Subject: اصول طراحی کامپایلر (برنامه‌سازی)

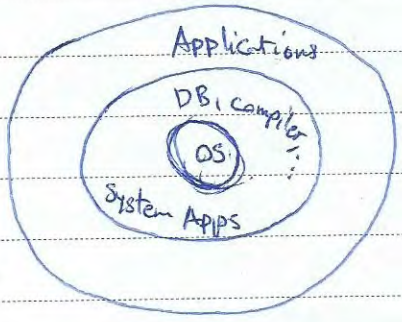
Year. ۹۴ Month. ۷ Day. ۲۲

بسم خدا

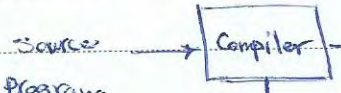
جزوه ۸ نمره

Compilers, principles, Techniques and tools

Aho, Sethi, Ullman, Lin (Dragon Book)



محدودیت های نحوه ← Time / Space



زبان سطح بالا: C, Java, Fortran

زبان اسمبلی: Assembly

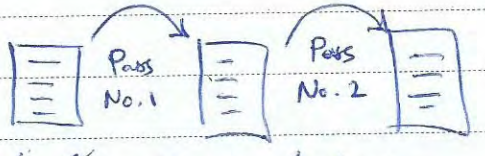
Formula Translation ← اولین زبان برنامه نویسی سطح بالا

کامپایلر ساخته شده برای Fortran ، ۱۸ نفر سال کار برد.

Pascal ← آخرین زبان برنامه نویسی سطح بالا ، C ← medium-level language

انواع کامپایلر:

- Single Pass → برنامه در یک Pass ترجمه می‌شود
- multi pass → هر بار خواندن کامل یک فایل و تولید فایل دیگر را یک Pass می‌گویند.
- load & go
- debugging
- optimizing → برنامه از یک Pass یا چند Pass بهینه تولید می‌شود.



Subject:
Year. Month. Day.

load & go → سریعترین کامپایلر برای استفاده مبتدی، سرعت بالا و بررسی دقیق خطاها برای عیب‌یابی

debugging → کامپایلر حساس، گسترده، تشخیص بهتر خطاها و درست کردن آن

optimizing → (target) کد تولید شده (برنامه‌نویسی از نظر سرعت و حافظه) + به مصرف کننده گسترده دارد

Source Prog — Compiler — Target Prog.

Source Prog. — Interpreter → مفسر کد منبع را می‌گیرد و اجرا می‌کند.

مزیت‌های مفسر: استقلال کد منبع برای اجرا شدن به Source Prog دارد و بررسی کامل دسترسی است.

وقتی برنامه‌ها کوچک هستند می‌توان برنامه منبع را در هر بار اجرا دوباره کد کامپایل شده

Portability مهم‌تر است، زیرا زبان‌های کامپایل شده (برنامه منبع) تولید شده توسط کامپایلر، برای هر ماشین خصوصی است و برای هر ماشین باید کامپایل شود. پس مفسر برنامه را اجرا کند.

زبان جاوا استفاده می‌کند (بایت کد) Source → byte code → JVM → input

کامپایلر داخل آن به سورس بایت کد و خود برنامه کامپایلر است سورس باید برای هر ماشین جدا کامپایل شود.

نتیجه: برنامه‌ها در هر ماشین نیاز به ترجمه مجدد خواهد بود.

زبان جاوا کد میانی به همراه کد برنامه (input) به Virtual Machine داده می‌شود و برنامه اجرا می‌شود، پس از این تغییر،

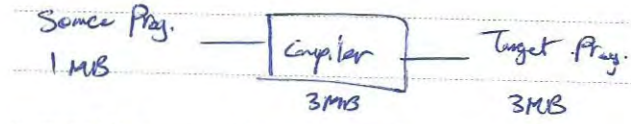
Subject
Date

Portable هست و این به خاطر JVM واسطه بین کد و سیستم است.

پس کامپایلر جاوا بهتر است یا مفسر برای Portability؟

زیرا حافظه:

خود کامپایلر حدود 3MB است؛ همیشه هر سورس کدی که کامپایل می‌شود



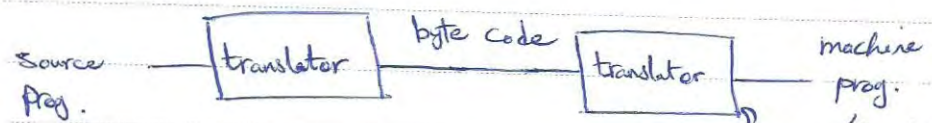
Interpreter: Source Prog. + 3MB

Compiler: Source Prog. * 3

فرق برنامه اجرا شده با برنامه کامپایل شده: برنامه کامپایل شده دیگر نیاز به کامپایلر ندارد و خودش اجرا می‌شود.

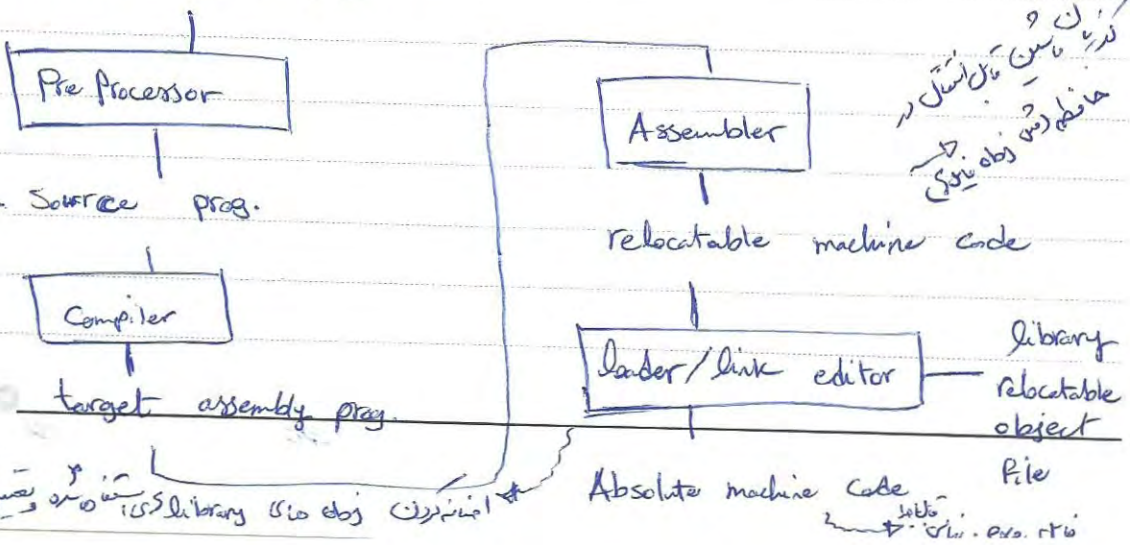
برای چنین برنامه‌هایی کامپایلر باید همراه برنامه باشد تا هر بار اجرا شد، کامپایل شود.

Just_in_time Compiler:



روند اجرای یک برنامه می‌تواند به شکل زیر باشد:

مثلا برای برنامه‌ای که include دارد نیاز است.



Subject
Date

کامپایلر به طور کلی از دو قسمت تشکیل شده است: Synthesis & Analysis

مراحل Analysis به طور کلی:

1) Linear Analysis (lexical Analysis, Scanning) تحلیل لغوی

→ Scanner یا liner Analyzer

در این مرحله این کار را انجام می‌دهد.

کاری که Scanner انجام می‌دهد این است:

لیست کاراکترها لغوی جدا می‌شود.

مثال در زبان طبیعی:

علی سیب را خورد

اسم | اسم | حرف اضافه | فعل

گروه اسمی | گروه اسمی | فعل

گروه فعلی

Position := initial + rate * 60

token: id

id | assignment symbol | id | plus sign | num | mult sign

حال token ها به Parser می‌رود.

2) Syntax Analysis (hierarchical analysis, Parsing) تحلیل نحوی (تجزیه)

→ از کاراکترها برای جدا کردن لغات استفاده می‌شود

این کار را Parser انجام می‌دهد

مثال:

Parse tree →

Assignment statement

id | := | exp

Position

exp | + | exp

id | exp | * | exp

initial | id | num

rate | 60

خروجی سمت چپ ← غلط

علی سیب را خورد ← درست

ریشه هر زیردرخت، سمت چپ یکی از قواعد است و فرزندان آن سمت راست آن قاعده است (گام‌ها)

مثال: exp → exp + exp

Subject
Date

تحلیل معنایی

3) Semantic Analysis

سپس علی را خورد.

↓

جمله از نظر Syntax درست است.

ولی از لحاظ معنایی و مفهوم اشکال دارد.

از لحاظ معنا شناختی بررسی می‌شود که آیا این عبارت از نظر معنایی

مجاز است یا نه.

برای "سیب" یک type تعریف می‌کنیم

که نشان می‌دهد آدم نمی‌تواند بخورد.

بعضی از این کنترل‌ها در زمان اجرا و بعضی در زمان کامپایل انجام می‌شود.

طرح درس:

فصول ۱، ۳، ۴، ۷ و ۸

(البته ویرایش اول کتاب)

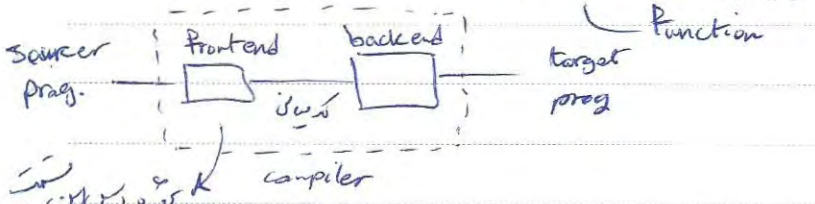
۸ نمره میان‌ترم
۸ نمره پایان‌ترم
} → ۶۰٪ نمره کل

۸ نمره تکه (آزمون)

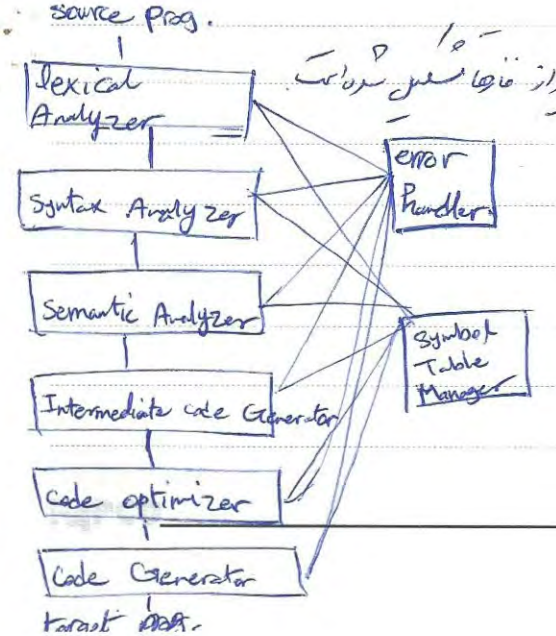
۲-۳ lex
۳-۲ yacc
۴ کدنویسی

زبان‌گرایی ۳

expression
statement
Function



کدنویسی ۴ درس این است



بخش‌های کامپایلر را فاز یا phase می‌گوییم. کامپایلر از فازها تشکیل شده است.

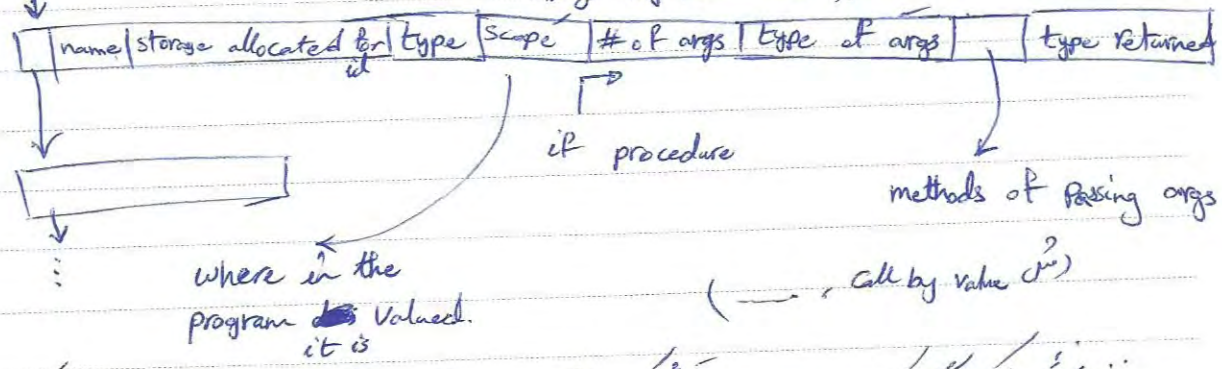
وظیفه: اطلاعات درست آوردن

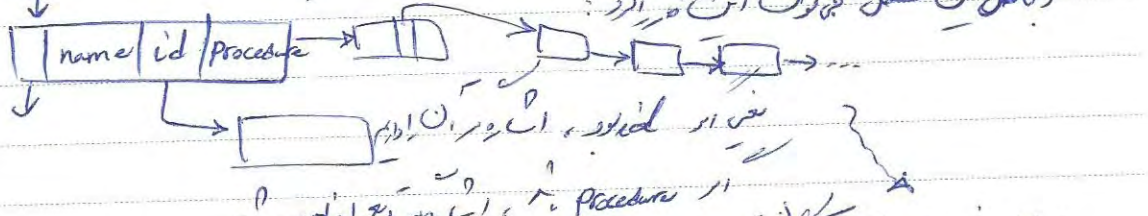
Source prog. و اطلاعات

تولید کد را بررسی می‌کند.

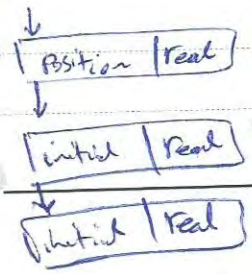
(برای تبادل اطلاعات)

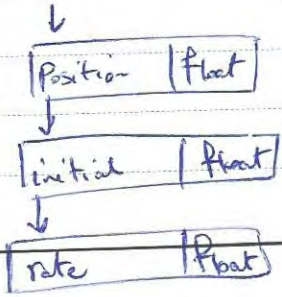
Subject
Date






```
var  position, initial, rate : real;
ftbat, position, initial, rate;
```





Subject
Date

Error production and Reporting:

از جمله وظایف کامپایلر، گزارش خطا و بازیابی در صورت بروز آن است.

انواع بازیابی در خطاها:

۱) در مرحله lexical Analyzer، با کمک Symbol Table و token ها، خطاهای املایی را گزارش می‌دهد.

Parser - وظیفه‌ی بررسی Syntax ها را دارد و در صورت بروز Syntax error آن را گزارش می‌دهد.

Semantic Analyzer - type mismatch را بررسی می‌کند.

Runtime error و logical error - مثل تقسیم بر صفر

$\left(y=0;\ \frac{x}{y};\right)$

Position := initial + rate * 60

↓ lexical analyzer

id := id + id * 60

↓ Syntax analyzer

```
      :=
   id     +
        id    *
            id   60
```

↓ Semantic analyzer

```
      :=
   id     +
        id    *
            id   int to real
                      |
                     60
```

↓ Intermediate Code Generator

temp1 := int to real (60)

temp2 := id3 * temp1

temp3 := id2 * temp2

id1 := temp3

Syntax Tree

three-address code

l: x := y or z

Subject
Date

از معنایی

Code optimizer

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Code Generator

```
LDF R2, id3
MULF R2, R1, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

Symbol Table

1	Position
2	initial
3	rate

token → شماره ثبت عبارت در جدول نمادها
lexeme → ثبت initial
lexical value → ارجاع به عبارت در Symbol Table (ارزش لغوی)

مثلا برای initial، ارزش لغوی 2 دارد

در Syntax Tree، برگ‌های داخلی عبارت‌ها هیچ مقدار ندارند و در برگ‌های خارجی عملوندها قرار دارند.

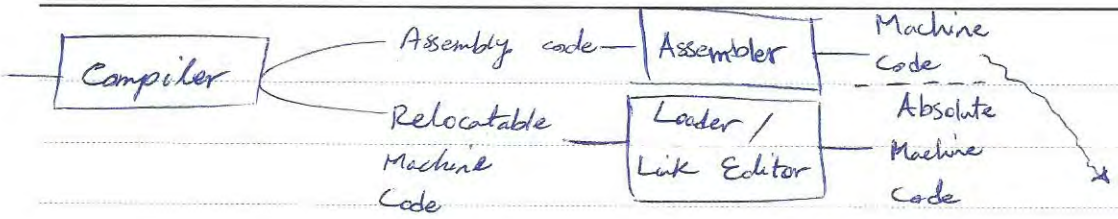
یکی از مشکلات در تولید کد، محدودیت رجیسترها است.

کدهایی که ماشین هدف پس می‌دهد، تولید کدهای اولیه کامپایلر انجام دادن و روی ماشین مقصد قرار دارد.

می‌توانست در خروجی تولید کدهای بهینه‌تری تولید کند ولی تعداد کار زیاد می‌شد و به خاطر این هدف اصلی کامپایلر نیست.

این کار را انجام دهد (بستگی دارد)

Subject
Date



اجرای برنامه

Relocatable خروجی اسمبلر: آدرس‌ها را تعیین نکرده و به صورت Absolute خواهد بود

مثال:

آدرس‌دهی نسبی (pass اول)

آدرس‌دهی مطلق

```
(b = a + 2)
MOV   a, R1
ADD   #2, R1
MOV   R1, b
```

a ۰
b ۴

op code | Register | آدرس | آدرس یا عدد

load 0000 | 01 | 00 | 0000 0000 *

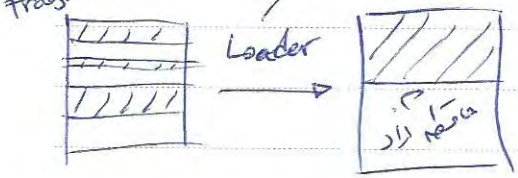
add 0001 | 01 | 00 | 0000 0000 *

store 0010 | 01 | 00 | 0000 0000 *

چون آدرس‌های a و b را نمی‌دانیم، فقط جای آن را خالی می‌گذاریم

همان جا را برای ذخیره‌ی آدرس ثابت (متغیر) نگه می‌داریم

فایل خروجی اسمبلر Relocatable Machine Code است.



پس از انجام کار loader، برنامه‌های پویا خود را در حافظه قرار داده و آدرس a و b را مثلاً ۱۵ و ۱۹ تعیین می‌کند. (آدرس متغیرها موقت است)

پس خروجی Absolute Machine code است.

(متغیرها مثل یک پست امانتی موقت بودن آدرس حافظه را نشان می‌دهد)

Link Editor: برای متغیرهای برون‌مرزی از برنامه تعریف شده در سایر فایل‌ها از دستور extern استفاده می‌شود.

پس از ترجمه کردن فایل‌ها، فایل obj. ساخته می‌شود. هر فایل جداگانه کامپایل می‌شود Seperate compilation است.

با دستوری مثل make، از link editor استفاده می‌شود و فایل نهایی پس از کامپایل کل برنامه ساخته می‌شود.

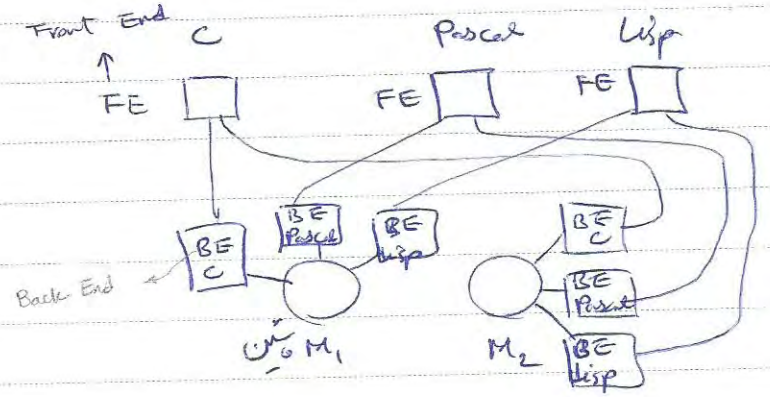
Subject
Date

کامپایلر را می‌توان به دو بخش front_end (مستقل از ماشین) و back_end (وابسته به ماشین) تقسیم کرد.

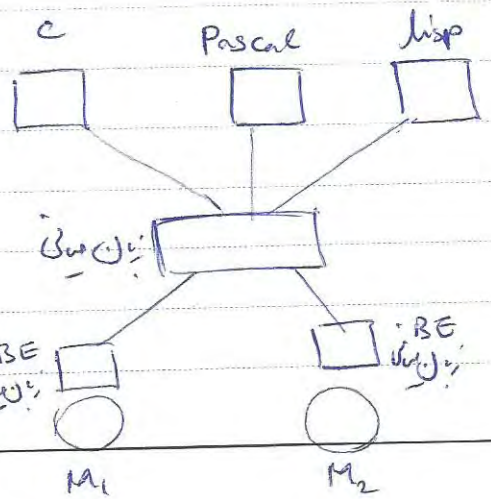
Front_End = Symbol Table + Syntax Analysis + Semantic Analysis + Intermediate Code Generation + Certain amount of code optimizer + related Part of Error handler

Back_End = Part of Symbol Table + Part of Code Generation + Part of code optimization + related Part of Error handler

مثال:



فقط کافی است FE بنویسیم.



یا می‌توان این کار را کرد:

(به این روش‌ها اصطلاحاً پورتابل گفته می‌شود)

چون که انواع مختلف زبان وجود دارد که هیچ‌کدام زبان‌ها را به زبان میانی واحد تبدیل کرد.

(تمامی این‌ها برای زبان‌های سطح بالا مطرح شده)

۹۳/۷/۵

Back Patching تکنیک ، برای single pass اگر بخواهیم کامپایلری بنویسیم که

goto L ≡

اگر بخواهیم برنامه مقابل را خط به خط ترجمه کنیم ، جلوی خود آدرس L

goto L ≡

مشخص نیست ، بعد دوباره وقتی که L را دیدیم آدرس L را داریم

L: ≡

دستور goto کنیم ، پس می‌شود دوبار خواندن (2 pass)

در روش Back Patching ، از ابتدا شروع به خواندن کد می‌کنند و ترجمه می‌کنند

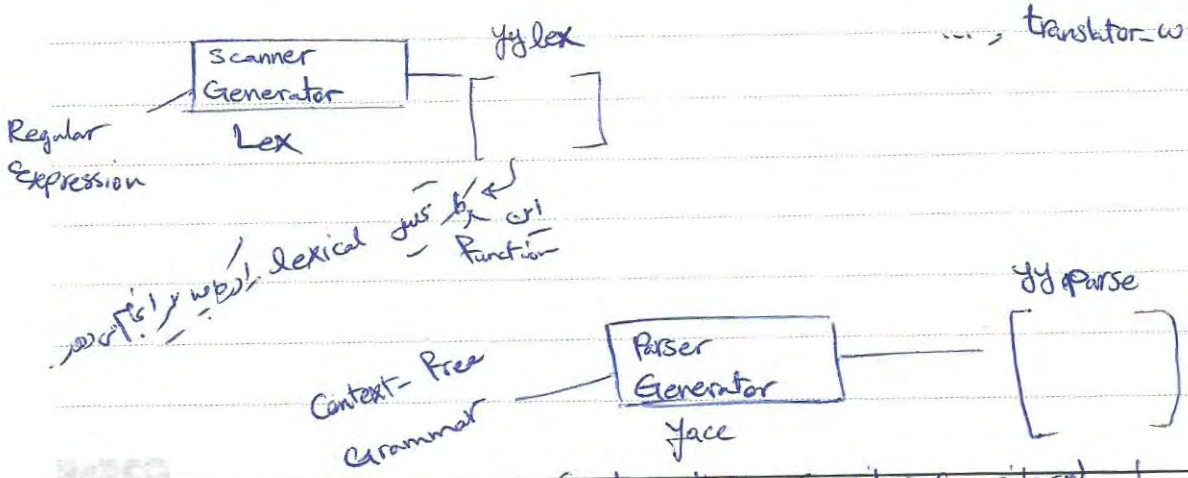
در ابتدا که می‌رسند آدرس L را نمی‌دانند ، باید لیست نویسی آدرس L را نگه می‌دارند.

(Random Access) بودن

```
100   goto L  (1400)
...
500   goto L  (1400)
...
1400  L:
```

پس به جای خواندن دوباره فایل ، از آن استفاده می‌کنیم.

ابزارهایی برای ساختن کامپایلر وجود دارد ، مثل Compiler_Compilers یا Compile_Generators ، ... ، translator_writing



Subject
Date

Parse tree → Syntax directed translation engine → Collection of routines that walks the parse tree generating intermediate code

one or more translations are associated with each node of the parse tree

این عمل در درخت تجزیه انجام می‌شود.

collection of rules defining the translation of each operation of the intermediate language → Automatic code Generator → ...

intermediate code + extra info → Data flow engine → Performs different task of Data Flow Analysis

مثالی در مورد تعریف زبان‌ها با BNF:

زبان انگلیسی را در نظر بگیرید. یک جمله از دو قسمت تشکیل شده است.

<sentence> → <noun phrase><verb phrase>

<noun phrase> → <adjective><noun phrase>

<noun phrase> → <noun>

<noun> → tree | book | ...

<adjectives> → little | big | ...

PgPCG:

Subject (derivation / اشتقاق) derivation

Date * derivation

$G = (V, T, P, S)$: Context-Free Grammar

- V: finite set of variables (non-terminals)
- T: finite set of terminals
- P: finite set of productions
- S: start symbol ($S \in V$)

$V \cap T = \emptyset$

$A \rightarrow \alpha$

- A: Non-Terminal
- α: sequence of non-terminals & terminals

α می‌تواند تهی باشد (λ یا ε)

$A \rightarrow \alpha_1$

$A \rightarrow \alpha_2$ $\Rightarrow A \rightarrow \alpha_1 | \alpha_2$

مثال:

1) $list \rightarrow list + digit$
2) $list \rightarrow list - digit$
3) $list \rightarrow digit$
4) $digit \rightarrow 0 | 1 | 2 | \dots | 9$

13)

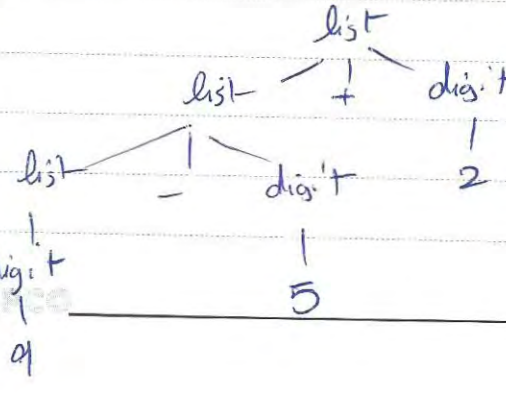
برای استخراج رشته $9-5+2$ چندین راه وجود دارد.

Parse tree ± derivation

derivation: $list \rightarrow list + digit \Rightarrow list - digit + digit \rightarrow$

$digit - digit + digit \xrightarrow{*} 9 - 5 + 2$ ✓

Parse tree:



تفاوت بین Parse tree و derivation: ترتیب اعمال قواعد تولید است.

در derivation ترتیب اعمال قواعد تولید مشخص است.

Subject
Date

۹۳/۷/۱

block → begin opt_stmts end

opt_stmts → stmt_list | ε

stmt_list → stmt_list ; stmt | stmt

stmt →

گرامر زیر را در نظر بگیرید:

مرتبط)

۱) کلمه را طوری تغییر دهید که قبل از end نباشد.

2) " " نگوید سه تا نباشد.

3) " " block ها ی تودرتو داشته باشیم.

4) " " block ها تودرتو ننظم داشته باشیم.

1) block → begin opt_stmts ; end

2) block → begin opt_stmts ε ; end | begin opt_stmts end

3) block → begin block end | ....

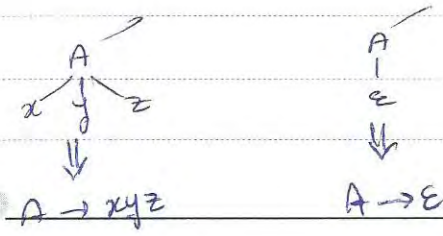
→ [S → (S) | ε]

4) block → begin block end block | ...

→ [S → (S)S | ε ، S → S(S) | ε ، S → S(S)S | ε]

هر سه درست هستند و معادل است.

Parse tree:



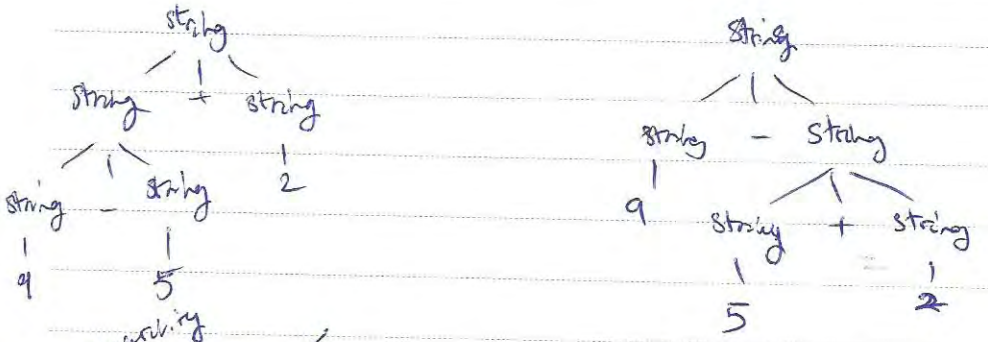
Subject
Date

اگر در Parse tree برگ‌های زیر را از چپ به راست بخوانیم yields of the tree نامیده می‌شود.

برنامه‌هایی که می‌توانند Parse tree یک عبارت را بسازند parser (تجزیه‌گر) نامیده می‌شوند.

مثال)

String → string + string | string − string | 0 | 1 | 2 | ... | 9



left-associativity

مثال) چرا اولویت + و − یکسان است؟ چرا + و − left-associativity دارند؟

* دارای اولویت بالاتری نسبت به + و − است.

چرا ضرب اولویت (Precedence) بالاتری نسبت به جمع دارد؟

روش رفع ابهام گرامر:

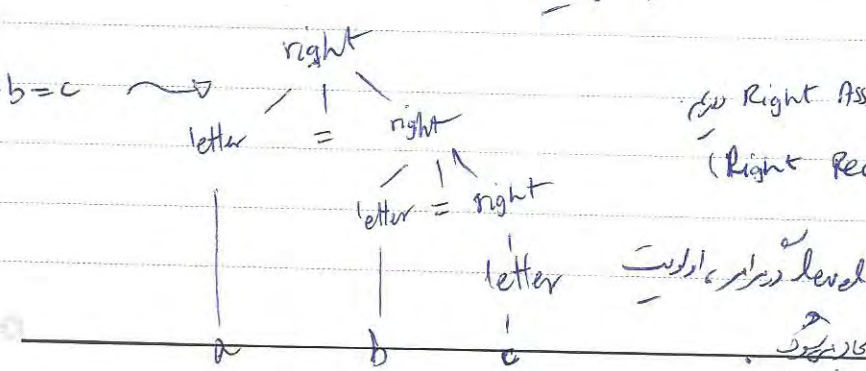
right → letter = right | letter

مثال)

letter → a | b | c | ... | z

این گرامر راست‌ترکیب است.

a = b = c



این گرامر Right Associativity دارد (Right Recursive)

با ایجاد level در گرامر، اولویت عملگرها را در آن ایجاد می‌کنیم.

Subject
Date

مثال) Exp → Exp + Exp | Exp - Exp | Exp * Exp | Exp / Exp | (Exp) | digit

این گرامر مبهم است ! رفع ابهام کنید :

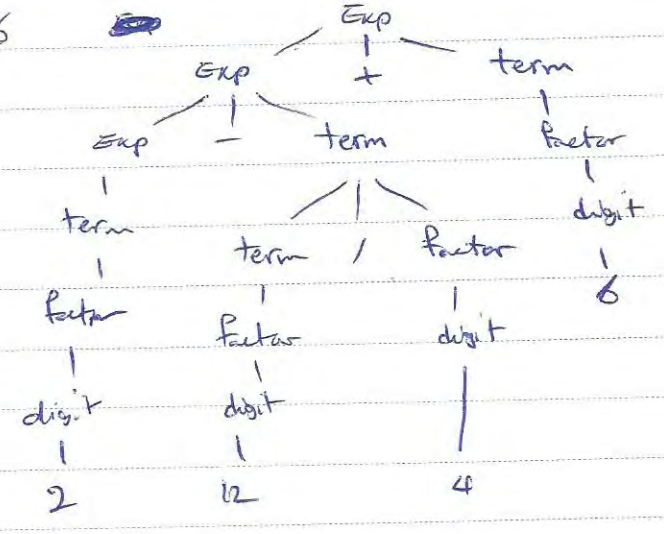
Exp → Exp + term | Exp - term | term

term → term * factor | term / factor | factor

factor → (Exp) | digit

این نوع با Left_Associativity و precedence رفع ابهام می‌شود.

2+12/4+6



بعضی از گرامرها ذاتاً مبهم هستند و نمی‌توان برای آن‌ها رفع ابهام کرد.

مثال)

stmt → id := expr
| if expr then stmt
| if expr then stmt else stmt
| while expr do stmt
| ~~begin opt_stmts end~~

opt_stmts → stmt_list | ε

stmt_list → stmt_list ; stmt | stmt

Subject:

Year. Month. Date. ( )

مثال:

Exp → Exp + EXP | Exp - EXP | Exp * EXP | Exp / Exp | (EXP) | digit

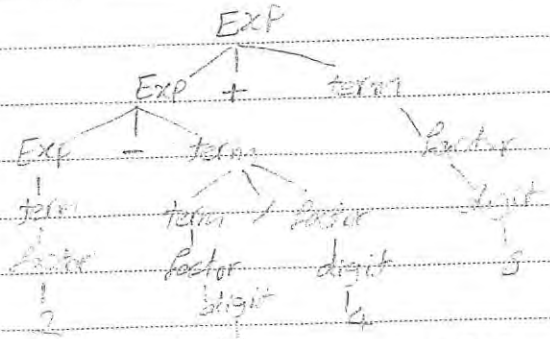
طوری تغییر دهید که * و / اولویت بیشتر از + - و left asso شود.

Exp → Exp + term | Exp - term | term

term → term * factor | term / factor | factor

factor → (exp) | digit

2 - 12 / 6 + 6



می توان همین کار را انجام داد (تقدم بیشتر) yacc در: %left + -
%left * /

%perc

مثال:

Stmt → id := expr | if expr then Stmt | if expr then Stmt else St
| while expr do Stmt | begin opt_Stmts end

opt_Stmts → Stmt_list | ε

Subject:

Year. Month Date. ( )

Stmt_list → Stmt_list Stmt | Stmt

reserved words مثل if و then

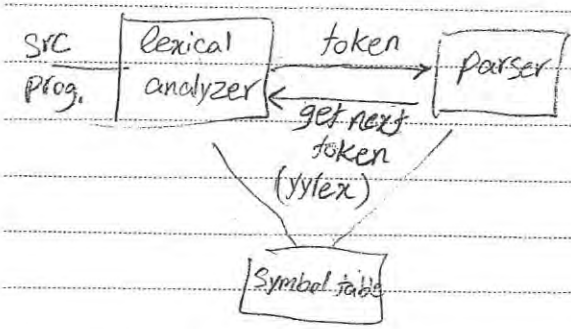
کلمات کلیدی بیشتر برنامه را برای ما خواناتر می‌کند همچین کامپایلر با

سرعت بیشتری مسیرش را تشخیص می‌دهد مثلا اگر در while expr do Stmt

while را نگذاریم وقتی به do می‌رسد می‌فهمد که نوع دستور حلقه بوده و باید به expr

برگردد اما وقتی می‌گذاریم از همان ابتدا که while را ببیند نوع دستور را می‌فهمد

lexical Analyzer



علاوه بر پیدا کردن token وظایف دیگری هم دارد

Possible Secondary tasks of lexical Analyzer

- removes comments, blocks, tabi, newline, white space
- keeps trace of numbers of new line character
- makes copy of src. prog. with error message marked in it

Preprocessing

جلسه ۲

چرا در کامپایلر lexical Analyzer و Parser را جدا می‌کنیم و از یک ماجول که کار هر دو را انجام می‌دهد نمی‌گذاریم؟

① Simpler Design

② efficiency is improved

PTO

این جلسه ۱ صفحه بوده

۹۳/۷/۱۲

Subject:
Year. Month. Date.

(3) Compiler portability is enhanced

اگر ماجولار باشد کنترل راحتتر است اگر نباشد سرعت بالاتر است و در کامپایلر ماجولار کنند که کنترل بهتر باشد

در lexical Analyzer خواندن فایل زمان بر است و تلاش می‌شود سرعت آن زیاد شود recursion از Parser استفاده می‌کند که از itteration سرعت کمتری دارد پس lexical An. و Parser را جداسازی کردند تا سرعت را بهبود دهند هر چند که جداسازی ماجول‌ها سربار دارد اما ارزشش را دارد و Portable می‌شود و با یک تغییر کوچک در یک ماجول می‌توان آن را برای زبان دیگری استفاده کرد

Sample lexems	token	Informal description of patterns
If	If	If
else	else	else
<, <=, =, <>, >, >=	relop عملگر	< or <= or = or <> or > or >=
pi, count, D5	id	letter followed by letters or digit
"core dumped"	literal	any characters between " and " except "

مثال:

printf("total = %d\n", s);

(?)

PaPCG

Subject:

Year. Month. Date. ( )

Lexeme match Pattern

↓

token

معمولا دسته های زیر در زبان ها وجود دارند:

keywords, operators, Identifiers, Constants, literal Strings

Punctuation Symbols : parenthesis, Comma, Semicolon

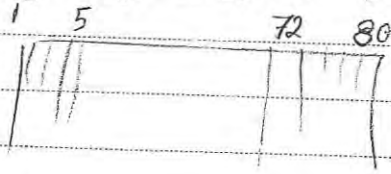
(token, attribute)

مثال:

Position 1 Position → (id, 1)

نوع ← ↑ آدرس در خانه حافظه

برای عمل نوشتن دستورات قوانینی دارد مثلا فقط در ستون های 72-6 Fortran



برای نوشتن برنامه بود پس lexical Ana آن

باید محتویات هر ستون را نیز چک می کرد (در زبان های بعدی این محدودیت وجود ندارد)

همچنین در Fortran ، کاراکتر بین حروف متغیرها مثلا Po si tion=4 مجاز بود (زیرا

اینکار باعث اشتباهی زیاد بود پس اگر خطا گرفته می شد تعداد خطاها زیاد می شد) پس

lexical ana باید در مثال زیر تا پیش برود تا بفهمد نوع دستور چیست assignment

Do 5 I = 1, 21 Do 5 I = 1.25

5 Continue

Subject:

Year. Month. Date. ( )

پیدا کردن token:

مثال:

if then then=else ; else else. then;

(؟) lexical an. طی کامپایلر تشخیص دهد که then متغیر است یا کلمه کلیدی

+ Pattern → (token, att)

attribute

نیز متغیر بولین است.

مثال

E = M * C ** 2

(id, pointer to symbol-table entry for E)

(assign_op, —)

(id, pointer to ... M)

(mult_op, —)

(id, pointer ... C)

(exp_op, —)

(num, value 2)

(num, 5)

پیدا کردن خطا

مثال:

fi (a == f(x))

این استوری توان if را شناسایی کند یا یک متد با نام fi را شناسایی کند

اگر a=f(x) شرط true باشد یک مقدار را برگرداند fi(2)

پس lexical an. باید تشخیص دهد که کدام حالت است. مثلا symbol table را چک کند

PAPCO

Subject:

Year. Month. Date. ( )

مثلا اگر begin را bigen بنویسیم lexical ana تشخیص error recovery داده و آن را درست کند. هر اصلاح را یک واحد زمانی در نظر می‌گیریم و به دنبال این هستیم که با کمترین زمان ویرایشی (Minimum distance error correction) آن را اصلاح کنیم

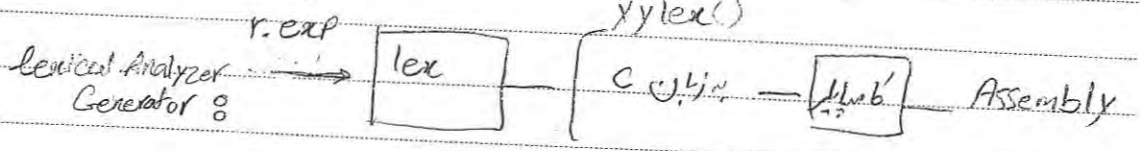
NP-Complete

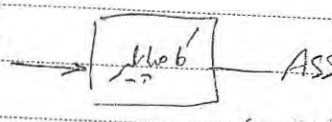
جلسه A ۹۳/۷/۱۴

1- lexical Analyzer Generator : lexical ana... سه زبان برای نوشتن

2- C

3- Assembly





از بالا به پایین سرعت بیشتر سهولت کار کمتر می‌شود

مثال :

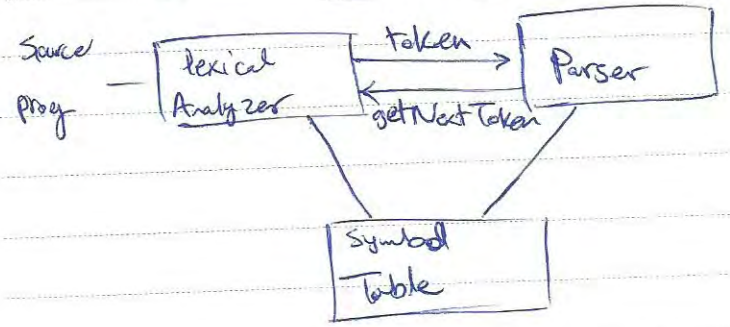
Declare (arg1, arg2, ... argn)

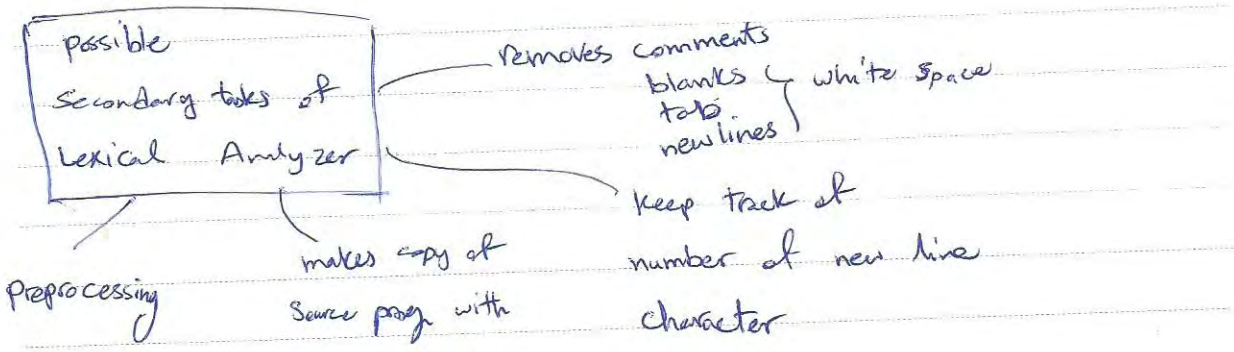
می‌تواند یک آرایه باشد یا یک Func. اگر بعد از آن (باشد Func است اگر = باشد آرایه. پس lex ana باید تا رسید به = یا (قبلی ها را در بافر نگه دارد و وقتی به = یا (رسید شروع به پردازش کند

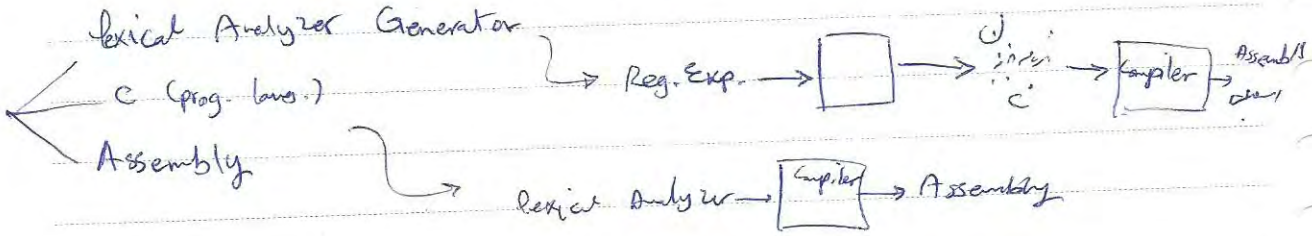
Subject

Date ۹۴/۷/۷

(reserved words)

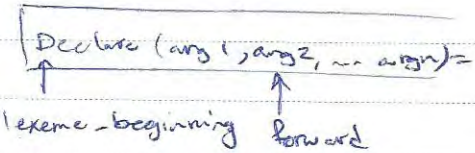






Subject
Date

Declare (arg1, arg2, ..., arg n) =




```
if forward at end of first half then
    begin
      reload second half;
      forward := forward + 1;
    end
else if forward at end of second half then
    begin
      reload first half;
      move forward to beginning of first half;
    end
else forward := forward + 1
```

عملگرهای روی زبان‌ها:

Union: $L_1 \cup L_2$ → $\{s \mid s \in L_1 \text{ or } s \in L_2\}$

Concatenation: $L_1 L_2$ → $\{st \mid s \in L_1 \text{ and } t \in L_2\}$

Closure: L^*

L^* ← Kleene closure of $L = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup \dots$; $L^0 = \{\varepsilon\}$

L^+: positive closure of $L = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup \dots$

Regular Expression:

هر R.E. نشان‌دهنده یک زبان است.

$\varepsilon \longrightarrow \{\varepsilon\}$

$a \in \Sigma \longrightarrow \{a\}$

r, s are R.E. and $L(r), L(s)$:

1) $(r) \mid (s) \longrightarrow L(r) \cup L(s)$ — زبان‌های تعریف‌شده

2) $(r) \cdot (s) \longrightarrow L(r) L(s)$

3) $(r)^* \longrightarrow (L(r))^*$

4) $(r) \longrightarrow L(r)$

$(r) \mid (s) \triangleq (r) + (s)$

$(r) \cdot (s) \triangleq (r)(s)$

اولویت عملگرها: * بیشترین اولویت را دارد، سپس ·، سپس |

· و | هر دو left-associative هستند.

$a \mid b^* . c \longrightarrow (a \mid ((b^*) . c))$

$(a \mid b)^* = (a^* b^*)^*$

دو عبارت منظم R.E. معادل‌اند اگر زبان‌های تعریف‌شده معادل باشند.

r, s R.E.:

1) $r \mid s = s \mid r$ — | is commutative (جابه‌جایی)

2) $r \mid (s \mid t) = (r \mid s) \mid t$ — | is associative

3) $r(st) = (rs)t$ — . is associative

4) $r(s \mid t) = (rs) \mid (rt)$

5) $(s \mid t)r = (sr) \mid (st)$

Concatenation distributes over | (پخش‌پذیری)

6) $\varepsilon r = r\varepsilon = r$

7) $r^* = (r \mid \varepsilon)^*$

8) $r^{**} = r^*$

Regular Definition:

$d_1 \rightarrow r_1$ — R.E.

$d_2 \rightarrow r_2$ — R.E. روی $\Sigma \cup \{d_1\}$

$\vdots$

$d_n \rightarrow r_n$ — R.E. روی $\Sigma \cup \{d_1, d_2, \dots, d_{n-1}\}$

در این تعریف، هر r_i می‌تواند از d_1 تا d_{i-1} استفاده کند.

forward = forward + 1;

if forward ↑ = EOF then begin

if forward at end of first half then

begin

reload second half;

forward := forward ——118 مرحله بعد—→ forward = beginning of second buffer;

end

else if forward at end of second half then

begin

reload first half;

move forward to beginning of first half.

end

else terminate lexical analysis

. با این کار، مشکل look-ahead داشتیم

end

۹۲/۷/۱۹ Strings and Languages:

« Alphabet = finite set of symbols

« String over an alphabet: finite sequence of symbols drawn from that alphabet

• (sentence or word)

« length of a string s: |s| = number of symbols

|book| = 4 ← {a, ..., z}: الفبای انگلیسی

« empty string: ε |ε| = 0

« prefix of s: book ← book رشته‌ای است، تعدادی از حروف ابتدای رشته‌ها

« suffix of s: book → رشته‌ای است، تعدادی از حروف انتهای رشته‌ها

« substring of s: book → رشته‌ای است، تعدادی از حروف پشت سر هم رشته‌ها

« subsequence of s: book → رشته‌ای است، تعدادی از حروف رشته‌ها (حروف حذف شده نباید کنار هم باشند)

« proper suffix: suffix ای که برابر خود رشته نیست

proper prefix: prefix ای که برابر خود رشته نیست

proper substring: زیررشته‌ای که

« Language: set of strings over some fixed alphabet

« φ: empty set → {ε}

تعریف زبان: {p | p is a syntactically well-formed pascal prog}

« اگر x و y دو رشته باشند، Concatenation این دو، xy خواهد شد.

« $S\varepsilon = \varepsilon S = S$

« $S^0 = \varepsilon$, $S^i = S^{i-1} S$

۱۹/I

Subject
Date

Example -

letter → A|B|c|...|z|a|b|...|z

digit → 0|1|2|...|9

id → letter (letter| digit)*

از لحاظ ظاهری شبیه تعریف گرامر است ولی خیلی فرق دارد!

مثلاً در بالا، A به معنای نوشتن دو تا حرف است.

letter → A
letter → B ⇒ letter → A|B

letter → A|B

یعنی A و B در یک سطح هستند.

یعنی R.E. ، یعنی A یا B

یا → {A, B}

Example -

برای تعریف اعداد در زبان برنامه نویسی

digit → 0|1|...|9

digits → digit digit*

optional_fraction → . digits | ε

optional_exponent → (E (+|-|ε) digits) | ε

num → digits optional_fraction optional_exponent

00123
↳ leading zero

12.300
↳ trailing zero

این عبارت چه اشکالاتی دارد؟ leading zero, trailing zero

مثلاً 0.1 یا 1.0 یا 0.0 را نباید داشته باشیم.

بهتر عبارت زیر:

non-zero digit → 1|2|...|9

⋮

Subject

Date ۹۳/۷/۲۴

Regular Expression	Language
r	$L(r)$
r^+	$(L(r))^+$
a^+	رشته‌ای با یک یا بیشتر a
$r^* = r^+ \mid \varepsilon$	
$r^+ = rr^*$	
$r? = r \mid \varepsilon$	

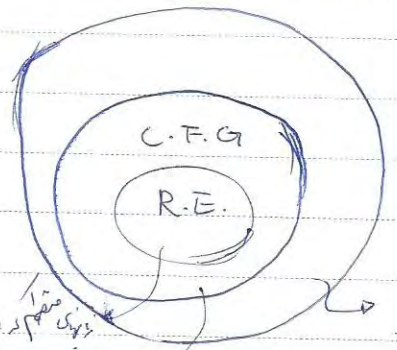
توان مثال قبل را نشان دهد.

digit → 0 | ... | 9

digits → digit$^+$

optional_fraction → (. digits)?

optional_exponent → (E(+|-)?digits)?



انواع زبان‌ها:

زبان‌های منظم یا R.E. پس دارند.

CFG دارند ولی R.E. نیستند. مثال: $\{wcw \mid w \in \{a,b\}^*\}$

مثال: $\{a^n b^n \mid n \geq 0\}$ ← $S \Rightarrow aSb \mid \varepsilon$

زبان‌های CFG که R.E. نیستند. مثال: $\{ww^R \mid w \in \{a,b\}^*\}$

← با استفاده از PDA می‌توان نشان داد.

مثال: گرامر زیر داده شده است:

(این مثال خودرس است)

stat → if expr then stat

| if expr then stat else stat

| ε

expr → term relop term

| term

relop → Relational Operator

term → id | num

Subject

Date ۹۳/۷/۲۶ R.D. = Regular Definition

ابتدا دستورات را به terminal و non_terminal تبدیل می‌کنیم.

این دستورات را به صورت R.E. می‌نویسیم.

if → if ← این بخش از R.E. است

then → then

else → else

relop → < | <= | > | >= | <> | =

id → letter (letter | digit)*

num → digit+ (. digit+)? (E (+|-)? digit+)?

ws → delim+ ~~~> white Space

delim → blank | tab | newline

R.D. ⇒ این بخش از R.E. است

سپس Reg. Exp. را می‌نویسیم.

در جدول کلمات کلیدی و ... برای اسامی از Symbol Table استفاده می‌شود.

R.E.	Token	Attribute Values
ws	—	—
if	if	—
then	then	—
else	else	—
id	id	pointer to table entry (Symbol table)
num	num	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

از اینجا به بعد را می‌توان به کد در آن Lex Analyzer را می‌دهد.

کلیه توکن‌ها به صورت relop، id، num و ... را می‌نویسیم.

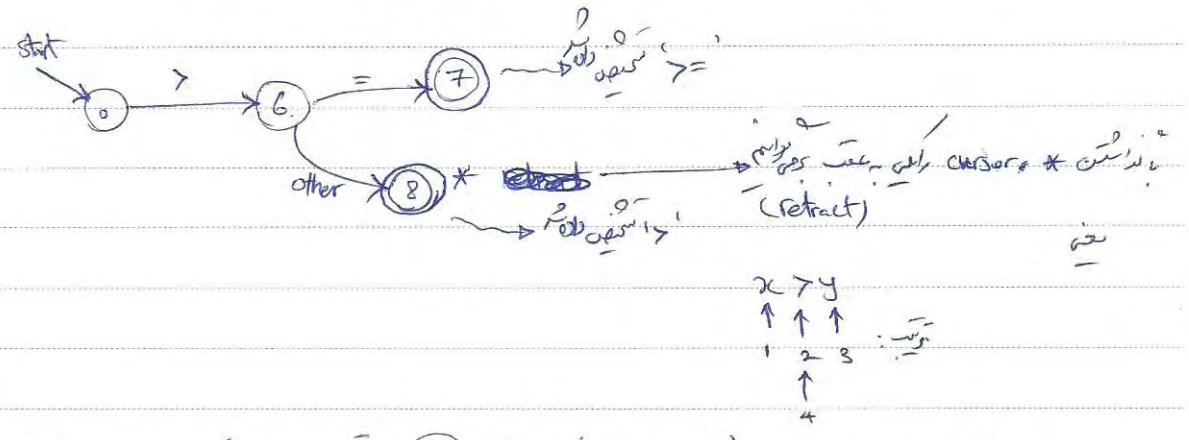
Attribute مشخص کننده Token است.

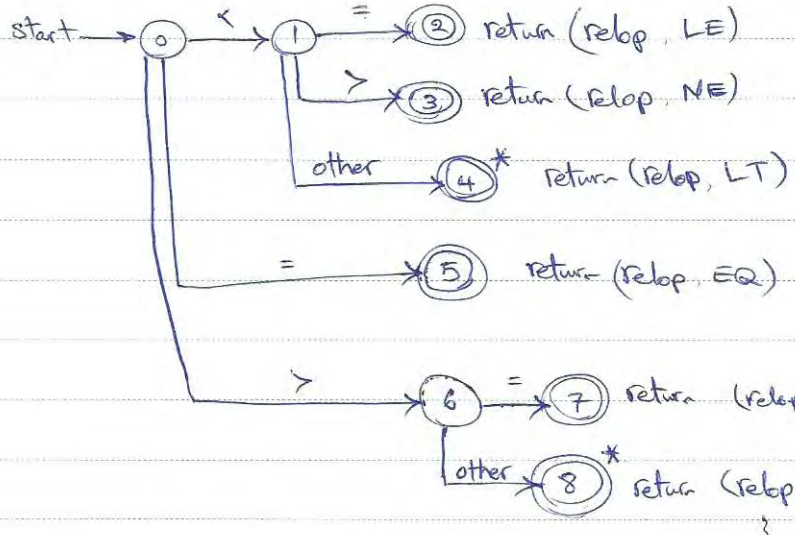
اما ما کد کلی با عملگرهای ریاضی می‌نویسیم.

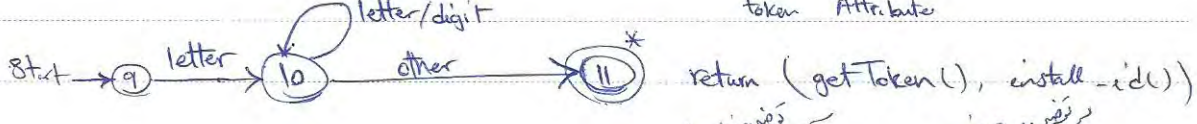
۴۲

Subject

Date ۹۳/۷/۲۴







Check symbol table for the lexeme

if lexeme = keyword

return (corresponding keyword)

else

return (id)

if
then
else

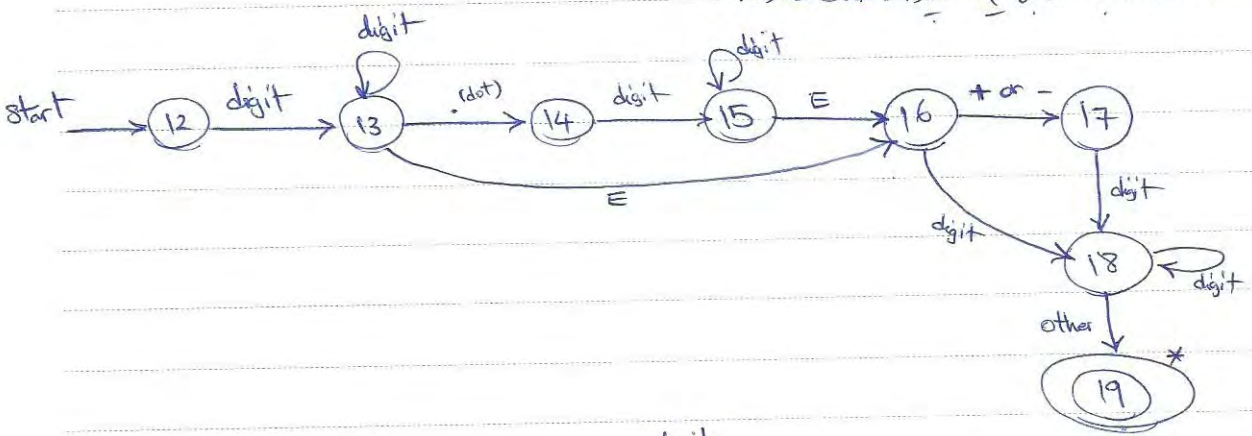
Subject
Date ۹۴/۷/۲۹

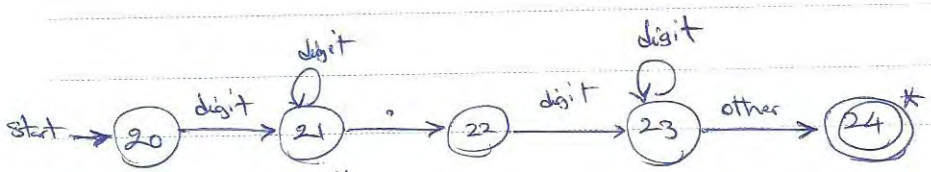
تابع install_id() Function :

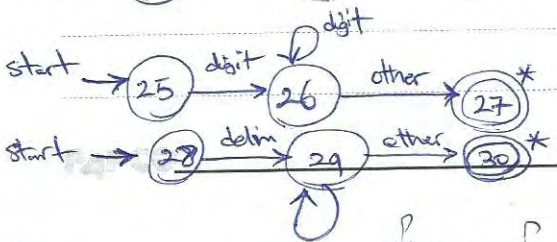
```
check symbol table for the lexeme
if (lexeme in symbol table) {
    if lexeme = keyword
        return (0)
                zero
    else
        return (pointer to the symbol table entry)
}
else{
    install (lexeme as variable into symbol table)
    return (pointer to the entry created)
}
```

مثال داده شده که اگر طبق فرمول استاد برای زبانی مثل C پیش برویم، حدود ۱۰۰ state خواهیم داشت که بسیار زیاد است.

شرط ها را حذف کرده، خیلی بهتر خواهد شد.







ترتیب state ها اهمیت دارد. (ریل خطر)

نکته: white space ها چیست و چه کاربردی دارند؟

۲۵

Subject
Date

۲. عوامل انتقال، خطاهایی که ما باعث ایجاد می‌شوند و بیان می‌کنند که باعث ایجاد چه چیزی را طور.

(ادامه داشت بعد ←)

:دیگر Automata پیاده

```
int state=0, start=0;
int lexical_value;
int fail(){
  forward = token_beginning;
  switch (start){
   Case 0: start:=9 ;break;
   Case 9: start:=12; break;
   Case 12: start:=20; break;
   Case 20: start=25; break;
   Case 25: recover(); break;
   default: /* Compiler error */
  }
 return start;
}

token nextToken(){
  while(1){
   switch (state){
   Case 0: c=nextchar();
   if (c==blank || c==tab || c==newline){
       state=0;
       lexeme_begining++;
   }
   else if (c=='<') state=1;
   else if (c=='=') state=5;
   else if (c=='>') state=6;
   else state=fail();
   break;
   ..... /* Cases for 1 to 8 */
```

Subject

Date

```
Case 9:  c = nextchar();
         if (isletter(c))  state = 10;
         else  state = fail();
         break;

Case 10: c = nextchar();
         if (isletter(c))  state = 10;
         else if (isdigit(c))  state = 10;
         else  state = 11;
         break;

Case 11:
         retract(1);
         install_id();
         return (gettoken());
         /* Cases 12-24 here */

Case 25: c = nextchar();
         if (isdigit(c))  state = 26;
         else  state = fail();
         break;

Case 26: c = nextchar();
         if (isdigit(c))  state = 26;
         else  state = 27;
         break;

Case 27: retract(1);
         install_num();
         return (num);
    }
}
```

Subject کامپایلر
Date ۹۳/۷/۲۸

Bottom_Up (Shift_reduce) parser:

LR(K)
- look ahead
- Right most deviation in reverse
- left to right scan

LL(K)
- look ahead
- left most deviation
- left to right scan
- ← Top-down parser

Example -

S → aAcBe
A → Ab | b
B → d

abcde*

S	
aAcBe	
aAbcde	
aAbcde	
abbcde	

(Right most deviation in reverse)

Example -

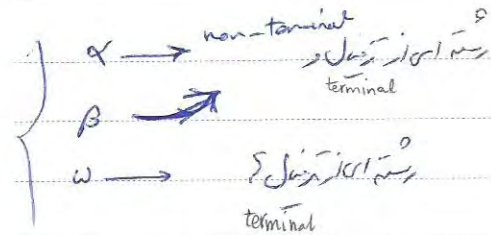
S → aABe
A → Abc | b
B → d

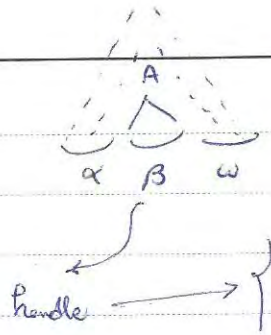
ab(bc)*de

S	
aABe	
aAde	
aAbcde	
abbcde	

Subject
Date

handle pruning





Example -

$E \to E + E$

$E \to E * E$

$E \to (E)$

$E \to id$

$id_1 + id_2 * id_3$

$E + id_2 * id_3$

$E + E * id_3$

$E + E * E$

$E + E$

E

$id_1 + id_2 * id_3$

$E + id_2 * id_3$

$E + E * id_3$

$E * id_3$

$E * E$

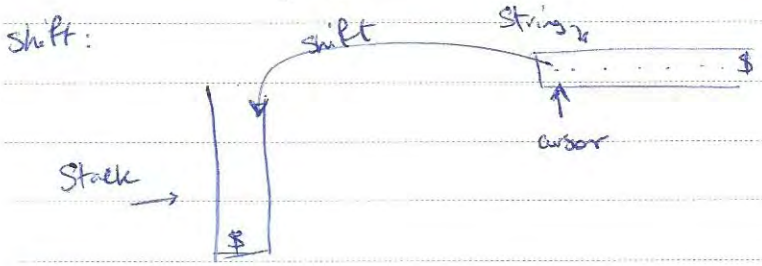
E

Right most derivation

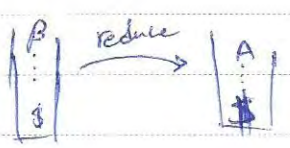
Shift - reduce conflict

: Shift - reduce

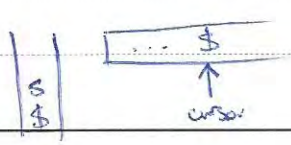
Shift:



Reduce:



accept:



Subject
Date ۹۳/۸/۳

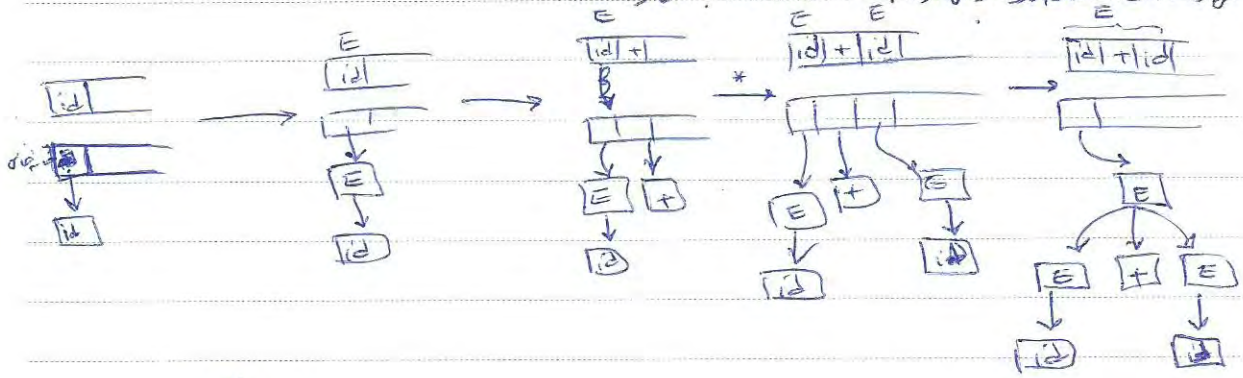
مثال قبل اجرا میشه:

1)	$	$id_1+id_2*id_3\$$	
2)	$id_1	$+id_2*id_3\$$	
3)	$E	$+id_2*id_3\$$	
4)	$E+	$id_2*id_3\$$	
5)	$E+id_2	$*id_3\$$	دوبار دیگه Shift داریم.
6)	$E+E	$*id_3\$$	
7)	$E+E*	$id_3\$$	$E	← دوبار دیگه میرسیم Reduce
8)	$E+E*id_3	$\$$	$E*
9)	$E+E*E	$\$$	$Eid_3	Shift/Reduce conflict این مشکل
10)	$E+E	$\$$	$E*E
11)	$E	$\$$	$E+

پس باید جمله را اینجا بزنیم و بعد جمله را با ضرب ادامه دهیم.

ساختار yacc: این شکل است که دستورهای حرف زده شده را نگه میداریم و وضعیت‌هایی که از Reduce و Stack

می‌شود آن قطعه کد اجرا می‌شود. نمونه کدی که برای این شکل نوشته می‌شود:



Stack

yacc با استفاده از Pattern Recognition روش‌های خودش را انجام می‌دهد. بعضی شکل‌ها برای این است که match

شده را خانه خانه می‌کند و استفاده می‌کند. OCR است.

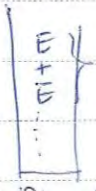
$S \rightarrow \underset{\text{بازگشت}}{B}: S \mid \underset{\text{بازگشت}}{A} S \mid \underset{\text{بازگشت}}{C} S \mid \dots \mid Z S$

بازگشت به حرف A — بازگشت به حرف C — بازگشت به حرف Z

Subject ..
Date ..

Viable Prefix:

The set of prefixes of right sentential form that can appear on the stack of a shift-reduce parser.



Viable Prefix

Shift-Reduce Conflict:

stmt → if expr then stmt
| if expr then stmt else stmt

مثال: if expr then if expr then stmt else stmt

if expr then stmt

↓

Reduce یا Shift

Reduce/Reduce Conflict:

stmt → id (parameter_list)
stmt → expr := expr

parameter_list → parameter_list ; parameter | parameter
parameter → id
expr → id (expr_list) | id
expr_list → expr_list , expr | expr

Subject
Date

Reduce/Reduce Conflict و Shift/Reduce Conflict

Symbol Table

Context-free

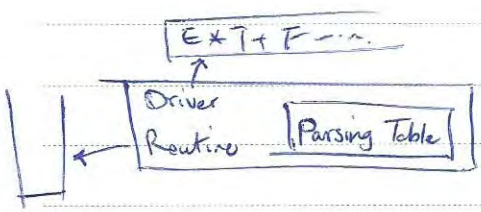
1 $E \to E+T$
2 $E \to T$
3 $T \to T*F$
4 $T \to F$
5 $F \to (E)$
6 $F \to id$

State	Action							GoTo		
	id	+	*	(	)	$		E	T	F
0	s_5			s_4				1	2	3
1		s_6				acc				
2		r_2	s_7		r_2	r_2				
3		r_4	r_4		r_4	r_4				
4	s_5			s_4				8	2	3
5		r_6	r_6		r_6	r_6				
6	s_5			s_4					9	3
7	s_5			s_4						10
8		s_6			s_{11}					
9		r_1	s_7		r_1	r_1				
10		r_3	r_3		r_3	r_3				
11		r_5	r_5		r_5	r_5				

LR

Parsing Table

Knuth



S: shift
S5: state 5 ، shift
r: reduce
r5: 5 ، reduce

State ، GoTo

non-terminal ، GoTo ، terminal : Action

Subject

Date ۹۴/۱/۳۰

این قسمت مهم است

top

Stack محتوی ↓ ورودی ↓

$(S_0 X_1 S_1 X_2 \cdots X_m S_m , a_i a_{i+1} \cdots a_n \$)$

مثلا :

shift S

$(S_0 X_1 S_1 \cdots X_m S_m a_i S , a_{i+1} \cdots a_n \$)$

S ↓ state

Goto جدول ← S state

reduce $A \to b$

$(S_0 X_1 S_1 \cdots X_{m-r} S_{m-r} A S , a_i a_{i+1} \cdots a_n \$)$

تعداد سمبل های سمت راست : r

Goto $[S_{m-r}, A] = S$

state

Stack از سمبل 2r سمبل را

حذف میکنیم

جدول تجزیه را روی ورودی $id * id + id \$$ و Stack تهی (S_0) تشکیل میدهیم.

Stack	input
0	id*id+id$
0id5	*id+id$
0F3	*id+id$
0T2	*id+id$
0T2*7	id+id$
⋮	⋮
0E1+6F3	$
0E1+6T9	$
0E1	$

Subject

Date

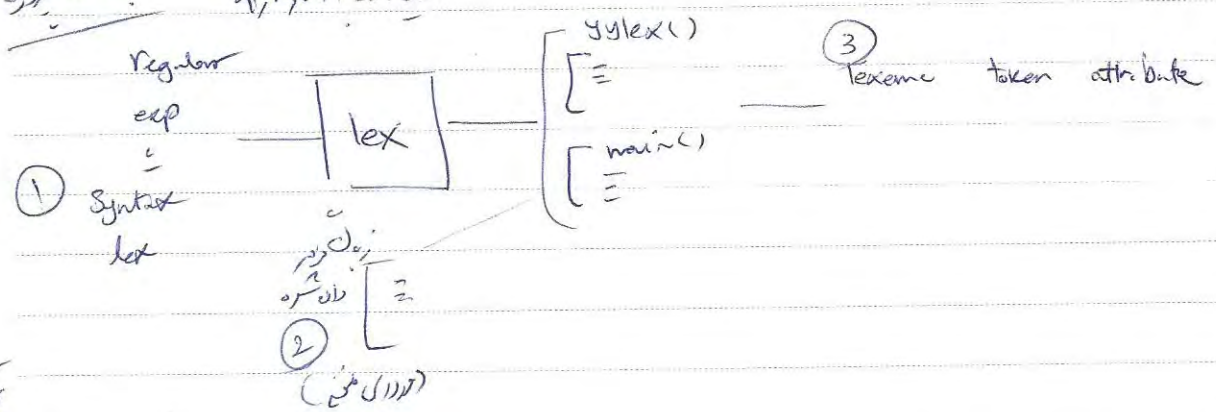
: Parsing Tree انواع

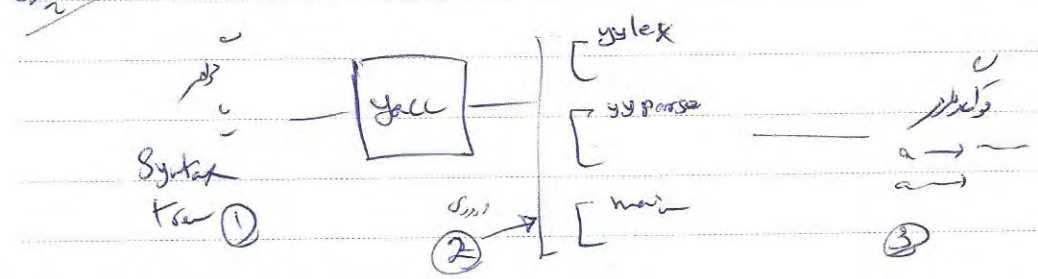
1) Simple LR
2) LALR
3) Canonical LR

For Pascal

SLR
LALR → Same # of states — Several hundred states

Canonical LR → Several thousands states





print → 1, 2, 3

print → 1, 2, 3

Subject
Date

Symbol Table:

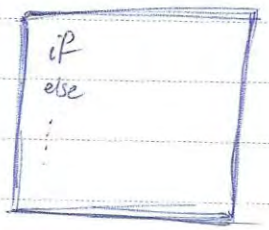
برای ذخیره سازی اطلاعات به دست آمده از فازهای مختلف کامپایلر استفاده می‌شود و بعد از آن نیز برای Context-Free مورد

انواع پیاده‌سازی Symbol Table:
- Hash table
- linear list

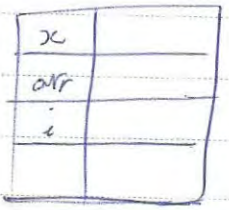
عملیات مورد نظر روی Symbol table عبارت است از insert، find، delete.

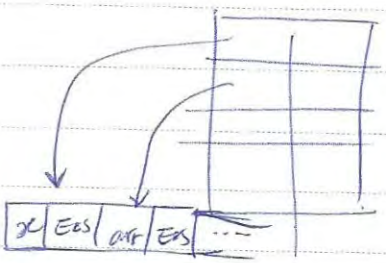
اسامی که برای متغیرها تعریف می‌کنیم در Symbol Table قرار می‌گیرند.

کلمات کلیدی یا Reserved word هستند و نیاز نیست در جدول نمادها قرار بگیرند.



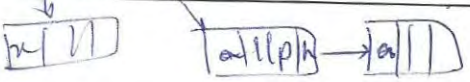
در حالت ساده جدول نمادها را به صورت یک آرایه (linear list) در نظر می‌گیریم و اسامی را به ترتیب در آن قرار می‌دهیم.





مشکل: نیاز به پیمایش کل آرایه است تا به Eos برسیم.

کارایی این حالت بهتر است.



Subject
Date

Symbol table ... Scope ... Scope ...

... Scope ...

مثال)

Program Test;

var i,j,k : integer;

Procedure P;
var i,j : char;
  Procedure Q;
  var i,k : real;
  begin
  i := 24.3;
  end
begin
i := 'c';
end

Procedure R;
var k : real;
begin
k := 2.4;
end

begin
i := 7;
end

i int
j int
k int
i char
j char
i real
k real

... Scope ... Scope ...

... linear list ...

insert → O(n)
Search → O(n)

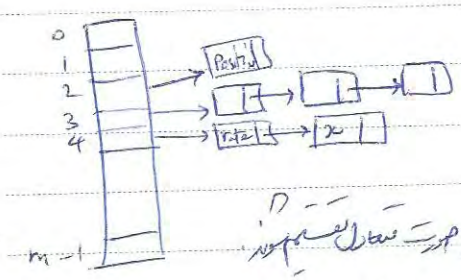
روش ساده : Hash table

h('position') = m-1 عددی بین ۰ تا

مثلا $h('rate') = 4$

$h('id') = 4$

$h('position') = 2$



اگر تابع Hash اهمیت دارد نه نوع token و ترتیب قرارگیری آن‌ها.

ساختار جدول نیز اهمیت دارد. اگر ساخته شده با تابع Hash هم خوانی دارد.

insert → O(1)
Search → O(1)

(مگر در چند حالت، بیشتر از ۳۰٪ درایه‌ها پر است یا بیشتر شد)

یعنی m، عددی اول انتخاب شود تا توزیع مقادیر در جدول به درستی انجام شود.

می‌توان hash را به صورت زیر محاسبه کرد

$h_0 = 0$

$h_i = \alpha h_{i-1} + c_i$ for $1 \le i \le k$

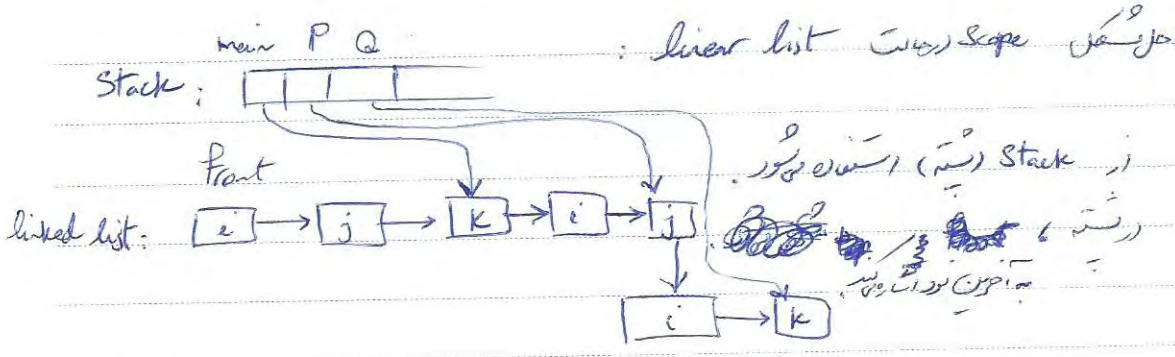
(c_i: کاراکتر i‌ام رشته)

let $h = h_k$, where k is the length of the string

hash value = h mod m

α ضریبی است که می‌تواند مقدار ثابتی باشد یا تابع hashing، تابعی از i باشد.

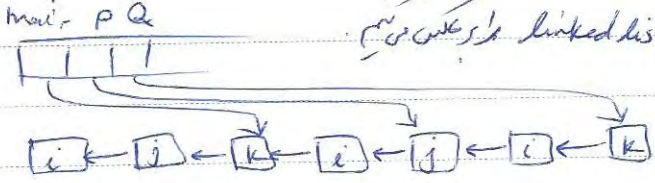
Subject
Date



برای حل مشکل Scope در حالت linear list :

از Stack (پشته) استفاده می‌شود.

در پشته، آخرین نود اشاره‌گر به آخرین نود اشاره می‌کند.



حال اشاره‌گرهای linked list را برعکس می‌کنیم.

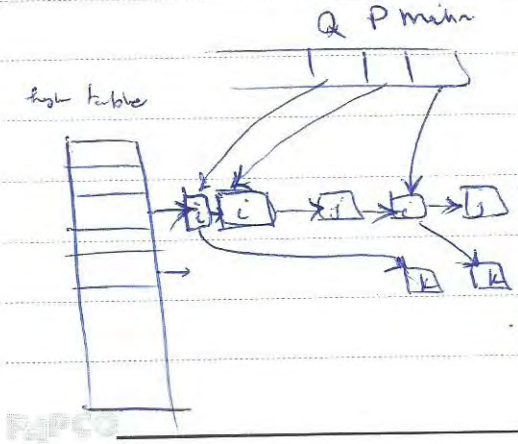
حال برای 3 = 2+1 = 2+4-3 یک نود می‌سازیم، جایگاه P را آن اشاره می‌دهد.

پس از اتمام کار، Q را از Stack، pop و متغیرهای مربوط به آن (یعنی P) را آزاد می‌کنیم.

این روند را برای بقیه نودها تکرار می‌کنیم...

این روش برای Compiler one-pass مناسب است، چون Symbol Table در حافظه نگه داشته می‌شود.

(در پایان کار، Symbol Table خالی خواهد بود)



برای حل مشکل Scope در حالت hash table :

در واقع اول اشاره‌گر لیست عوض می‌شود و درست است بعد تغییر می‌کند.

بقیه هم لیست عوض است با اشاره‌گرهای خودش.

Subject Compiler

Date ۹۴/۸/۱۷ COMPILER

کامپایلر نیاز به ساختن Stack دارد که بعداً می‌بینیم.

Runtime environment:

هر چیزی (شیء یا داده) که در حافظه نیاز به ذخیره دارد data object

activation of a procedure ← execution of a procedure

procedure ← procedure که recursive نیست

Procedure definition:

header → procedure P(.....) ← Formal Parameters

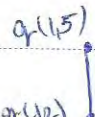
body → begin
=
end

P(——) ;
↳ actual Parameters

lifetime of an activation of procedure p: مدت زمانی که procedure p در حال اجرا است

علاوه بر زمانی که تعریف می‌شود، زمانی که فراخوانی می‌شود.

یعنی تابع P، تابع q را فراخوانی کرده است.
(procedure)



Call ،q(1,2) ، q(1,5) را

P، q را Call کرده و q، r را Call کرده است.

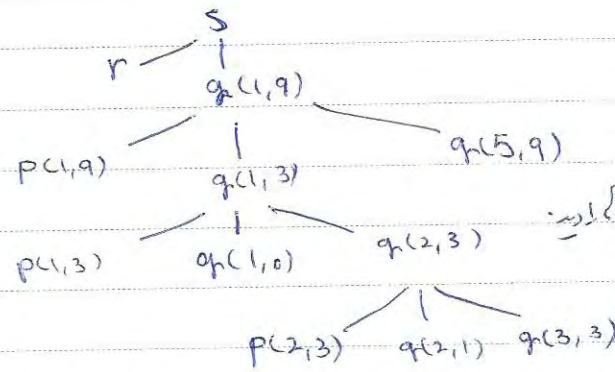
Subject

Date ۹۳/۸/۷

اگر گراف فراخوانی procedure ها را رسم کنیم به آن Activation Tree گفته می‌شود.

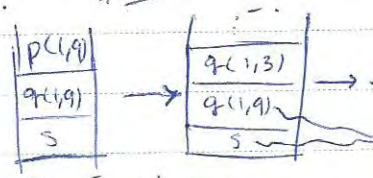
ریشه درخت برنامه

مثال: الگوریتم Quick_Sort:



پیمایش DFS معادل ترتیب فراخوانی‌ها در درخت است.

از این گراف می‌توان به ساختمان داده‌ی پشته رسید. مثلاً در مثال فوق، می‌بینیم که باید پشته‌ای برای نگهداری اطلاعات مربوط به هر procedure در زمان اجرا داشته باشیم.



Activation Record ها در پشته کنترل (control Stack) قرار می‌گیرند و با عملیات Pop و Push حذف و اضافه می‌شوند.

Activation Record شامل Parameteres, local data, ... است.

برای رسم این گراف باید در زمان اجرا، دستورات فراخوانی توسط برنامه پیمایش شوند.

Data Object: محل حافظه‌ای که می‌توان مقداری به آن تخصیص داد. (شیء داده‌ای)

environment: name → (environment) → Storage

حافظه‌ای که به name تخصیص داده شده را برمی‌گرداند.

State: Storage → (State) → Value

حافظه‌ای را برمی‌گرداند که مقدار می‌گیرد.

(روش‌های مورد استفاده)

name: x
storage: 112
value: 15

Subject
Date ۹۴/۸/۱۷

x is bound to s ، متغیر x به حافظه s تخصیص داده شده ، یا x bind شده به s .

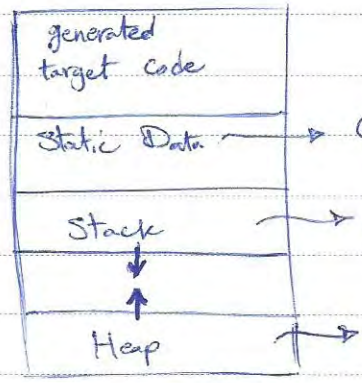
name → (environment) → Storage → (state) → Value

اگر رابطه بین متغیر و حافظه S ، bind شده .

feature های زبان به دو دسته تقسیم می‌شوند :

Static	Dynamic
Definition of a procedure	Activation of a procedure
Definition of a name	binding of a name
scope of a Declaration	lifetime of binding

حافظه هایی که برای یک برنامه لازم است ، بدین شرح است (!) :



حافظه مربوط به Stack ، Heap ثابت است ولی کوچک یا بزرگ شدن حافظه این دو برعکس یکدیگر است .

توان از این حافظه استفاده بهینه‌ای کرد .

در برنامه ها ، اگر حافظه های تخصیص داده شده ، آزاد نشود ، Heap را پر می‌کند . در این زبان‌ها garbage collector وجود دارد .

Subject

Date ۹۳/۸/۱۷

حفظ فراخوانی یک procedure اطلاعات زیر را لازم دارد:

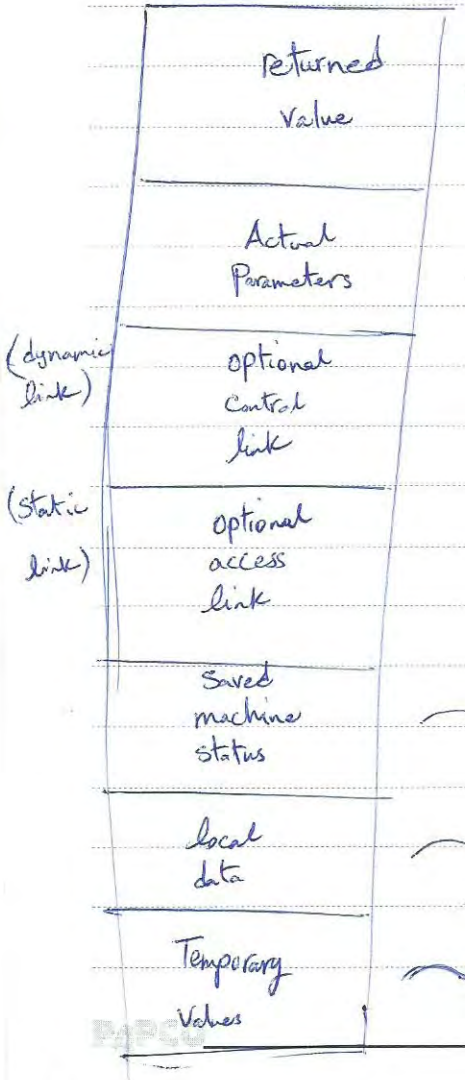
محتویات ثبات ها در هنگام فراخوانی را باید ذخیره کنیم (Status)

محل return های فراخوانی را هم ذخیره می کنیم.

پس از این که procedure، محتویات ثبات ها و Status ذخیره شده برگردد، دستورالعمل را انجام می دهد.

در Activation Record این اطلاعات ذخیره شود.

یک Activation Record شامل این موارد خواهد بود:



مقداری که تابع برمی گرداند

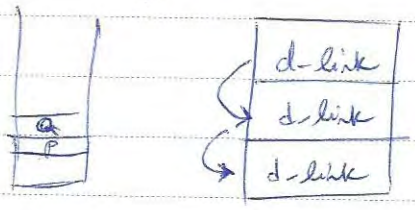
مقادیر ورودی تابع که ارسال می شود

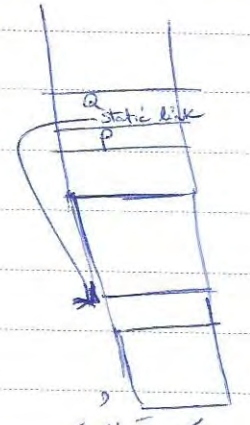
محتویات ثبات و محل بازگشت

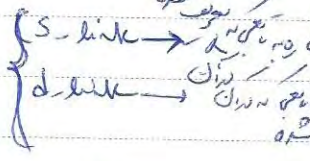
داده هایی که در زمان تابع تعریف شده

برای متغیرهای جدیدی که ممکن است در تابع تعریف شود.

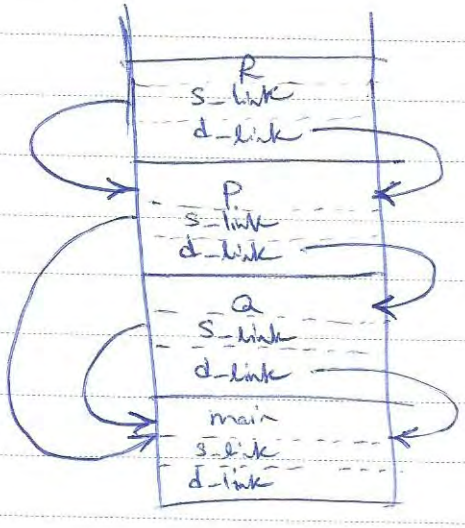
Subject
Date








```
program main
    proc P
        proc R
        begin
        end
    begin
        R
    end
    proc Q
    begin
        P
    end
begin
    Q
end
```



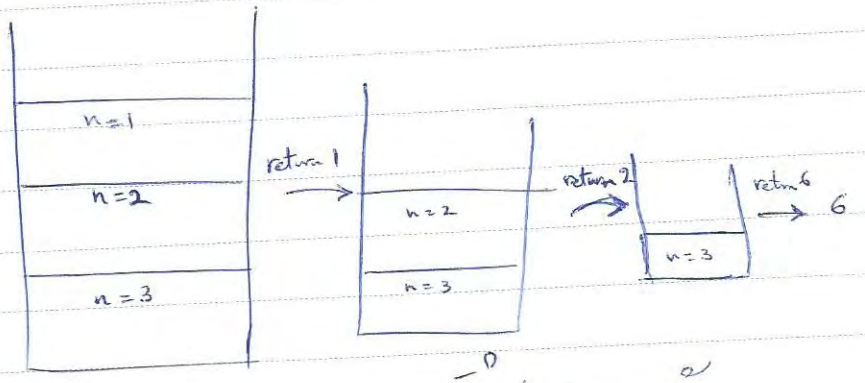
(*note that the nesting depth of q must be exactly one less than the nesting depth of p. (p is nested immediatly)

Subject:
Year. ۹۲ Month. ۸ Date ۱۷ ( )

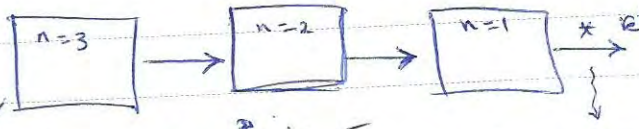
Recursion:

```
function nfact (n: integer)
  if n=1 then
      return (1)
  else
      return (n*nfact-(n-1))
```

nfact(3);
[4] ↑ ↓ [1]
nfact(2);
[3] ↑ ↓ [2]
nfact(1);



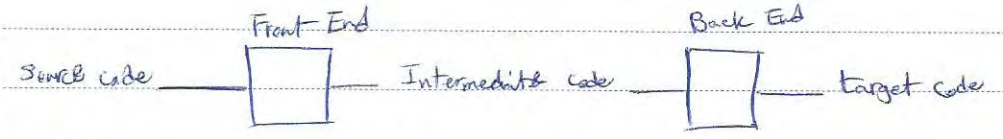
control stack

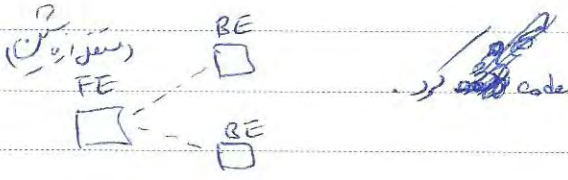


Subject:

Year. Month. Date. ( )

Intermediate Code Generation:





code optimization

Syntax tree

postfix

three address code

l: $x := y$ op z

$x + y * z \Longrightarrow$ $T_1 := y * z$

$T_2 := x + T_1$

(1) $x := y$ op z

→ binary arithmetic

logical operator

(2) $x :=$ op y

→ unary operator

- Unary minus

~ logical negation

<<, >> shift left, shift right

int_to_real int to real

(3) $x := y$

(4) Goto L

(5) if x relop y Goto L

↳ <, <=, =, >, >=, <>

(6) $p(x_1, x_2, \ldots, x_n)$ ⟹ Param x_1 (مقدار دهی آرگومان)

Param x_2

⋮

Param x_n

Call p, n

↙ procedure ↘ تعداد پارامترها

(مثال)

Param x_1

Param y_1 } → خود این یک فراخوانی بود

Param y_2 } و پارامترهای آن می‌شود.

Call Q, 2

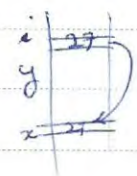
Param x_3

Call P, 3

return y : دستور برگشت

(7) $x = y[i]$

$x[i] := y$



(8) $x := \&y$ ⟶ آدرس y در متغیر x (که pointer است) قرار گیرد

$x := *y$ ⟶ مقدار y در متغیر x قرار گیرد

$*x := y$

(طبق ترتیب، ۱ تا ۴ دستورهایی که کمتر دیده شده‌اند)

Normal Forms: $A \to BC$, $A \to a$ — $A, B, C \in V$, $a \in T$

$A \to ax$ — $A \in V$, $x \in V^*$, $a \in T$

Subject:

Year. Month. Date.

$E \to E + E$

E.code

attribute

$E \to E + E$

E.code = 'T1 := y * z T2 := x + T1'

E.val

E.place

E.type

x + y + z

Syntax directed definition:

CFG + each grammar symbol has an associated set of attributes partitioned into two subsets called synthesized and inherited

Context-Free Grammar

Concatnation

$S \to id := E$ — S.code := E.code || gen(id.place ':=' E.place)

$E \to E_1 + E_2$ — E.place := new temp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)

$E \to E_1 * E_2$ — E.place := new temp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)

$E \to -E_1$ — E.place := new temp; E.code := E_1.code || gen(E.place ':=' 'Uminus' E_1.place)

$E \to (E_1)$ — E.place := E_1.place; E.code := E_1.code

$E \to id$ — E.place := id.place; E.code := ''

Subject:

Year. 9[۲] Month. ۸ Date 19 ( )

در gen ها نحوه تولید می‌شود و تمهای زیر از روش‌های تولید سری، استفاده می‌شود.

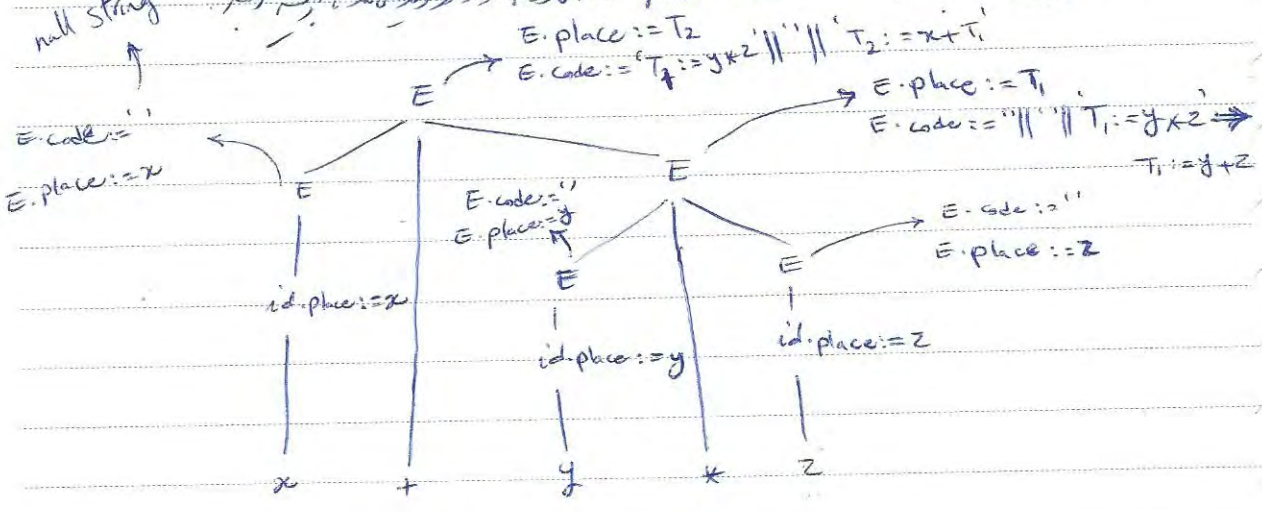
T_i new temp یک تابع است که در هر بار فراخوانی یک نام جدید می‌دهد. (یک نام متغیر جدید)

(در نحوه بررسی، نیازی به روش‌های متنی نیست، فقط روش‌های Sgen، Printf و ... نوشته می‌شود.

نحوه صدا زدن می‌شود و در نهایت Yacc کد مورد نظر تولید می‌کند.)

مثال: $x+y*z$ به پرسر روش (Yacc) داده می‌شود:

در روش بالا، Parse tree ساخته شده، و اطلاعات مورد نیاز پیوست می‌شود.



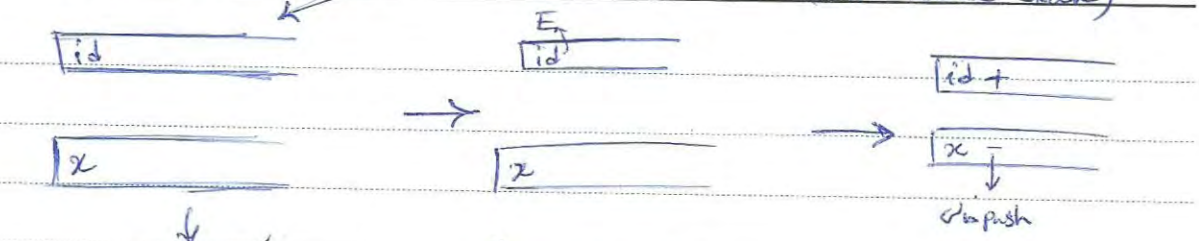
پس از طی کردن درخت، مقدار E.code شامل "$T_1:=y*z$ $T_2:=x+T_1$" می‌شود.

و در ادامه روش مورد استفاده در دوره را بررسی می‌کنیم: سیستم

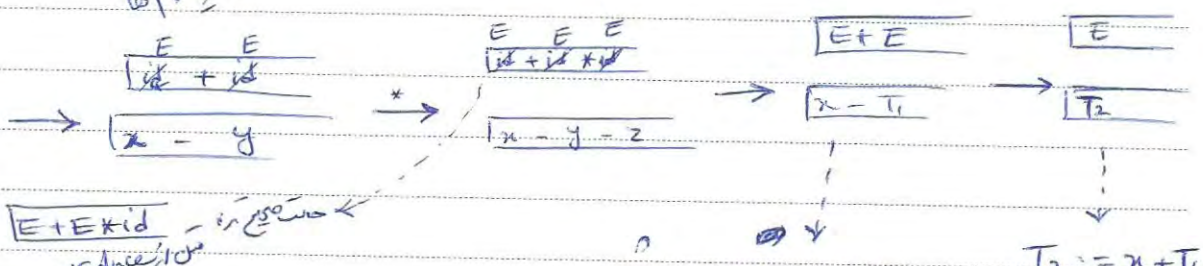
Subject:

Year. 98 Month. 8 Date. 2-24

(shift-reduce stack)



(reduce E : id)



E+E*id

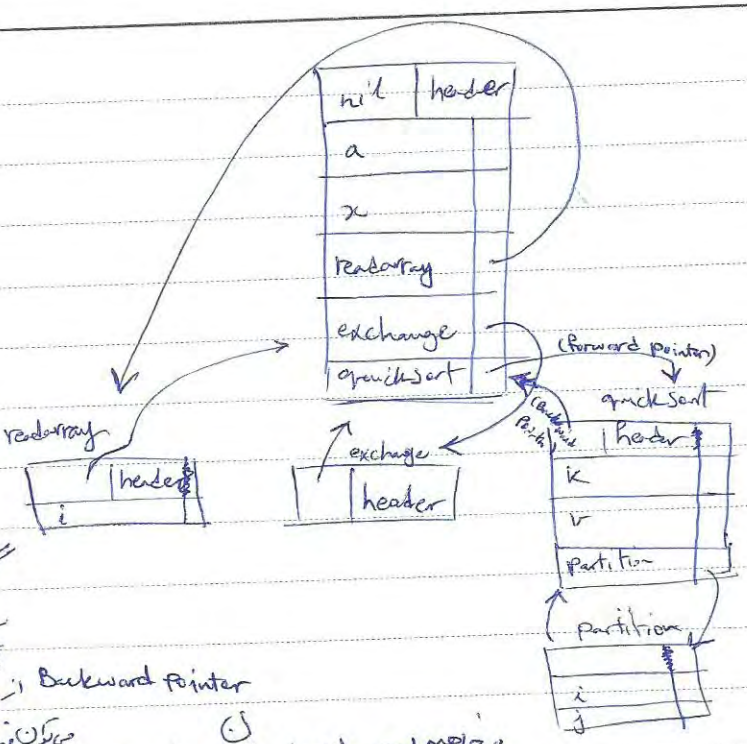
$T_1 := y \times z$

$T_2 := x + T_1$

: Symbol Table

```
program sort ( input, output)
:
var a : integer
    x : real
procedure read array :
    var i
    [
procedure exchange (i, j : integer) ;
    [
procedure quicksort (m, n : integer)
    var k, v : integer ;
    function partition (y, z : integer) : integer ;
        var i, j : integer.
        [
        [
        [
```

Subject:
Year. Month. Date.



$P \rightarrow MD$ {addwidth (top (tblptr), top(offset));
pop(tblptr); pop(offset); }

$M \rightarrow \varepsilon$ { t := mktable(nil); push (t, tblptr); push (0, offset)}

$D \rightarrow D_1 ; D_2$ { t := top (tblptr); addwidth (t, top (offset)); pop(tblptr);

$D \rightarrow$ proc id ; N D_1 ; S { ... pop(offset); enterproc(top(tblptr), idname, t) }

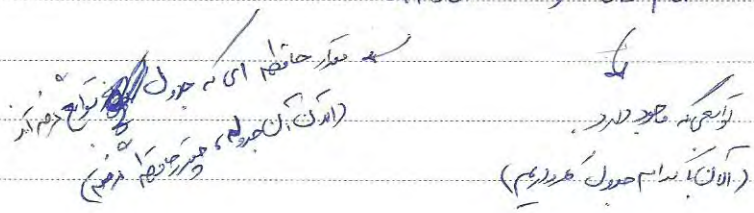
$D \rightarrow$ id : T { enter (top (tblptr), idname, T.type, top(offset));
top(offset) := top(offset) + t.width }

$N \rightarrow \varepsilon$ { t := mktable (top(tblptr)); push (t, tblptr); push (0, offset)
}

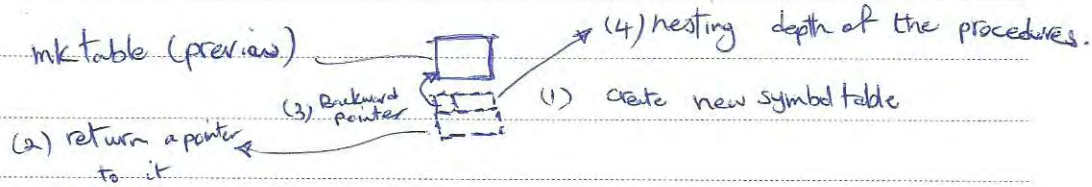
Subject:

Year. 93 Month. 8 Date 24

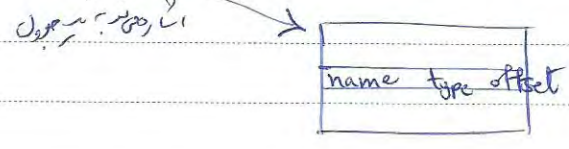
offset , tblptr ... Stack ...



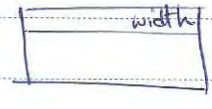
mktable (previous)



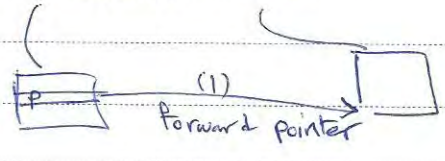
enter (table, name, type, offset)



addwidth (table, width)



enterproc (table, name, newtable)



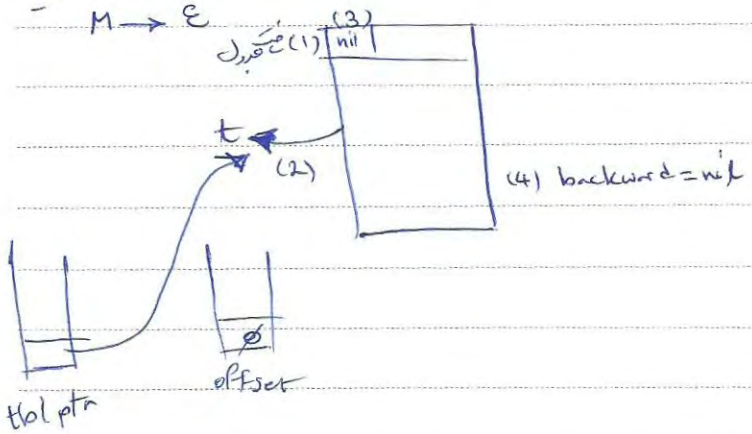
DwordDeclaration → تعریف متغیر

→ تعریف متغیر

Subject:
Year 93 Month 8 Date 24 ( )

reduction

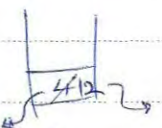
$M \rightarrow \varepsilon$



(5)

$D \rightarrow id : T$

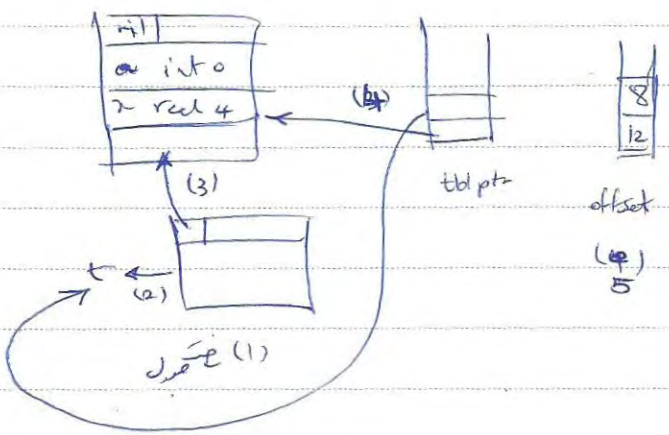
(1), (3)



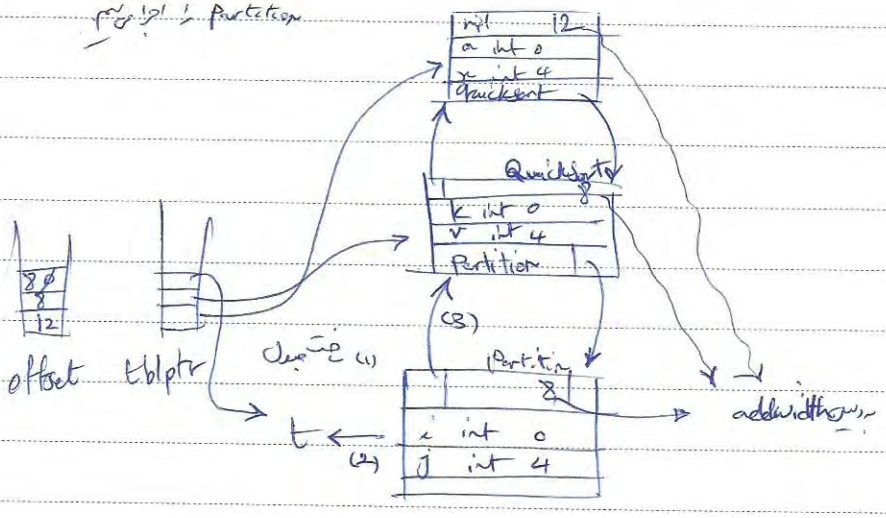
int + real = 4 + 8 = 12

offset (2), (4)

$D \rightarrow proc\ id : ND_1 ; S$



Subject:
Year. 93 Month. 8 Date 24 ( )



Pop ... Stack ...

... Partition ... Forward Pointer ... Ebp ...

... addwidth ... Quicksort ...

Pop ...

... forward pointer ... quicksort

quicksort ...

... Forward و Backward ... quicksort ...

... identifier ... (lookup(id)) ...

... قواعد ...

Subject:
Year. 93 Month. 8 Date. 26 ( )

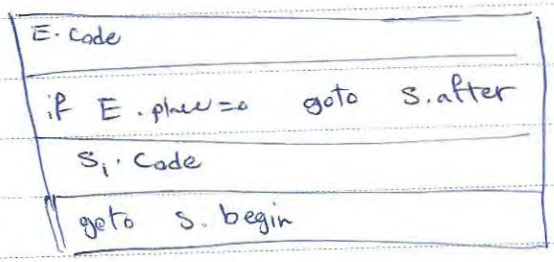
$S \rightarrow id := E$ { p := lookup (id.name);
if p ≠ nil then emit (p ':=' E.place)
else error }

$E \rightarrow id$ { p := lookup (id.name);
if p ≠ nil then emit (p ':=' E.place)
else error }

پس زمانی که Parser با Symbol Table کار می‌کند، برای دسترسی به آن باید روال‌هایی مثل lookup و insert را داشته باشد. Scope به صورت پشته‌ای می‌شود.

می‌خواهیم برای چند ساختار کد تولید کنیم.

$S \rightarrow$ while E do S_1



Semantic rule:

S.begin = new label;
S.after = new label;
S.code = gen(S.begin ':') || E.code ||
gen('if' E.place '=' '0' goto S.after) ||
S_1.code ||
gen('goto' S.begin) || gen(S.after ':')

Subject:

Year. Month. Date. ( )

در استفاده از ترجمه دستوری برای:

Code E

code S1

S.begin ':'

if E place '=' '0' goto S.after

'goto' S.begin

S.after ':'

این روش ترجمه برای while به این روش را trick ترجمه می‌گویند!

(ترجمه Symbol Table به روش ترجمه‌ای):

P → M D

M → ε [offset := 0]

D → D ; D

D → id : T [enter (id.name, T.type, offset); offset := offset + T.width;

T → integer [T.type := integer; T.width := 4]

T → real [T.type := real; T.width := 8]

که می‌خواهیم از یک متغیر موقتی به متغیرهای موقتی از new temp() می‌زنیم و reuse می‌کنیم.

c = 0

$c

c = c - 1

یک راه این است که:

PaPCO

فایل‌ها از متغیرهای موقت استفاده می‌کنند!!

(Temporary Variable Used)

Subject:
Year 93 Month 8 Date 26 ( )

وقتی متغیری تولید می‌شود:

$c

c = c + 1 (Temporary Generated)

مثال:

x := a*b + c*d - e*f

Statement	Value of c
	0
$0 := a*b	1
$1 := c*d	2
$0 := $0 + $1	1
$1 := e*f	2
$0 := $0 - $1	1
x := $0	

$0, $1 استفاده می‌شود و $0 هم اکد می‌شود. پس تنها یکی می‌شود. (-1)

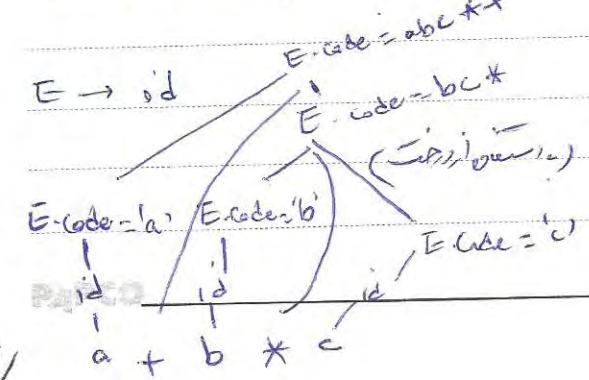
مثال بالا: به ترتیب ترجمه می‌شود.

اشکال روش بالا: وابسته به ماشین هدف است. ما نمی‌خواهیم.

می‌خواهیم یک بیاییم که infix را postfix تحویل بدهد:

Production	Semantic action
$E \to E^{(1)}\ op\ E^{(2)}$	$E.code := E^{(1)}.code \,||\, E^{(2)}.code \,||\, 'op'$
$E \to (E^{(1)})$	$E.code := E^{(1)}.code$
$E \to 'id'$	$E.code := 'id'$

مثال: a+b*c ← abc*+



(بازنویسی رشته، شبیه reduce در رشته جملات)

Subject:
Year. 93 Month. 9 Date. 1 ( )

۳ روش برای نمادگذاری بدینی در Single pass کامپایلر: (نمادگذاری درواسطه)

1) Quadruples 2) Triples 3) Indirect triples

مثال: $A := -B * (C + D)$

(کد سه آدرسی)

$T_1 := -B$

$T_2 := C + D$

$T_3 := T_1 * T_2$

$A := T_3$

Quadruples:

	op	Arg1	Arg2	result
(0)	Uminus	B	—	T_1
(1)	+	C	D	T_2
(2)	*	T_1	T_2	T_3
(3)	:=	T_3	—	A

(در دوره از این روش استفاده می‌کنیم)

سپس از برنامه مثال می‌زنیم

عمل

در ردیف سوم result را خوانده و جای نشانی مطلب قرار می‌دهد.

Triples:

	op	Arg1	Arg2
(0)	Uminus	B	—
(1)	+	C	D
(2)	*	(0)	(1)
(3)	:=	A	(2)

در این روش (جدول) دیگر از T_1 و T_2 ... استفاده نمی‌کنیم.

(0), (1), ... را می‌نویسیم.

در این مثال کاملا مشخص است که چه چیزی جای چه چیزی قرار می‌گیرد.

~~در سطری با این تفاوت که در جدول دوم از ... استفاده نمی‌کنیم~~

PAPCO

(ادامه مطلب)

Subject:
Year. Month. Date. ( )

Indirect Triples:

	op	Arg1	Arg2
(14)	Uminus	B	-
(15)	+	C	D
(16)	*	(14)	(15)
(17)	:=	A	(16)

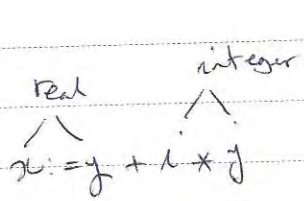
	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)

	op	arg1	arg2
(0)	=[]	y	i
(1)	assign	x	(0)

$x := y(i)$

	op	arg1	arg2
(0)	[]=	x	i
(1)	assign	(0)	y

$x[i] := y$



$t_1 := i$ int* j

$t_3 := \text{int\_to\_real}\ t_1$

$t_2 := y$ real+ t_3

$x := t_2$

Subject:

Year. Month. Date. ( )

$E \rightarrow E_1 + E_2$

$E.type \rightsquigarrow$ attribute

```
E.place := new temp;
if E1.type = integer and E2.type = integer then
    begin
    emit(E.place ':=' E1.place 'int+' E2.place);
    E.type := integer;
else end
;
else if E1.type = integer and E2.type = real then
    begin
    u := new temp;
    emit(u ':=' 'int to real' E1.place);
    emit(E.place ':=' u 'real+' E2.place);
    E.type := real;
end
;
```

PAPCO

Subject:
Year. Month. Date. ( )

Boolean expression

and or not

E → E or E | E and E | not E | (E) | id relop id | true | false

not
and
or

true ≠0 ≥0
false 0 0 <0

Pascal C

k = 2;
if x=4 or g(y) then
[

function g(—)
[k=16;

side effect

Borland C

and then

Subject:

Year. Month. Date. ( )

۱۰/۱

مرور سریع بر ترجمه Boolean Expressions :

مثال:

روش ۱)

a < b

Quadruple ← سه آدرسی

100 : if a < b goto 103

101 : t := 0

102 : goto 104

103 : t := 1

104 :

(استفاده از t و مقدار دهی خروجی)

$E \rightarrow E_1$ or E_2

{ E.place := newtemp;

emit(E.place ':=' E_1.place 'or' E_2.place); }

$E \rightarrow E_1$ and E_2 { E.place = new temp;

emit(E.place ':=' E_1.place 'and' E_2.place); }

$E \rightarrow$ not E_1

{ E.place = new temp;

emit (E.place ':=' 'not' E_1.place);

$E \rightarrow (E_1)$ { E.place = E_1.place; }

~~[illegible]~~

$E \rightarrow id_1$ relop id_2

|<|<=|=|#|>|>=|

E.place = new temp

emit ('if' id_1.place relop.op id_2.place,

'goto' nextstate +3);

emit (E.place ':=' '0');

emit ('goto' nextstate+2)

emit (E.place ':=' '1'); }

nextstate ← اشاره‌گر

PdPCO

Program Counter

Quadruple

Subject:
Year. ۹۷ Month. ۹ Date. ۳ ( )

E → true { E.place = newtemp;
emit (E.place ':=' '1'); }

E → false { E.place = newtemp;
emit (E.place ':=' '0'); }

روش ۲) E طوری ترجمه می‌شود که اگر مقدار آن true است، کنترل به E.true و اگر false است، به E.false پرش کند.

(E.false, E.true برای label های استفاده می‌شود)

if a<b goto E.true
goto E.false

E.true: _____

E.false: _____

Semantic Rules:

E → E1 or E2
E1.true := E.true;
E1.false := newlabel;
E2.true := E.true;
E2.false := E.false;
E.code := E1.code || gen(E1.false ':') || E2.code;

Subject:

Year. Month. Date. ( )

$E \rightarrow E_1$ and E_2 — $E_1.true := newlabel;$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \,||\, gen(E_1.true ':') \,||\, E_2.code;$

$E \rightarrow$ not E_1 — $E_1.true := E.false;$ $E_1.false := E.true;$ $E.code := E_1.code;$

$E \rightarrow (E_1)$ — $E_1.true := E.true;$ $E_1.false := E.false;$ $E.code := E_1.code;$

$E \rightarrow id_1$ relop id_2 — $E.code := gen('if' id_1.place relop.op id_2.place 'goto' E.true) \,||\, gen('goto' E.false)$

$E \rightarrow$ true — $E.code := gen('goto' E.true)$

$E \rightarrow$ false — $E.code := gen('goto' E.false)$

OR-ی:

E (E_1.false:)
- E_1: goto $E_1.true$ → $E.true$
- E_1: goto $E_1.false$ → $E.false$
- E_2: goto $E_2.true$ → $E.true$

(سورکد ... سورکد ... label جدید ... $E_1.true$ ... $E_2.true$ ... $E.false$ ... $E.true$ ...)

Subject: Compilers
Year. ۹۳ Month. ۹ Date. ۴ ( )

Mixed mode Boolean Expression:

false = 0
true = 1

$(a<b) + (b<a)$ — if a<b then 0, else 1

$E \rightarrow E_1 + E_2$

$E_2.true$: E.place := E_1.place + 1
goto nextstate+1
$E_2.false$: E.place := E_1.place

(این کد در شرایطی که E_2 درست شد و عبارت E_1 صفر شود و E_2 عبارت منفی و پس اگر E_2 درست شد، عبارت E_1 صفر شود و یک می‌شود و درنتیجه با E_1 جمع می‌شود.)

E.type := arith;
if E_1.type = arith and E_2.type = arith then begin
E.place := new temp;
E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)
end
else if E_1.type = arith and E_2.type = bool then begin
E.place := new temp;
E_2.true := new label;
E_2.false := new label;
E.code := E_1.code || E_2.code || gen(E_2.true ':' E.place ':=' E_1.place + 1) || gen('goto' nextstate+1) || gen(E_2.false ':' E.place ':=' E_1.place)
end

Subject:
Year. Month. Date. ( )

روش ۳)

اول خروجی بصورت زیر خواهد بود.

Back patching:

goto L ⟶ 100: goto 407

104 → L

241

goto L ⟶ 211: goto 407

L: ... 407: ...

در حالت عادی با رسیدن به L پرش می‌کند.

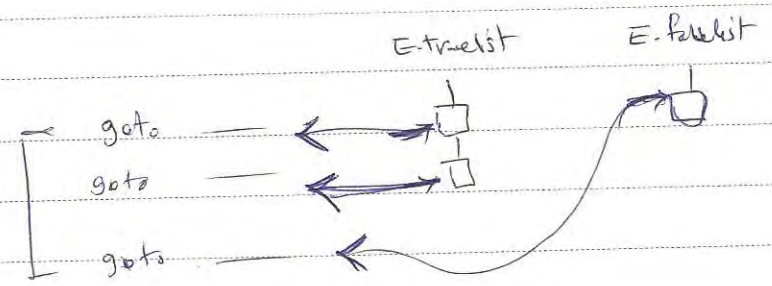
makelist (i) ⟶

merge (P_1, P_2) ⟶ ادغام دو لیست

backpatch (P, i) ⟶ پر کردن جاهای خالی با i

E. truelist

E. falselist



Subject:
Year. ۹۴ Month. ۹ Date. ۳ ( )

$E \rightarrow E_1$ or M E_2 { backpatch (E_1.falselist, M.quad);
E.truelist := merge (E_1.truelist, E_2.truelist);
E.falselist := E_2.falselist }

$E \rightarrow E_1$ and M E_2 { backpatch (E_1.truelist, M.quad);
E.truelist := E_2.truelist;
E.falselist := merge (E_1.falselist, E_2.falselist); }

$E \rightarrow$ not E_1 { E.truelist := E_1.falselist;
E.falselist := E_1.truelist;
}

$E \rightarrow id_1$ relop id_2 { E.truelist := makelist (nextquad);
E.falselist := makelist (nextquad+1);
emit ('if' id_1.place relop.op id_2.place 'goto —');
emit ('goto —'); }

$E \rightarrow$ true { E.truelist = makelist (nextquad);
emit ('goto —'); }

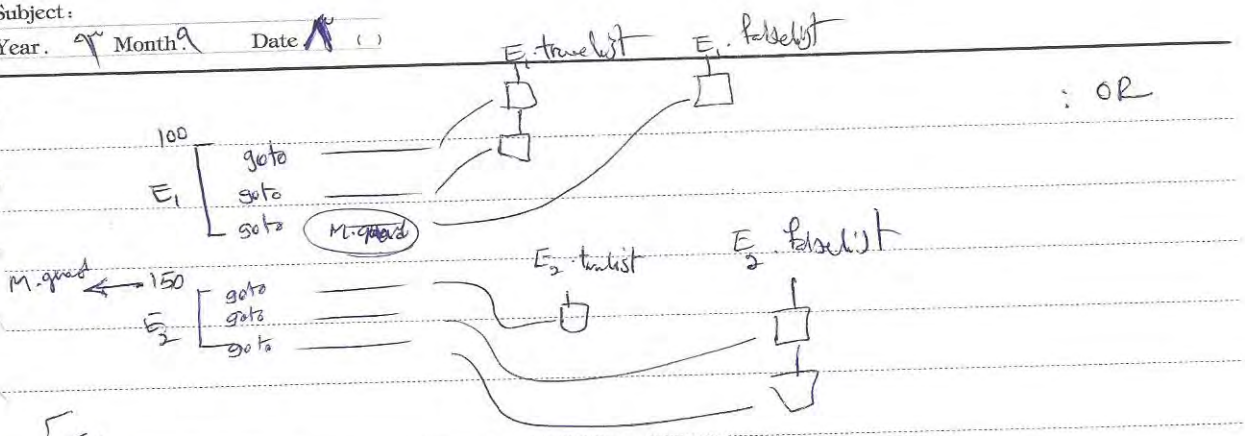
$E \rightarrow$ false { E.falselist = makelist (nextquad);
emit ('goto —') }

$M \rightarrow \varepsilon$ { M.quad := nextquad }

100 if a<b goto —
101 goto —

(The marker nonterminal M records the current value of nextquad)

Subject:
Year. Month. Date ( )



E.falselist ← to falselist ← AND ← E1.falselist, E2.falselist

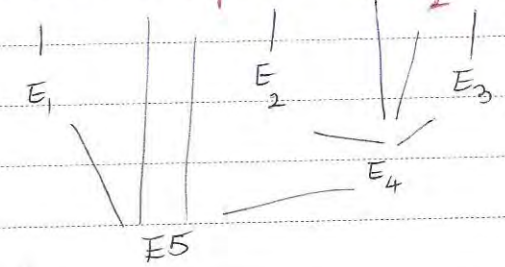
E.truelist ← to truelist ← OR ← E1.truelist, E2.truelist

relop :

E.falselist, E.truelist ← goto

: مثال

a<b or M1 c<d and M2 e<f



E_1.truelist	E_1.falselist
100	101

100 شروع :

$E_1 \rightarrow a<b$

100 if a<b goto —

101 goto 102

$M_1 \rightarrow \varepsilon$

M_1.quad = 102

$E_2 \rightarrow c<d$

102 if c<d goto 104

103 goto —

E_2.t.list	E_2.f.list
102	103

$M_2 \rightarrow \varepsilon$

M_2.quad = 104

$E_3 \rightarrow e<f$

104 if e<f goto —

105 goto —

E_3.t.l	E_3.f.l
104	105

Subject:

Year 93 Month 9 Date 8 ( )

$E_4 \rightarrow E_2$ and M_2 E_3 } 102 پرش به 104

E_4.T.l	E_4.F.l
104	103
	105

$E_5 \rightarrow E_1$ or M_1 E_4 } 101 پرش به 102

E_5.T.l	E_5.F.l
100	103
104	105

Statement:

$S \rightarrow$ if E then S

| if E then S else S

| while E do S

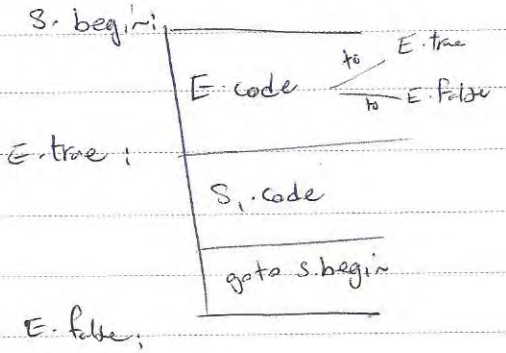
| begin L end

| A

$L \rightarrow L ; S$

| S

while برای:



$M \rightarrow \varepsilon$ { M.quad := nextquad } E.false;

Subject:

Year. Month. Date. ( )

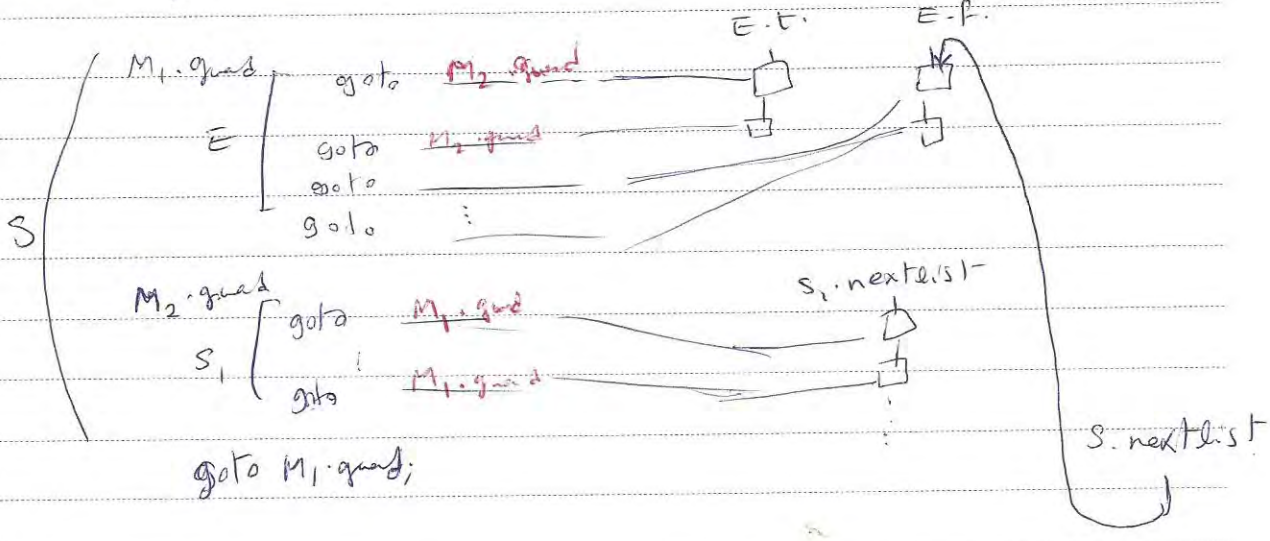
S → while M_1 E do M_2 S_1

{ backpatch (S_1.nextlist, M_1.quad);

backpatch (E.truelist, M_2.quad);

S.nextlist := E.falselist;

emit('goto' M_1.quad) }



S → call id (Elist)

Elist → Elist , E

Elist → E

E_1 [T_1 → E_1 ...

E_2 [T_2 → E_2 ...

E_n [T_n → E_n ...

param T_1

param T_2

param T_n

call p,n

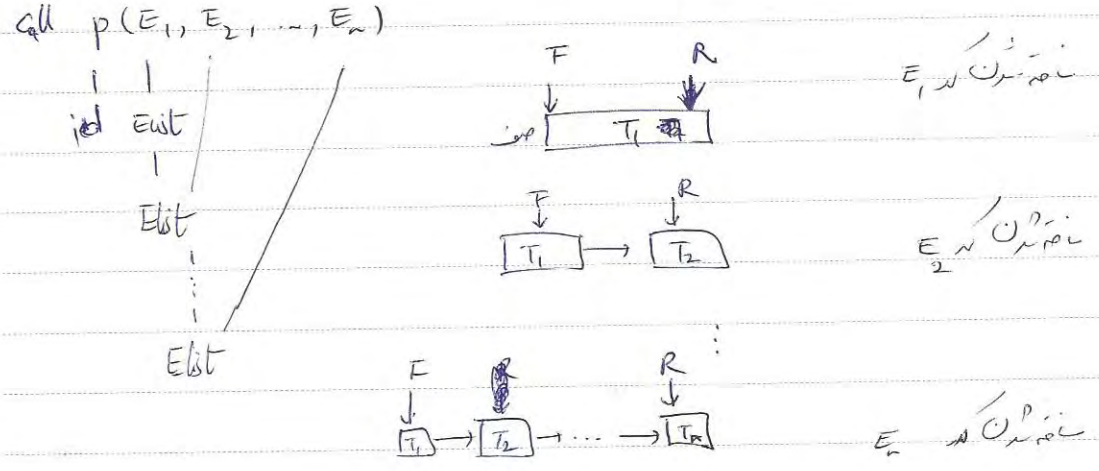
p (E_1, E_2, ... , E_n)

Subject:
Year. Month. Date. ( )

Elist → E { code for E
initialize queue to contain only E.place }

Elist → Elist, E { code for E
append E.place to the end of queue. }

S → call id (Elist) { for each item p on queue do
emit ('param p');
emit ('call' id.place); }



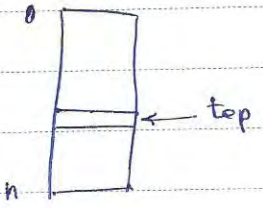
پس از کد نهایی، پس از آن یک for برای E ها قرار می‌گیرد، param ها چاپ می‌شود.

سپس (Counter) تعداد پارامترها چاپ می‌شود. call p,n چاپ می‌شود.

Subject: 9/29/17 ← (1)

Year. Month. Date:

Stack



push(x) : top = top − 1 ;

*top = x ;

Param T ⟶ push (T)

param T_1
Param T_2
⋮
Param T_n
Call p,n

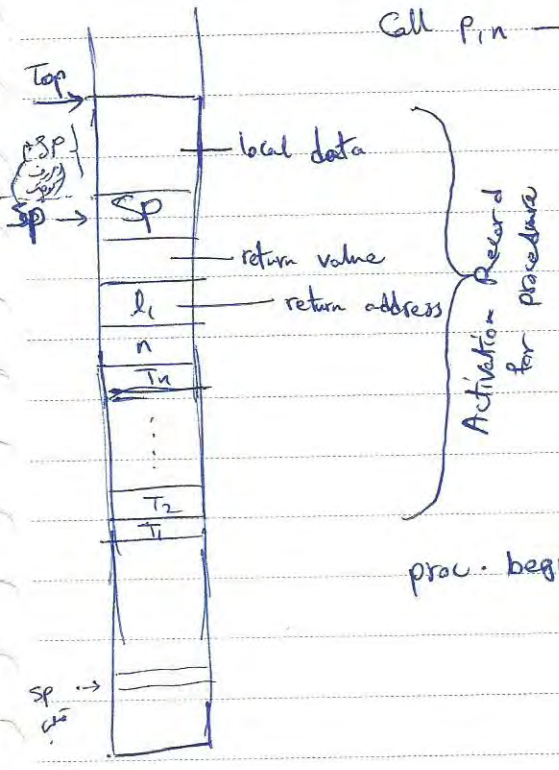
Call p,n ⟶ push (n) /* Store the arg Count */

push(l_1) /* l_1 is the label of the return address */

push( ) /* leave space for the return value */

push (sp) /* store the old stack pointer */

goto l_2 /* l_2 is the first statement of the called procedure */



proc. begin ⇒

Sp := Top

Top := Sp − sp /* sp is the size of P, is # of words taken by the local data for P */

return (exp) ⇒

A[Sp+1] ⟵ 1 [sp] := T /* 1 is the offset for the location of the return value */

Top := Sp + 2 /* Top now points to the return address */

Sp := * Sp /* restore sp */

L := * Top /* The value of L is now the return address */

Subject:
Year. 9۴ Month. 9 Date. ۱۰

Top := Top + 1 /* Top points to the arg counter */

Top := Top + 1 + *Top /* *Top is the # of parameters of P */

Goto L

Borland C :

short jump

long jump

Recursive descent parser, predictive parser, LL(k), LR(k) : Parser

LL(k) → lookahead
↳ Left-most derivation
↳ left-to-right scan

LR(k) → lookahead
↳ Right most derivation in reverse
↳ left-to-right scan

S → cAd

A → ab | a

Recursive Descent Parser

Subject:

Year. 4۲ Month. 9 Date. 1۰ ( )

```
procedure S();
begin
  if input_symbol = 'c' then
    begin
      ADVANCE(); ~~~> cursor حرکت
      if A() then
        if input_symbol = 'd' then
          begin
            ADVANCE();
            return true;
          end
    end
  return false;
end
```

```
procedure A();
begin
  isave := input_pointer;
  if input_symbol = 'a' then
  begin
      ADVANCE();
      if input_symbol = 'b' then
      begin
          ADVANCE();
          return true;
      end
  end
    input_pointer := isave;
```

Subject:

Year 9۳ Month. 9 Date. ۱۵ ( )

```
if input_Symbol = 'a' then
begin
    ADVANCE();
    return true;
end
else
    return false;
end
```

(:)

سوال این تمرین از روش Recursive Descent برای پیاده‌سازی پارسر است.

هر تابع پارسر یک procedure (یا تابع) است که از ADVANCE برای خواندن توکن بعدی استفاده می‌کند.

مشکلات این روش:

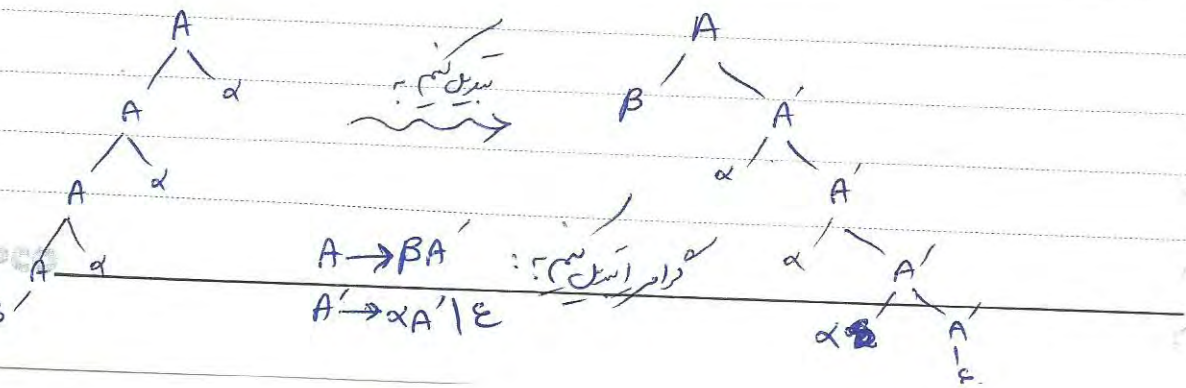
- اگر گرامر Recursion (چپ‌گرد) داشته باشد، به مشکل می‌خورد! (Left recursion)
- نیاز به Back Tracking دارد.

مثلاً در رشته‌ی ورودی Ca db اگر قانون S → Sd|a را اعمال کنیم، پیشوند Ca درست است و return true می‌دهد، اما کل رشته پذیرفته نمی‌شود.

باید بررسی کرد که رشته تمام شده باشد و سپس return true داده شود.

راه‌حل برای left recursion:

$A \rightarrow A\alpha \mid \beta$



Subject:

Year. Month. Date. ( )

(۱) حذف left Rec. (مثال) ... حذف right recursion - مثال

$$E \to E+T \mid F \quad (\alpha \mid \beta) \longrightarrow E \to TE'$$
$$E' \to +TE' \mid \varepsilon$$

$$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

↓ حذف

$$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$$

حذف immediate left Rec.

$S \to Aa$
$A \to Sb$ ← non-immediate left-Rec.

راه حل این مثال (وقت گیر است!)

راه حل دوم: Back Tracking

$S \to ab \mid ac$ → LL(1) نیست ← با یک bakahead نمی‌توان تصمیم گرفت

$S \to ab \mid cb$ → LL(1) است

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

↓

$A \to \alpha A'$
$A' \to \beta_1 \mid \beta_2$

با این روش مشکل حل می‌شد: (left-factoring)

آیا همه left-factoring ها را می‌توان حذف کرد؟

PdPCO

Subject:
Year. ۹۴ Month. ۹ Date. ۱۵ ( )

آیا همه‌ی CFG ها را می‌توان با left-factoring برای Back Tracking آماده کرد؟ (مثال در پارسر) خیر!

stmt → if exp then stmt | if exp then stmt else stmt | a

exp → b

Factoring

stmt → if exp then stmt stmt' | a

stmt' → else stmt | ε

exp → b

if b then a else

این مشکل را می‌توان به زبان ماشین منتقل کرد اما در کامپایلر حل نمی‌شود.

Non-Context-Free Language Constructs:

$L_1 = \{wcw \mid w \in (a|b)^*\}$

$L_2 = \{a^n b^m c^n d^m \mid n, m \geq 1\}$ هیچ‌کدام از زبان‌های CF نیست (Context-Free)

$L_3 = \{a^n b^n c^n \mid n \geq 0\}$

ثابت اینکه این زبان CF نیست، نشان می‌دهد که ماشین‌هایی که تعریف شده‌اند اصلاً قادر به تشخیص آن نیستند.

$x = w \cdots x = a$

$L_1' = \{wcw^R \mid w \in (db)^*\}$ هم CF است.

$S \rightarrow aSa \mid bsb \mid \varepsilon$

Subject:
Year. Month. Date. ( )

L2 نشان دهنده ساختار procedure call در زبان برنامه نویسی است.

proc P($a_1, \dots, a_n$) [...] → definition
proc Q($b_1, \dots, b_m$) [...]

CFG می‌تواند دو حالت مراجعه زیر را داشته باشد.

Call P($x_1, \dots, x_n$) ، Call هر کدام که تعداد پارامترهای ورودی آن با تعداد پارامترهای ورودی تعریف شده برابر است.
Call Q($y_1, \dots, y_m$)

مثال:

$L_2' = \{a^n b^m c^m d^n \mid n, m \geq 1\}$

$S \to aSd \mid aAd$

$A \to bAc \mid bc$

$L_2'' = \{a^n b^n c^m d^m \mid n, m \geq 1\}$

$S \to AB$

$A \to aAb \mid ab$

$B \to cBd \mid cd$

L3 نشان دهنده n تا کاراکتر است، سپس n تا بار عقب می‌رویم و روی تک تک کاراکترها underbar (_) می‌گذاریم، (این هم حالتی برای پاک کردن کاراکترها در Type writer) نیز CFG دارد.

مثال: $(abc)^n$ ، به این صورت که با کاراکتر a، سپس backspace و سپس underbar می‌زنیم و روی b نیز همین کار را انجام می‌دهیم، CFG می‌نویسیم.

$S \to abcS \mid abc$

PAPCO

Subject:
Year. ۹۳ Month: ۹ Date: ۱۵ ( )

راه حل حذف left-Recursion غیرمستقیم: (این شیوه در صفحه ۷۹ کتاب "دقت کامپایلر" است.)

حالت مستقیم: $A \rightarrow A\alpha \mid \beta \Longrightarrow A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \varepsilon$

حالت غیرمستقیم: $S \rightarrow Aa \mid b$, $A \rightarrow Ac \mid Sd \mid \varepsilon$

الگوریتم حذف بازگشت غیرمستقیم:

شروط لازم برای کارکرد این الگوریتم: ۱) حلقه نداشته باشد. $A \xrightarrow{+} A$ را نداشته باشد.

۲) قاعده‌ی $A \rightarrow \varepsilon$ را نداشته باشد.

(ریشه در Compiler Writing نوشته Sorenson دارد و در کتاب ایشان این روش را توضیح داده است)

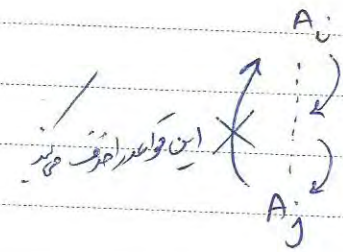
شرح الگوریتم:

1. Arrange the nonterminals in some order $A_1 \dots A_n$
2. for $i := 1$ to n do begin
   - for $j := 1$ to $i-1$ do begin
     - replace each production of the form $A_i \rightarrow A_j \gamma$ by the production $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all the current A_j-productions.
   - end
   - eliminate the immediate left recursion among the A_i-productions
3. end

۷۹/

Subject:

Year. Month. Date. ( )



این قواعد اجازه می‌دهد

(ترکیب از terminal و non-terminal)

مثال را با این روش انجام می‌دهیم:

S A ← non-terminal

A_1 A_2

(استثنا: این قاعده را به برنامه‌ها نمی‌دهد)

پس:

$A_1 \rightarrow A_2 a \mid b$

$A_2 \rightarrow A_2 c \mid A_1 d \mid \varepsilon$

$i := 1 \rightarrow$ هیچ قواعد A_1 وجود ندارد $\rightarrow$ برنامه را می‌گیرد

$i := 2 \rightarrow j := 1 \rightarrow$ $A_2 \rightarrow A_2 c \mid A_2 a d \mid bd \mid \varepsilon$ (α_1 α_2 β_1 β_2)

$A_1 \rightarrow A_2 a \mid b$

پس هیچ قواعد A_2

$A_2 \rightarrow bdA_2' \mid \varepsilon A_2'$

$A_2' \rightarrow cA_2' \mid adA_2' \mid \varepsilon$

✓ دیاگرام انتقال: Transition Diagram رسم کرده و پس آن را پیاده‌سازی کنیم که ماشین‌های تجزیه‌گر ساخته شود.

Subject:
Year. Month. Date. ( )

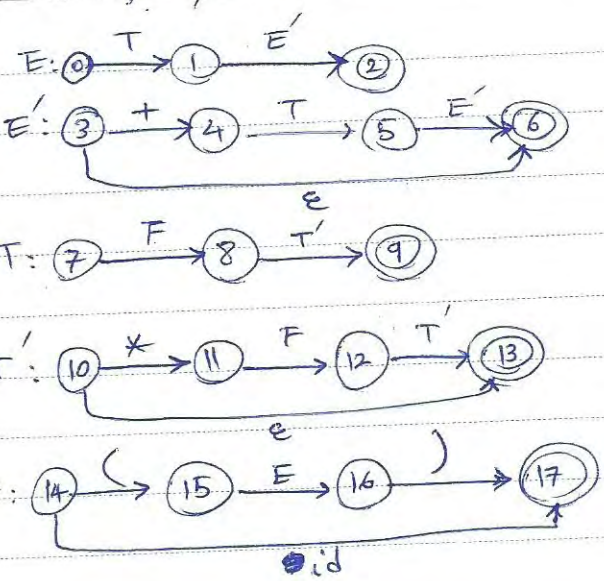
$E \to TE'$

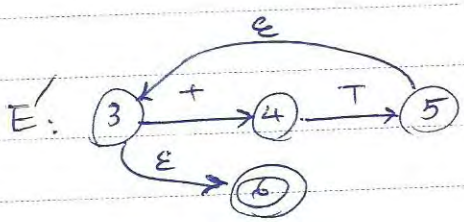
$E' \to +TE' \mid \varepsilon$

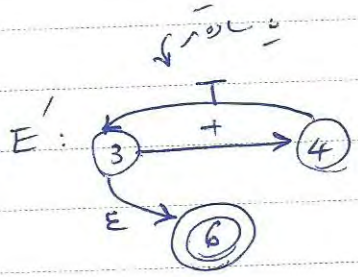
$T \to FT'$

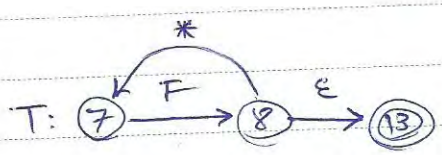
$T' \to *FT' \mid \varepsilon$

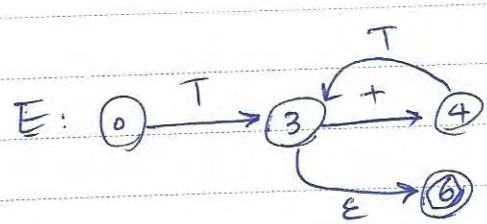
$F \to (E) \mid id$

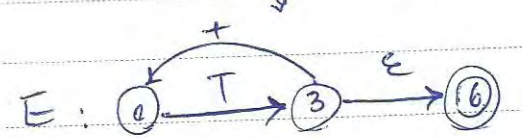












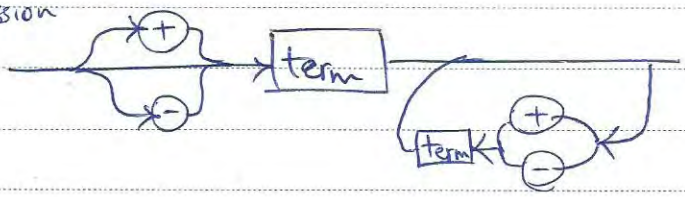
Subject:
Year. 9۳ Month. 4 Date. ۱۵ ( )

روشی دیگر برای پیاده‌سازی تحلیل‌گر نحوی (درخت با فعال‌سازی شده):

Syntax Diagram:

دایره ← terminal
مستطیل ← non-terminal

Expression

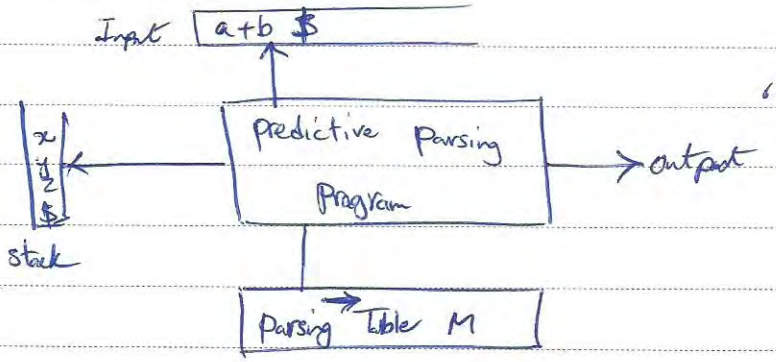


Term

لینک:
primepuzzle.com/tp2/syntax_diagrams.html

Predictive Parser:

(non-recursive predictive parsing)



زبان‌ها، stack استفاده می‌کند،
و سطر و ستون را برمی‌گرداند.

برای شروع، Start Symbol ← S / $

Subject: (Recursive Descent parser)
Year. Month. Date. ( )

(non-recursive predictive parser)

Start Symbol ← S
dollar ← $

input: a

Stack: x

حالت‌های ممکن:

(1) $x = a = \$$ ⟶ parser halts and announces successful completion

(Top-Down)

(2) $x = a \neq \$$ ⟶ Pop x off the stack, and advance the input pointer

(3) x is a non-terminal ⟶ $M[x, a] : \{x \to uvw\}$ ، تعریف شده

x ⟹ u v w

$x \to uvw$ را برمی‌دارد a و x را می‌بیند، $M[x,a]$ در جدول به سطر x و ستون a

(حرکت left-most derivation است، پس از چپ به راست، روی رشته می‌رود)

LL(k)

گرامر زیر را در نظر می‌گیریم:

$E \to TE'$

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

$T \to FT'$

$T' \to *FT' \mid \varepsilon$

$F \to (E) \mid id$

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Stack	input
$E	id+id*id$
$E'T	id+id*id$
⋮	⋮
$	$

روش‌های تجزیه به دو دسته تقسیم می‌شوند:

Object:
Year. Month. Date. ( )

First, Follow ؟ حيث يستخدم هذا

α any string of grammar symbol

$First(\alpha) = \{a \mid \alpha \xrightarrow{*} aw$ and a is a terminal or $\alpha \xrightarrow{*} \varepsilon, a = \varepsilon\}$

$Follow(A) = \{a \mid$ a is a terminal symbol
Non-Terminal $S \xrightarrow{*} \alpha A a \beta$ for some α and β
or
$a = \$$ and A can be a right most symbol
in some right sentential form $\}$

$First(X)$ for all grammar symbol X

(1) if X is a terminal $\Rightarrow First(X) = \{X\}$

(2) if $X \rightarrow \varepsilon \Rightarrow \varepsilon \in First(X)$

(3) if X is a non-terminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$

if $\exists i \quad \exists a \in First(Y_i)$
$\varepsilon \in First(Y_1), \varepsilon \in First(Y_2), \cdots, \varepsilon \in First(Y_{i-1})$
$\Downarrow$
$a \in First(X)$

if $\forall j = 1, \cdots k \quad \varepsilon \in First(Y_j) \Rightarrow \varepsilon \in First(X)$

PAPCO

Subject:

Year. ۹۲ Month. ۹ Date. ۲۴ ( )

مثال سبق)

توضیح کامل مثال در صفحه ۸۲

stack	input
$E	id+id*id$
$E'T	id+id*id$
$E'T'F	id+id*id$
$E'T'id	id+id*id$
$E'T'	+id*id$
$E'	+id*id$
$E'T+	+id*id$
$E'T'F	id*id$
$E'T'id	id*id$
$E'T'	*id$
$E'T'F*	*id$
$E'T'F	id$
$E'T'id	id$
$E'T'	$
$E'	$
$	$

First, Follow ادامه

Follow (A) for all non-terminal A

(1) $ ∈ Follow (S)

(2) if $A \to \alpha B \beta \Rightarrow (First(\beta) - \varepsilon) \subset Follow(B)$

(3) if $A \to \alpha B$ or $A \to \alpha B \beta$ and $\varepsilon \in First(\beta) \Rightarrow Follow(A) \subset Follow(B)$

Subject: (پارسر Top Down (بالا به پایین)
Year. Month. Date. ( )

First(E) = First(T) = First(F) = { (, id } : مثال از کتاب اژدر صفحه ۸۲

First(E') = { +, ε }

First(T') = { *, ε }

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

Follow(T) = Follow(T') = { +,), $ }

Follow(F) = { +, *,), $ }

حالا جدول را اینطوری می‌سازیم:

For each production A → α do

1 a ∈ First(α)

2 ε ∈ First(α)
   b ∈ Follow(A)

(مفهوم دو خط بعدی این است که دو حرف ترم چیست)

نکته: اگر پس از ساخت جدول دیدیم، در خانه‌ای بیش از یک rule نوشته شده، گرامر LL(1) نیست.

(مثلاً گرامر S → e | ε در حالت if-then و if-then-else)

نکته: گرامرهای LL همیشه راست‌گرد هستند، ولی LR نیستند.

Subject:
Year. Month. Date. ( )

راهی ساده‌تر برای پیدا کردن LL(1) بودن گرامر:

A grammar G is LL(1) $\iff$ for $A \to \alpha | \beta$ we have:

(1) $\begin{cases} \alpha \overset{*}{\Rightarrow} aw \\ \beta \overset{*}{\Rightarrow} aw' \end{cases}$ ← اشتراکی نباشند

(2) at most one of $\alpha \overset{*}{\Rightarrow} \varepsilon$ or $\beta \overset{*}{\Rightarrow} \varepsilon$

(3) if $\beta \overset{*}{\Rightarrow} \varepsilon$ then α does not derive any string with a terminal in Follow(A)

← که دوتاشون در یک ترمینال نباشند.

Constructing : در ادامه به طریقه ساخت جدول می‌پردازیم

SLR Parsing :

(SLR Grammar, SLR parser, SLR parser)

LR(0) items

$A \to .xyz$ ← که dot بعد از چه چیزی قرار گرفته یعنی چه چیزی پارس شده

$A \to x.yz$

$A \to xy.z$

$A \to xyz.$

$A \to \varepsilon$ → عضو ثابت داریم.
$A \to .$

~~Chapter~~

تا حالا $S' \rightarrow S$ اضافه نمیکردیم. چیت آفر (Acceptance) از این طریق حاصل میشد و گفتیم میشه

Augmented Grammar نمیشود. این کار برای این بود که بتونیم Acceptance رو بگیریم.

Closure (I)

I is a set of items

1. Initially, every item in I is added to closure(I)

2. If $A \rightarrow \alpha . B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production then add the item $B \xrightarrow[\text{dot}]{} .\gamma$ to I, if it isn't already there.

We apply this rule until no more item can be added to closure(I).

(مثال

$E' \rightarrow E$ ⟹ Augment

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

closure($\{[E' \xrightarrow[\text{dot}]{} .E]\}$) = ??

$E' \rightarrow .E$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

closure روی تمام item های کنار هم میزنیم ←

Subject:
Year 93 Month. 9 Date 29 ( )

function closure (I) :

begin

J := I

repeat

for each item $A \to \alpha . B\beta$ in J and each production $B \to \gamma$ of G such that $B \to .\gamma$ is not in J do

add $B \to .\gamma$ to J

until no more items can be added to J.

return J

end

Kernel items : $S' \to .S$, all items whose dots are not at the left end.

Non-Kernel items : dot at the left end (نقطه در ابتدای سمت چپ باشد)

(نقطه در ابتدای سمت چپ نباشد)

تعریف تابع goto :

goto (I, X) = closure $(A \to \alpha . X . \beta)$ (dot)

$I : A \to \alpha . X \beta$ (dot)

نقطه از کجا جلوتر رود، سمت راست آن قرار می‌گیرد.

(مثال)

$\{ [E' \to E.],$
$[E \to E. + T] \}$

goto (I, +) = ? = closure $[E \to E + .T]$ =

$E \to E + . T$
$T \to . T * F$
$T \to . F$
$F \to . (E)$
$F \to . id$

Subject: ترجمه
Year 93 Month 09 Date 29

Cannonical collection of sets of LR(0) items :

Procedure items (G'):

begin

$C := \{ \text{closure}(\{[S' \to .S]\}) \}$

repeat

for each set of items I in C and each grammar symbol X such that goto (I, X) is not empty and not in C, do

add goto (I, X) to C

until no more sets of items can be added to C.

end

مثال: گرامر زیر:

I_0 : $E' \to .E$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_1 : $E' \to E.$
$E \to E.+T$

I_2 : $E \to T.$
$T \to T.*F$

I_3 : $T \to F.$

I_4 : $F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_5 : $F \to id.$

I_6 : $E \to E+.T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

Subject:
Year 03 Month 9 Date 29 ( )

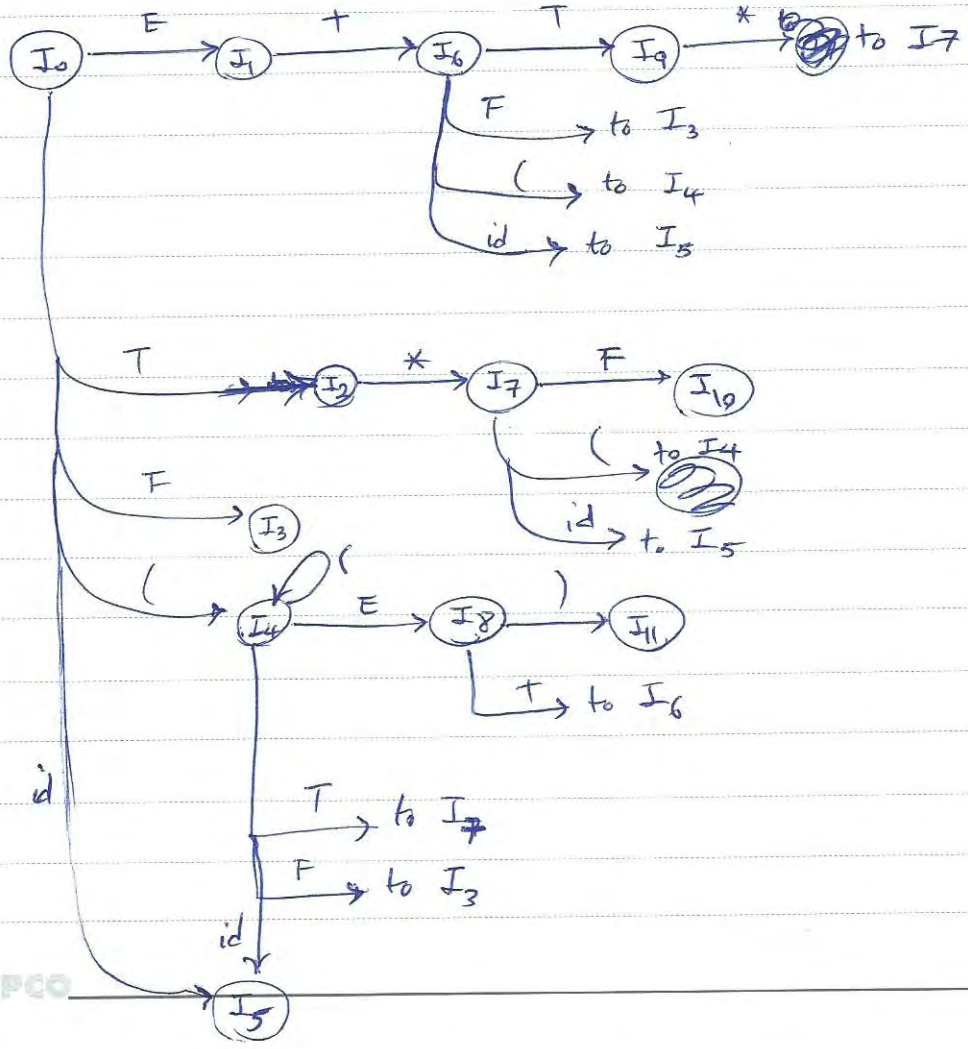
I_7 : $T \to T * . F$
$F \to . (E)$
$F \to . id$

I_8 : $F \to (E .)$
$E \to E . + T$

I_9 : $E \to E + T .$
$T \to T . * F$

I_{10} : $T \to T * F .$

I_{11} : $F \to (E) .$



PdPCO

ay

Subject:

Year. ۲۰۲۴ Month. ۹ Date. ۲۹ ( )

(یعنی viable prefix تعریف شده)

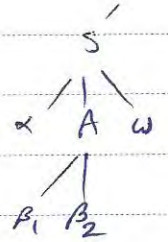
Valid items:

item $A \to \beta_1 \cdot \beta_2$ is valid for a viable prefix $\alpha\beta_1$

(یعنی $\alpha\beta_1$ از راست‌ترین مشتق ممکن به دست آمده)

if there is a derivation $S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1 \cdot \beta_2 w$

Right most derivation



اگر reduce به β_2 و اگر shift به β_2

	action			goto		
	id	+	$	E	T	=
states						

Algorithm: Constructing an SLR parsing Table

input: an Augmented grammar G'

output: SLR Parsing Table ← برای goto و action

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$ the collection of sets of LR(0) items for G'

2. State i is constructed from I_i, The parsing action for state i are determined as follow:

a) if $[A \to \alpha \cdot a\beta]$ is in I_i and $goto(I_i, a) = I_j$

$\Rightarrow$ action $[i, a]$ = "shift j"

(a is a terminal)

Subject:
Year. 93 Month. 9 Date 29 ( )

b) if $[A \to \alpha.]$ is in I_i

$\Rightarrow$ action $[i, a]$ = "reduce $A \to \alpha$"

for all $a \in$ Follow (A)

Here A may not be S'

c) if $[S' \to S.]$ is in I_i, then set action $[i, \$]$ to accept

3. The goto transitions are constructed for state i for all non-terminals A using:

goto $(I_i, A) = I_j \Rightarrow$ goto $[i, A] = j$

4. all entries not defined by rules 2,3 are made error

5. The initial state is the one constructed from the set of items containing $[S' \to .S]$

مثال درسی:

$F \to .(E)$ ... action $[0, (]$ shift 4

$F \to .id$ ... action $[0, id]$ shift 5

$E' \to E.$ ... actia $[1, \$]$ = accept

$E \to E.+T$ ... action $[1, +]$ shift 6

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

action $[2, \$]$ = action $[2, +]$ = action $[2,)]$ reduce $E \to T.$

action $[2, *]$ = shift 7 ...

Subject:
Year. 93 Month. 10 Date. 1 ( )

$E+T*$ $\quad I7$ (مثال برای Valid items صفحه ۹۲)

$\alpha\beta_1$

$I7: T \to T*.F \rightsquigarrow T \to \overbrace{T*}^{\beta_1}.\overbrace{F}^{\beta_2}$

$F \to .(E)$

$F \to .id$

$E' \Rightarrow E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow$

$E+T*F*id$

این valid item است.

$E' \Rightarrow E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*(E)$

$E' \Rightarrow E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id$

همانگونه که در DFA ظاهر شدند، این valid items

می‌توانند تعیین کنند که LR است یا LL و (SLR است یا نیست).

اگر توانایی تعیین نداشتند، می‌توان SLR نیست.

مثال)

$S \to L=R$

$S \to R$

$L \to *R$

$L \to id$

$R \to L$

$I_0: S' \to .S$

$S \to .L=R$

$S \to .R$

$L \to .*R$

$L \to .id$

$R \to .L$

$I_1: S' \to S.$

$I_2: S \to L.=R$

$R \to L.$

$I_3: S \to R.$

$I_4: L \to *.R$

$R \to .L$

$L \to .*R$

$L \to .id$

$I_5: L \to id.$

$I_6: S \to L=.R$

$R \to .L$

$L \to .*R$

$L \to .id$

$I_7: L \to *R.$

$I_8: R \to L.$

$I_9: S \to L=R.$

PqPC

برگردان این سه‌تا گرامر ... شیفت/ریدیوس کانفلیکت I_2 دارد.

action[2, =] = shift 6

action[2, $\in$ FOLLOW(R)] = action[2, =] = reduce $R \to L$

(پایان)

E. ~~tr~~ true ⇒ E.t = make(next); emit(goto _);

① if exp M S₁ ⇒ backpath(E.t, M.q); S.next = merge(S₁.n, e.f);
② S → if e M S₁ N else M₂ S₂ ⇒ back(e.t, M.q); backp(e.f, M₂.q);
S.N = merge(S₁.N, N.N); S.N = merge(S₁.N, S₂.N);

S → while M₁ B M₂ S ⇒ backpatch(S.N, M₂.q); backp(B.t, M₂.q); S.N = B.F; emit(goto M₁.q)

---

predictive parsing: ① x = a = $ ⇒ successful ② x = a ≠ $ ⇒ pop stack and Advance
③ x is a nonterminal ⇒ M[x, a] = {x → uvw}

First(A) ① if A is terminal ⇒ First(X) = {x} ② if x ⇒ ε ⇒ ε ∈ First(A)
③ if A is NT and x ⇒ $Y_1 Y_2 Y_3 \dots Y_n$, $Y_1 \dots Y_i \Rightarrow \varepsilon$, $Y_{i+1} \Rightarrow a$ ⇒ a ∈ first(A)

Follow(A) ① $ ∈ Follow(A) ② if A → αBβ : First(β) − ε ⊂ Follow(B)
③ if (A → αB) or A → αBβ, β ⇒ ε (ε ∈ first β) ⇒ Follow(A) ⊂ Follow(B)

---

table of predictive parsing: ① if a ∈ First(α) ⇒ | a | A → α | ② ε ∈ First(α), b ∈ Follow(A) ⇒ | | b | | A | A → α |
(A → α)

LL(1) ⇒ ① A → α | β ① α ⇒ aw, β ⇒ aw ✗ ② حداکثر α ⇒ ε یا β ⇒ ε ③ if β ⇒ ε then α ∩ Follow(A) = ∅
↑ برای آن نباید چیزی در اول شروع شود Follow A

---

closure(E' → .E) : A → α.Bβ, B → γ ⇒ add B → .γ
Goto(I, x) ≡ closure(A → αx.β)
canonical collection : add all goto(I, x) to C
table of SLR :
① a) if A → α.aβ is in I_i, goto[I_i, a] = I_j ⇒ [i, a] = sj
b) if A → α. ∈ I_i ⇒ [i, $] = reduce A → α for all a ∈ follow(A)
c) if S' → S ∈ I_i ⇒ [i, $] = accept
② goto transitions goto[i, A] = j ⇒ from C or STD

---

goto(1 ; متغیر
statement(2
prec 3 ;
عملیات(4

NDFA$_\lambda$ → NDFA

$f' = f \cup \{q\}$ if λ-closure(q) contain a final state

$S'(q,a) = \lambda\text{-closure}(S(\lambda\text{-closure}(q), a))$

NDFA → DFA

$q_0 \xrightarrow{a} q_1$, $q_0 \xrightarrow{b} q_0 q_1$, ...

Complement of DFA

change final state of a complete DFA

DFA اشتراک دو

$q_0 = [q_0, p_0]$, $F = [r,s] \in F$

$Q = [r,s]$

$S([r,s], a) = [r', s']$

Left Direct Recursion:

$A \to A\alpha \mid \beta \Rightarrow A \to \beta A'$, $A' \to \alpha A' \mid \varepsilon$

$A \to \lambda$ حذف تولید λ

Find nullable variable ($x \overset{*}{\to} \lambda$)

بعضی تغییرهای پیرو λ تولید و (وقتی و کی) و ...

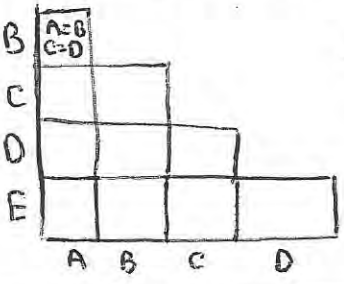
indirect left recursion

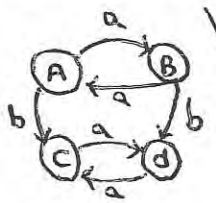
$A_i \to A_j \gamma$ ($i > j$ from $i = 1$)

$A_i \to S_1 \gamma \mid S_2 \gamma \mid \ldots$

$A_6 \to S_1 \mid S_2 \mid \ldots$

Find minimal DFA





(complete DFA? No)

or:

	a	b
A	B	C
B	A	D
C	D	—
D	C	—

کامپایلر: سرعت و امنیت

مفسر: قابل حمل

Lexical analyzer:

① pre processing ② Remove comment, blank, tab ③ keep track of number of newline ④ make copy of source program with error message mark in it

digits → zero | nonzerodigit (zero | nonzerodigit)* Leading zero

op-fraction → .zero | (.(zero | nonzerodigit)* nonzerodigit | ε ← trailing zero

id → letter (letter | digit)*

num → digits$^+$ (.digit$^+$)? (E(+|-)? digit)?

Lex: letter = [a-z A-Z]

id = {letter} ({letter} | {digit})*

ws = {delim}$^+$

lexical analyzer و parser علت جدایی:

① simpler design

② efficiency is improved

③ portability is enhanced

code %% deffinition %% rule ← lex

<yy:instal>{"if" { sysout(); } ... {int num}{

return ID;} ... .{}}

Top Down : LL : S → target

Bottom up : LR : target → S → in reverse

yacc: %language "java" %type <string> exp S

%token INT LE ... %code {...} %left OR

OR → AND → assign → LT, GT, ... → plus, minus → mult, Div → Else

%% program: S a I {...};() %% java code

Def of a procedure — activation of a procedure

Declaration of a name — Binding of name

scope of declaration — Lifetime of binding

return value, actual parameter, optional control link, optional access link, saved machine status, local data, temporary value

S → id := E S.code = E.code || gen(id.p ':=' E.p)

E → E + E E.p = newtemp; E.code = E.code || ... || gen(...);

E → id E.p = id.p , E.code = "";

E → E$_1$ or E$_2$ ⇒ E$_1$.true = E.true; E$_1$.false = newLable

E$_2$.false = E.false; E$_2$.true = E.true; E.code = E$_1$.code || gen(E$_1$.false) || E$_2$.code

E → E$_1$ and E$_2$ ⇒ E$_1$.true = newLable; E$_1$.f = E.f; E$_2$.t = E.t; E$_2$.f = E.f ; E.c = E$_1$.c || gen(E$_1$.t ':') || E$_2$.c

E → not E ⇒ change true and false

E → id relop id ⇒ E.code := gen('if' id relop id.p 'goto' E.true) || gen('goto' E.false)

E → true ⇒ goto E.true

P → M D { addwidth(top(tblptr), top(offset)); pop(tblptr); pop(offset); }

M → ε { t = maketable(nil); push(t, tblptr); }

N → ε ⇒ push(0, offset); }

D → proc id; M { t = top(tblptr); addwidth(t, top(offset)); pop(tblptr); pop(offset); enterproc(top(tblptr), id.name); }

D → id : T { enter(top(tblptr), id.name, type, top(offset)); top(offset) = top(offset) + t.width }

E → E$_1$ or M E$_2$ ⇒ backpatch(E$_1$.f, m.q); E.t = merge(E$_1$.t, E$_2$.t); E.F = E$_2$.f

E → E$_1$ and M E$_2$ ⇒ backpatch(E$_1$.t, m.q); E.F = merge(E$_1$.F, E$_2$.F); E.t = E$_2$.t

E → id$_1$ relop id$_2$ ⇒ E.t = makelist(nextq); E.F = makelist(nextq+1); quad.add(if ...); quad.add(goto _);

First(X) for all grammar symbol X:

(1) if X is a terminal ⟹ First(X) = {X}

(2) if X → ε ⟹ ε ∈ First(X)

(3) if X is a nonterminal and X → $Y_1 Y_2 \dots Y_k$

if ∃i ∃a ∈ First(Y_i) ε ∈ First(Y_1), ε ∈ First(Y_2) ... ε ∈ First(Y_{i-1})

⟹ a ∈ First(X)

if ∀j = 1...k ε ∈ First(Y_i) ⟹ ε ∈ First(X)

Follow(A) for all nonterminal A:

(1) $ ∈ Follow(S)

(2) if A → αBβ ⟹ (First(β) − ε) ⊂ Follow(B)

(3) if A → αB or A → αBβ and ε ∈ First(β) ⟹ Follow(A) ⊂ Follow(B)

For each production A → α do :predictive

(1) a ∈ First(α) — A | A → α (a)

(2) ε ∈ First(α), b ∈ Follow(A) — A | A → α (b)

A grammar G is LL(1) ⟺ for A → α | β we have:

(1) α →* aw, β →* aw' ⟹ a ∉ Follow(A)

(2) at most one of α →* ε or β →* ε

(3) if β →* ε then α doesn't derive any string with a terminal in Follow(A).

closure(I) I is a set of items

1. Initially, every item in I is added to closure(I).
2. if A → α.Bβ is in closure(I) and B → γ is a production then add the item B → .γ to I, if it isn't already there. We apply this rule until no more item can be added to closure(I).

Function closure(I):

begin

J := I

repeat

for each item A → α.Bβ in J and each production B → γ of G such that B → .γ is not in J do

add B → .γ to J

until no more items can be added to J.

return J;

end

kernel items ⇒ S' → .S, all items whose dots are not at the left end.

nonkernel items ⇒ dot at the left end.

goto(I, X) = closure(A → αX.β)

Canonical collect. of sets LR(0) items:

procedure items(G'):

begin

C := {closure({S' → .S})}

repeat

for each set of items I in C and each grammar symbol X such that goto(I, X) is not empty and not in C do

add goto(I, X) to C

until no more sets of items can be added to C.

end

SLR description:

1. Construct C = {$I_0, I_1, \dots I_n$} the collection of sets of LR(0) items for G'.
2. state i is constructed from I_i, The parsing action for state i are determined as follows:
   a) if [A → α.aβ] is in I_i and goto(I_i, a) = I_j ⟹ action[i, a] = "shift j" (a is a terminal)
   b) if [A → α.] is in I_i ⟹ action[i, a] = "reduce A → α" for all a ∈ Follow(A), here A may not be S'.
   c) if [S' → S.] is in I_i, then set action[i, $] to accept.
3. The goto transitions are constructed for state i for all nonterminals A using: goto(I_i, A) = I_j ⟹ goto[i, A] = j
4. all entries not defined by rules 2, 3 are made error.
5. The initial state is the one constructed from the set of items containing [S' → .S]

Valid items:

item A → $\beta_1.\beta_2$ is valid for a viable prefix $\alpha\beta_1$ if there is a derivation $S' \xrightarrow[rm]{*} \alpha A w \xrightarrow[rm]{*} \alpha\beta_1.\beta_2 w$

SLR

S → L = R

S → R

L → *R

L → id

R → L

shift/Reduce

if exp M S_1 → bp(e.tl, M.q); S.next = merge(S_1.next, e.fl);

if exp M S_1 N else M_2 S_2 pac; bp(e.tl, M_1.q); bp(e.fl, M_2.q);

S.n = merge(S_1.N, N.N); S.N = merge(S.N, S_2.N);

while M_1 B M_2 S → bp(S.N, M_1.q); bp(B.tl, M_2.q); S.N = B.fl; emit(goto M_1.q);

OR < AND < Assign < Relop < plus, minus < mult, div < else

NDFA$_\lambda$ → NDFA

$\delta'(q,a) = \lambda\text{-closure}(\delta(\lambda\text{-closure}(q), a))$

$F' = \begin{cases} F \cup q & \text{if } \lambda\text{-closure}(q) \text{ contains at least a final state in } F \\ F & \text{o.w.} \end{cases}$

NDFA → DFA

	a	b
A	B	C
B	A	D
C	D	—
D	C	—

minimum

Simple Grammar:

1) A → aX a∈V, α∈T, X∈V*

2) ...

(LL(1))

A → Ax$_1$ | Ax$_2$ | ... | Ax$_n$

A → y$_1$ | y$_2$ | ... | y$_m$

⇓

A → y$_1$ | y$_1$Z | y$_2$ | y$_2$Z | ... | y$_m$ | y$_m$Z

Z → x$_1$ | x$_1$Z | x$_2$ | x$_2$Z | ... | x$_n$ | x$_n$Z

A ⇒* λ → A is Nullable

A → BC   A → a   A → λ

A, B, C ∈ V, a ∈ T

A → aX   a∈T, X∈V*

Error → Lexical → ST
Error → Syntax → ...
Error → Semantic → type mismatch
Error → Runtime → Divide by zero

lexical analyser → Syntax analyser → Semantic routine → Intermediate Code Generator → Code Optimizer → Code Gen.

Associativity level ...

exp → exp+term | exp-term | term

term → term*factor | term/factor | factor

factor → (exp) | digit

(< . < * ← ... (left-Associative)

1) simpler design.
2) efficiency is improved
3) Portability enhanced

digits → zero | non-zero (zero | non-zero)*

op-fraction → . zero | . (zero | non-zero)* non-zero | ε

id → letter (letter | digit)*

num → digits (. digits)? (E (+|-)? digits)?

r? ⇒ r|ε

R.E.	Token	Attribute
id	id	pointer to S.T.
num	num	^
relop	relop	LT

return(getToken(), install_id()); ... S.T. ...

Bottom-Up ⇒ (Shift-Reduce) ⇒ LR(k) → Right-most in Reverse

Top-Down ⇒ LL(k) → left-most

S →* terminal, terminal → S

... LR(k) ... LL(k) ...

Shift/Reduce: ...

Rn/Rn: ...

ST: ... ← Rn/Rn ...

... ← Sn/R ...

Action → terminal

Go To → Non-terminal

... id 5 ...

... non-terminal ...

O F3 id → o T2 *id ...

Simple LR LALR Canonical LR

→ ...

ST → Hash Table
↳ Linear List

Hash: ...

P → ... q

Activation Tree ← to procedure ...

... ← DFS ...

name → environment → state → storage → value

Act. Record:
Returned Value + Actual Parameter +
optional Control Link (d-link) +
optional access or (s-link) +
Saved machine status +
local data + Temp. values

Error → Lexical → ST ...
Error → Syntax → ...
Error → Semantic → type mismatch
Error → Runtime → Divide by zero

lexical analyser → Syntax analyser → Semantic routine → Intermediate Code Generator → Code Optimizer → opt. Gen.

Associativity level ...

P → MD { addwidth(top(tblptr), top(offset)); pop(tblptr); pop(offset); }

M → ε { t := mktable(nil); push(t, tblptr); push(0, offset); }

D → D$_1$; D$_2$

D → proc id ; ND ; S { t := top(tblptr); addwidth(t, top(offset)); pop(tblptr); pop(offset); enterproc(top(tblptr), idname, t); }

D → id : T { enter(top(tblptr), idname, T.type, top(offset)); top(offset) := top(offset) + T.width; }

N → ε { t := mktable(top(tblptr)); push(t, tblptr); push(0, offset); }

Single pass → Quadruples → op, arg1, arg2, result
→ Triples → op, arg1, arg2
→ Indirect Triple → ...

S → id := E { S.code := E.code || gen(id.place ':=' E.place); }

E → E$_1$ + E$_2$ { E.place := newtmp(); E.code := E$_1$.code || E$_2$.code || gen(E.place ':=' E$_1$.place '+' E$_2$.place); }

E → id { E.place := id.place; E.code := ""; }

... ...

Boolean Exp: a<b → 100: if a<b goto 103 | 102: t := 0 | 103: goto 104 | 104: t := 1;

E → E$_1$ or E$_2$ { E.place := newtmp(); emit(E.place ':=' E$_1$.place 'or' E$_2$.place); }

E → id$_1$ relop id$_2$ { E.place := newtmp(); emit('if' id$_1$.place relop.op id$_2$.place 'goto' nextstate+3); emit(E.place ':=' '0'); emit('goto' nextstate+2); emit(E.place ':=' '1'); }

E → true { E.place := newtmp(); emit(E.place ':=' '1'); }

... ...

First, Follow:
α any string of grammar Symbols

First(α) = {a | α →* aω and a is a terminal or α →* ε, a = ε}

Follow(A) = {a | a is a terminal symbol, S →* αAaβ for some α, β or a = $ and A can be a right most symbol in some right sentential form}

E → E$_1$ or E$_2$ { E$_1$.true := E.true; E$_1$.false := newlabel; E$_2$.true := E.true; E$_2$.false := E.false; E.code := E$_1$.code || gen(E$_1$.false ':') || E$_2$.code; }

E → id$_1$ relop id$_2$ { E.code := gen('if' id$_1$.place relop.op id$_2$.place 'goto' E.true) || gen('goto' E.false); }

E → true { E.code := gen('goto' E.true); }

Backpatch:

E → E$_1$ or M E$_2$ { bp(E$_1$.flist, M.quad); E.tlist := merge(E$_1$.tl, E$_2$.tl); E.fl := E$_2$.fl; }

E → id$_1$ relop id$_2$ { E.tl := makelist(nextquad); E.fl := mklist(nextquad+1); emit('if' id$_1$.place relop.op id$_2$.place 'goto' _); emit('goto' _); }

E → true { E.tl := mklist(nextquad); emit('goto' _); }

M → ε { M.quad := nextquad; }

S → while M$_1$ E do M$_2$ S$_1$ { bp(S$_1$.nextlist, M$_1$.quad); bp(E.tl, M$_2$.quad); S.nextlist := E.fl; emit('goto' M$_1$.quad); }

Function call:

Elist → E { Code for E; initialize queue to contain only E.place }

Elist → Elist, E { Code for E; append E.place to the end of queue }

S → call id (Elist) { for each item p on queue do emit('param' p); emit('call' id.place); }

Recursive descent parser → LL
↳ non-Terminal ⇒ procedure → ...
↳ Terminal ⇒ ADVANCE() ...
↳ left-Recursion, Back Tracking

... ADVANCE ... input_ptr

... left-Factoring ...

A → αβ$_1$ | αβ$_2$ ⇒ A → αA'   A' → β$_1$ | β$_2$

... left-recursion ...

A$_1$ ... A$_n$ ... to non-terminal ...

for i := 1 to n do begin
  for j := 1 to i-1 do begin
    replace each production of the form A$_i$ → A$_j$γ
    by the productions A$_i$ → δ$_1$γ | δ$_2$γ | ... | δ$_k$γ where
    A$_j$ → δ$_1$ | δ$_2$ | ... | δ$_k$ are all the current A$_j$-productions
  end
  eliminate the immediate left-recursion among the A$_i$-productions
end

predictive parser: ... input ... stack ...

(1) x = a = $ → parser halts and announces successful completion

(2) x = a ≠ $ → pop x and advance the input pointer

(3) x is a non-terminal → M[x, a] = {x → uvw}

explist ⟶ exp | explist, exp

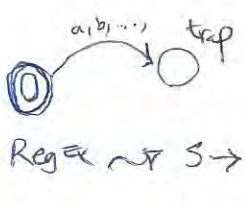
relop ⟶ >|<| # |>=|<=|=

کلیه کلماتی که حرف اولشان بزرگ است همانند For کلمه کلیدی هستند و پایانه بقیه کلمات نماد های گرامر می باشند شناسه ها و اعداد همانند بقیه زبانهای برنامه نویسی می باشند. کلمات کلیدی And Then , Or Else همان and . or معمولی می باشند که مقدار برگشتی آنها از نوع bool می باشد و در مورد آنهاهمانند اکثر زبانهای برنامه نویسی اتصال کوتاه انجام می شود. همچنین در این زبان هر عدد مخالف صفر از لحاظ منطقی مقدار true و عدد صفر مقدار false در عبارتهای منطقی دارد و از این لحاظ رفتار این زبان در ارزیابی عبارتهای منطقی کاملاً شبیه به زبان C می باشد.

به عنوان مثال برنامه زیر را در نظر بگیرید:

```
program sample1;
int fac1:=101,fac2;
real div1:=100.01,div2;
bool flag:=false;
function fac(int n,real m):real
    int counter:=1;
    real result:=0.0;
begin
    for counter:=1 to n do
    result := result * counter;
     return result
    end;
function facr(int n):int
begin
    if (n=0) then
    return 1
else return n*facr(n-1)
end;
begin
    fac1:=fac(5,2);fac2:=facr(5);
    flag:=fac2=120;
    div2:=div1 + flag
end ;
```


Relop



program ⟶ Program id; declist block;

declist ⟶ dec | declist dec| ε

dec ⟶ vardec | procdec | funcdec

type ⟶ Int | Real | Bool

iddec ⟶ id | id:=exp

idlist ⟶ iddec | idlist , iddec

vardec ⟶ type idlist;

procdec ⟶ Procedure id (paramdecs) declist block;

funcdec ⟶ Function id (paramdecs) : type declist block;

paramdecs ⟶ paramdec | paramdecs ; paramdec

paramdec ⟶ type paramlist

paramlist ⟶ id | paramlist , id

block ⟶ Begin stmtlist End | stmt

stmtlist ⟶ stmt | stamtlist; stmt

lvalue ⟶ id

stmt ⟶ lvalue := exp |
If exp Then block |
If exp Then block Else block |
While exp Do block |
For lvalue:=exp To exp Do block |
For lvalue:=exp To exp Downto block |
Case exp caseelement End |
Return exp |
exp

exp ⟶ exp And Then exp

exp ⟶ exp Or Else exp

exp ⟶ exp + exp

exp ⟶ exp - exp

exp ⟶ exp * exp

exp ⟶ exp / exp

exp ⟶ ~~exp exp~~ ⟶ exp relop exp

exp ⟶ intnumber | realnumber | True | False | lvalue

exp ⟶ id(explist)

caseelement ⟶ intnumber : block ; | caseelement intnumber : block;

71 — 13

12 Conflict ← ... exp → exp > exp | exp # exp ... relop → ...

15 Conflict ← ... Program → Program id; block; ... declist → ε ...

Procdec → ... declist

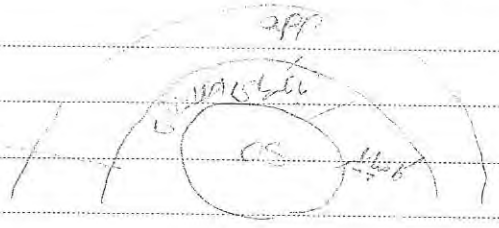
Funcdec → ...

کامپایلر

Compiler principles, techniques and tools

Subject:

Year. Month. Date.



وظیفه: هر چیزی که در کامپیوتر وجود دارد time و space است پس باید کامپایلر خوبی بنویسید.

کامپایلر: نرم افزاری است که Source prog (برنامه مبدا) را به عنوان ورودی می‌گیرد و برنامه هدف را به عنوان خروجی می‌دهد

Source Prog. → Compiler → target prog.

مثلا Pascal و fortran و C به زبان سطح میانی

زبان ماشین / Assembly

اولین زبانی که برای کامپیوتر ساخته شد Formula Translation (Fortran) بود

ساخت این کامپایلر ۱۸ نفر سال طول کشید!

man year

کامپایلر برای تبدیل Source Prog به Target Prog یک سری فاز را طی می‌کند. علاوه بر ترجمه برنامه خطاها را نیز نشان می‌دهد (error message)

حافظه کم نیازمند

optimizing, debuging, load & go, multi pass, Single Pass

عبور

1 Pass

عبور ← یک بار خواندن فایل و تولید خروجی توسط کامپایلر

Subject:
Year. Month. Date. ( )

Load & go ← خواندن و اجرا (برای تازه کارها مناسب است)

debugging ← خطاها را تشخیص و اصلاح می‌کند (کندتر است اما خطاهای بیشتری را می‌یابد؟)

برای کارهای حساس مثل کاربردهای نظامی

optimizing ← در زمان کم بیشترین استفاده (کد ایجاد شده Space و time کمتری داشته باشد) ← کاربردهای تجاری مثل بانکهای اطلاعاتی که هرچه سریعتر باشند بازار را بیشتر به دست می‌گیرند.

جلسه ۲ ۹۳/۶/۲۴

Interpreter برنامه مبدا (Src prog.) را می‌گیرد و همان موقع اجرا می‌کند
مفسر

برای زبانی استفاده می‌شود که دستوراتش تا حد امکان مستقل باشند

خود کامپایلر و Interpreter نیز اگر به زبانی مثل C باشند باید به زبان ماشین ترجمه شوند

کامپایلر Src را به برنامه‌ای تبدیل می‌کند که به زبان سطح پایین است پس به راحتی

نمی‌توان آن را دستکاری کرد اما Src در Interp... قابل دستکاری است

اما برنامه‌ای که کامپایلر تولید می‌کند مخصوص یک ماشین است و برای ماشین دیگر

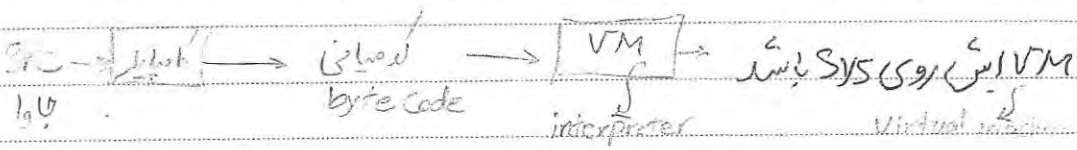
دوباره باید کامپایل شود. ولی Src در Interp... Portable است

Rapco

Subject:
Year. Month. Date. ( )

معمولی کامپایلر : امنیت معمولی Inter- : Portable

ترکیب کامپایلر و Inter- : کامپایلر یک کدمیانی تولید می‌کند که قابل خواندن نباشد سپس یک Inter- آن را اجرا می‌کند این کدمیانی Portable است اگر VM ایش روی Sys باشد

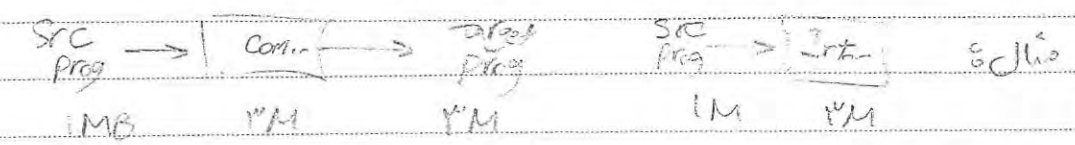


امنیت این کدمیانی نسبت به target pro بالایی که کامپایلر تولید می‌کند کمتر است (ممکن است بتوان آن را به برنامه اصلی برگرداند و دستکاری کرد.)

بررسی حافظه مورد نیاز کامپایلر و Inter- :

(در هنگام اجرای برنامه)

تبدیل کد به زبانی مثل C به اسمبلی تقریبا حافظه را ۳ برابر می‌کند



Src فایل + VMB (inter..)

Src فایل x ۳

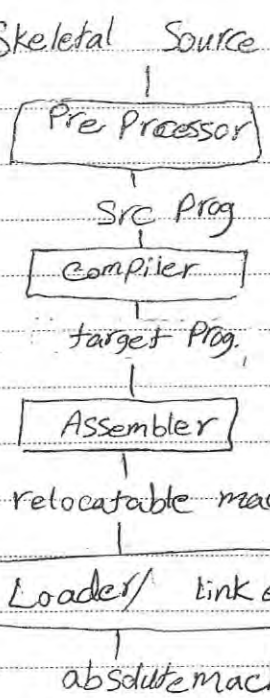
(در هنگام اجرا فقط target فایل نیاز است)

پس برای برنامه‌های کوچک Com.. و بزرگ inter.. مناسب است.

Just in-time Compiler

Subject:
Year. Month. Date. ( )

تجزیه parsing تحلیل analysis



کامپایلر از دو قسمت analysis , Synthesis (جمع کردن) تشکیل شده است

انواع analysis :

① Linear Analysis (lexical analysis, Scanning)
تحلیل خطی (تحلیل لغوی ، پویش)

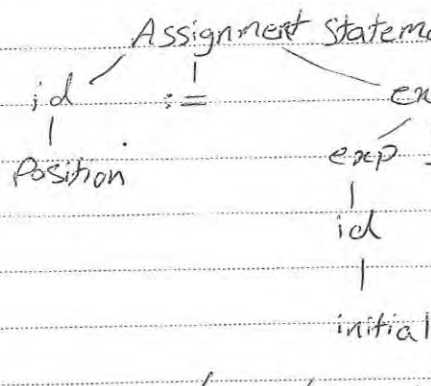
مثال:
Position := initial + rate * 60
Taken s identifier) id id (num
assignment symbol plus sign mult sign

علی سیب را خورد
اسم اسم حرف اضافه فعل
فاعل مفعول فعل
از نظر گرامری جمله درست است

② Syntax analysis (hierarchical analysis, Parsing)
تحلیل نحوی

Subject:
Year. Month. Date. ( )

در این مرحله درخت تولید می شود :



براساس گرامر زبان درخت ساخته می شود و چک می شود که از نظر گرامری درست باشد

③ Semantic analysis تحلیل معنایی

مثال:

سیب علی را خورد

از نظر گرامری جمله درست است

برای سیب و علی type تعریف می شود که سیب از type است که می تواند علی را از type مشخصی بخورد

این عملگر برای این دو type تعریف شده است

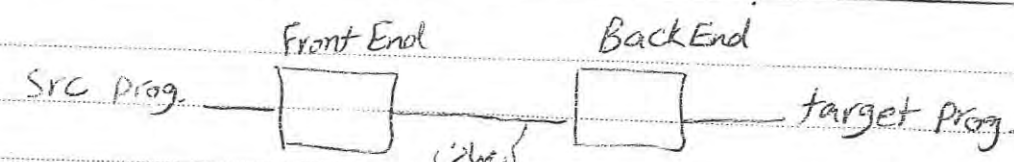
مثال:

rate * 60

float int ⇒ نادرست

۹۳,۶,۲۹ سه‌شنبه ۳

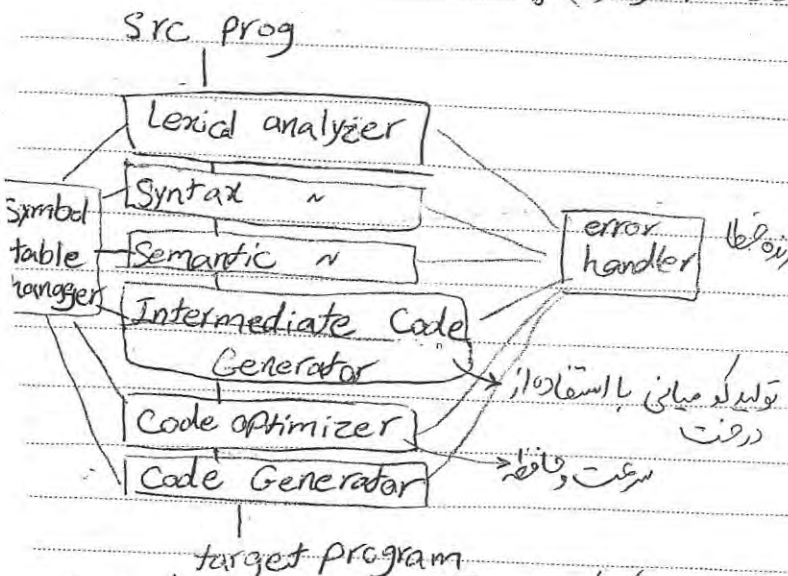
Subject:
Year Month Date ( )



پروژه این درس

در پروژه کدمیانی به زبان C است و نیازی به Semantic analy نیست

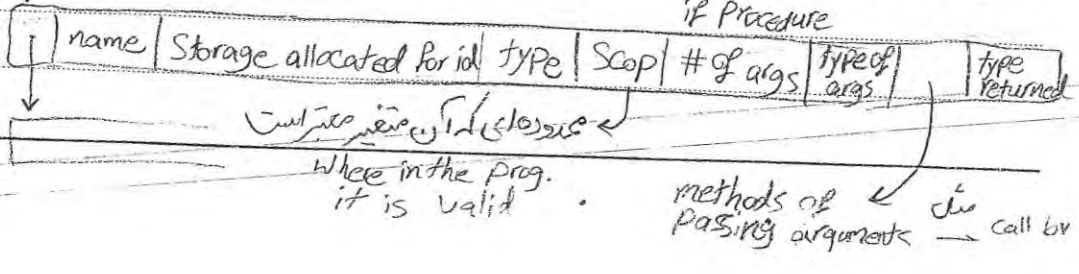
بخش های کامپایلر ← Phase (فاز) ؛



در جدول نمادها ابتدا نمادهایی وجود دارد که کامپایلر با آن کار می کند و سپس کامپایلر نیز به این نمادها اضافه می کند.

وقتی به خطایی در حین کامپایل برسیم error handler با توجه به نوع آن پیغام مناسبی را نمایش میدهد

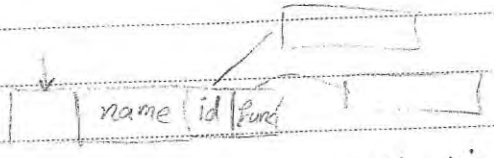
در جدول نمادها
برای هر متغیر یا func یک نود داریم و در کل یک لیست برای کل برنامه



Subject:
Year. Month. Date. ( )

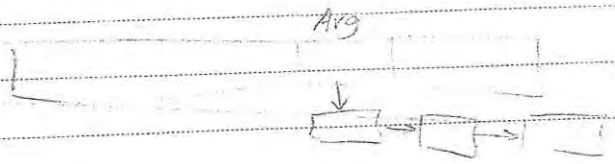
این ساختمان داده یک اشکال دارد و آن خالی ماندن برخی فیلدها است.

راه حل:



در این راه حل یک خانه اضافه می‌شود اما فیلد خالی نداریم

اشکال دیگر: تعداد Arg ها برای func ها مشخص نیست. پس به آن حافظه ثابت نمی‌دهیم لینک لیست می‌دهیم

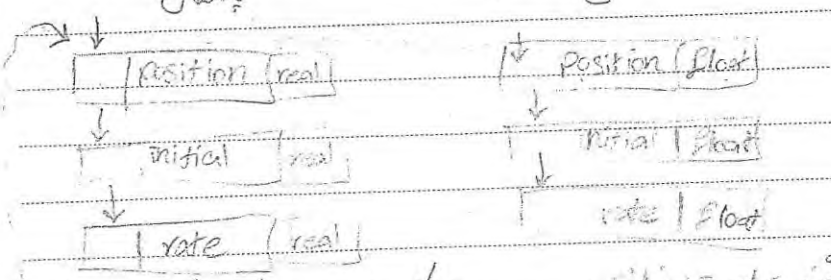


مقایسه زبان C و پاسکال:

Var position, initial, rate: real ; → مناسب برای آموزش

float position, initial, rate; → سریعتر کار می‌کند

lexical analyzer:



در پاسکال نیاز است یک یا دیگر به اولین پارامتر pointer و type را بنویسیم

در ابتدا type مشخص است در زمان به برنامه نیست (سریعتر)

error production and reporting

خطاهای هر فاز:

Subject: ۹۳/۶/۳۱ جلسه ۴
Year. Month. Date. ( )

① خطا در lexical analyzer : مثال نام متغیر غیرمجاز باشد مثلا با عدد شروع شود 2L ✗ L2 ✓

② خطا در Syntax : مثال برای if و then نوشتیم

③ Semantic : real * int

④ Runtime : در x/y مقدار y برابر صفر شود

جلسه ۴

برنامه‌ای بدون خطا در ابتدا نمی‌توانید بنویسید چون کامپیوتر بسیار حساس است مثلا اگر در مکالمه بین دو انسان شخصی منظور دیگری را نفهمد با سوال ابهام را رفع می‌کند اما کامپیوتر نمی‌تواند ابهام را رفع کند و خطا می‌دهد.

ادامه مثال قبل

Position := initial + rate * 60

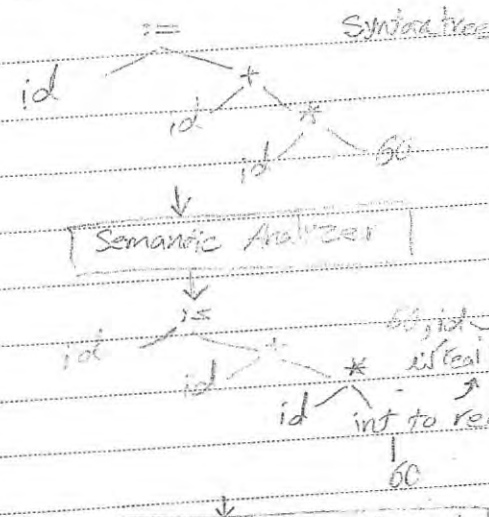
↓ lexical Analyzer

id := id + id * 60

↓ Syntax Analyzer

↓

Subject:
Year. Month. Date. ( )



Symbol table

1	position
2	initial
3	rate

token
lexeme
lexical value
lexical value of rate = 3

P: X := Y op Z

تعداد reg ها محدود است اما در تولید کد میانی راحتتر از temp ها استفاده میکنیم (جزء ۱) همچنین این کد وابسته به عاملش نیست (جزء ۲)

Code Optimizer : افزایش سرعت یا کاهش حافظه

(در این مثال ۲ جزء را نه حافظه مورد نیاز را کاهش داد)

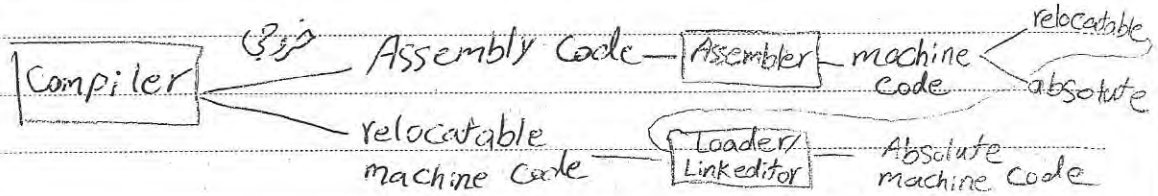
Subject:

Year. Month. Date. ( )

در اینجا ابتدا کدمیانی تولید می‌شود سپس بهینه می‌شود چرا در ابتدا کد بهینه تولید نمی‌کند

زیرا در تولید کدمیانی در هر مرحله از پیمایش درخت کد تولید می‌شود و به بقیه قدم‌ها

کاری ندارد اگر بخواهد بقیه قدم‌ها را نیز در نظر بگیرد پیچیده‌تر می‌شود



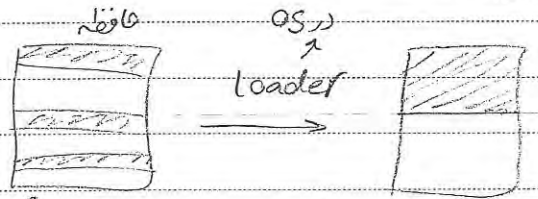
مثال

```
MOV  a, R1   Assembler
ADD  #2, R1  ------>   a  0
                       b  4
MOV  R1, b   b و a به آدرس‌های حافظه
```

```
Load  0001  01  00  000000   *
Add   0011  01  10  ----10
Store 0010  01  00  0--100   *
```

opcode | reg | نوع آدرس | آدرس یا عدد

آدرس‌های موقت ← relocatable machine code ← .obj



a 0 → 15
b 4 → 19

```
0001  d  00  0000 1111
0011  01  10  000000 10   ------> absolute machine code ------> .exe
0010  01  00  000100 11
```

Subject:

Year. Month. Date. ( )

Separate Compilation

برنامه‌های بزرگ را به چند فایل تقسیم می‌کنند و جداگانه کامپایل می‌کنند

link editor این قسمت‌های جدا را به هم وصل کرده و یک برنامه می‌سازد

* کامپایلر را به دو قسمت Front end, back end تقسیم می‌کنند که قسمت Front End به ماشین وابسته نیست اما پیاده‌سازی Back end با توجه به ماشین است

Front End:

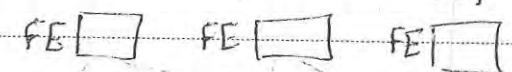
Symbol table + lexical Analysis, Syntax, Semantic, Intermediate Code Generation, Certain amount of Code optimization + related part of error handler

Back End:

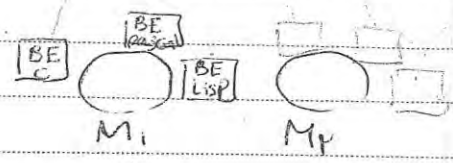
part of Symbol table + part of optimization + ?

جلسه ۵ ۹۳/۷/۲۵

Front End — C, pascal, lisp



Back End

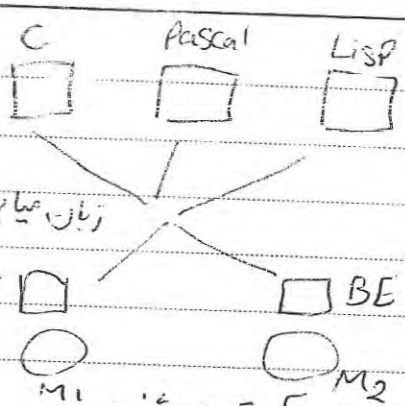


برای هر زبان یک FE نیاز است و به ازای هر ماشین یک BE برای آن زبان سپس کامپایلر ایجاد می‌شود

PgPG

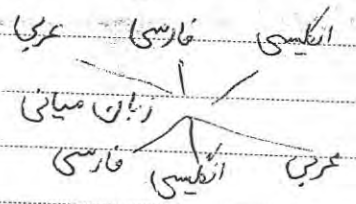
Subject:

Year. Month. Date. ( )

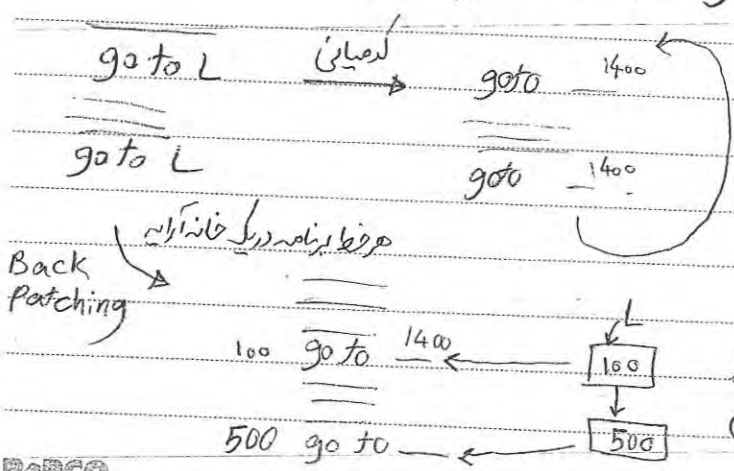


این روش اگر پیاده سازی شود بهتر است اما پیاده سازی آن دشوار است و تاکنون نتوانسته اند زبان میانی بسازند که همه زبان ها به آن تبدیل شوند

همین ایده در زبان های طبیعی نیز مطرح شد اما به دلیل اینکه زبان های مختلف قدرت متفاوت دارند (مثلا دامنه لغات متفاوت) نتوانسته اند زبان میانی که شامل خاصیت های مختلف زبان ها باشد بسازند



خواندن یک فایل ورودی یک Pass است. برای اینکه کامپایلر را طوری بنویسیم که در یک Pass خوانده شود نیاز به Back Patching داریم. مثال



هرگاه مقدار L مشخص شد به خط های مورد نظر رفته و مقدار L را قرار می دهیم پس نیازی به Pass دیگر نیست

PAPCO

Subject:
Year. Month. Date. ( )

ابزارهایی که به ساخت کامپایلر کمک می‌کنند:

Compiler - Compilers
~ Generators
translator_writing

مثال: نرم‌افزارها:

Lex
regular expression → Scanner Generator → yylex()

lexical analyzer کامپایلر را انجام می‌دهد

yacc: yet another compiler compiler
Context free grammar → Parser generator → yyParse()

parser (یا تجزیه‌کننده) را انجام می‌دهد

Parse tree → Syntax directed translation engines → Collection of routines that walks the parse tree generating intermediate code

one or more translations are associated with each nodes of parse tree

Collection of rules defining the translation of each operation of the intermediate language → Automatic code generator → ۷

intermediate code + extra info → Data flow Engine → Performs different task of Data Flow Analysis

تحلیل جریان داده

Control Flow: ترتیب انجام دستورات

Subject:

Year. Month. Date. ( )

گرامر از نظر گوی دست باشند

<Sentence> → <noun phrase> <verb phrase>

<noun phras> → <adjective> < np >

< np > → <noun>

<noun> → tree | book

<adjective> → little | ...

∴ Context Free Grammar

$G = (V, T, P, S)$

V: Finite set of variables (nonterminal)

T: terminals

S: Start Symbol

P: finite set of productions

$A \rightarrow d$ (A: nonterminal; d: nonterminals and terminals)

$V \cap T = \emptyset$

$A \rightarrow \Sigma$

$A \rightarrow \lambda$

مثال :

$List \xrightarrow{1} List + digit$

$List \xrightarrow{2} list - digit$

$List \xrightarrow{3} digit$

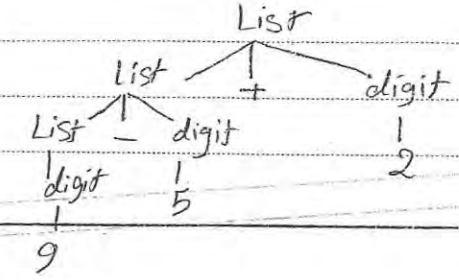
$digit \rightarrow \overset{4}{0} | 1 | 2 | \ldots \overset{13}{19}$

رشته‌ای از ارقام با یا بدون یک یا با طول بیشتر که بیشتر + یا - است

آیا جمله 2+5-9 جزو زبانی که این گرامر تولید می‌کند هست یا نه ؟

① derivation : $List \xrightarrow{1} List + digit \xrightarrow{2} List - digit + digit \xrightarrow{3} digit - digit + digit \xrightarrow{13} 9 - digit + digit \rightarrow \ldots \rightarrow 9-5+2$

② Parse tree



Subject:
Year. Month. Date. ( )

تفاوت دو روش : روش ① یک بعدی و روش ② دو بعدی است پس درک
روش ② ساده‌تر است. اما در روش ① ترتیب استفاده از گرامر مشخص است

جلسه ۶ ۹۳/۷/۱۷

block → begin opt_stmts ; end | begin opt_stmts end → ②

opt_stmts → stmt_list | ε

stmt_list → stmt_list ; stmt | stmt

stmt →

تمرین ① گرامر را تغییر دهید به طوری که ; قبل از end بیاید

② 〃 〃 〃 〃 〃 〃 〃 بتواند بیاید یا نیاید

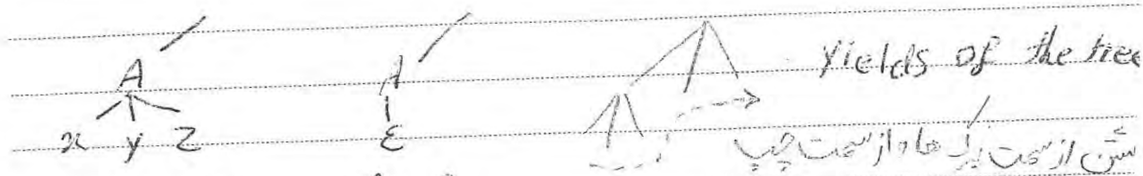
③ 〃 〃 〃 〃 block های تو در تو داشته باشیم ← ((( ))) مشابه پرانتز

④ 〃 〃 〃 و کنار هم داشته باشیم ← (()())

مشابه S → (S) | ε

③ block → begin block end | begin opt_stmts end

④ S → (S)S | ε
S → S(S) | ε
S → S(S)S | ε → مبهم



A → xyz   A → ε

PaPCO

۸

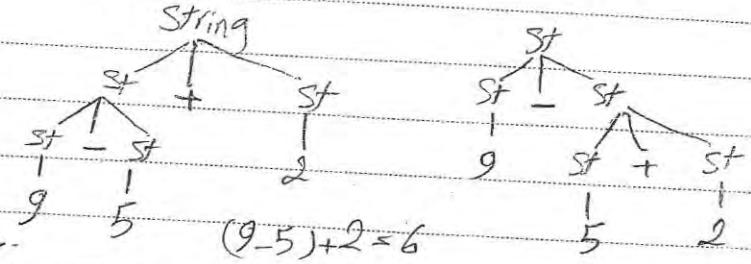
Subject:

Year. Month. Date. ( )

گرامر مبهم (ambiguous) ای که درخت parse یکسان باشد

مثال:

String → String + String | String - String | 0 | 1 | 2 | ... | 9



associativity انجمن پذیر

left associative از سمت چپ انجمن پذیر

precedence اولویت اولویت * بیشتر از - 9-5*2 = -1

تمرین ① اگر همه اولویت ها برابر بود و فقط left asso بود چه مشکلی پیش می آمد

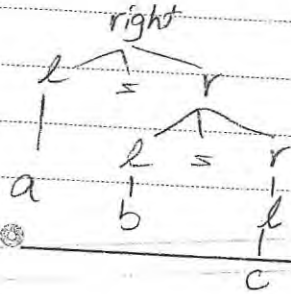
تمرین ② اگر همه اولویت ها متفاوت بودند و دیگر نیازی به left asso نبود

روش های رفع ابهام:

مثال:

right → letter = right | letter

letter → a | b | c | ... | z (a=(b=c))



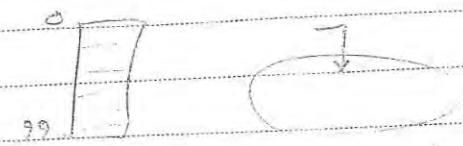
این گرامر مبهم نیست چون right recursion داریم

پس در درخت right asso داریم

۹۳/۷/۱۴

Subject:
Year. Month. Date. ( )

بافر سیگ :



خواندن از دیسک کند است .

تا بافر اول خوانده می شود بافر دوم از دیسک می خوانیم پس از بافر دوم می خوانیم و بافر اول از دیسک می خوانیم پس خواندن از دیسک کند است .

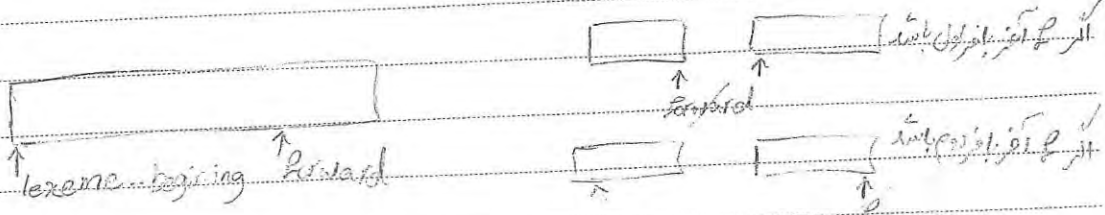
last ahead

زمان ما را می گیرد (تعداد بافرها بستگی به سرعت خواندن دارد اگر تعداد زیاد بافر باعث کند شدن برنامه می شود)

در مثال قبل با داشتن ۲ بافر همواره تضمین می شود به ازای یک بافر دیگر را داریم

| ... Declare | arg1, ... |

اگر یک بافر بود در این حالت باید بافر را قبل از Declare به اندازه بگوئیم

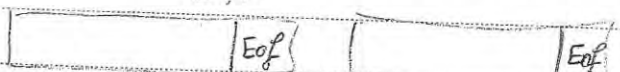



```
if forward at end of first half then
begin
reload second half ;
forward:= forward + 1 ;
end
else if forward at end of second half then
begin
reload first half
```

Subject:
Year. Month. Date.

move forward to beginning of first half
end
else forward = forward + 1

اشکال این برنامه: حالتی که انتهای دو نیمه بافر هستیم فقط دو بار است و دیگر از این دو حالت برای خانه‌های دیگر بافر دو شرط بیهوده چک می‌شوند پس سرعت کم می‌شود. راه حل: مشابه سرافاصله در لینک لیست (Sentinel نگهبان)



خانه آخر را با Eof مشخص می‌کنیم

forward := forward + 1
if forward ↑ = eof then begin
  if forward at end of first half then
    begin
      reload second half
      forward := forward + 1
    end
  else if forward at end of second half then
    begin
      reload first half
      move forward to beginning of ...
    end
  else terminate lexical analysis
end

اگر به آخر فایل اصلی رسیدیم

در این حالت حافظه بیشتری نیاز است اما سرعت بالایی دارد

جلسه ۹

Subject:
Year. Month. Date. ( )

استفاده از بافر در بیشتر کامپایلرها به دلیل lookahead است (سرعت زیادتر است)

Strings & languages

Alphabet : finite set of symbols $\{a, \ldots, z\} \cup \{0,1\}$

String over an alphabet finite sequence of symbols drawn from that alphabet

Sentence, word

length of a string s $|s|$ $|book| = 4$

empty string, ε $|\varepsilon| = 0$

Prefix of s

Suffix of s book

Substring of s book

Subsequence of s

Proper Prefix

language : set of strings over some fixed alphabet

مثال اگر alphabet = {0,1} باشد یک زبان {001, 010}

$\emptyset$ Empty set $\{\varepsilon\}$

{p | p is a syntatically well-formed Pascal prog}

x, y string → Concatenation x and y XY

$S\varepsilon = \varepsilon S = S$

$S^0 = \varepsilon$

$S^i = S^{i-1} S$

۱۴

Subject:

Year. Month. Date. ( )

Union Concatenation closure بستن

$L \cup M = \{\Delta \mid \Delta \in L \text{ or } \Delta \in M\}$

$LM = \{\Delta t \mid \Delta \in L \text{ and } t \in M\}$

$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup \dots$ ($L^0 \to \{\varepsilon\}$)

Kleene Closur of L

$L^+ = \bigcup_{i=1}^{\infty} L^i$

Positive closur of L

مثال:

$L = \{A, B, \dots, Z, a, \dots, z\}$ $D = \{0, 1, \dots, 9\}$

$L \cup D = \{A, B \dots Z, a, \dots z, 0, 1 \dots 9\}$

$LD = \{A0, \dots\}$

$L^4 = \{AAAA, AAAB, \dots$

L^* رشته‌هایی با طول‌های مختلف از حروف

$L(L \cup D)^*$ رشته‌هایی که با یک حرف شروع می‌شود بعدش حرف یا عدد

D^+ رشته‌هایی با حداقل طول 1 از اعداد

نمایش regular exp مجموعه یک زبان است

regular exp	زبان
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
r, S are regular exp	$L(r), L(s)$
a) $(r) \mid (s)$	$L(r) \cup L(s)$
b) $r \cdot s$	$L(r)L(s)$
c) r^*	$(L(r))^*$
d) r	$L(r)$

Subject:
Year. Month. Date. ( )

اولویت ها به ترتیب: * ، . ، | (left asso. و)

مثال

(a|((b*)c)) یا a یا رشته ای از b ها و سپس c

a|b {a, b}

(a|b)(b|a) {aa, ab, ba, bb}

a* {ε, a, aa, aaa, ...}

(a|b)*, (a*b*)* ترکیب صفر یا چند a و b، شامل ε

a|a*b {a, b, ab, aab, ...}

دو عبارت منظم معادل اند اگر زبانی که تعریف می کنند یکسان باشد (a|b)*, (a*b)**

r, s regular exp.

r|s = s|r | is commutative

r|(s|t) = (r|s)|t | is associative

r(st) = (rs)t . " "

r(s|t) = rs|rt Concatenation distributer over |

(s|t)r = sr|tr

εr = rε = r

r* = (r|ε)*

r** = r*

تعریف reg exp:

regular definition

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

$d_i \rightarrow r_i$

$d_n \rightarrow r_n$ که در آن r_i بر روی $d_1, \dots, d_{i-1}$ تعریف شده است

مثال ⇒ letter → A|B|...|a|...|z

digit → 0|1|2|...|9

id → letter(letter|digit)*

Subject:
Year. Month. Date. ( )

تفاوت تعریف regular exp و گرامر

گرامر
letter → A|B
یعنی
letter → A
letter → B

reg exp
letter → A|B
یعنی
اعمال عملگر | بر روی دو reg exp
A و B که زبان {A, B} را می‌سازند

مثال:

digit → 0 | 1 | ... | 9
digits → digit digit*
optional_fraction → (.digits | ε)?
optional_exponent → (E(+|-|ε) digits) | ε)?
num → digits optional_fraction optional_exponent

00123 leading zeros
12.300 trailing zero
نشان دهنده دقت

مثال قبل را طوری تغییر دهید که leading zeros, trailing z نداشته باشد!
۰.۱ ، ۱.۰ و ۰.۰ باید نشان داده شوند

NZdigit → 1 | 2 | ... | 9
opt_frac → .digit* NZdigit | .0 | ε
digits → NZdigit digit*

جلسه ۱۰ ۹۳/۷/۲۴

reg. exp	زبان
r	L(r)
r*	(L(r))*
a^+	
$r^* = r^+ \mid \varepsilon$	
$r^+ = rr^*$	
r?	r | ε

Subject
Date

$a|b|c$ — lex → $[a\ b\ c]$

$a|b|\dots|z$ → $[a-z]$

ide → $[A\text{-}Z\ a\text{-}z][A\text{-}Z\ a\text{-}z\ 0\text{-}9]^*$

reg exp ⊂ Context free grammar (cfg)

cfg مثال → $\{a^n b^n \mid n>0\}$

$S \to aSb \mid \varepsilon$

$\{wcw^R \mid w \in \{a,b\}^*\}$

$S \to asa \mid bSb \mid c$

$\{wcw \mid w \in \{a,b\}^*\}$ ←

چون حافظه نداریم که ببینیم برای این زبان چی قرار وقع نوشت

مثالی از تراجل زیر:

Stmt → If expr then Stmt

| If expr then stmt else Stmt

| ε

expr → term relop term | term

term → id | num

non-terminal ها terminal else ، then ، If ، num ، id ، relop

reg. definition

If → If → reg. exp → i.f

then → then

else → else

relop → < | <= | > | >= | <> | =

bject
ate

id → letter | (letter | digit)*

num → digit⁺ (. digit⁺)? (E (+ | -)? digit⁺)?

WS → delim⁺
↓ White space

delim → blank | tab | newline

reg. exp	token	attribute values
WS	—	—
If	If	—
then	then	—
else	else	—
id	id	Pointer to table entry
num	num	"
<	relop	LT → relop نشخص نوع
<=	"	LE
=	"	EQ
>	"	GT
>=	"	GE
<>	"	NE

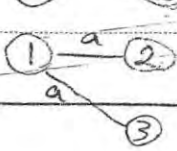
به Parser فرستاده می‌شود

مراد از token ها relop ها LE LT و... را توسیم کرده‌ایم
کار را بعدا انجام می‌دهیم و با relop ای که در آن هست فعلا کار می‌کنیم (فعلا)

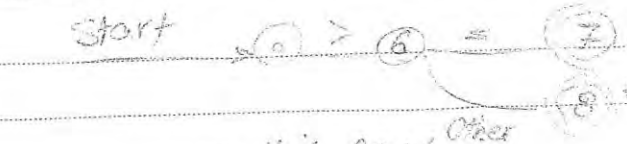
مرحله بعد رسم transition diagram است:
نمودار انتقالی

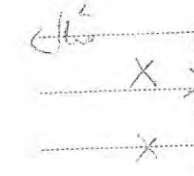
nsition diagram → Deterministic
با دیدن a فقط به یک حالت می‌رود

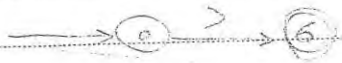
non deterministic

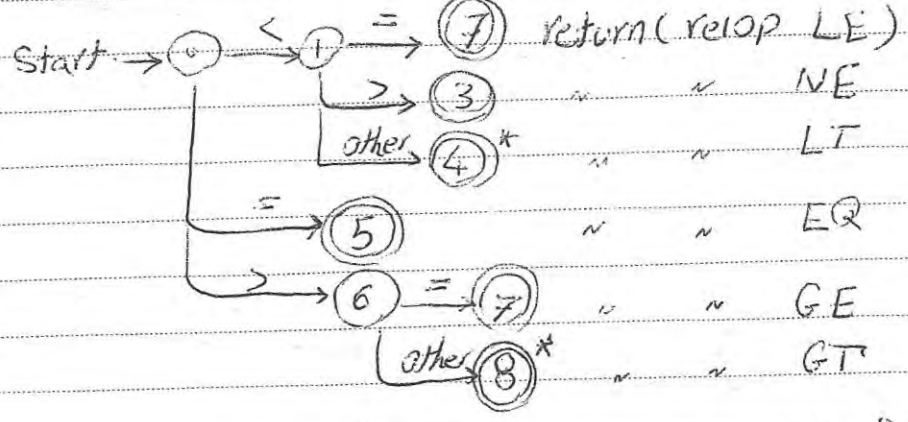


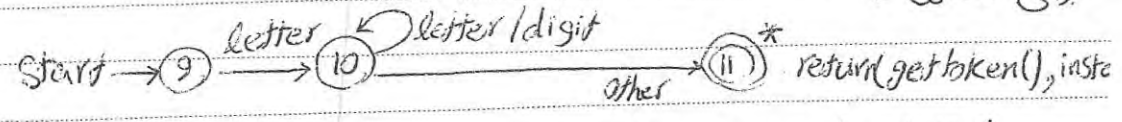
Subject
Date









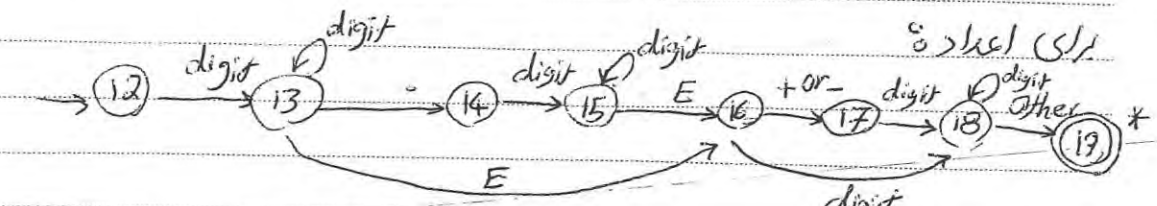


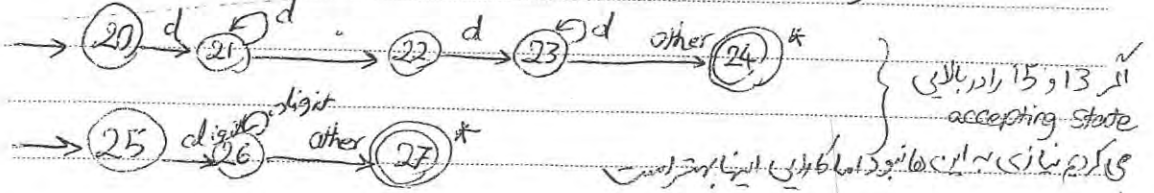

```
gettoken() {
Check Symbol table for the lexeme.
 if lexeme = keyword
    return (corresponding keyword)
 else
    return (id)
```

Subject
Date

```
install_id ( ) {
Check Symbol table for the lexeme
  if ( lexeme in symbol table )
    { if (lexeme = keyword)
        return (0)
      else
        return (pointer to the symbol table entry)
    }
  else
    { install ( lexeme as variable into Symbol table)
      return ( Pointer to the entry created )
    }
```

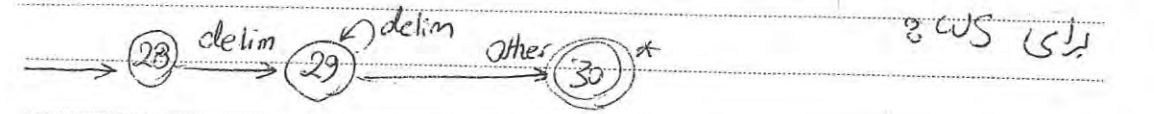
نکته: برای زبان های معمولی مثل C اگر با این مراحل (اول نمودار انتقالی بعدی) پیش بریم تعداد State ها کمتر از ۱۰۰ است در غیر این صورت چندصد state





اگر 13 و 15 را در بالایی accepting state می کردیم نیازی به این ها نبود اما کارایی اینها بهتر است

برای C++ ؟



اگر هیچ کدام از این ها را طی نکنیم ← error

جلسه ۱۱

۹۳/۷/۲۱

Subject
Date

چون این نمودارهای انتقالی از بالا به پایین چک می‌شوند اگر معادل کد برنامه متداول‌تر باشد مثل case ها، کدهایشان را بالاتر قرار دهیم بهینه‌تر می‌شود

جلسه ۱۱
کد مربوط به نمودارهای انتقالی

```
int State = 0, start = 0;
int lexical_value;
int fail()
{ forward = token_beginning;
   switch(start){
   case 0: start = 9; break;
        9  start = 12   "
        12   "   20   "
        20   "   25   "
        25   recover()  "
   default; /* Compiler error */
   }
  return start
}

token nextToken()
{ while(){
   switch(state){
    case 0; c = nextChar();
            if(c == blank || c = tab || c = newline);
            State = 0;
            lexeme_beginning ++;
}
```

Subject
Date

```
        else if (c == '<') state = 1;
             "    c == '=' state = 5;
             "    "   >   "   6;
        else state = fail();
        break;
        /* cases for 1-8 here */
case 9 : c = nextchar();
        if (isletter(c)) state = 10;
        else state = fail();
        break;
se 10 : c = nextchar();
        if (isletter(c)) state = 10
        else if (isdigit(c)) state = 10;
        else state = 11;
        break;
11 : retract(1);
     install-id();
     return(gettoken())
     /* cases 12-24 here */
;  : c = nextchar()
     if (isdigit(c)) state = 26
     else state = fail();
     break;
     c = nextchar()
     if (isdigit(c)) state = 26
     else          27
     break;
```

Subject
Date

```
27: retract(1);
    install_num();
    return (num);
}}}
```

reg. def. → lexical analyzer → همین کدها

Bottom up (Shift_reduce) parser

token → parser → Parse ساخت درخت ← Top Down Parser LL(k)
Bottom UP Parser از این نوع است LR(k)

LL(k) → Left most derivation
Left to right scan (خواندن فایل ورودی)
look ahead → چند تا char جلو را نگاه کند که Parser کدام تابع را فراخوانی کند

LR(k) → right most derivation

اگر گرامر مبهم بود LR یا LL ندارد

مثالی از یک گرامر که با LR کار می‌کند

S → a A c B e
A → A b | b → a b⁺ c d e
B → d

Subject
Date

مثال برای رشته abbcde زبان

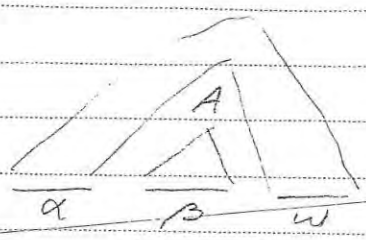
S
aACBe → nonterminal سمت راست ترین ← right most derivation
aAcde
aAbcde ↑ کاهش طول (reduction)
abbcde

جهت حرکت

مثال دیگر

$S \to aABe$
$A \to Abc \mid b$ زبان $ab(bc)^*de$
$B \to d$

S
aABe
aAde
aAbcde
abbcde



handle Pruning

handle { $A \to \beta$
Position β شروع

مثال دیگری برای روش Shift reduce

$E \to E + E$
$E \to E \times E$ → ضرب
$E \to (E)$
$E \to id$

Subject
Date

$id_1 + id_2 * id_3$

$E + id_2 * id_3$

$E + E * id_3$

$E + E * E$ right most derivation

$E + E$

E

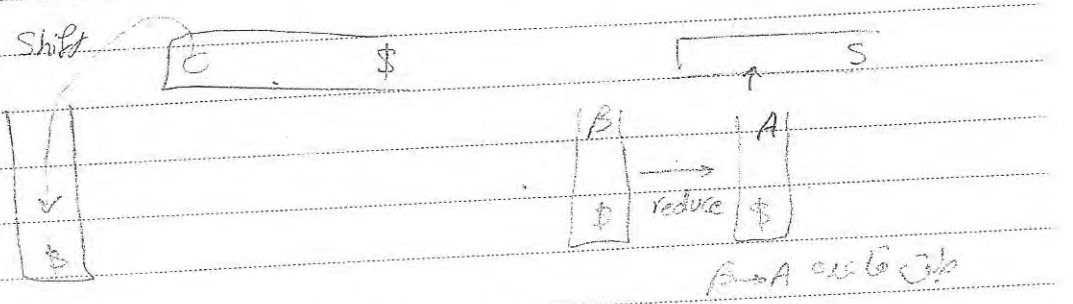
$E + E * id_3$

$E * id_3$ right most derivation

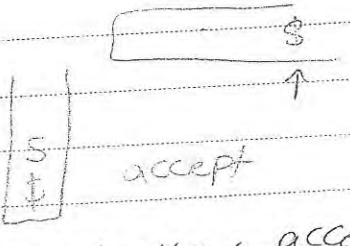
$E * E$

E

Shift/reduce conflict



$\beta \to A$ حالتی که



error ، accept ، reduce Shift هر حالتی یکی از

Subject
Date

$	$id_1 + id_2 * id_3$		
$ id1	$+id_2 * id_3$		
$ E	$+ id_2 * id_3$		
$ E +	$id_2 * id_3$		
$ E + id2	$* id_3$		
$ E + E	Shift ⟵ $* id_3$	reduce ⟶	$ E
$ E + E *	id_3		$ E *
$ E + E * id3	$		$ E * id3
$ E + E * E	$		$ E * E
$ E + E	$		$ E
$ E	$		

در yac با %prec اولویت ی دهیم و ابهام رفع ی شود مثال

```
%left + -
%left * /
```

جلسه ۱۲ ۹۳،۸،۳

ساختار yac

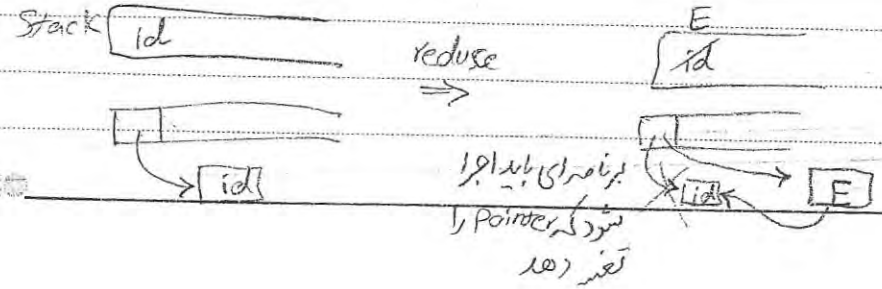
$E \to E + E$ { برنامه C }

$E \to E - E$

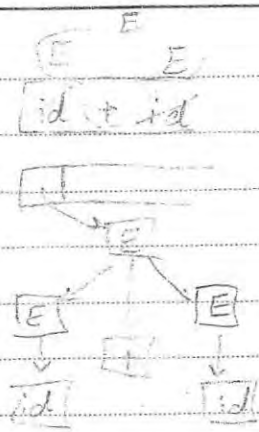
;

جلوی هر قاعده می توان برنامه ای نوشت که هرگاه آن قاعده استفاده شود آن برنامه اجرا شود

مثال



Subject
Date

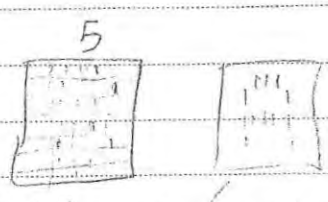


OCR تبدیل نوشته‌های داخل تصویر به text

با yac می‌توان OCR نوشت به کمک خاصیت Pattern recognition

یعنی yac هرچه در Stack هست را با الگوهای سمت راست قاعده‌ها منطبق

دهد مثلاً وقتی در Stack بالا $\begin{bmatrix} E \\ + \\ E \end{bmatrix}$ است از قاعده اول استفاده می‌کند



$S \rightarrow B$ { printf('B'); }

$S \rightarrow A$ { printf('A'); }

البته متن همیشه این‌قدر ساده نیست نوشته‌های متعارف ... همان است خطوط اضافه‌تر

حروف باشد و ...

Viable Prefix (پیشوند معتبر):

The set of prefixes of right sentential form that can appear on the stack of a shift-reduce parser

پیشوندهایی در Stack که ما می‌توانیم سمت راست یک قاعده شویم

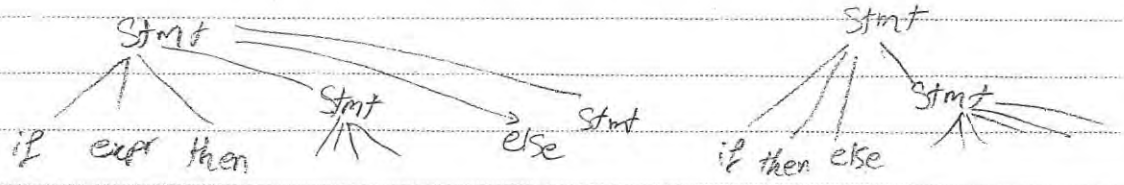
Subject
Date

یکی از دستوراتی که Shift reduce Conf ایجاد می‌کنند if است.

مثال:

Stmt → if expr then stmt
if expr then stmt else stmt

رفع ابهام با prec → %prec



if expr then stmt else.... Shift یا reduce؟

رفع ابهام با تعریف قاعده: else به نزدیک‌ترین if برمی‌گردد.

reduce / reduce conflict:

وقتی دو قاعده سمت راستشان یکی باشد و ندانیم به کدام reduce کنیم

مثال: تعریف تابع
Stmt → id (Parameter_list)
Stmt → expr := expr → asignment
Parameter_list → Parameter_list , Parameter | parameter
Parameter → id
expr → id (expr_list) → فراخوانی
expr → id
expr_list → expr_list , expr
expr_list → expr

id(id → {Parameter → id, expr → id} reduce reduce conflict

Subject
Date

مثال: id اول را وقتی می‌خواند تشخیص نمی‌دهد که متغیر است یا تابع

با در Symbol table نام token را تغییر می‌دهیم مثلاً Proc_id برای تابع

این مثال‌ها نشان می‌دهد که Symbol table نمی‌تواند توسط Context free پیاده‌سازی شود.

Parser

جدول LR Parser را yacc می‌سازد

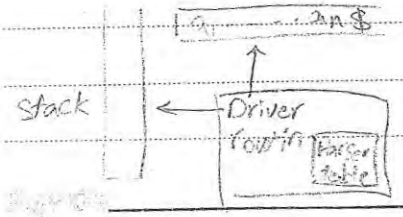
	State	Action						GOTO		
		id	+	*	(	)	$	E	T	F
1 $E \to E+T$	0	S5			S4			1	2	3
2 $E \to T$	1		S6				acc			
3 $T \to T*F$	2		r_2	S7		r_2	r_2			
4 $T \to F$	3		r_4	r_4		r_4	r_4			
5 $F \to (E)$	4	S5			S4			8	2	3
6 $F \to id$	5		r_6	r_6		r_6	r_6			
	6	S5			S4				9	3
	7	S5			S4					10
	8		S6			S11				
	9		r_1	S7		r_1	r_1			
	10		r_3	r_3		r_3	r_3			
	11		r_5	r_5		r_5	r_5			

State 5 با Shift

یک reduce با طول 1

این جدول را knuth اولین بار رسم کرده است.

Tree



تشخیص می‌دهد Shift یا reduce است Driver routin

Parser table دستورالعمل‌ها را دارد

Subject
Date

مثال: ساختار Stack : $(S_0 X_1 S_1 X_2 S_2 \cdots X_m S_m \quad a_i a_{i+1} \cdots a_n \$)$

top ↑ ورودی ↑

Shift S

S 5 $(S_0 X_1 S_1 X_2 S_2 \cdots X_m S_m a_i \; S \; , a_{i+1} \cdots a_n \$)$

goto$[S_m a_i]$

Reduce $A \to b$

r 5 r تعداد عملیاتها

به اندازه ۲r از Stack پاک می‌شود

$S_0 X_1 S_1 \cdots X_{m-r} S_{m-r} A \; S \quad a_i, a_{i+1} \cdots a_n \$)$

GOTO$[S_{m-r}, A] = S$

جدول تجزیه برای دستور زیر تشکیل شده

Stack	ورودی	
0	id * id + id $	State=0, id → S5
0 id 5	* id + id $	~ 5, * → r6
~~0 F 3~~	~~* id + id $~~	
0 T 2	* id + id $	
0 T 2 * 7	id + id $	
0 T 2 * 7 id 5	+ id $	
0 T 2 * 7 F 10	+ id $	
0 T 2	+ id $	0 E 1 + 6 F 3 $
0 E 1	+ id $	0 E 1 + 6 T 9 $
0 E 1 + 6	id $	0 E 1 $
0 E 1 + 6 id 5	$	

Subject
Date

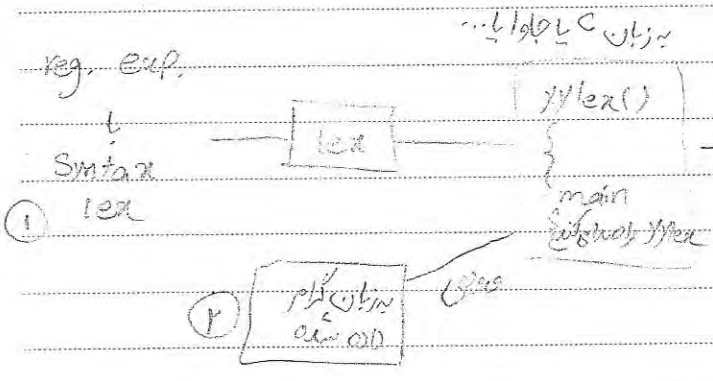
از این نوع جدول ها ۳ نوع وجود دارد { Simple LR, LALR, Canonical LR

افزایش قدرت ↓

(3)

SLR — Same # of states — For pascal Several hundred States
LALR — " " thousands "
Canonical LR

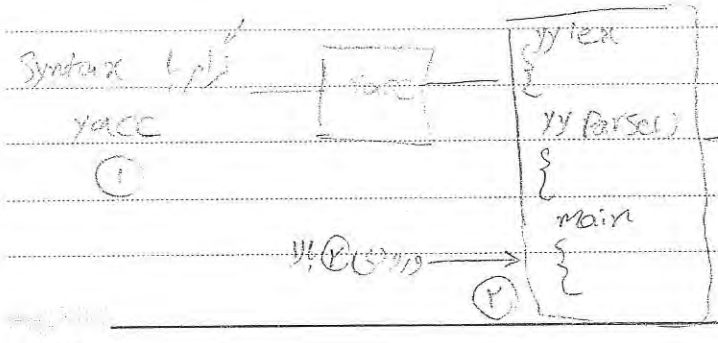
قسمت اول پروژه



* Print یک و دو و سه را تحویل می دهیم lex ۹۴,۸,۱۹ تحویل
yacc " ۹,۳

* آخرین روز کلاس تحویل کل پروژه

قسمت دوم پروژه



Subject
Date

Symbol table:

یک جدول است که وقتی فازهای مختلف کامپایلر اطلاعاتی تولید می‌کنند در آن قرار می‌دهند و از آن استفاده می‌کنند. به طور کلی آن جداول را طبقه‌بندی کنیم

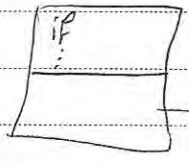
- Hash table ← سرعت بالا
- linear list

از Symbol table در آخر برنامه زیاد استفاده می‌شود. در مراحل اول با token کار می‌کنیم ولی در آخر اساسی از جدول استخراج شده و عبارت جواب را تولید می‌کنیم

عملیات: insert find delete

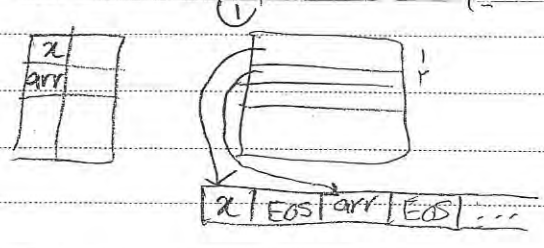
Hash table:

reserved word ها را نمی‌توان برای نام متغیرها استفاده کرد و این کلمات در بالای Symbol table قرار می‌گیرند



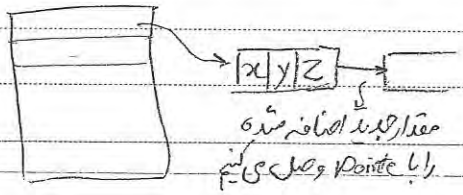
اگر از if برای نام متغیر استفاده کنیم error یا warning می‌دهد

linear list: برای حالتی که Scope نداریم (کد تو در تو نداریم)



آخر آن با EOS مشخص می‌شود

اشکال: Pointer زمان بر است



؟

Subject
Date

```
Program test;
Var i,j,k : integer;
Procedure P;
var i,j : char;
   Procedure Q;
    var i,k : real;
    begin
    i := 2.43;
    end;
  begin
  i := 'c';
  end
  Procedure R;
  Var x : real
  begin
  k := 2.4;
  end
begin
  i := 7
end
```

جدول خانه‌ها به صورت آرایه‌ای پوش

Scope با نام‌های که فقط دارد

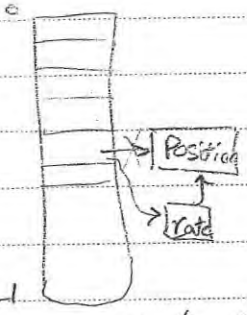
وقتی که از بالا به پایین برنامه حرکت از پایین به بالا اولین نمادی که دیده شده مورد نظر است

→	i	int
	J	int
	k	int
→	i	char
	j	char
→	i	real
	k	real

44

Subject
Date

راه دیگر: هرگاه می خواهیم متغیری را وارد جدول کنیم اول آن را به hash می دهیم که روشن ارایه ولینک لیست



h('Position') = 4

h('rate') = 4

در این روش hash func را باید طوری انتخاب کنیم که کلمات را مناسب توزیع کند مثلا همیشه عدد ۴ را بدهد مناسب نیست

همچنین سایز آرایه هم مهم است و m معمولا عدد اول است تا در توزیع مناسب تاثیر داشته باشد

در روش قبلی
insert O(n)
Search O(n)

hashing { insert O(1)
Search O(1)

مثال:

$h_0 = 0$

$h_i = a h_{i-1} + c_i$ for $1 \le i \le k$

let $h = h_k$ wher k is the length of the string

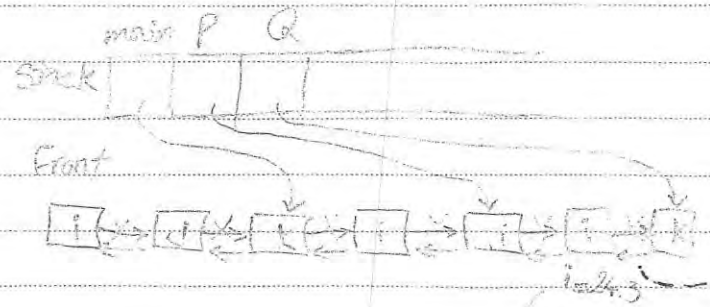
hash value = h mod m

با a های متفاوت و exclusive OR یا جمع می توان func های متفاوت ساخت

Subject
Date

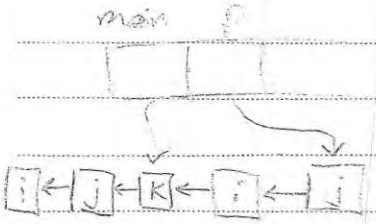
روش Stack و لینک لیست

برای برنامه‌های که صدا زده شده



جهت دلخواهی نمی‌خواهیم

از Q که مقادیر متغیرها را نگه می‌داریم با P بازگشت می‌کنیم



۲ ویژگی یا دو اشکال این روش

① Lex Analyzer ، Symbol table را می‌سازد با آن کار می‌کند و پاک می‌کند پس

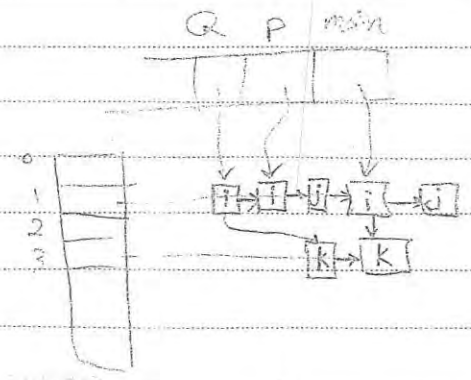
برای کامپایلر one pass مناسب است

اگر بعد از برنامه Symbol table را بخواهیم مثلا وقتی debug er دارم این روش مناسب نیست

② با نام یا ما تعریف شده کار می‌کند (cin)

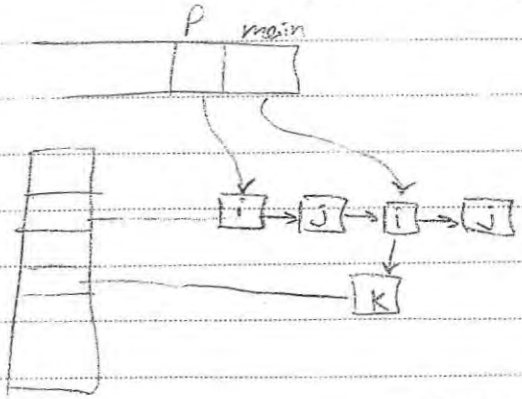
روش Stack ، hash

فرض کنید $h(i)=1$ و $h(j)=3$ و $h(k)=2$



Subject
Date

از Q که خارج می شویم از Pointer یکی Pointer i یعنی هم را delete می کنیم
عمومی عمومی

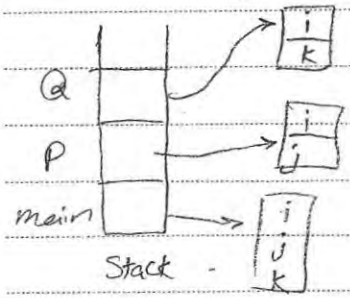


با استفاده از hash برای درج و جستجو O(1) است

روش Stack و آرایه

معمولا این روش برای Symbol table استفاده می شود

وقتی Q تمام شود از Stack یک POP می شود و آرایه اش هم پاک خواهد شد



همه این روش ها مشکل پاک شدن Symbol table را دارند

در linear list هزینه درج و جستجو O(n) است اما با درخت دودویی متوازن AVL می شود O(log n) و اگر hash func خوب پیدا کنیم O(1)

اگر به متغیرها شماره دهیم و مرتبشان کنیم linear list هم می تواند مانند AVL عمل کرده و O(log n) شود

Subject
Date

Runtime environment محیط زبان اجرای برنامه

src. prog. → [کامپایلر] → target prog.

↓

چیدمان داده‌ها در حافظه برای متغیرهای این برنامه در طول اجرا

data object : شیء داده‌ای که حافظه نیاز دارد

execution of Procedure : Activation of a Procedure

Procedure is recursive : در این صورت چند فعالیت از Proc در حال اجرا

Proc definition : header

- Procedure P(— — —) ← formal parameters

body

- begin

- end

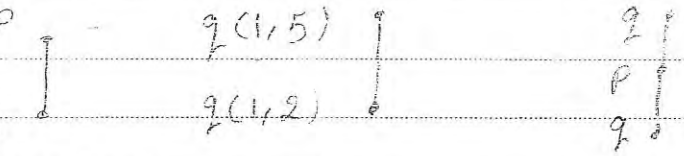
در هنگام call کردن Proc : actual parameters

P(— — —)

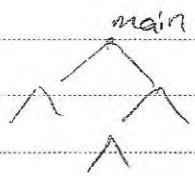
برنامه به صورت Sequential اجرا می‌شود اما وقتی به Proc برسد body آن را اجرا کرده و برمی‌گردد

life time of activation of a Proc : برای body آن

Subject
Date



ترتیب اجرای p و q و فراخوانی‌های زیر



در هنگام اجرای برنامه این درخت پیاده‌سازی می‌شود.
activation tree