

بسم الله الرحمن الرحيم



دانشگاه صنعتی اصفهان

دانشکده مهندسی برق و کامپیوتر

## پروژه درس کامپایلر

طراحی کامپایلر برای زبان برنامه نویسی Xlang

استاد درس

دکتر زینب زالی

ترم ۴۰۰۲

# فهرست مطالب

صفحه	عنوان
سه	فهرست مطالب
۱	تعریف پروژه
۲	فصل اول: تحلیلگر لغوی
۲	۱-۱ هدف
۲	۲-۱ حساسیت به حروف کوچک و بزرگ
۳	۳-۱ کلمات کلیدی
۳	۴-۱ متغیرها
۳	۵-۱ کامنت
۳	۶-۱ مقادیر ثابت
۳	۱-۶-۱ رشته و کاراکتر
۳	۲-۶-۱ اعداد
۴	۷-۱ عملگرها
۴	۸-۱ توکن‌های خاص
۴	۹-۱ تشخیص توکن‌ها
۴	۱۰-۱ خروجی تحلیلگر لغوی
۵	۱۱-۱ بخش امتیازی
۶	فصل دوم: تحلیلگر نحوی
۶	۱-۲ هدف
۶	۲-۲ قواعد نحوی زبان
۷	۱-۲-۲ نوع داده‌ها
۷	۲-۲-۲ متغیرها و آرایه‌ها
۷	۳-۲-۲ دستورات کنترلی
۸	۴-۲-۲ توابع و متدها
۱۰	۵-۲-۲ عملگرها
۱۰	۳-۲ گرامر مرجع
۱۲	۴-۲ خروجی تحلیلگر نحوی

۱۴	۵-۲ بخش امتیازی
----	-----------------

## ۱۵ فصل سوم: تحلیلگر معنایی و تولید کد

۱۵	۱-۳ هدف
۱۶	۲-۳ قواعد معنایی زبان
۱۶	۱-۲-۳ Scope قواعد
۱۷	۲-۲-۳ Locations
۱۷	۳-۲-۳ انتساب
۱۷	۴-۲-۳ فراخوانی توابع و بازگشت
۱۸	۵-۲-۳ دستورات کنترلی
۱۹	۶-۲-۳ عبارات
۲۰	۳-۳ جمع‌بندی قواعد معنایی
۲۲	۴-۳ تولید کد اسمبلی
۲۲	۱-۴-۳ نمونه سودو کدهای تولید کد اسمبلی
۲۴	۵-۳ خروجی تولیدکننده کد

## ۲۷ پیوست‌ها

۲۷	۱- پیوست ۱
----	------------

## تعریف پروژه

در این پروژه قصد داریم یک کامپایلر برای زبانی به نام *Xlang* طراحی و پیاده سازی کنیم. *Xlang* یک زبان دستوری ساده شبیه به *C* یا *Pascal* است. پیاده سازی این کامپایلر با استفاده از دو ابزار *Flex* و *Bison* انجام می شود و کامپایلر هدف باید بتواند یک فایل حاوی کد نوشته شده به زبان *Xlang* را دریافت کرده و با در نظر گرفتن سه بخش تحلیل لغوی<sup>۱</sup>، نحوی<sup>۲</sup> و معنایی<sup>۳</sup> و دیگر مفاهیم لازم که در درس کامپایلر مطالعه خواهید کرد یک کد خروجی به زبان اسمبلی تولید نماید.

هدف از طراحی این پروژه این است که کامپایلر مذکور را قدم به قدم و همگام با مطالبی که در درس می آموزید پیاده سازی کنیم تا با جنبه های عملی نوشتن یک کامپایلر ابتدایی، آشنا شوید.

---

<sup>۱</sup>Lexical Analysis

<sup>۲</sup>Syntax Analysis

<sup>۳</sup>Semantic Analysis

## فصل اول

### تحلیلگر لغوی

#### ۱-۱ هدف

در این فاز از پروژه می‌بایست یک تحلیلگر لغوی را با استفاده از ابزار *flex* نوشته و در محل مشخص شده درون سامانه آپلود کنید.

جهت ارزیابی، یک فایل حاوی قطعه کدی به زبان *Xlang* که در ادامه توصیف خواهد شد به تحلیلگر لغوی شما داده می‌شود، در صورتی که کد برنامه قواعد لغوی زبان برنامه نویسی *Xlang* را رعایت کرده باشد، شما باید در خروجی، توکن‌های آن برنامه را چاپ کنید و در غیر این صورت، بعد از مواجه شدن با خطا، بدون تولید هرگونه توکنی، می‌بایست خطای مناسب چاپ گردد.

#### ۱-۲ حساسیت به حروف کوچک و بزرگ

تمام کلمات کلیدی در زبان *Xlang* با حروف کوچک نوشته می‌شوند. کلمات کلیدی و شناسه‌ها<sup>۱</sup> حساس به حروف کوچک و بزرگ هستند<sup>۲</sup> مثلاً `if` یک کلمه کلیدی هست ولی `IF` نام یک متغیر است یا به طور مثال `foo` و `Foo` دو نام متفاوت برای اشاره به دو متغیر متفاوت هستند.

---

<sup>۱</sup>Identifiers

<sup>۲</sup>Case-Sensitive

## ۳-۱ کلمات کلیدی

در زبان *Xlang* کلمات کلیدی شامل موارد زیر است :

boolean	break	callout	class	continue	else	for	if
false	return	true	void	int			

## ۴-۱ متغیرها

در زبان *Xlang* متغیرها<sup>۱</sup> ترکیبی از حروف، اعداد انگلیسی و خط تیره<sup>۲</sup> هستند که حتماً باید با یک حرف و یا خط تیره آغاز شوند و هیچ تغییری نمی‌تواند با عدد آغاز شود.

## ۵-۱ کامنت

کامنت‌ها با // شروع می‌شوند و با پایان خط<sup>۳</sup> خاتمه می‌یابند. توجه! اصولاً کامنت‌ها به وسیله preprocessor پردازش می‌شوند و کامپایلر وظیفه پردازش آنها را ندارد، اما چون در این پروژه، preprocessor وجود ندارد باید به وسیله‌ی تحلیلگر لغوی پردازش شوند.

## ۶-۱ مقادیر ثابت

### ۱-۶-۱ رشته و کاراکتر

رشته‌ها ترکیبی از `<char>` ها هستند که در داخل "" قرار می‌گیرد. یک کاراکتر شامل یک `<char>` است که در داخل '' قرار می‌گیرد. منظور از `<char>` هر کاراکتر اسکی قابل چاپ (کاراکترهایی که کد اسکی نظیر آنها از ۳۲ تا ۱۲۶ است به جز کاراکترهای single quote (') ، backslash (\) و double quote (")) به علاوه پنج دنباله کاراکتری شامل (\') برای نمایش single quote ، (\\) برای نمایش backslash ، (\") برای نمایش double quote ، (\n) برای نمایش newline و (\t) برای نمایش tab می‌باشد.

### ۲-۶-۱ اعداد

اعداد در زبان *Xlang* ۳۲ بیتی و علامت‌دار هستند. همچنین در زبان *Xlang* فقط با اعداد صحیح<sup>۴</sup> کار می‌کنیم. اعداد صحیح به یکی از دو فرم زیر بیان می‌شوند :

<sup>1</sup>Variables

<sup>2</sup>Underscore OR \_

<sup>3</sup>Newline OR \n

<sup>4</sup>Integer

- دسیمال<sup>۱</sup> : مقادیر دسیمال از 2147483648- تا 2147483647 است.
- هگزا دسیمال<sup>۲</sup> : اگر یک دنباله با 0x آغاز شود و بعد از آن دنباله‌ای از کاراکترهای نشأت گرفته شده از [a-fA-F0-9] بیاید آنگاه دنباله مذکور بیانگر یک عدد هگزا دسیمال است.

## ۲-۱ عملگرها

عملگرهایی که در زبان ورودی مجاز هستند شامل عملگرهای محاسباتی، منطقی و شرطی می‌شوند که لیست آنها در زیر آمده است :

+ - \* / % < > != <= >= == && || ! = -= +=

## ۸-۱ توکن‌های خاص

توکن‌های خاص به توکن‌هایی گفته می‌شود که نه متغیر هستند نه کلمه کلیدی و نه عملگر که لیست آنها در زیر آمده است :

( ) { } [ ] ; ,

## ۹-۱ تشخیص توکن‌ها

توکن‌ها از طریق فاصله<sup>۳</sup> و یا از طریق توکن‌های خاص از هم جدا می‌شوند.  
توجه ! هر تعداد فاصله که بین دو توکن وارد شود بی‌تاثیر است و باید نادیده گرفته شود.

## ۱۰-۱ خروجی تحلیلگر لغوی

همانگونه که پیش تر اشاره شد، چنانچه یک برنامه‌ی صحیح به تحلیلگر شما داده شود باید بتواند توکن‌های آن را استخراج کند. به منظور استخراج توکن‌ها از برنامه ورودی، تنها نام آن توکن و مقدار آن را در خروجی چاپ نمایید (ابتدا نام توکن و سپس مقدار آن).

<sup>1</sup>Decimal

<sup>2</sup>Hexadecimal

<sup>3</sup>Whitespace: Tab, Space, ...



خروجی تحلیلگر لغوی شما برای یک نمونه کد صحیح به صورت زیر خواهد بود :

Input Code :

```
int x;
x = 5;
```

Analyzer Output :

```
TOKEN_INTTYPE int
TOKEN_WHITESPACE [space]
TOKEN_ID x
TOKEN_SEMICOLON ;
TOKEN_WHITESPACE [newline]
TOKEN_ID x
TOKEN_WHITESPACE [space]
TOKEN_ASSIGNOP =
TOKEN_WHITESPACE [space]
TOKEN_DECIMALCONST 5
TOKEN_SEMICOLON ;
```

خروجی تحلیلگر لغوی شما برای یک نمونه کد حاوی خطا به صورت زیر خواهد بود :

Input Code :

```
int 9comp;
9comp = 3;
```

Analyzer Output :

```
TOKEN_INTTYPE int
TOKEN_WHITESPACE [space]
error in line 1 : wrong id definition
```

در پیوست ۱ لیست کلیه توکن‌های موجود در زبان *Xlang* همراه با نام هر توکن آورده شده است.

توجه ! دو توکن `main` و `Program` در فاز بعدی به تفصیل شرح داده خواهند شد.

## ۱۱-۱ بخش امتیازی

در صورتی که تحلیلگر لغوی شما بتواند بعد از مواجه شدن با خطا ضمن چاپ پیغام مناسب ، از خطای موجود

عبور کرده و مابقی توکن‌ها را نیز استخراج کند، نمره امتیازی کسب خواهید کرد.

## فصل دوم

### تحلیلگر نحوی

#### ۱-۲ هدف

در این فاز از پروژه می‌بایست یک تحلیلگر نحوی را با استفاده از ابزار *bison* نوشته و در محل مشخص شده درون سامانه آپلود کنید.

جهت ارزیابی، یک فایل حاوی قطعه کدی به زبان *Xlang* که در ادامه ساختار جملات آن توصیف خواهد شد به تحلیلگر نحوی شما داده می‌شود، در صورتی که کد برنامه قواعد نحوی زبان برنامه نویسی *Xlang* را رعایت کرده باشد، شما باید در خروجی، *Syntax Tree* آن برنامه را چاپ کنید و در غیر این صورت، بعد از مواجه شدن با خطا، بدون تولید هیچ درختی، می‌بایست خطای مناسب چاپ گردد.

لازم به ذکر است این تحلیلگر نحوی می‌بایست توکن‌های برنامه را از خروجی تحلیلگر لغوی پیاده سازی شده در فاز پیشین دریافت نماید.

#### ۲-۲ قواعد نحوی زبان

یک برنامه نوشته شده به زبان *Xlang* شامل کلاسی تحت عنوان Program می‌باشد که از دو بخش *field* *declaration* و *method declaration* تشکیل شده است. بخش *field declaration* حاوی تعریف متغیرهایی

است که به صورت سراسری توسط تمامی متدهای برنامه قابل دسترسی و استفاده هستند و بخش `method` `declaration` حاوی تعریف توابع برنامه می باشد. کلاس `Program` الزماً می بایست شامل متدی تحت عنوان `main` بدون هیچ گونه آرگومان ورودی ای باشد. لازم به ذکر است نقطه شروع برنامه `Xlang` متد `main` خواهد بود.

## ۱-۲-۲ نوع داده ها

دو نوع داده اصلی در زبان `Xlang` تعریف می شود. این دو نوع داده، داده های صحیح و `true` یا `false` هستند که اختصاراً با کلمات کلیدی `int` و `boolean` نشان داده می شود.

## ۲-۲-۲ متغیرها و آرایه ها

در زبان `Xlang` متغیرها و آرایه هایی از نوع `int` و `boolean` قابل تعریف و استفاده هستند و تعریف آن ها به صورت زیر خواهد بود.

```
Variable Declaration:
    int var1, var2, var3;
    boolean v1;
```

```
Array Declaration:
    int arr[10];
    boolean a[3], b[5];
```

آرایه ها می بایست تنها در بخش `field declaration` کلاس `Program` تعریف شوند و تمامی آرایه ها تک بعدی بوده و آرگومان سائز نظیر آن ها یک مقدار ثابت خواهد بود و این مقدار به صورت ورودی از کاربر دریافت نمی شود. در این زبان هیچ گونه تعریفی برای آرایه های پویا نخواهیم داشت.

## ۳-۲-۲ دستورات کنترلی

دستورات کنترلی در زبان `Xlang` شامل دستورات شرطی و حلقه ها است که در ادامه به شرح آن ها می پردازیم:

### دستورات شرطی

شرط `if` ممکن است در کدها وجود داشته باشد. در این صورت ساختار آن به صورت زیر خواهد بود.

```
if (expr) {
    //if body
}
```

به علاوه شرط if ممکن است با else نیز همراه شود در این صورت ساختار آن به صورت زیر خواهد بود.

```
if (expr) {
    //if body
}
else {
    //else body
}
```

#### حلقه ها

حلقه for ممکن است در کدها وجود داشته باشد در این صورت ساختار آن به صورت زیر خواهد بود.

```
for x = expr , expr {
    //for body
}

for x = 1 , 10 {
    //for body
}
```

در کد فوق متغیر x را اندیس حلقه گوئیم. نخستین expr ، شروع حلقه و دومین expr ، انتهای حلقه را معین می سازد برای مثال در حلقه فوق، مقدار اولیه اندیس حلقه برابر 1 قرار داده شده و پس از هر مرتبه اجرای حلقه یک واحد به مقدار این اندیس افزوده می گردد تا زمانی که اندیس حلقه از مقدار 10 کوچکتر باشد دستورات موجود در بدنه اجرا خواهند شد.

لازم به ذکر است expr می تواند هر عبارتی باشد که معادل یک عدد صحیح است. برای مثال می تواند خروجی یک تابع یا حاصل یک عملیات ریاضیاتی نیز باشد.

#### ۴-۲-۲ توابع و متدها

متدها می توانند حداکثر چهار آرگومان ورودی داشته باشند. در صورتی که یک متد بیش از چهار آرگومان ورودی داشته باشد به منزله نقض قواعد زبان است.

بدین صورت تعریف و فراخوانی متدها به صورت زیر خواهد بود.

Method Declaration:

```
int method_name(int agr1, boolean arg2) {
    // method body
}
```

Method Call:

```
method_name(10, true);
```

لازم به ذکر است متدهایی که خروجی ندارند (از نوع void هستند) ضمن فراخوانی تنها می توانند در قالب یک جمله استفاده شوند و قابل استفاده در عبارات نیستند. (برای مثال اگر متد foo دارای خروجی صحیح باشد عبارت  $foo(args) + 3$  یک عبارت معتبر تلقی می شود در حالی که اگر متد foo بدون خروجی باشد تنها می توان این متد را به صورت  $foo(args)$  و در قالب یک جمله فراخوانی نمود)

همچنین در صورتی که یک متد خروجی داشته باشد می توان از آن هم در قالب بخشی از عبارات و هم در قالب یک جمله استفاده نمود که در اینصورت خروجی آن نادیده گرفته می شود. (برای متد foo که دارای یک خروجی صحیح است هم فراخوانی در قالب یک جمله یعنی  $foo(args)$  صحیح است و هم فراخوانی در قالب بخشی از یک عبارت همانند  $2 + foo(args) - 3$ )

توجه! بررسی خروجی متدها مربوط به فاز تحلیلگر معنایی است لذا در این فاز شما تنها می بایست ضمن تعریف گرامر هر دو حالت ذکر شده را در نظر بگیرید.

#### فراخوانی توابع آماده از کتابخانه‌های مختلف

زبان *Xlang* دارای یک روش برای فراخوانی توابع آماده در سیستم در زمان اجرای برنامه است، مانند توابعی در کتابخانه استاندارد زبان *C* یا توابع تعریف شده توسط کاربر با زبان‌هایی غیر از *Xlang* که با ابزارهای استاندارد کامپایل شده و موقع اجرا به برنامه *Xlang* لینک می‌شوند.

در واقع `callout` خود تابعی آماده در زبان *Xlang* است که به صورت زیر تعریف شده.

```
int callout ( <string_literal> [, <callout_arg>+, ] )
```

واضح است که نام تابعی که در کتابخانه‌ای خارج از برنامه فعلی موجود است و قصد فراخوانی آن را داریم به همراه آرگومان‌های مورد نیاز آن به `callout` پاس داده می‌شوند. عباراتی از نوع `int` و `boolean` به صورت عدد صحیح<sup>۱</sup> و رشته‌ها یا عباراتی از نوع آرایه به صورت اشاره‌گر<sup>۲</sup> به تابع مذکور پاس داده می‌شوند.

همچنین مقدار خروجی تابع مذکور به صورت عدد صحیح بازمی‌گردد و مقدار بازگشتی زمانی معتبر و قابل استفاده است که تابع مذکور در واقع مقداری از نوع مناسب را بازگرداند.

ضمناً بدیهی است که کاربر موظف است به تعداد مورد نیاز تابعی که قصد فراخوانی آن را دارد، آرگومان از نوع مناسب از طریق تابع `callout` به تابع موردنظر پاس دهد.

بدین استفاده از `callout` به صورت زیر خواهد بود.

```
callout("strcmp", "string 1", "string 2");
```

---

Integer<sup>۱</sup>  
Pointer<sup>۲</sup>

## ۵-۲-۲ عملگرها

عملگرها در این زبان به دو دسته تک عملوندی<sup>۱</sup> و دو عملوندی<sup>۲</sup> تقسیم می شوند. برای مثال عملگر ! یا همان نقیض یک عملگر تک عملوندی و عملگر && یک عملگر دو عملوندی محسوب می شود.

## ۳-۲ گرامر مرجع

همانطور که می دانید اصلی ترین بخش یک تحلیلگر نحوی تعریف گرامر مناسب برای زبان ورودی است. بدین منظور گرامر زیر به صورت اولیه برای این زبان تعریف شده است. بدیهی است در مواردی که گرامر مبهم باشد رفع ابهام بخش های مورد نیاز بر عهده شماست.

به این معنا که foo غیرترمینال است.	$\langle foo \rangle$
(با خط درشت) به این معنا است که foo ترمینال است.	foo
به این صورت که یک توکن یا بخشی از یک توکن است.	$[x]$
به معنای ظاهر شدن حداکثر یک $x$ (صفر یا یک رخداد) است به نحوی که $x$ اختیاری می باشد.	$x^*$
توجه داشته باشید که براکت در گیومه ، ' ' [ ' ' ] ، ترمینال است.	$x^+$
به معنای ظاهر شدن صفر یا بیشتر $x$ است.	$\{ \}$
یک لیست شامل حداقل یک $x$ که با کاما جدا شده باشند.	
کروشه بزرگ برای گروه کردن استفاده میشود.	
توجه داشته باشید که کروشه در گیومه ، ' ' { ' ' } ، ترمینال است.	
عملگر or	

$\langle \text{program} \rangle$	$\rightarrow$	<code>class Program '{' <math>\langle \text{field\_decl} \rangle^* \langle \text{method\_decl} \rangle^* \text{'}</math></code>
$\langle \text{field\_decl} \rangle$	$\rightarrow$	<code><math>\langle \text{type} \rangle \{ \langle \text{id} \rangle \mid \langle \text{id} \rangle \text{'['} \langle \text{int\_literal} \rangle \text{'}]' }^+, ;</math></code>
$\langle \text{method\_decl} \rangle$	$\rightarrow$	<code><math>\{ \langle \text{type} \rangle \mid \text{void} \} \{ \langle \text{id} \rangle \mid \text{main} \} ( [ \{ \langle \text{type} \rangle \langle \text{id} \rangle \}^+, ] ) \langle \text{block} \rangle</math></code>
$\langle \text{block} \rangle$	$\rightarrow$	<code>'{' <math>\langle \text{var\_decl} \rangle^* \langle \text{statement} \rangle^* \text{'}</math></code>
$\langle \text{var\_decl} \rangle$	$\rightarrow$	<code><math>\langle \text{type} \rangle \langle \text{id} \rangle^+, ;</math></code>
$\langle \text{type} \rangle$	$\rightarrow$	<code><code>int</code>   <code>boolean</code></code>
$\langle \text{statement} \rangle$	$\rightarrow$	<code><math>\langle \text{location} \rangle \langle \text{assign\_op} \rangle \langle \text{expr} \rangle ;</math>   <code><math>\langle \text{method\_call} \rangle ;</math></code>   <code><code>if</code> ( <math>\langle \text{expr} \rangle</math> ) <math>\langle \text{block} \rangle</math> [<code><code>else</code> <math>\langle \text{block} \rangle</math>]</code>   <code><code>for</code> <math>\langle \text{id} \rangle = \langle \text{expr} \rangle , \langle \text{expr} \rangle \langle \text{block} \rangle</math></code>   <code><code>return</code> [<math>\langle \text{expr} \rangle</math>] ;</code>   <code><code>break ;</code></code>   <code><code>continue ;</code></code>   <code><math>\langle \text{block} \rangle</math></code></code></code>
$\langle \text{assign\_op} \rangle$	$\rightarrow$	<code><code>=</code>   <code>+=</code>   <code>--</code></code>
$\langle \text{method\_call} \rangle$	$\rightarrow$	<code><math>\langle \text{method\_name} \rangle ( [ \langle \text{expr} \rangle^+, ] )</math>   <code><code>callout</code> ( <math>\langle \text{string\_literal} \rangle [ , \langle \text{callout\_arg} \rangle^+, ] )</math></code></code>
$\langle \text{method\_name} \rangle$	$\rightarrow$	<code><math>\langle \text{id} \rangle</math></code>
$\langle \text{location} \rangle$	$\rightarrow$	<code><math>\langle \text{id} \rangle</math>   <code><math>\langle \text{id} \rangle \text{'['} \langle \text{expr} \rangle \text{'}]'</math></code></code>
$\langle \text{expr} \rangle$	$\rightarrow$	<code><math>\langle \text{location} \rangle</math>   <code><math>\langle \text{method\_call} \rangle</math></code>   <code><math>\langle \text{literal} \rangle</math></code>   <code><math>\langle \text{expr} \rangle \langle \text{bin\_op} \rangle \langle \text{expr} \rangle</math></code>   <code><code>-</code> <math>\langle \text{expr} \rangle</math></code>   <code><code>!</code> <math>\langle \text{expr} \rangle</math></code>   <code><code>(</code> <math>\langle \text{expr} \rangle</math> <code>)</code></code></code>
$\langle \text{callout\_arg} \rangle$	$\rightarrow$	<code><math>\langle \text{expr} \rangle \mid \langle \text{string\_literal} \rangle</math></code>
$\langle \text{bin\_op} \rangle$	$\rightarrow$	<code><math>\langle \text{arith\_op} \rangle \mid \langle \text{rel\_op} \rangle \mid \langle \text{eq\_op} \rangle \mid \langle \text{cond\_op} \rangle</math></code>
$\langle \text{arith\_op} \rangle$	$\rightarrow$	<code><code>+</code>   <code>-</code>   <code>*</code>   <code>/</code>   <code>%</code></code>
$\langle \text{rel\_op} \rangle$	$\rightarrow$	<code><code>&lt;</code>   <code>&gt;</code>   <code>&lt;=</code>   <code>&gt;=</code></code>
$\langle \text{eq\_op} \rangle$	$\rightarrow$	<code><code>==</code>   <code>!=</code></code>
$\langle \text{cond\_op} \rangle$	$\rightarrow$	<code><code>&amp;&amp;</code>   <code>  </code></code>
$\langle \text{literal} \rangle$	$\rightarrow$	<code><math>\langle \text{int\_literal} \rangle \mid \langle \text{char\_literal} \rangle \mid \langle \text{bool\_literal} \rangle</math></code>
$\langle \text{id} \rangle$	$\rightarrow$	<code>TOKEN_ID</code>
$\langle \text{int\_literal} \rangle$	$\rightarrow$	<code><math>\langle \text{decimal\_literal} \rangle \mid \langle \text{hex\_literal} \rangle</math></code>
$\langle \text{decimal\_literal} \rangle$	$\rightarrow$	<code>TOKEN_DECIMALCONST</code>
$\langle \text{hex\_literal} \rangle$	$\rightarrow$	<code>TOKEN_HEXADECIMALCONST</code>
$\langle \text{bool\_literal} \rangle$	$\rightarrow$	<code>TOKEN_BOOLEANCONST</code>
$\langle \text{char\_literal} \rangle$	$\rightarrow$	<code>TOKEN_CHARCONST</code>
$\langle \text{string\_literal} \rangle$	$\rightarrow$	<code>TOKEN_STRINGCONST</code>

## ۴-۲ خروجی تحلیلگر نحوی

همانگونه که پیش تر اشاره شد، چنانچه یک برنامه‌ی صحیح به تحلیلگر شما داده شود باید بتواند *Syntax Tree* آن را استخراج کند و به صورت پیشوندی<sup>۱</sup> چاپ کند.

توجه داشته باشید به تحلیلگر شما یک آرگومان پاس داده می‌شود که اگر مقدار این آرگومان 1 باشد، باید Token Name برای ترمینال‌ها نمایش داده شود و در صورتی که این آرگومان 0 باشد، باید Token Value برای ترمینال‌ها چاپ شود.

بدین ترتیب، تحلیلگر شما باید این آرگومان را به صورت زیر دریافت کند:

\$ ./syntaxParser 0 or 1

خروجی تحلیلگر نحوی شما برای یک نمونه کد صحیح به صورت زیر خواهد بود:

Input Code :

```
class Program {
    int add(int a, int b){
        return a + b;
    }
    void main(){
        int a, b;
        a = 3;
        add(a);
    }
}
```

Analyzer Output (print by Token\_Name):

```
<program> TOKEN_CLASS TOKEN_PROGRAMCLASS TOKEN_LCB <method_decl> <type>
TOKEN_INTTYPE <id> TOKEN_ID TOKEN_LP <type> TOKEN_INTTYPE <id> TOKEN_ID
TOKEN_COMMA <type> TOKEN_INTTYPE <id> TOKEN_ID TOKEN_RP <block> TOKEN_LCB
<statement> TOKEN_RETURN <expr> <expr> <location> <id> TOKEN_ID <bin_op>
<arith_op> TOKEN_ARITHMATICOP <expr> <location> <id> TOKEN_ID
TOKEN_SEMICOLON TOKEN_RCB <method_decl> TOKEN_VOIDTYPE TOKEN_MAINFUNC
TOKEN_LP TOKEN_RP <block> TOKEN_LCB <var_decl> <type> TOKEN_INTTYPE <id>
TOKEN_ID TOKEN_COMMA <id> TOKEN_ID TOKEN_SEMICOLON <statement> <location>
<id> TOKEN_ID <assign_op> TOKEN_ASSIGNOP <expr> <literal> <int_literal>
<decimal_literal> TOKEN_DECIMALCONST TOKEN_SEMICOLON <statement>
<method_call> <method_name> <id> TOKEN_ID TOKEN_LP <expr> <location> <id>
TOKEN_ID TOKEN_RP TOKEN_SEMICOLON TOKEN_RCB TOKEN_RCB
```



Analyzer Output (print by Token\_Value):

```
<program> class Program { <method_decl> <type> int <id> add ( <type> int
<id> a , <type> int <id> b ) <block> { <statement> return <expr> <expr>
<location> <id> a <bin_op> <arith_op> + <expr> <location> <id> b ; }
<method_decl> void main ( ) <block> { <var_decl> <type> int <id> a , <id> b
; <statement> <location> <id> a <assign_op> = <expr> <literal> <int_literal>
<decimal_literal> 3 ; <statement> <method_call> <method_name> <id> add (
<expr> <location> <id> a ) ; } }
```

خروجی تحلیلگر نحوی شما برای یک نمونه کد صحیح به صورت زیر خواهد بود :

Input Code :

```
class Program {
    int globalVar;
    void main(){ }
}
```

Analyzer Output (print by Token\_Name):

```
<program> TOKEN_CLASS TOKEN_PROGRAMCLASS TOKEN_LCB <field_decl> <type>
TOKEN_INTTYPE <id> TOKEN_ID TOKEN_SEMICOLON <method_decl> TOKEN_VOIDTYPE
TOKEN_MAINFUNC TOKEN_LP TOKEN_RP <block> TOKEN_LCB TOKEN_RCB TOKEN_RCB
```

Analyzer Output (print by Token\_Value):

```
<program> class Program {<field_decl> <type> int <id> globalVar ;
<method_decl> void main ( ) <block> { } }
```

خروجی تحلیلگر نحوی شما برای یک نمونه کد حاوی خطا به صورت زیر خواهد بود :

Input Code :

```
class Program {
    int globalVar
    void main(){ }
}
```

Analyzer Output :

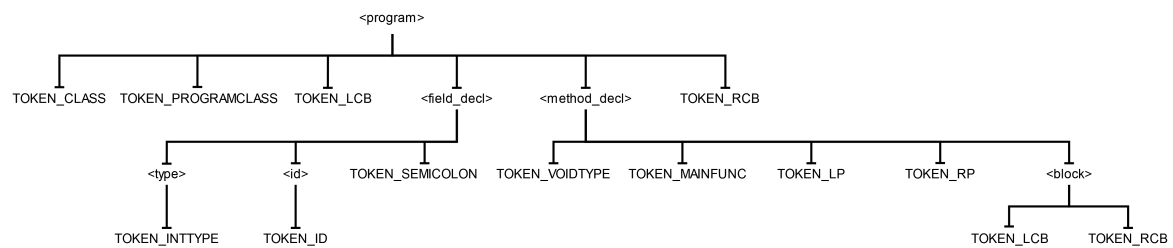
```
syntax error in line 2 : [expected ; at the end of statement]
```

## ۵-۲ بخش امتیازی

نمایش *Syntax Tree* به صورت دو بُعدی به شکل افقی یا عمودی، دارای نمره اضافه است.

به طور مثال، *Syntax Tree* برای قطعه کد زیر، رسم شده است.

```
class Program {
    int globalVar;
    void main(){ }
}
```



شکل ۱-۲: نمایش *Syntax Tree* به صورت دو بُعدی به شکل عمودی برای نمونه کد فوق

## فصل سوم

### تحلیلگر معنایی و تولید کد

#### ۱-۳ هدف

در فاز سوم و نهایی پروژه ، می‌بایست یک کامپایلر برای زبان *Xlang* را با استفاده از ابزارهای *flex* و *bison* نوشته و در محل مشخص شده درون سامانه آپلود کنید.

لازم به یادآوری است که در فازهای قبلی ، توانستیم برنامه ورودی را از لحاظ ایرادهای لغوی و نحوی بررسی کنیم و Syntax Tree آن را رسم کنیم ؛ اما همانطور که می‌دانید وظیفه کامپایلر صرفاً به تولید Syntax Tree ختم نمی‌شود بلکه خروجی یک کامپایلر باید یک برنامه به زبان اسمبلی باشد.

برای تولید کد اسمبلی ، کافیت semantic action های فاز قبل را به گونه‌ای تغییر بدهید که منجر به تولید کد اسمبلی شود بنابراین در این فاز ، همان برنامه‌ای که برای فاز اول و دوم پروژه آماده کرده بودید را تکمیل خواهید کرد.

زبان اسمبلی که برای کد خروجی در نظر گرفته‌ایم ، زبان MIPS است. البته قصد نداریم که از دستورات پیچیده آن استفاده کنید و صرفاً آشنایی مقدماتی با آن برای انجام این فاز کافی است.

در انتهای این فاز ، برنامه شما مشابه یک کامپایلر واقعی عمل خواهد کرد به این صورت که بعد از دریافت یک فایل حاوی قطعه کدی به زبان *Xlang* که قواعد لغوی، نحوی و معنایی زبان برنامه نویسی *Xlang* را رعایت

کرده باشد، یک فایل به زبان MIPS تولید کند به نحوی که قابلیت اجرا در محیط<sup>۱</sup> SPIM را داشته باشد و در غیر این صورت بدون تولید فایل خروجی و با نمایش خطای مربوطه خارج شود. لازم به ذکر است که اگر برای اجرای کد خروجی کامپایلر تان در فضای SPIM، به تعریف برخی ماکروها نیاز باشد بهتر است کامپایلر بتواند خودش آن‌ها را تولید کند. برای فهم بهتر از چگونگی تعریف ماکروها در صورت نیاز می‌تونید به این لینک مراجعه کنید.

## ۲-۳ قواعد معنایی زبان

هنگام پیاده سازی بخش تحلیل معنایی، قواعد زیر را در نظر داشته باشید :

### ۱-۲-۳ قواعد Scope

زبان *Xlang* قواعد scope ساده و محدودی دارد. تمام شناسه‌ها<sup>۲</sup> باید قبل از استفاده، تعریف شده باشند. به طور مثال :

- یک متغیر باید قبل از استفاده، تعریف شده باشد.
- یک تابع را فقط می‌توان با کدی که بعد از header آن ظاهر می‌شود، فراخوانی کرد. (پس توجه داشته باشید که توابع بازگشتی مجاز هستند.)

در هر نقطه از برنامه *Xlang* حداقل دو scope معتبر وجود دارد : Global Scope و Method Scope.

**global scope** شامل اسامی متغیرها و توابعی هست که در کلاس **Program** تعریف شده‌اند و **method scope** شامل اسامی متغیرها و پارامترهایی است که در داخل بدنه‌ی یک تابع تعریف شده‌اند. علاوه بر این دو، **local scope** نیز در هر *block* در کد وجود دارد. این می‌تواند بعد از **if** و یا **for** بیاید و یا در هر جایی از یک *statement* اضافه شود.

همچنین توجه داشته باشید که یک شناسه در یک **method scope** می‌تواند همانم شناسه‌ای در **global scope** باشد و به صورت مشابه، یک شناسه در **local scope** می‌تواند همانم شناسه‌ای در **scope** بالاتر (تابع یا سراسری) باشد که در این شرایط از شناسه‌ای که متعلق به همان **scope** هست استفاده می‌شود نه **scope** های بالاتر. به علاوه دقت کنید که هیچ شناسه‌ای نمی‌تواند بیش از یکبار در یک **scope** واحد تعریف شود.

<sup>۱</sup> یک برنامه‌ی شبیه‌سازی برای اجرای کدهای MIPS است  
<sup>۲</sup> Identifiers

### ۲-۲-۳ Locations

زبان برنامه نویسی *Xlang* دو نوع location دارد: متغیرهای اسکالر محلی/سراسری و آرایه‌های سراسری. هر location یک نوع دارد. location های از نوع داده **int** شامل مقادیر integer و location های از نوع داده **boolean** شامل مقادیر boolean هست تبعاً. Location هایی از انواع **int [N]** و **boolean [N]** نیز عناصر آرایه‌ای هستند. از آنجایی که آرایه‌ها سائز ثابتی دارند در فضای استاتیک داده‌های برنامه، حافظه می‌گیرند و نیازی به استفاده از heap ندارند.

هر location هنگامی که تعریف می‌شود به صورت دیفالت مقداردهی نیز می‌شود. اعداد صحیح به صورت دیفالت با صفر مقداردهی شده و **boolean** ها با مقدار **false**. همچنین متغیرهای محلی هنگامی که خط اجرای برنامه به scope که در آن تعریف شده‌اند می‌رسد، مقداردهی می‌شود و آرایه‌ها نیز هنگام شروع برنامه به صورت دیفالت مقداردهی می‌شوند.

### ۳-۲-۳ انتساب

انتساب فقط برای مقادیر اسکالر مجاز است. برای انواع داده **int** و **boolean** زبان *Xlang* از قواعد value-copy استفاده می‌کند به بیان دیگر، در انتساب  $\langle location \rangle = \langle expr \rangle$  مقدار معادل  $\langle expr \rangle$  در  $\langle location \rangle$  کپی می‌شود و یا در  $\langle location \rangle += \langle expr \rangle$  مقدار ذخیره شده در  $\langle location \rangle$  به اندازه مقدار معادل  $\langle expr \rangle$  افزایش خواهد یافت و این تنها زمانی معتبر است که  $\langle expr \rangle$  و  $\langle location \rangle$  هر دو از نوع داده **int** باشند به صورت مشابه برای  $\langle location \rangle -= \langle expr \rangle$  مقدار ذخیره شده در  $\langle location \rangle$  به اندازه مقدار معادل  $\langle expr \rangle$  کاهش خواهد یافت و این تنها زمانی معتبر است که  $\langle expr \rangle$  و  $\langle location \rangle$  هر دو از نوع داده **int** باشند.

در هر انتساب  $\langle expr \rangle$  و  $\langle location \rangle$  باید هر دو از یک نوع داده باشند. برای آرایه‌ها،  $\langle expr \rangle$  و  $\langle location \rangle$  باید به یک عنصر خاص از آرایه که یک مقدار اسکالر است، اشاره داشته باشند.

بدیهی است که انتساب به پارامترهای داخل بدنه‌ی یک تابع نیز مجاز هست و این انتساب تنها در scope تابع تاثیرگذار هست.

### ۴-۲-۳ فراخوانی توابع و بازگشت

فراخوانی تابع شامل سه مرحله است:

۱. پاس دادن آرگومان‌ها از فراخوانی‌کننده<sup>۱</sup> به فراخوانی‌شده<sup>۲</sup>

۲. اجرای بدنه تابع

۳. بازگشت خط اجرا از فراخوانی شده به فراخوانی کننده، معمولاً همراه با یک مقدار بازگشتی.

پاس دادن آرگومان‌ها تحت عنوان انتساب انجام می‌شود به این صورت که پارامترهای رسمی تابع، متغیرهایی در `scope` تابع تعریف می‌شوند و مقداردهی آنها با مقادیری که فراخوانی‌کننده به تابع می‌فرستد انجام می‌شود. مقداردهی آرگومان‌ها از چپ به راست انجام می‌شود.

سپس بدنه تابع خط به خط اجرا می‌شود.

تابعی که خروجی نداشته باشد (`void`) فقط می‌تواند به صورت یک جمله تنها استفاده شود و نباید در یک عبارت استفاده گردد. در این توابع وقتی خط اجرا به دستور `return` یا در صورت عدم وجود این دستور، به انتهای بدنه تابع می‌رسد، کنترل را به فراخوانی‌کننده برمی‌گرداند.

توابعی که مقداری را برمی‌گردانند، می‌توانند در یک عبارت استفاده شوند که در این صورت، مقدار بازگردانده شده تابع در عبارت مذکور استفاده می‌شود. این توابع را می‌توان به صورت یک جمله تنها نیز استفاده کرد که در این صورت مقدار بازگردانده شده تابع، نادیده گرفته می‌شود.

### ۳-۲-۵ دستورات کنترلی **if**

دستور **if** قاعده نحوی مشابه زبان برنامه نویسی *C* دارد. ابتدا  $\langle expr \rangle$  ارزیابی می‌شود. اگر ارزش آن **true** باشد آنگاه بدنه **if** اجرا می‌شود در غیر این صورت، بدنه **else** (در صورت وجود) اجرا می‌گردد. از آنجایی که در زبان *Xlang* برای مشخص کردن بدنه **if** و **else** از **کروشه** استفاده می‌شود (حتی اگر بدنه هر یک از این دو دستور شامل تنها یک جمله باشد). پس ابهامی در اینکه هر **else** متعلق به کدام **if** است پیش نخواهد آمد.

### **for**

دستور **for** مشابه دستور **do** در زبان *Fortran* می‌باشد.  $\langle id \rangle$  همان اندیس حلقه است و نیازی نیست که قبلاً تعریف شده باشد (توجه داشته باشید که اگر قبل از حلقه متغیری با همان نام اندیس حلقه تعریف شده باشد، در بدنه حلقه از آن استفاده نمی‌شود و مقدار آن تغییر نخواهد کرد). اندیس حلقه یک متغیر از نوع **int** است که `scope` آن محدود به حلقه نظیر آن می‌باشد. اولین  $\langle expr \rangle$  مقدار آغازین اندیس حلقه و دومین  $\langle expr \rangle$  مقدار نهایی اندیس حلقه را مشخص می‌کند. هر یک از این  $\langle expr \rangle$  ها یکبار قبل از اجرای دستور حلقه، ارزیابی می‌شود. نوع هر یک از  $\langle expr \rangle$  ها باید **int** باشد. بدنه حلقه زمانی اجرا می‌شود که مقدار فعلی اندیس حلقه از مقدار نهایی آن کمتر باشد. بعد از اجرای بدنه حلقه، مقدار اندیس یک واحد افزایش می‌یابد و مقدار جدید با مقدار نهایی مقایسه می‌شود تا بررسی شود آیا حلقه باید یک بار دیگر اجرا شود یا خیر.

## ۳-۲-۶ عبارات

عبارت‌ها از قواعد متداول و نرمال برای ارزیابی مقدار نهایی‌شان پیروی می‌کنند. در غیاب محدودیت‌های دیگر، عملگرهایی از یک سطح اولویت، از چپ به راست محاسبه می‌شوند. پرانتز بالاترین اولویت را دارد. لیترال‌های صحیح معادل مقدار صحیح نظیرشان هستند و لیترال‌های کاراکتری معادل ارزش اسکی نظیرشان هستند مثلاً 'A' بیانگر عدد صحیح 65 است.

جدول اولویت عملگرها از بالاترین به پایین‌ترین :

عملگرها	توضیحات
-	منفی تک‌عملوندی
!	not منطقی
* / %	باقیمانده، تقسیم، ضرب
+ -	تفریق، جمع
< <= >= >	رابطه‌ای
== !=	عدم تساوی، تساوی
&&	and شرطی
	or شرطی

در فاز اول و دوم پروژه به ازای همه عملگرهای محاسباتی توکن یکسان TOKEN\_ARITHMATICOP در نظر گرفته شد.

لیکن جهت تولید کد در این فاز نیاز است این عملگرها را تفکیک کرده و به ازای هر یک توکن جداگانه‌ای در نظر بگیرید. این کار را می‌توانید با ایجاد تغییرات بسیار اندک در فاز ۱ و ۲ انجام دهید. بدین ترتیب در این فاز با در نظرگیری توکن‌های زیر به گونه‌ای مناسب اولویت دهی به عملگرهای محاسباتی را درون گرامر لحاظ نمایید.

<i>Token</i>	<i>Token Name</i>
+	TOKEN_ARITHMATICOP_ADD
-	TOKEN_ARITHMATICOP_SUB
*	TOKEN_ARITHMATICOP_MUL
/	TOKEN_ARITHMATICOP_DIV

تقسیم بر 0 را چه به صورت صریح و چه به صورت غیرصریح کنترل کنید و در صورت وجود تقسیم بر 0، صرفاً یک هشدار چاپ کنید. (دقت کنید که خطایی تولید نمی‌شود یعنی کد اسمبلی تولید می‌شود و تنها یک هشدار به کاربر داده می‌شود.)

### ۳-۳ جمع‌بندی قواعد معنایی

این قوانین، محدودیت‌های بیشتری را روی برنامه‌های زبان *Xlang* اعمال می‌کنند. هر برنامه‌ای که از نظر گرامر زبان، خوش-فرم باشد و تناقضی با قوانین معنایی زیر نداشته باشد، یک برنامه *legal* نامیده می‌شود. یک کامپایلر قدرتمند باید هریک از قوانین زیر را چک کرده و در صورت مواجه شدن با خطا، پیغام ارور مناسب را چاپ نماید.

۱. هیچ شناسه‌ای در یک *scope* واحد، دو بار تعریف نمی‌شود.
۲. هیچ شناسه‌ای پیش از تعریف شدن، استفاده نمی‌شود.
۳. برنامه شامل یک تابع به نام **main** است که هیچ پارامتر ورودی ندارد. (توجه داشته باشید از آنجایی که نقطه شروع برنامه‌های *Xlang* تابع **main** است، پس هر تابعی که بعد از آن تعریف شود هیچگاه اجرا نخواهد شد.)
۴. در تعریف آرایه، اندازه آرایه باید بزرگتر از ۰ باشد.
۵. در فراخوانی تابع، تعداد و نوع مقادیر پاس داده شده به تابع باید با تعداد و نوع آرگومان‌ها در تعریف تابع یکسان باشد.
۶. اگر در یک عبارت، فراخوانی تابع وجود داشت، آن تابع باید حتما خروجی داشته باشد و از نوع **void** نباشد.
۷. دستور **return** نباید شامل مقدار بازگشتی باشد مگر آنکه در تعریف تابع، خروجی از نوع **int** یا **boolean** لحاظ شده باشد.
۸. نوع مقدار بازگشتی در دستور **return** باید با نوع خروجی در تعریف تابع یکی باشد.
۹. هرگاه  $\langle id \rangle$  به عنوان  $\langle location \rangle$  استفاده می‌شود باید قبلاً به عنوان یک متغیر سراسری یا محلی و یا پارامتر رسمی در تابع، تعریف شده باشد.
۱۰. برای هر  $location$  به فرم  $\langle id \rangle [\langle expr \rangle]$

( $\bar{A}$ )  $\langle id \rangle$  باید نام متغیری که بیانگر یک آرایه است، باشد و



(ب) نوع  $\langle expr \rangle$  باید **int** باشد.

۱۱. نوع  $\langle expr \rangle$  در دستور **if** باید **boolean** باشد.

۱۲. نوع عملوندهای عملگرهای محاسباتی و رابطه‌ای باید **int** باشد.

۱۳. نوع عملوندهای عملگرهای تساوی و عدم تساوی باید یکی باشد، یا **int** یا **boolean**.

۱۴. نوع عملوندهای عملگرهای شرطی و **not** منطقی (!) باید **boolean** باشد.

۱۵. در یک انتساب به فرم  $\langle location \rangle = \langle expr \rangle$  باید نوع  $\langle expr \rangle$  و  $\langle location \rangle$  یکی باشد.

۱۶. در یک انتساب افزایشی/کاهشی به فرم  $\langle location \rangle += \langle expr \rangle$  و  $\langle location \rangle -= \langle expr \rangle$  باید نوع  $\langle expr \rangle$  و  $\langle location \rangle$  **int** باشد.

۱۷.  $\langle expr \rangle$  شروع و  $\langle expr \rangle$  پایان حلقه‌ی **for** باید از نوع **int** باشند.

۱۸. تمام دستورات **break** و **continue** فقط باید در بدنه‌ی حلقه‌های **for** آمده باشند.

۱۹. اندیس آرایه حتماً باید در محدوده اندازه‌ی آرایه باشد.

۲۰. کنترل نباید بدون آنکه مقداری را بازگرداند به انتهای تابعی که باید خروجی داشته باشد، برسد.

### ۴-۳ تولید کد اسمبلی

برای پیاده سازی بخش تولید کد، به چند نکته توجه کنید :

۱. نگران پرش‌های با طول زیاد نباشید، فرض می‌شود که آدرس‌های توابع و `jump` با هر دستور پرش قابل دسترس هستند.

۲. برای نگهداری آرایه‌ها و متغیرها از حافظه استاتیک استفاده کنید و نیاز به استفاده از `heap` نیست.

### ۱-۴-۳ نمونه سودوکدهای تولید کد اسمبلی

در ادامه چند نمونه از تبدیل ک ورودی به کد اسمبلی خروجی آورده شده است.

تذکر : این کدهای اسمبلی به زبان MIPS نیست و صرفاً جنبه سودوکد دارد تا شما با فرایند تولید کد آشنا شوید.

#### عملیات unary

Input Code : -3

Pseudo Code:

```
movl    $3, %eax; // EAX register contains 3
neg     %eax;     // now EAX register contains -3
```

#### عملیات binary

در مثال زیر، ابتدا باید کدهای مربوط به  $E1$  را تولید کنیم و مقدار آن را در استک ذخیره کنیم و سپس کدهای مربوط به  $E2$  را تولید می‌کنیم و مقدار آن را نیز محاسبه می‌کنیم. حال مقدار  $E1$  را از استک برداشته و عملیات جمع را انجام می‌دهیم.

Input Code :  $E1 + E2$

Pseudo Code:

```
<CODE FOR E1 GOES HERE>
push    %eax;          // save value of E1 on the stack
<CODE FOR E2 GOES HERE>
pop     %ecx;          // pop E1 from the stack into ECX register
addl    %ecx, %eax;    // add E1 to E2, save results in EAX
```

#### عملیات binary اتصال کوتاه

عملیات‌هایی مثل `&&` و `||` که ممکن است نیازی به اجرای کل دستور نباشد را در این دسته قرار می‌دهیم. همان مراحل قبل را طی می‌کنیم با این تفاوت که اگر پس از محاسبه  $E1$  نتیجه محاسبات به صورت قطعی تعیین گردد پس دیگر نیازی به محاسبه  $E2$  نیست و به سراغ کامپایل خط بعدی در برنامه می‌رویم.

Input Code : E1 || E2

Pseudo Code:

```

<CODE FOR E1 GOES HERE>
    cmpl    $0, %eax; // check if E1 is true
    je      _clause2; // E1 is 0, so we need to evaluate clause 2
    movl    $1, %eax; // we didn't jump, so E1 is true and therefore result is 1
    jmp     _end;
_clause2:
    <CODE FOR E2 GOES HERE>
    cmpl    $0, %eax; // check if E2 is true
    movl    $0, %eax; // zero out EAX without changing 2F
    setne   %al;      // set AL register (the low byte of EAX) to 1 iff E2 != 0
_end:
    jmp     _end;

```

### عبارت‌های شرطی

مقدار  $E1$  را محاسبه کرده و آن را با 0 مقایسه می‌کنیم چنانچه برابر با صفر بود آنگاه کد مربوط به قسمت **else** باید اجرا گردد و در غیر این صورت، تنها کد مربوط به  $E2$  باید انجام شود.

Input Code : if (E1) { E2 } else { E3 }

Pseudo Code:

```

<CODE FOR E1 GOES HERE>
    cmpl    $0, %eax;
    je      _E3; // if (E1) == 0, E1 is false so execute E3
    <CODE FOR E2 GOES HERE> // we're still here so E1 must be true. execute E2.
    jmp     _post_conditional; // jump over E3
_E3:
    <CODE FOR E3 GOES HERE> // we jumped here because E1 was false. execute E3.
_post_conditional: // we need this label to jump over E3

```

### فراخوانی تابع

• توابع موجود در کلاس **Program** :

Input Code : foo(1, 2, 3)

Pseudo Code:

```

    push    $3; // save parameters in registers/stack
    push    $2;
    push    $1;
    call    _foo; // call method
    add     $0xC, %esp; // remove input arguments from stack
_foo:

```

```

push    %ebp; // Save the stack start address corresponding to the
           // caller function
mov     $esp, %ebp; // initialize stack for foo function
<DO STUFF> // execute function body
mov     %ebp, %esp; // Delete all variables taken from the stack
           // during function execution
pop     %ebp; // Restore stack information related to caller function
ret; // return to previous function

```

• توابع آماده از کتابخانه‌ها مختلف با استفاده از **callout** :

Input Code : `callout("atoi", "12");`

Pseudo Code:

```

push    $0x3132; // ascii code for "12" is 0x3132
call    atoi;

```

### ۵-۳ خروجی تولیدکننده کد

خروجی چند نمونه کد اسمبلی MIPS به صورت زیر می باشد. لازم به ذکر است خروجی کد اسمبلی نظیر یک برنامه نوشته شده به زبان سطح بالا، لزوماً یکتا نیست.

نمونه اول:

Input Code :

```

class Program{
    void main(){
        int s=3;
        int k=9;
        k=k+s;
    }
}

```

Output Code:

```

.globl main
main:
    addi $s0, $zero, 3
    addi $s1, $zero, 9
    add  $t0, $s1, $s0
    move $s1, $t0
    li  $v0, 10
    syscall

```

## نمونه دوم:

Input Code :

```

class Program{
    int func(int a,int b){
        return a+b;
    }
    void main(){
        int t;
        t = func(3, 7);
    }
}

```

Output Code:

```

.globl main
func:
    add $t0, $a0, $a1
    move $v0, $t0
    jr $ra
main:
    addi $a0, $zero, 3
    addi $a1, $zero, 7
    jal func
    move $s0, $v0
    li $v0, 10
    syscall

```

## نمونه سوم:

Input Code :

```

class Program{
    void main(){
        int k;
        k = 5 * 3 + 6 / 2;
        int t;
        t = (k - 10) * 4 + 2 - (2 * 3);
    }
}

```

Output Code:

```

.globl main
main:
    addi $t0, $zero, 15
    addi $t1, $zero, 3
    add $t2, $t0, $t1
    move $s0, $t2
    addi $t0, $s0, -10
    addi $t1, $zero, 4

```

```

mul  $t1, $t0, $t1
addi $t0, $t1, 2
addi $t1, $zero, 6
sub  $t2, $t0, $t1
move $s1, $t2
li   $v0, 10
syscall

```

نمونه چهارم:

Input Code :

```

class Program{
    void main(){
        int k;
        k = 9;
        for i = 0, 5 {
            k = k + 1;
        }
    }
}

```

Output Code:

```

.globl main
main:
    addi  $s0, $zero, 9
    addi  $s1, $zero, 0
LOOP1:
    addi  $t0, $zero, 5
    slt   $t0, $s1, $t0
    beq   $t0, $zero, L1
    addi  $t1, $s0, 1
    move  $s0, $t1
    addi  $t1, $s1, 1
    move  $s1, $t1
    j     LOOP1
L1:
    li    $v0, 10
    syscall

```

<i>Token</i>	<i>Token Name</i>
boolean	TOKEN_BOOLEANATYPE
break	TOKEN_BREAKSTMT
callout	TOKEN_CALLOUT
class	TOKEN_CLASS
continue	TOKEN_CONTINUESTMT
else	TOKEN_ELSECONDITION
false	TOKEN_BOOLEANCONST
for	TOKEN_LOOP
if	TOKEN_IFCONDITION
int	TOKEN_INTTYPE
return	TOKEN_RETURN
true	TOKEN_BOOLEANCONST
void	TOKEN_VOIDTYPE
Program	TOKEN_PROGRAMCLASS
main	TOKEN_MAINFUNC
<variables>	TOKEN_ID
+	TOKEN_ARITHMATICOP
-	TOKEN_ARITHMATICOP
*	TOKEN_ARITHMATICOP
/	TOKEN_ARITHMATICOP
%	TOKEN_ARITHMATICOP
&&	TOKEN_CONDITIONOP
	TOKEN_CONDITIONOP
<=	TOKEN_RELATIONOP
<	TOKEN_RELATIONOP
>	TOKEN_RELATIONOP
>=	TOKEN_RELATIONOP
!=	TOKEN_EQUALITYOP
==	TOKEN_EQUALITYOP
=	TOKEN_ASSIGNOP
+=	TOKEN_ASSIGNOP
-=	TOKEN_ASSIGNOP
!	TOKEN_LOGICOP
(	TOKEN_LP
)	TOKEN_RP
{	TOKEN_LCB
}	TOKEN_RCB
[	TOKEN_LB
]	TOKEN_RB
;	TOKEN_SEMICOLON
,	TOKEN_COMMA
\n [newline]	TOKEN_WHITESPACE
\t [tab]	TOKEN_WHITESPACE
[space]	TOKEN_WHITESPACE
//[some string until \n]	TOKEN_COMMENT
3 [or other decimal integers]	TOKEN_DECIMALCONST
0xFF [or other hexadecimal integers]	TOKEN_HEXADECEMALCONST
"[some string]"	TOKEN_STRINGCONST
'a'[or other characters]	TOKEN_CHARCONST