در این مستند با زبان ساده TesLang آشنا می‌شویم. در گام‌های تمرین عملی درس طراحی کامپایلر، بخش‌هایی از یک مترجم برای این زبان نوشته می‌شوند. قواعد این زبان در ادامه‌ی این مستند بیان می‌شود.

- این زبان دارای چهار نوع داده‌ای است:
  - int برای اعداد صحیح
  - vector برای آرایه‌ها
  - str برای رشته‌ها
  - boolean برای مقادیر دوگانه
- برنامه‌های این زبان در یک فایل نوشته می‌شوند و هر فایل شامل تعدادی تابع است.
- در این زبان متغیر سراسری (Global) وجود ندارد.
- خط اول هر تابع شامل تعریف نام (identifier) آن تابع و ورودی و نوع خروجی آن است.
- بدنه هر تابع بین دو توکن { و } قرار گرفته و شامل تعدادی عبارت (statement) است.
- در حلقه for بعد از keyword مربوطه‌ی آن به یک شناسه که برای مشخص کردن نقطه شروع به آن است نیاز داریم.
- کامنت‌ها در کد با توکن %> تعریف می‌شوند و از مکانی که این توکن استفاده شود تا پایان که دوباره توکن <% استفاده شود را نادیده می‌گیریم.
- متغیرهای محلی در هر block از کد به شکل زیر تعریف می‌شوند (هر دو شکل تعریف که با مقادیر اولیه و بدون مقادیر اولیه هستند را در نظر داشته باشید):

```
a :: int = 10;                <% this is a comment %>
b :: vector = [1, 2, 3, 4];
c :: str = "Hello world!";
d :: int;
e :: vector;
f :: str;
```

- مقدار خروجی تابع با استفاده از کلمه‌ی کلیدی return مشخص می‌شود:

```
fn sum(a as int, b as int) <int>
{
    result :: int = 0;
    result = a + b;
    return result;
}
```

- در صورتی که تابع چیزی برنگرداند، نوع داده‌ای باید از نوع void باشد.
- در مثال زیر، یک لیست را به عنوان ورودی به تابع می‌دهیم و عناصر آن را با هم جمع می‌کنیم:

```
fn sum(numList as vector) <int>
{
    result :: int = 0;

    for (i = 0 to length(numList))
    begin
        result = result + numList[i];
    end

    return result;
}
```

- مثال زیر یک نوع استفاده از if را در این زبان نشان می‌دهد.

```
fn fact(n as int) <int>
{
    if [[ n < 2 ]]
    begin
        return 1;
    end

    return n * fact(n - 1)
}
```

- برای مثال در زبان TesLang یک تابع sum بدون بدنه به شکل زیر نوشته شده است:

```
fn sum(a as int, b as int) <int> return a+b;
```

- جداول توابع داخلی:

تابع	توضیح
scan()	یک عدد را از ورودی استاندارد می‌خواند و برمی‌گرداند.
print()	یک عدد را در ورودی استاندارد چاپ می‌کند.
list(n)	یک لیست با n عنصر برمی‌گرداند.
length(arr)	یک لیست (arr) بعنوان ورودی گرفته و طول لیست را برمی‌گرداند.
exit(n)	برنامه را با کد برگشتی داده شد (n) به پایان می‌رساند.

قواعد تجزیه‌ی زبان TesLang

در ادامه، ساختار BNF زبان TesLang نمایش داده شده است. اولویت عملگرها در زبان TesLang مشابه اکثر زبان‌های برنامه‌نویسی JavaScript, C, Python, ... است. چون در گرامری که در ادامه نمایش داده می‌شود اولویت عملگرها مشخص نشده است، گرامر مبهم است.

گرامر زبان

```
prog :=                 |
        func prog


body :=                 |
        stmt body

func :=     FN iden (flist) <type> { body } |
            FN iden (flist) <type> => return expr ;

stmt :=     expr;                                           |
            defvar;                                         |
            func;                                           |
            IF [[ expr ]] stmt                              |
            IF [[ expr ]] stmt ELSE stmt                    |
            WHILE [[ expr ]] stmt                           |
            DO stmt WHILE [[ expr ]]                        |
            for (iden = expr TO expr) stmt                  |
            BEGIN body END                                  |
            RETURN expr ;


defvar :=   iden :: type        |
            iden :: type = expr


flist :=                |
        iden AS type    |
        iden AS type, flist

clist :=                |
        expr            |
        expr , clist

type :=     int         |
            vector      |
            str         |
            bool        |
            null
```

```
expr :=     expr [ expr ]       |
            [ clist ]           |
            expr ? expr : expr  |
            expr + expr         |
            expr - expr         |
            expr * expr         |
            expr / expr         |
            expr > expr         |
            expr < expr         |
            expr == expr        |
            expr >= expr        |
            expr <= expr        |
            expr != expr        |
            expr || expr        |
            expr && expr        |
            ! expr              |
            + expr              |
            - expr              |
            iden                |
            iden = expr         |
            iden ( clist )      |
            number


iden    := [a-zA-Z_][a-zA-Z_0-9]*
number  := [0-9]+
comment := Implement commenting system with <%%>. Provided regex must support
           nested commenting as well.
string :=  Must start with ‘ or “ and end with one of those
           respectively. Must support escaping as well.
```

درBNF داده شده به نکات زیر توجه کنید:

- کلمات کلیدی زبان در فرم UPPERCASE نوشته شده‌اند.
- علامت‌های | و =: برای تعریف قواعد گرامرها استفاده می‌شوند و از ورودی‌های زبان نیستند.

عبارات منظم (Regex) مربوط به مرحله نحوی:

```
iden    := [a-zA-Z_][a-zA-Z_0-9]*
number  := [0-9]+
comment := Implement commenting system with <%%>. Provided regex must support
           nested commenting as well.
string :=  Must start with ‘ or ” and end with one of those
           respectively. Must support escaping as well.
```

iden: عبارت منظم مربوط به یک identifier را نشان می‌دهد که اولین حرف می‌تواند حروف بزرگ یا کوچک یا _ باشد و در ادامه می‌تواند شامل اعداد نیز باشد.

number: از یک یا بیشتر رقم تشکیل می‌شود.

string: رشته‌ای از کاراکترها به صورت پشت هم که می‌تواند با ‘ یا ” شروع شود و می‌تواند شامل خود ‘ یا ” در میان رشته شود؛ ولی باید به صورت escape شده باشد.

comment: شامل عبارات میان <%%> است (که در هنگام tokenize کردن نباید در نظر گرفته شود).

- ## قاعده آغازین برنامه

```
prog     :=  empty    |
             func prog
```

در صورت وجود تابع وضعیت، func تجزیه خواهد شد و سپس مجدد prog اجرا می‌شود. در غیر آن صورت به وضعیت پایانی می‌رسیم.

- ## تعریف تابع

```
func :=  FN iden (flist) <type> { body }   |
         FN iden (flist) <type> => return expr
```

قاعده اول یک تابع دارای بدنه است و قاعده دوم، یک تابع بدون بدنه است که صرفاً یک مقدار را برمی‌گرداند. تابع با کلمه کلیدی FN شروع خواهد شد و بعد از تعریف نام تابع، بین دو پرانتز flist قرار می‌گیرد و در آخر نوع داده‌ای آن خواهد آمد. در ادامه به بررسی بدنه تابع می‌پردازیم، به قانون زیر رجوع می‌کنیم، در صورت عدم برقراری آن به قاعده دوم (تابع بدون بدنه) رجوع می‌کنیم.

- ## بدنه زبان

```
body :=              |
           stmt body
```

در صورت وجود یک دستور وضعیت stmt تجزیه خواهد شد و سپس مجدد body اجرا می‌شود. در غیر این صورت به وضعیت پایانی می‌رسیم.

- ## عبارات (statements) زبان

```
stmt :=     expr;                                   | #1
            defvar;                                 | #2
            func;                                   | #3
            IF [[ expr ]] stmt                      | #4
            IF [[ expr ]] stmt ELSE stmt            | #5
            WHILE [[ expr ]] stmt                   | #6
            DO stmt WHILE [[ expr ]]                | #7
            for (iden = expr TO expr) stmt          | #8
            BEGIN body END                          | #9
            RETURN expr ;                             #10
```

بعد از توکن (بدنه اجرا خواهد شد که می‌تواند تک خطی هم باشد (مانند مثالی که در مستند قبلی به آن اشاره شد).

۱. به اتمام یک عملیات در وضعیت expr اشاره می‌کند.
۲. تعریف متغیر را نشان می‌دهد.
۳. این دستور باعث تشکیل توابع تو در تو (nested functions) در این زبان می‌شود.
۴. دستور شرطی if را به صورت تنها نشان می‌دهد.
۵. دستور شرطی if...else را نشان می‌دهد.
۶. دستور مربوط به حلقه مرسوم while است که تا زمان برقراری expr موردنظر، stmt را اجرا می‌کند.
۷. دستور مربوط به حلقه do...while است که تا زمان برقراری expr مورد نظر، stmt را اجرا می‌کند.
۸. دستور مربوط به حلقه for را نشان می‌دهد که به صورت زیر یک مقدار اولیه به آن داده می‌شود (به عنوان اندیس شروع):

```
for (i = 0 to length(numList))
begin
        result = result + numList[i];
end
```

۹. این دستور مربوط به یک بدنه می‌باشد که مجددا درون آن وضعیت body اجرا خواهد شد و بلوک‌های کد تو در تو را تشکیل می‌دهد (nested blocks).
۱۰. این دستور مربوط به خروجی تابع است که با کلمه کلیدی return مشخص می‌شود و سپس یک مقدار به صورت expr در جلوی آن می‌آید.

برای درک بهتر این مورد، به مثال زیر توجه کنید:

```
fn testFunc1(a as int) <int>
{
    fn testFunc2() <int>
    {
        return a + 10;
    }
    return testFunc2();
}

print(testFunc1(5));
```

- ## تعریف متغیر (defining variables) در زبان

```
defvar :=   iden :: type          | #1
            iden :: type = expr     #2
```

۱. این قانون به تعریف متغیر بدون انتساب مقدار اولیه اشاره می‌کند.

۲. این قانون به تعریف متغیر به همراه انتساب مقدار اولیه اشاره می‌کند.

نکته: توجه کنید که ابتدا نام متغیر قرار دارد، سپس :: و بعد نوع داده‌ای متغیر و در قانون دوم هم برای انتساب مقدار اولیه کافیست یک = قرار بگیرد و بعد از آن به سراغ تجزیه expr باید رفت.

- ## لیست آرگومان‌های توابع (function’s argument list)

```
flist :=                  |
          iden AS type    |
          iden AS type, flist
```

۱. یک تابع می‌تواند صفر یا بیشتر آرگومان داشته باشد. این قانون نشان می‌دهد که هنگام تجزیه، آرگومان بعدی می‌تواند خالی باشد.

۲. این قانون نشان‌دهنده یک آرگومان است، به این نحو که ابتدا نام متغیر آمده است سپس کلمه کلیدی as و بعد از آن نوع متغیر و تجزیه پس از آن پایان می‌یابد.

۳. این قانون نیز مانند قانون دوم اجرا می‌شود، اما با این تفاوت که پس از مشاهده توکن ,_ مجددا به تجزیه flist می‌پردازیم.

- ## لیست فراخوانی توابع و ورودی‌ها (call list)

```
clist :=                |
          expr          |
          expr , clist
```

۱. یک تابع می‌تواند صفر یا بیشتر ورودی داشته باشد. این قانون نشان می‌دهد که می‌توان تجزیه را بدون وجود expr ادامه داد.
۲. این قانون نشان‌دهنده یک ورودی است که پس از آن تجزیه clist خاتمه می‌یابد.
۳. این قانون مانند قانون دوم است، اما با این تفاوت که پس از مشاهده توکن , مجددا به تجزیه clist می‌پردازیم.

- ## انواع داده‌ای (data types) در زبان

```
type :=     int         |
            vector      |
            str         |
            bool        |
            null
```

۱. نشان‌دهنده نوع داده‌ای از نوع اعداد صحیح است.
۲. نشان‌دهنده نوع داده‌ای آرایه است.
۳. نشان‌دهنده نوع داده‌ای رشته است.
۴. نشان‌دهنده نوع داده‌ای بولین است.
۵. نشان‌دهنده نوع داده‌ای تهی است.

عبارات و اصطلاحات (expressions)

```
expr :=     expr [ expr ]       | #1
            [ clist ]           | #2
            expr ? expr : expr  | #3
            expr + expr         |
            expr - expr         |
            expr * expr         |
            expr / expr         | #7
            expr > expr         |
            expr < expr         |
            expr == expr        |
            expr >= expr        |
            expr <= expr        |
            expr != expr        | #13
            expr || expr        | #14
            expr && expr        | #15
            ! expr              |
            + expr              |
            - expr              | #18
            iden                | #19
            iden = expr         | #20
            iden ( clist )      | #21
            number              |
            string
```

۱. مربوط به خواندن یک خانه از آرایه است. expr سمت چپ باید از نوع vector و expr سمت راست باید از نوع number باشد.

۲. روشی از ساخت یک آرایه می‌باشد.

۳. نشان‌دهنده یک ternary operator است که ابتدا صحیح بودن expr سمت چپی را بررسی کرده و در صورت درست بودن آن expr وسط را اجرا می‌کند و در غیر این صورت expr سمت راست را اجرا می‌کند.

۴. تا قانون شماره ۷ نشان‌دهنده عملیات محاسباتی است (arithmetic aoperations).

a. نکته: برای پیاده‌سازی این بخش، باید رفع ابهام صورت گیرد تا از سمت چپ محاسبات دارای اولویت باشند. به ترتیب اولویت عملگرهای * / % از + – بیشتر است.

۵. تا قانون شماره ۱۳ نشان‌دهنده عملیات مقایسه‌ای در زبان است.

۶. ۱۴ و ۱۵ نشان‌دهنده عملیات short circuit هستند.

۷. تا قانون شماره ۱۸ برای عبارات یگانه (unary) است که می‌تواند قبل از آن توکن‌های – + ! استفاده کرد.

۸. ۱۹ یک شناسه (identifier) را برمی‌گرداند.

۹. ۲۰ یک انتساب را نشان می‌دهد.

۱۰. ۲۱ یک فراخوانی تابع (function call) را نشان می‌دهد.

۱۱. یک عدد صحیح را برمی‌گرداند.

۱۲. یک رشته از حروف را برمی‌گرداند.