برنامه و قابل اجرا بودن آنرا انجام می دهد لینکر یا پیوند دهنده گویند.

برنامه ی قابل اجرا (فرمی واحد لینکر) بایستی برروی حافظه بارشود تا بتواند توسط CPU اجرا گردد. به واحد بارکننده اصطلاحا loader می گویند.

Subject: اصول طراحی کامپایلر
Year. ۹۰ Month. ۸ Date. ۴ ( ) ۱۵:۴۵ ۱۸:۰۰ چهارشنبه

جلسه چهارم

استاد سلیمانی تبار

گرامرهای Context Free و زبانهای برنامه نویسی

تعریف گرامر: هر چهارتایی به فرم $G=(V_N, V_T, S, P)$: Context Free

یک Context Free Grammer گوییم به طوریکه V_N مجموعه‌ای محدودی از Noun Terminal ها، V_T مجموعه‌ی محدودی از Token های زبان $S \in V_N$ سمبل شروع گرامر و P مجموعه‌ی قوانین تولید گرامر

$$\alpha \rightarrow \alpha \quad / \quad \begin{matrix} a \in V_N \\ \alpha \in (V_N \cup V_T)^* \end{matrix}$$

تعریف درخت اشتقاق: درخت اشتقاق درختی است که ریشه‌ی آن سمبل شروع گرامر و گره‌های میانی درخت Noun Terminal های گرامر و برگهای درخت Token های گرامر به همراه ε می‌باشد، به طوریکه Source Program داده‌شده در برگهای درخت token به token از چپ به راست می‌آیند.

تعریف گرامر مبهم: گرامر Context Free بنام G را مبهم گوییم اگر و تنها اگر برای حداقل یک رشته متعلق به زبان گرامر $L(G)$ بیش از یک درخت اشتقاق وجود داشته باشد.

واضح است که تعداد رشته های یک گرامر Context Free Grammer در حالت کلی نامحدود است. نتیجه می‌گیریم مشخص کردن مبهم بودن یا نبودن یک گرامر امری حل‌شدنی نیست (تصمیم ناپذیر)

در کامپایلر ابزاری وجود دارد که با استفاده از آنها می‌توانیم گرامر را مورد آزمایش

Subject:
Year. Month. Date. ( )

قرار دهیم و اگر از آن ابزار مربوطه بیرون بیاید قطعا گرامر مبهم نیست! اگر سر بلند بیرون نیاید نمی توانیم در مورد مبهم بودن یا نبودن آن استدلال کنیم.

چنانچه یک گرامر مبهم باشد بایستی آنرا با روشهای محدودی رفع ابهام کنیم که معمولا زبان گرامر تغییری کند.

مثال: نمونه ای از گرامر زبان PLT برای تولید جملات if-then-else در زیر آمده است. باتوجه به Source Program داده شده نشان دهید این گرامر ابهام دارد. (بارسم درختهای اشتقاق)

Grammer:

stmts → if-stmt | asn-expr | ...

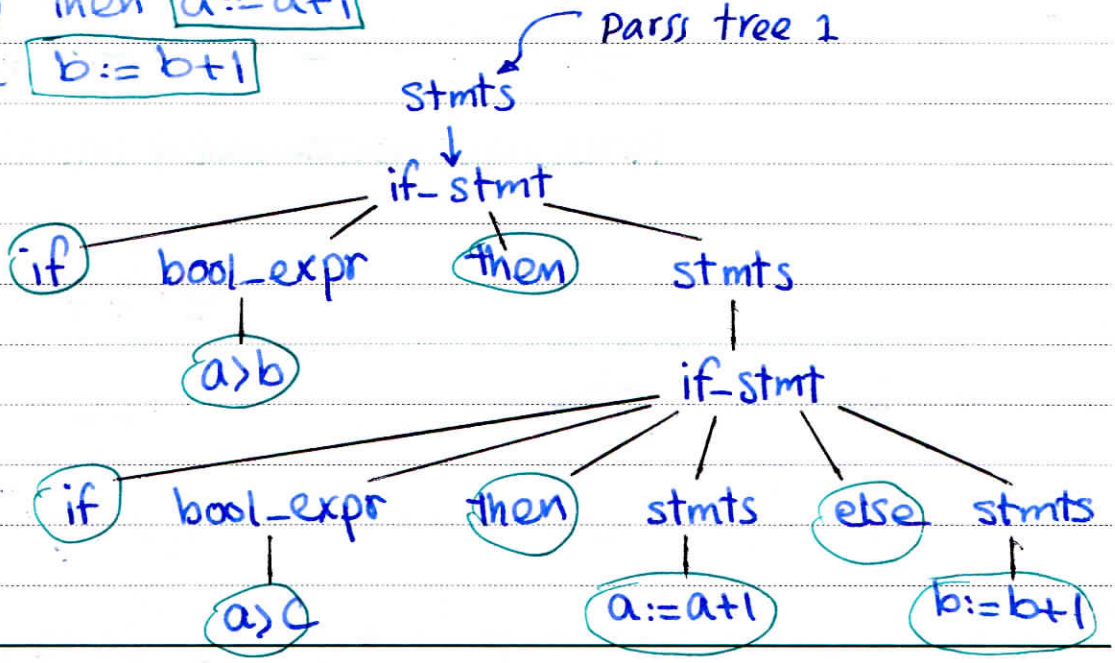
if-stmt → if bool-expr then stmts

if-stmt → if bool-expr then stmts else stmts

Source Program:

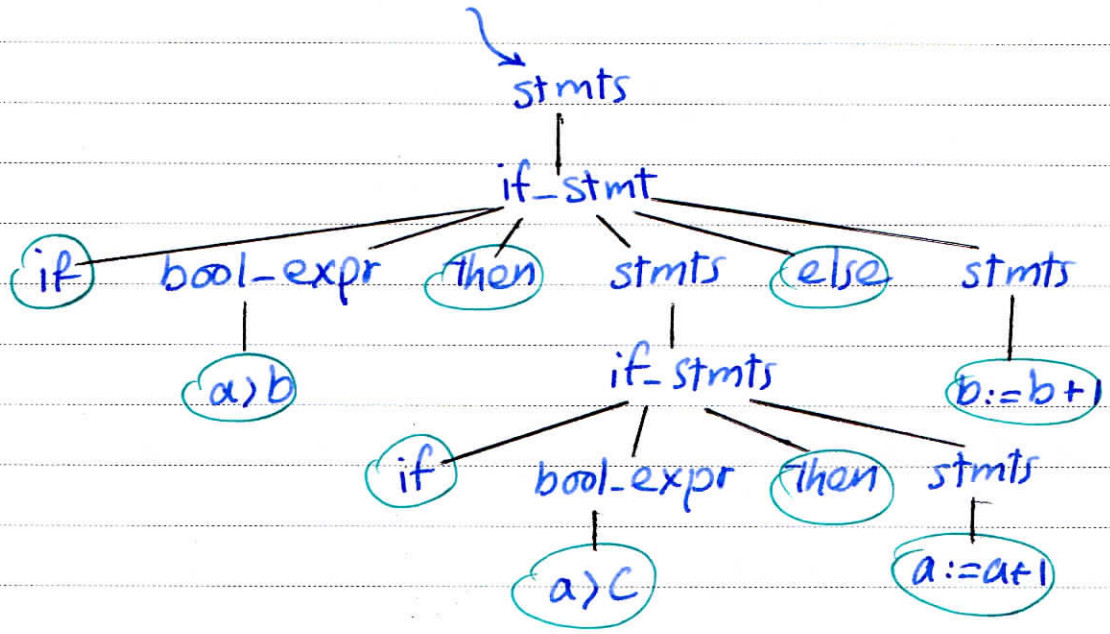
```
if (a>b) then
    if (a>c) then a:=a+1
    else b:=b+1
```

Parss tree 1



Subject:
Year. Month. Date. ( )

Parss tree 2



چنانچه با Parss Tree دوم ماشین دستور را اجرا کنیم درصورت عدم برقراری شرط a>b دستور b:=b+1 اجرا خواهد شد. درصورتیکه ما در منطق برنامه نویسی خودمان داریم همیشه else مربوط به نزدیکترین if می باشد ولذا منطق برنامه نویسی باید توسط Pars tree اول این دستور اجرا شود.

از رسم درختهای اشتقاق فوق، نتیجه گرفتیم که گرامر فوق ابهام دارد.

اشتقاق سمت چپ ترین و اشتقاق سمت راست ترین

RMD(right mostderivation) LMD (left most derivation)

تعریف اشتقاق سمت چپ ترین:

اگر در مراحل اشتقاق یک رشته از گرامر اولویت جایگزینی Noun Terminal های یک گرامر با سمت چپ ترین Noun Terminal های موجود باشد به اشتقاق حاصل اشتقاق سمت چپ ترین گوییم. (به طور مشابه اشتقاق سمت راست ترین تعریف می شود)

Subject:
Year. Month. Date. ( )

مثال: گرامر Context free زیر را در نظر گرفته و برای Source Program داده شده تمامی اشتقاق‌های سمت چپ‌ترین و تمامی اشتقاق‌های سمت راست‌ترین و تمامی درخت‌های اشتقاق را به دست آورید.

P_G: $E \to E+E$

$E \to E*E$

$E \to id$

Source Program:

$id_1 + id_2 * id_3$

LMD: Left most derivation

1: $E \to E+E \Rightarrow id_1 + E \to id_1 + E*E \to id_1 + id_2 * E \to$

$\to id_1 + id_2 * id_3$

Source Program

2: $E \to E*E \to E+E*E \to id_1 + E*E \to id_1 + id_2 * E \to$

$\to id_1 + id_2 * id_3$

Source Program

RMD: right most derivation

1: $E \to E+E \to E+E*E \to E+E*id_3 \to E+id_2*id_3 \to id_1+id_2*id_3$

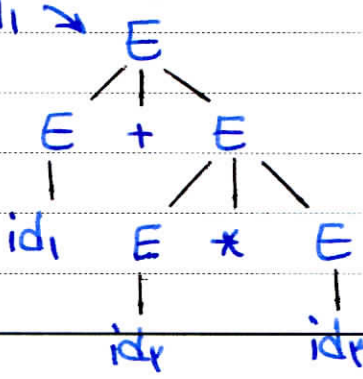
Source Program

2: $E \to E*E \to E*id_3 \to E+E*id_3 \to E+id_2*id_3 \to id_1+id_2*id_3$

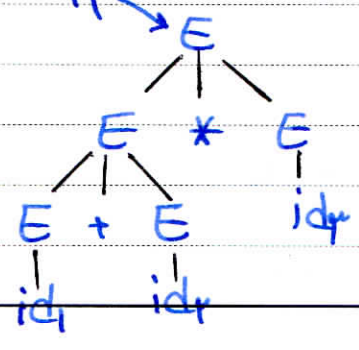
Source Program

Parss Tree

T_1



T_2



Subject:
Year. 9. Month. ۸ Date. ۱۸ ( )

جلسه پنجم

هر درخت اشتقاق هم درخت LMD دارد و هم RMD

تعریف کاملتری از گرامر مبهم:

گرامر Context Free Grammer را مبهم می نامیم اگر و تنها اگر برای حداقل یک رشته از زبان بیش از یک درخت اشتقاق (بیش از یک اشتقاق سمت چپ ترین) (بیش از یک اشتقاق سمت راست ترین) وجود داشته باشد.

گرامرهای بازگشتی از چپ و گرامرهای دارای فاکتور مشترک چپ.

تعریف گرامر بازگشتی از چپ: هر گرامری که حداقل یک قانون بازگشتی از چپ داشته باشد.

◄ تعریف قانون بازگشتی از چپ: هر قانون به فرم $A \to A\alpha$ را یک قانون بازگشتی از چپ گویند.

واضح است که برای قانون بازگشتی از چپ $A \to A\alpha$ حتماً می بایست قانونی به فرم $A \to \beta$ وجود داشته باشد و β بتواند رشته ای از Token ها را تولید کند تا بالطبع A، رشته ای از Token ها را تولید کند، در غیر اینصورت $A \to A\alpha$ قانونی useless خواهد بود.

مشکل یک قانون بازگشتی از چپ: برای تولید یک رشته نمی دانیم چند بار بایستی $A \to A\alpha$ را به کار ببریم.

مثل:

$P_G : A \to Ab$

$A \to a$