
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

عنوان **صفحه**

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 $Bison$ و $Flex$ 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[1]، نحوی[2] و معنایی [3] 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 می‌آموزید پیاده سازی کنیم تا با جنبه‌های عملی نوشتن یک کامپایلر ابتدایی، آشنا شوید.

[1] Lexical Analysis
[2] Syntax Analysis
[3] Semantic Analysis

فصل اول

تحلیلگر لغوی

۱ـ۱ 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.

۱ـ۲ حساسیت به حروف کوچک و بزرگ

تمام کلمات کلیدی در زبان $Xlang$ با حروف کوچک نوشته می‌شوند. کلمات کلیدی و شناسه‌ها [1] حساس به حروف کوچک و بزرگ هستند [2] 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.

[1] Identifiers
[2] Case-Sensitive

۱ـ۳ کلمات کلیدی

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

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

۱ـ۴ متغیرها

در زبان $Xlang$ متغیرها [۱] ترکیبی از حروف، اعداد انگلیسی و خط تیره [۲] هستند که حتما باید با یک حرف و یا خط تیره آغاز شوند و هیچ متغیری نمی‌تواند با عدد آغاز شود.

۱ـ۵ کامنت

کامنت‌ها با // شروع می‌شوند و با پایان خط [۳] خاتمه می‌یابند.
توجه ! اصولاً کامنت ها 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.

۱ـ۶ مقادیر ثابت

۱ـ۶ـ۱ 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.

۱ـ۶ـ۲ اعداد

اعداد در زبان $Xlang$ ۳۲ بیتی و علامت‌دار هستند. همچنین در زبان $Xlang$ فقط با اعداد صحیح [۴] کار می‌کنیم. اعداد صحیح به یکی از دو فرم زیر بیان می‌شوند :

---
[1] Variables
[2] Underscore OR _
[3] Newline OR $\backslash n$
[4] Integer

- دسیمال[1] : مقادیر دسیمال از 2147483648− تا 2147483647 است.
- هگزا دسیمال[2] : 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.

۱ـ۷ عملگرها

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

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

۱ـ۸ 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 :

```
(  )    {  }    [  ]    ;    ,
```

۱ـ۹ تشخیص توکن‌ها

توکن‌ها از طریق فاصله [3] 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.

۱ـ۱۰ 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).

---
[1] Decimal
[2] Hexadecimal
[3] 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.

۱ـ۱۱ بخش امتیازی

درصورتی 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

۲ـ۱ 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 می بایست توکن های برنامه را از خروجی تحلیلگر لغوی پیاده سازی شده در فاز پیشین دریافت نماید.

۲ـ۲ قواعد نحوی زبان

یک برنامه نوشته شده به زبان $Xlang$ شامل کلاسی تحت عنوان Program می باشد 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 ورودی‌ای باشد. لازم به ذکر است نقطه شروع برنامه $Xlang$ متد main خواهد بود.

۲ـ۲ـ۱ 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.

۲ـ۲ـ۲ متغیر ها و آرایه ها

در زبان $Xlang$ متغیر ها 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 سایز 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.

۲ـ۲ـ۳ 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 گوییم. 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.

۲ـ۲ـ۴ 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 3 + foo(args) 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 [۱] و رشته ها یا عباراتی از نوع آرایه به صورت اشاره‌گر [۲] به تابع مذکور پاس داده می‌شوند.

همجنین 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 نیازِ 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");
```

---

[۱] Integer

[۲] Pointer

۲-۲-۵ عملگرها

عملگر ها در این زبان به دو دسته تک عملوندی [۱] و دو عملوندی [۲] تقسیم می شوند. برای مثال عملگر ! یا همان نقیض یک عملگر تک عملوندی و عملگر && یک عملگر دو عملوندی محسوب می شود.

۲-۳ 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.

⟨ foo ⟩	به این معنا که foo غیرترمینال است.
foo	(با خط درشت) به این معنا است که foo ترمینال است.
$[x]$	به این صورت که یک توکن یا بخشی از یک توکن است. به معنای ظاهر شدن حداکثر یک x (صفر یا یک رخداد) است به نحوی که x اختیاری میباشد. توجه داشته باشید که براکت در گیومه ، '[' ']' ، ترمینال است.
x^*	به معنای ظاهر شدن صفر یا بیشتر x است.
x^+,	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

[۱] Unary operator
[۲] Binary operator

⟨program⟩ → **class Program** '{' ⟨field_decl⟩* ⟨method_decl⟩* '}'

⟨field_decl⟩ → ⟨type⟩ { ⟨id⟩ | ⟨id⟩ '[' ⟨int_literal⟩ ']' }+, ;

⟨method_decl⟩ → {⟨type⟩ | **void**} {⟨id⟩ | **main**} ([{⟨type⟩ ⟨id⟩}+,]) ⟨block⟩

⟨block⟩ → '{' ⟨var_decl⟩* ⟨statement⟩* '}'

⟨var_decl⟩ → ⟨type⟩ ⟨id⟩+, ;

⟨type⟩ → **int** | **boolean**

⟨statement⟩ → ⟨location⟩ ⟨assign_op⟩ ⟨expr⟩ ;
| ⟨method_call⟩ ;
| **if** (⟨expr⟩) ⟨block⟩ [**else** ⟨block⟩]
| **for** ⟨id⟩ = ⟨expr⟩ , ⟨expr⟩ ⟨block⟩
| **return** [⟨expr⟩] ;
| **break** ;
| **continue** ;
| ⟨block⟩

⟨assign_op⟩ → = | += | -=

⟨method_call⟩ → ⟨method_name⟩ ([⟨expr⟩+,])
| **callout** (⟨string_literal⟩ [, ⟨callout_arg⟩+,])

⟨method_name⟩ → ⟨id⟩

⟨location⟩ → ⟨id⟩
| ⟨id⟩ '[' ⟨expr⟩ ']'

⟨expr⟩ → ⟨location⟩
| ⟨method_call⟩
| ⟨literal⟩
| ⟨expr⟩ ⟨bin_op⟩ ⟨expr⟩
| - ⟨expr⟩
| ! ⟨expr⟩
| (⟨expr⟩)

⟨callout_arg⟩ → ⟨expr⟩ | ⟨string_literal⟩

⟨bin_op⟩ → ⟨arith_op⟩ | ⟨rel_op⟩ | ⟨eq_op⟩ | ⟨cond_op⟩

⟨arith_op⟩ → + | - | * | / | %

⟨rel_op⟩ → < | > | <= | >=

⟨eq_op⟩ → == | !=

⟨cond_op⟩ → && | ||

⟨literal⟩ → ⟨int_literal⟩ | ⟨char_literal⟩ | ⟨bool_literal⟩

⟨id⟩ → TOKEN_ID

⟨int_literal⟩ → ⟨decimal_literal⟩ | ⟨hex_literal⟩

⟨decimal_literal⟩ → TOKEN_DECIMALCONST

⟨hex_literal⟩ → TOKEN_HEXADECIMALCONST

⟨bool_literal⟩ → TOKEN_BOOLEANCONST

⟨char_literal⟩ → TOKEN_CHARCONST

⟨string_literal⟩ → TOKEN_STRINGCONST

۲ـ۴ 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 [۱] 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> TOKEN_LP <expr> <location> <id> TOKEN_ID
TOKEN_RP TOKEN_SEMICOLON TOKEN_RCB TOKEN_RCB
```

[۱] Pre-Order

```
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> ( <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 :

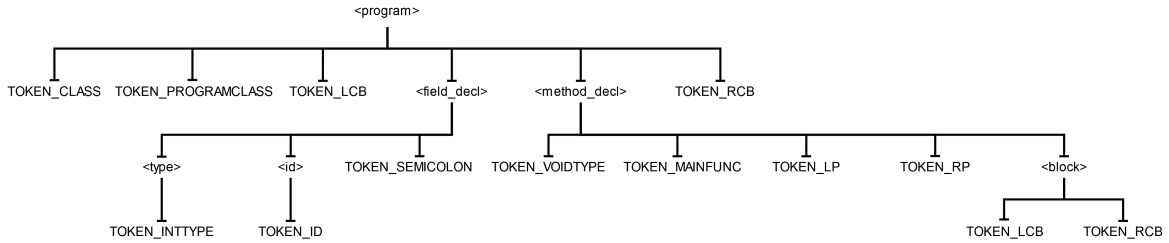
```
Input Code :

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


Analyzer Output :

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

۲ـ۵ 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 ۲ـ۱: نمایش $Syntax\ Tree$ به صورت دو بُعدی به شکل عمودی برای نمونه کد فوق

۱ـ۱ پیوست ۱

Token	*Token Name*
boolean	TOKEN_BOOLEANTYPE
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_HEXADECIMALCONST
"[some string]"	TOKEN_STRINGCONST
'a'[or other characters]	TOKEN_CHARCONST