

«به نام خدا»

## کامپایلر زبان Toorla

پروژه درس اصول طراحی کامپایلر

### مقدمه

هدف از این پروژه طراحی کامپایلر زبان Toorla می باشد. طراحی این کامپایلر به صورت فاز به فاز پیش خواهد رفت بنابراین فاز های بعدی ادامه همین قسمت خواهند بود. سند زبان Toorla در فایل ضمیمه در اختیار شما قرار گرفته است. در این فاز از شما انتظار می رود پس از مطالعه سند این زبان و آشنایی با قواعد آن، برای یک ورودی که قطعه کدی به زبان Toorla است خروجی مورد نظر که توضیحات آن در ادامه است را تولید نمایید. فاز یکم پروژه صرفا جهت آشنایی شما با قواعد زبان Toorla و ابزار ANTLR و فراگیری چگونگی خروجی گرفتن از توابع طراحی شده است و بسیار ساده می باشد.

### توضیحات

با توجه به ویدیویی که در اختیارتان قرار داده شده است به راه اندازی اولیه پروژه بپردازید. در این ویدئو چگونگی عملکرد گرامر ها و طرز کار با listener ها نیز توضیح داده شده است.

با توجه به ویدئو شما باید پس از ایمپورت کردن یک قطعه کد Toorla، با استفاده از Listener ها یک خروجی تولید نمایید. این خروجی نمایگر اجزای مختلف قطعه کد ورودی و جزئیات آن است.

شکل کلی خروجی مورد نظر به صورت زیر است. مواردی که داخل [ ] قرار ندارند نشان دهنده اجزای مختلف یک برنامه در حالت کلی می باشد (کلاس، اینترفیس، متغیر و ...) و باید عینا در خروجی نوشته شوند. موارد داخل [ ] وابسته به قطعه کد ورودی می باشد و در واقع توضیحی برای هر جزء هستند (نام کلاس ها، نام اینترفیس ها، نام متغیرها، نوع متغیر ها و ....) که باید توسط شما با توجه به قطعه کد ورودی تکمیل شوند. کد های خروجی شما تست خواهند شد بنابراین حتما مطابق فرمت داده شده خروجی را تعیین کنید، در غیر این صورت بخش زیادی از نمره را از دست خواهید داد.

توجه کنید لازم است تا یک نمونه فایل ورودی به منظور تست کد ارائه شده خود آماده کنید. این قسمت بخشی از نمره را در زمان تحویل پروژه شامل می شود.

program start {

    [program body]

}

class: [class name]/ class parent: [parent name]/ isEntry: boolean{

    [class body]

}

class constructor: [constructor name] / type: public | private{

    parameters list: [ ([[parameter type] [parameter name]], )+]])?

    [method body]

}

main method{

    parameters list: [ ([[parameter type] [parameter name]], )+]])?

    [method body]

}

class method: [method name]/ return type=[return type] / type: public | private {

    parameters list: [ ([[parameter type] [parameter name]], )+]])?

    [method body]

}

field: [field name]/ type=[type]

nested statement{

}

در ادامه یک نمونه ورودی و خروجی برای درک بهتر آورده شده است.

### Input:

```
class Operator inherits Test:
  private field result int;
  public function Operator() returns int:
    return 1;
  end

  public function subtractor(a:int , b:int) returns int:
    self.result = a - b;
    return self.result;
  end

  public function arrCollector(arr:int[]) returns int:
    int counter = 0;
    while(counter < arr.length):
      self.result = self.result + arr[counter];
    end
    return self.result;
  end

  public function comparator(a:int , b: int) returns string:
    if(a<b)
      int alaki = 3;
      if(a<0)
        print("a is negative")
      return "a is bigger than b";
    elif(a>b)
      if(b<0)
        print("b is negative");
        return "b is bigger than a";
      else
        return "a and b are equal";
      end
    end
  end
```

```

entry class MainClass:
  function main() returns int:
    int a = 5;
    int b = 6;
    arr = new int[4];
    int sum;
    int sub;
    string bigger;
    operator = new Operator();
    sub = operator.subtractor();
    sum = operator.arrCollector(arr);
    bigger = operator.comparator(a,b);
    return 1;
  end
end

```

### Output:

```

program start{
  class: Operator / class parent: Test / isEntry: false {
    field: result / type: int
    class constructor: Operator / return type: int/ type: public{
      parameter list: []
    }
    class method: subtractor / return type: int/ type: public {
      parameter list: [type: int / name: a, type: int / name: b]
    }
    class method: arrCollector / return type: int/ type: public {
      parameter list: [type: int[] / name: arr]
    }
    class method: comparator / return type: string/ type: public {
      parameter list: [type: int / name: a, type: int / name: b]
      nested {
        field: alaki / type: int
        nested{
        }
        nested{
        }
      }
    }
  }
}

```

```
class: MainClass / class parents: none / isEntry: true {  
  main method / type: int {  
    field: a / type: int  
    field: b / type: int  
    field: arr / type: int[]  
    field: sum / type: int  
    field: sub / type: int  
  }  
}  
}
```

توجه داشته باشید از شما خواسته شده است همانند مثال بالا دندان‌گذاری (Indentation) بلاک‌های کد را در خروجی برآورده سازید. به این معنی که خطوط خروجی می‌بایستند با توجه جایگاهشان در ساختار کد با فاصله مناسب از ابتدای خط چاپ شوند. هر indent level چهار عدد space می‌باشد. موفق باشید.

تیم حل پروژه: الهه متقین، محمدرضا تشکری، امیرعلی وجدانی فرد

# پروژه درس اصول طراحی کامپیوتر (فاز دوم و سوم)

دکتر امین طوسی

تدریس یاران : الهه متّین، محمدرضا تشکری، امیرعلی وجدانی فرد

مهلت تحویل : ۱۴۰۱/۱۱/۱۲

## مقدمه :

هدف این فاز : (۱) پیاده‌سازی تحلیل‌گر معنایی (۲) تشخیص خطا می‌باشد.

در مرحله اول اطلاعاتی را جمع‌آوری و در جدول علائم ذخیره می‌کنیم و در آخر جدول را نمایش می‌دهیم.

در مرحله دوم خطاها را توسط تحلیل‌گر معنایی بررسی و سپس چاپ می‌کنیم.

## توضیحات:

- جدول علائم ( Symbol Table )

ساختار داده‌ای است که برای نگهداری شناسه‌های (علائم) تعریف شده در کد ورودی استفاده می‌شود.

- طراحی جدول علائم:

برای طراحی این جدول می‌توان از روش‌های مختلفی (List, Linked List, Hash Table, ...) استفاده کرد که با توجه به نیاز، نوع زبان، پیچیدگی و نظر طراح انتخاب می‌شود.

ساده‌ترین نوع پیاده‌سازی این جدول استفاده از Hash Table می‌باشد. به این‌صورت که key آن نام شناسه و value آن مقدار (مجموعه مقادیر) ویژگی‌های مربوط به شناسه است.

هر جدول علائم دو متد اصلی دارد که اطلاعات مربوط به شناسه از طریق این دو متد در جدول ذخیره یا از جدول بازیابی می‌شوند.

```
insert (idefName, attributes)
lookup (idefName)
```

در زبان Toorla هر Scope یک جدول علائم مخصوص به خود دارد.

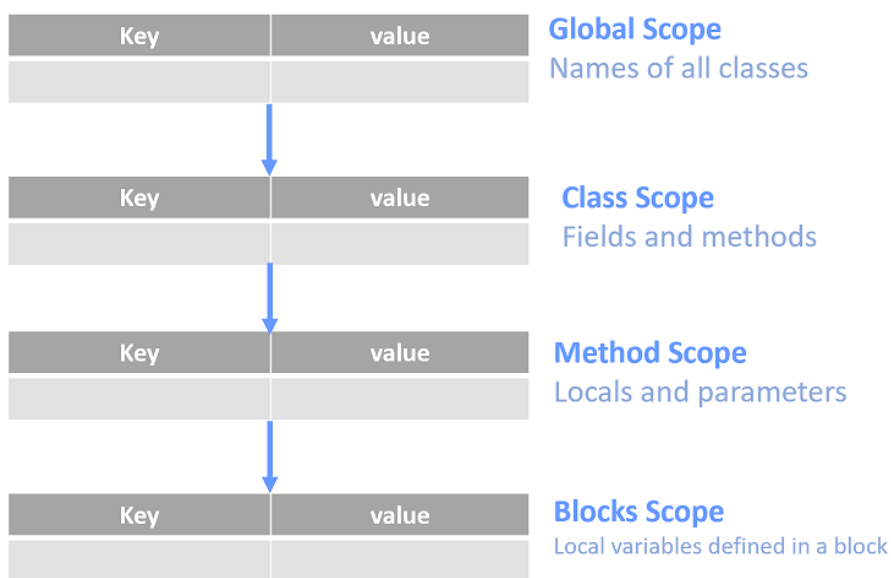
#### • Scopes:

هر یک از موارد زیر در زبان Toorla یک اسکوپ به حساب می آیند:

- تعریف برنامه
- تعریف کلاس
- تعریف constructor
- تعریف متد
- ساختار تصمیم گیری (شروع if و elif و else)
- ساختار تکرار (while و for)

#### اسکوپ ها و جداول علائم (صرفا جهت اطلاع)

همانطور که پیش تر گفته شد، هر اسکوپ شامل یک جدول علائم می باشد. بنابراین علائمی (شناسه هایی) که در هر اسکوپ تعریف می شوند در جدول علائم این اسکوپ ذخیره می شوند. از آنجایی که اسکوپ ها می توانند تو در تو باشند، جداول علائم اسکوپ ها با یکدیگر رابطه درختی دارند.



## در این دو فاز چه باید انجام دهیم؟

### فاز دوم:

در این فاز ابتدا چند برنامه به زبان Toorla بنویسید؛ سپس هر قطعه کد را به عنوان ورودی دریافت و اسکوپ‌های آن را پردازش کنید و جدول علائم مربوط به آن را بسازید و همه جداول را در یک خروجی و به ترتیب شماره خط شروع اسکوپ چاپ کنید. در ادامه مثالی از ورودی و خروجی به زبان Toorla آمده است.

Input:

```
۱ class Operator inherits Test:
۲   private field result int;
۳   public function Operator(a: int) returns int:
۴     return 1;
۵   end
۶   public function subtractor(a:int , b:int) returns int:
۷     self.result = a - b;
۸     return self.result;
۹   end
۱۰  public function arrCollector(arr:int[]) returns int:
۱۱    var counter = 0;
۱۲    while(counter < arr.length)
۱۳      self.result = self.result + arr[counter];
۱۴    return self.result;
۱۵  end
۱۶  public function comparator(a:int , b: int) returns string:
۱۷    if(a<b)
۱۸      begin
۱۹        var alaki = 3;
۲۰        return "b is bigger than a";
۲۱      end
۲۲    elif(a>b)
۲۳      begin
۲۴        if(b<0)
۲۵          print("b is negative");
۲۶          return "a is bigger than b";
۲۷        end
۲۸      else
۲۹        return "a and b are equal";
۳۰      end
۳۱ end
۳۲ entry class MainClass:
۳۳   function main() returns int:
۳۴     var a = 5;
۳۵     var b = 6;
۳۶     arr = new int[4];
۳۷     var sum=0;
۳۸     var bigger="";
۳۹     operator = new Operator();
۴۰     sub = operator.subtractor();
۴۱     sum = operator.arrCollector(arr);
۴۲     bigger = operator.comparator(a,b);
۴۳     return 1;
۴۴   end
۴۵ end
```



## Output:

```
-----program: 1 -----
Key: Class_ Operator | Value : Class (name: Operator) (parent: Test) (isEntry:
False)

Key: Class_ MainClass | Value : Class (name: MainClass) (parent: []) (isEntry:
True)

=====

----- Operator: 1 -----
Key : Field_result | Value : ClassField (name: result) (type: int, isDefiend:
True)

Key : Constructor_Operator | Value : Constructor (name : Operator) (return type:
[int]) (parameter list: [name: a, type: int, index: 1])

Key : Method_ subtractor | Value : Method (name : subtractor) (return type:
[int]) (parameter list: [name: a, type: int, index: 1], [name: b, type: int,
index: 2])

Key : Method_ arrCollector | Value : Method (name : arrCollector) (return type:
[int]) (parameter list: [name: arr, type: int[], index: 1])

Key : Method_ comparator | Value : Method (name : comparator) (return type:
[string]) (parameter list: [name: a, type: int, index: 1], [name: b, type: int,
index: 2])

=====

----- Operator: 3-----
Key : Field_a | Value : ParamField (name: a) (type: int, isDefiend: True)

=====

----- subtractor: 6-----
Key : Field_a | Value : ParamField (name: a) (type: [ classtyped= int, isDefiend:
True)

=====

----- arrCollector: 10-----
Key : Field_arr | Value : ParamField (name: arr) (type: [ classtyped= int[],
isDefiend: true)

Key : Field_counter | Value : MethodVar (name: counter) (type: [ loacalVar=
int, isDefiend: True)

=====

----- comparator: 16 -----
Key : Field_a | Value : ParamField (name: a) (type: int, isDefiend: True)
```

```

Key : Field_b | Value : ParamField (name: btest) (type: int, isDefiend: True)

----- while: 12 -----

=====

----- if: 17 -----
Key : Field_alaki | Value : MethodVar (name: counter) (type: [ loacalVar= int,
isDefiend: True)

=====

----- nested: 24 -----

=====

----- MainClass: 32 -----
Key : Method_main | Value : Method (name : main) (return type: [int]) (parameter
list: [])

=====

----- MainClass: 33 -----

Key : Field_a | Value : MethodVar (name: a) (type: [ loacalVar= int, isDefiend:
True)

Key : Field_b | Value : MethodVar (name: b) (type: [ loacalVar= int, isDefiend:
True)

Key : Field_arr | Value : MethodVar (name: arr) (type: [ loacalVar= int[],
isDefiend: True)

Key : Field_sum | Value : MethodVar (name: sum) (type: [ loacalVar= int,
isDefiend: True)

Key : Field_bigger | Value : MethodVar (name: bigger) (type: [ loacalVar=
string, isDefiend: True)

Key : Field_Operator | Value : MethodVar (name: Operator) (type: [ loacalVar=
Classtype:[Operator], isDefiend: True)

```

## مراحل گرفتن خروجی :

۱. برای هر SymbolTable باید دو تابع زیر فراخوانی شوند. تابع toString برای چاپ کردن مقادیر symbolTable و تابع getValue برای دریافت مقادیر از Hashmap استفاده می‌شوند.

```
public String toString() {  
    return "----- " + name + " : " + scopeNumber + " ----- \n" +  
        printItems() +  
        "----- \n";  
}
```

```
public String printItems(){  
    String itemsStr = "";  
    for (Map.Entry<String,SymbolTableItem> entry : items.entrySet()) {  
        itemsStr += "Key = " + entry.getKey() + " | Value = " + entry.getValue()  
+ "\n";  
    }  
    return itemsStr;  
}
```

۲. برای چاپ هر item نیز باید متد toString بنویسیم.

فرمت مثال زده شده صرفاً یک نمونه فرمت قابل قبول برای خروجی زبان Toorla می باشد و دیگر فرمت‌های خوانا، مرتب و نمایش‌دهنده تمام اجزای هر بخش قابل قبول می‌باشند. (اگر فرمت شما خلاقانه، مرتب و بسیار کامل باشد و به طور کاملاً واضح و زیبا نمایانگر تمام اجزا جدول علائم اسکوپ باشد می‌تواند شامل نمره اضافه شود.)

## نکات:

- شماره خط شروع هر اسکوپ را در ابتدا به همراه نام آن نمایش دهید:

```
----- Base : 18 -----
```

- در صورت خالی بودن یک جدول باز هم نیاز به نمایش دادن آن می‌باشد:

```
----- nested : 39 -----
```

- در هنگام ذخیره سازی هر یک از اجزا در Symbol table نیاز است نوع آن را در کنار نام آن ذخیره کنید  
به عنوان مثال در قطعه کد زیر نیاز است کلاس Base را به صورت class\_Base و متد set را به صورت  
method\_set در قسمت key ذخیره کنید.

```
class Base{  
    private int set() {  
    }  
}
```

**\*\* توجه کنید تشخیص نوع متغیرهای محلی از نوع Classtype دارای نمره اضافه است اما تشخیص نوع سایر  
انواع از پیش تعیین شده از متغیرهای محلی مانند int، string، و ... الزامی است.**

## فاز سوم:

در این فاز می‌خواهیم با استفاده از جدول علائم به بررسی خطاهای معنایی موجود در برنامه بپردازیم.

### فرمت گزارش خطا:

خطاهای موجود در برنامه را بر اساس فرمت زیر گزارش دهید:

line شماره خط ارور و column مکان آن را در یک خط نشان می‌دهد.

شما باید دو نوع خطایی که در ادامه آورده شده است پیاده‌سازی کنید.

۱. خطای تعریف دوباره متد/خصیصه/کلاس/متغیر محلی

- تعریف دوباره کلاس:

```
Error\۰\۰: in line [line:column] , method [name] has been defined already
```

- تعریف دوباره متد:

```
Error\۰\۲: in line [line:column] , method [name] has been defined already
```

- تعریف دوباره خصیصه:

```
Error\۰\۳: in line [line:column], field [name] has been defined already
```

- تعریف دوباره یک متغیر محلی در یک حوزه:

```
Error\۰\۴: in line [line:column], var [name] has been defined already
```

▪ نکته: دو نوع متفاوت میتوانند هم نام باشند به عنوان مثال اگر یک فیلد و متد هم اسم باشند مشکلی نیست.

▪ نکته: در صورت تعریف دوباره یک کلاس، متد و یا فیلد اسم آن را عوض میکنیم و به سیمبل تبیل اضافه میکنیم و اسم آن را به این صورت ذخیره میکنیم: `name_line_column`.

بعنوان مثال اگر متغیر `d` دوباره تعریف شود آن را به صورت `d_۳۴_۴۸` ذخیره می‌نماییم.

- نکته: هر کدام از موارد ذکر شده اگر دوبار تعریف شوند مورد دوم مطرح نیست و فرض میکنیم اصلا وجود ندارد و تنها از مورد اول استفاده میشود. به عنوان مثال اگر یک کلاس دوبار تعریف شده باشد تنها میتوان از کلاس اول استفاده کرد.

۲. خطاهای مربوط به ارث بری

- وجود دور در ارث بری :

```
Error410 : Invalid inheritance [classname1] -> [classname2] -> [classname3] ...
```

۳. نمره اضافه

- عدم تطابق نوع بازگشتی متد با نوع بازگشتی تعریف شده توسط متد:

```
Error210 : in line [line:column], ReturnType of this method must be [MethodReturnType]
```

- متدهای پرایوت یک کلاس در کلاس های دیگر قابل دسترسی نمی باشد.

```
Error310: in line [line:column], private methods are not accessible outside of class.
```

موفق باشید