تحلیل مفهومی

تحلیل مفهومی در حد Type Checking انجام می گیرد. زبانهایی که تحلیل مفهومی را انجام مـی دهنـد، اصطلاحا Strongly Typed Languages نامیده می شوند. در این زبانها معمولا هر متغیر از نـوع خـاص تعریف می شود. زبانهایی وجود دارد که در آنها متغیرها معرفی نمی شـوند، امـا عمـل Type Checking انجام می شود، در این دسته از زبانها نوع هر متغیر وابسته به نوع تخصیص داده شـده بـا آن مـی باشـد. بـرای مثال، اگر در داخل متغیر A مقداری از integer نوع باشد، نوع متغیر A نیز integer می باشد، ولـی اگر در زمان اجرا یک نوع Real به آن اختصاص بدهیم نوع متغیر A به Real تغییر می کند، لذا به این دسـته از زبانها Dynamic Typed Languages می گویند.

در زبانها عادی عمل آزمون نو در زمان کامپایل برای این منظور با استفاده از یک کلاس بـه نـام Symbol Tables هر گاه که متغیری در داخل جدول نمادها گنجانده می شود و در هنگامی که در داخل عبـارت یـا جملات یک متغیر استفاده می شود، نوع آن از داخل جدول نمادها مورد بررسی قرار می گیرد.

: Sematic Analyses

همانگونه که در بالا گفته شد، تحلیل مفهومی در حد آزمون نو صـورت مـی گیـرد، بـرای ایـن منظـور مـی بایست جدول نمادها را ایجاد کنیم.

در ادامه به ذکر چگونگی ایجاد آن و پر کردن ان در ضمن عمل تحلیل نحوی می پردازیم.

```
Const C1 = 5;

Type
  Student = record
    a : array [1..10] of integer;
    b : real;
  End;
  F = record
    i : integer;
    j :array [1..10] of char;
  End;

Var
  a : integer;
  d : real;
  c : char;
  T : Student;
```

در ضمن عمل کامپایل با استفاده از کلاس Symbol Table در هنگام تحلیل نحوی با مشاهده تعریف (Decleration) اسامی تعریف شده در داخل جدول نمادها گنجانده می شوند. اما ابتدا ابزار کار یعنی کلاس Symbol Table باید مشخص شود.

۱- تعیین ساختار کلاس (Data Structure) :

با توجه به مثال ارایه شده مشخص می کنیم که چه فیلدهایی برای تعیین دقیق ساختار جدول نمادها ضروری است.

Name	Token	Group	Offset/Value	Type
C1	-	S_Const	[5]	int
Student	-	S_Type	-	.
a	-	S_Var	0	integer
b	-	S_Var	2	real
C1	-	S_Var	6	char
f	-	S_Var	8	.
T	-	S_Var	00??	.

Record Descriptor		
Name	Offset	Type
a	0	.
b	20	real

Array Descriptor	
Element _Type	: int
Lower_Bound	: 1
Upper_Bound	: 10
No_Dim	: 1

(فرض می کنیم طول هر کلمه ۲ بایت است و فضای ذخیره سازی ۲ کیلو بایت)

```
class Buket {
        string Key; object Binding; Buket Next;
```

```
        Bucket(string  k,  object    b,  Bucket  n) { key = k; binding := b; next := n;}
        }


class HashT{
            final int size = 256;
            Bucket tabel[] = new Bucket[size];
             Int hash(string S_ )
             {
               int h = 0;
               for (int I = 0; I < s.length; I++)
               h := h * 65599 + s.charAt( I );  //S_ کاراکتر I ام در
               return h;
             }
             void insert(string S_, binding b)
             {
               int index = hash(S_) % size;
               table[index] = new Bucket(S_, b, table[index]);
             }
             object LookUp(string S_)
             {
               int index = hash(S_) % size;
               for (binding b = table[index]; b != null; b := b.next)
               if (s.equal = S_(b.key)) return b.binding;
               return null;
             }
             void pop(string S_)
             {
               int index = hash(S_) % size;
               table[index] := table[index].next;
             }
           }// end of class HashT
```

در مثال فوق مشاهده می کنید کـه جـدول نمادهـا بصـورت Hash Table و در قـالب کلاسـی بـه نـام HashT ارایه شده، و هر ردیف آن در قالب کلاسی به نام Bucket معین شده است.

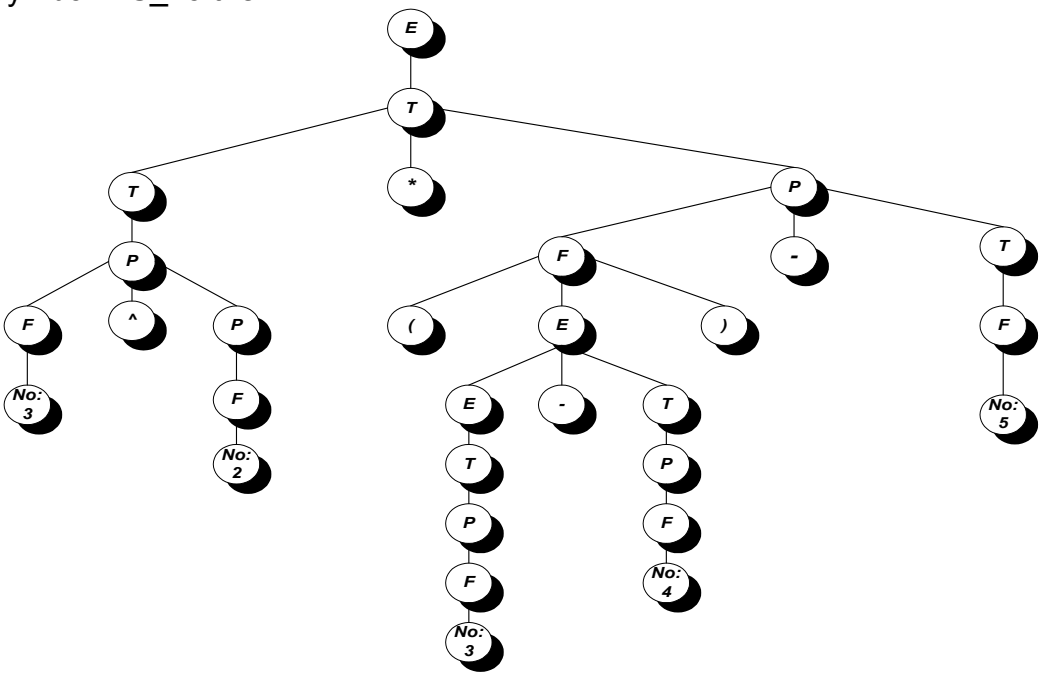
Size بعنوان یک Const مطرح شده است، توابع Insert و LookUp بیشتر از همه مـورد نیـار تحلیلـگر مفهومی می باشند، تحلیلگر مفهومی با استفاده از تابع LookUp به دنبال نام متغیر می گردد و نوع ان را پیـدا می کند، تابع Insert کار افزودن نانم جدید را به داخل جدول نمادها را بر عهده دارد.

نکته در اینجاست که چگونه کامپایلر با بهره برداری از توابع Insert و LookUp جدول نمادها را ایجاد می کند و برای تحلیلگر مفهومی مورد استفاده قرار می دهد.

```
Program›   Program id ';' BlockBody'.'
BlockBody›  [ConstDefpart] [TypeDefpart]
            [VarDefpart] { ProDefpart | FunDefpart}
            compound Stop
```

```
(* ConsDefpart ›   const ConstDef {ConstDef} *)
Procedure ConstDefpart(Stop : Stops);
Begin
//مانند قبلی
End;

(* ConstDef ›   id '=' ( id | no )';' *)
Procedure ConstDef(Stop : Stops);
Begin
  c := CurrentToken;
  SymTable.Insert(CurrentToken);
  Expect(S_id, Stop + [S_equal]);
  Expect(S_equal, Stop + [S_id, S_no]);
  If CurrentSymbol = S_no then
```




```
    Begin
      SymTable.Update(c, CurrentToken.Lexeme, S_var);
      NextSymbol;
    End
  Else
    Begin
      d := LookUp(CurrentToken);
      SymTable.Update(c, d.Value, S_var);
      Expect(S_id, Stop + [S_semicolon]);
    End;
  Expect(S_semicolon, Stop);
End;

(* VarDefpart ›   Var VarDef {VarDef} *)
```

```
Procedure VarDefpart(Stop  : Stops);
Begin
//مانند قبلی
End;

(* VarDef ›    id {, id} ':' (integer | real) ';' *)
Procedure VarDef(Stop : Stops);
Var
  s : stack;
Begin
  s.push(CurrentToken);
  Expect(S_id, …);
  While(CurrentSymbol = S_comma) do
    Begin
      NextSymbol;
      s.push(CurrentToken);
      Expect(S_id, …);
    End;
  Expect(S_colon, Stop + …);
  While not (s.isEmpty) do
    SymTable.Insert(s.pop, CurrentToken.Lexeme);
  If CurrentSymbol = S_real then
    NextSymbol
  Else
    Expect(S_int, Stop);
End;

(* F›   id | no |'('E')' *)
Procedure F(Stop : Stops; Var ResF : string; R : integer; TypeF : Symbols);
Begin
  If CurrentSymbol = S_id then
    Begin
      TypeF := SymTable.LookUp(CurrentToken);
      R := 0;
      ResF := CurrentToken.Lexeme;
    End
  Else If CurrentSymbol = S_no then
    Begin
      TypeF := S_const_int;
      R := 0;
      ResF := CurrentToken.Lexeme;
    End
  Else
    Begin
      Expect(S_OpenPar, …);
      E(Stop + [S_ClosePar], ResF, R, TypeF);
      Expect(S_ClosePar, …);
    End;
End;
```

```
(* T›   F { (* | / ) F} *)
Procedure T(Stop : Stops; Var ResT : string; R : integer; TypeT : Symbols);
Var
  OpCode, Type1 : Symbols;
  Operand1 : string;
  R1 : integer;
Begin
  F(Stop + [S_mul, S_div], ResT, R, TypeT);
  While (CurrentSymbol = S_mul) or (CurrentSymbol = S_div) do
  Begin
    OpCode := CurrentSymbol;
    NextSymbol;
    If (R =0) then
    Begin
      R := 1;
      Operand := NewTemp;
      EmitIn('mov ' + Operand + ',' + ResT);
      ResT := Operand;
    End;
    F(Stop + [S_mul, S_div], Operand, R1, Type1);
    If OpCode = S_ mul then
      Emit('mul ')
    Else
      Emit('div ');
    EmitIn(ResT + ',' + Operand);
    If R = 1 then
      RemTemp;
    Type1 := Compare(TypeT, Type1);
  End;
End;
```