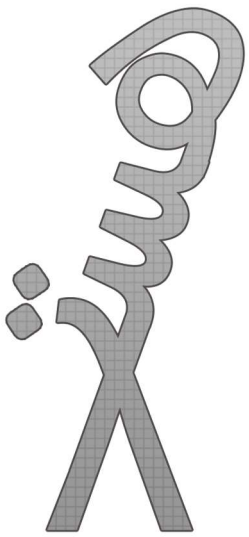
به نام خدا



زبان برنامه نویسی هستی

نویسنده مستندات، طراح و پیاده ساز زبان: آرین ابراهیم پور

استاد راهنما: دکتر ابوالقاسم میرروشندل

گروه:

- زبان های برنامه نویسی
- زبان های تابعی
- پروژه نهایی کارشناسی رشته کامپیوتر دانشگاه گیلان

مقدمه

زبان های برنامه نویسی سازه های اولیه را برای ساخت یک نرم افزار تهیه می‌کنند و همچنین از مهم‌ترین ابزارها برای ساختاردهی، اعتبار سنجی و قابلیت انعطاف و توسعه یک برنامه هستند.

در این پروژه سعی شده با دید به «برنامه نویسی تابعی» از گروه زبان های توصیفی، زبان برنامه نویسی فارسی ای توسعه داده شود که مناسب اهداف آموزشی است و با تمرکز به آموزش به برنامه نویسان با سن پایین‌تر که هنوز زبان انگلیسی را به خوبی نمی‌دانند، یاور آنان در این مسیر باشد.

جدول اطلاعات زبان

Design		
By	Aryan Ebrahimpour	(BCS Student)
Start Date	1 Ordibehesht 1398	
Paradigm	Multi-Paradigm	Declarative (With imperative support)
Group	Functional Programming Languages	ML Family
Blocks	With Indentation	
Flow Direction	Right to Left	

Implementation		
By	Aryan Ebrahimpour	(BCS Student)
Start Date	Experimental Impl at 1 Khordad 1398	
Language	F# on .NET Core, .NET Standard	Cross-Platform
Runtime-Type	Interpreted	Todo: Compile to MSIL
Compiler Parser	Parser Combinators	FParsec for F#

IDE/Tools		
By	Aryan Ebrahimpour	(BCS Student)
Name	Hasti IDE	
Features	Syntax Highlighting, AST Viewer, Code-Runner	
Language	C# and F# on .NET Core	
GUI Technology	WPF on .NET Core	

چرا زبان برنامه‌نویسی فارسی؟

در دنیا زبان های برنامه نویسی بسیاری وجود دارند که به اهداف متفاوتی ساخته می‌شوند. در این میان، تقریبا تمامی زبان های برنامه نویسی که به اهداف تجاری و در سطح وسیع استفاده می‌شوند به زبان انگلیسی هستند.

طبیعتا هدف این زبان برنامه نویسی فارسی جایگزینی آن با زبان های تجاری جهانی نیست، بلکه صرفا یک زبان آموزشی‌ست که اولا، آموزش برنامه نویسی را به توسعه‌دهندگان جوانی که هنوز به خوبی زبان انگلیسی را نیاموخته اند آسان می‌کند، و در ثانی، از آن‌جایی که اکثر زبان های تجاری زبان هایی از پارادایم دستوری و خانواده شی‌گرایی هستند، می‌کوشد با آموزش برنامه نویسی تابعی و توصیفی در کنار شی‌گرایی، دید جدیدی در رابطه با نحوه برنامه‌نویسی به یادگیرنده بدهد.

زبان های برنامه نویسی تابعی

همان طور که پیشتر گفته شد، گروه زبان های مختلفی در جهان وجود دارند که زبان های تابعی و زبان های شی‌گرایی دو گروه مهم از آن‌ها هستند. زبان های شی‌گرا از گروه زبان های دستوری هستند (اگرچه ممکن است برخی از آن‌ها نیز به نحوه دیگری طراحی و پیاده سازی شده باشند، اما ملاک ما پارادایم صنعتی شی گرایی است)، به این معنا که در هر مرحله برنامه نویس به زبان می‌گوید که چه کاری را انجام دهد.

تابع Sum در کتاب خانه دات‌نت به زبان سی شارپ:

```
public static int Sum(this IEnumerable<int> source) {
    if (source == null) throw Error.ArgumentNull("source");
    int sum = 0;
    checked {
        foreach (int v in source) sum += v;
    }
    return sum;
}
```

همین برنامه به صورت تابعی در زبان اف‌شارپ:

```
let rec sum list =
    match list with
    | [] -> 0
    | x :: xs -> x + sum xs
```

اگرچه در زبان سی‌شارپ به مرور زمان قابلیت های تابعی هم پیاده سازی شده است و جمع توابع با استفاده از تابع Sum از دید کاربر نهایی وجهه ای تابعی دارد، اما کلیت زبان و ساختار های آن به شکل شی‌گرا، وابسته به جهش و دستوری است. در حالی که در زبان تابعی‌ای چون اف‌شارپ پارادایم کلی زبان تابعی، توصیفی و بدون جهش است، اگرچه از ویژگی های دستوری نیز پشتیبانی می‌کند.

در زبان های تابعی از آن‌جایی که بر اساس ریاضیات پیش می‌رود، به جای متغیرها، انقیاد وجود دارند که قابل تغییر و یا اصطلاحا قابل جهش نیستند (به گروهی از زبان های تابعی که به هیچ‌وجه امکان تغییر مقدار یک متغیر وجود ندارد اصلاحا زبان های Pure Functional می‌گویند، مثل Haskell)، همچنین برخلاف زبان های دستوری که نتیجه تفکر ماشین های تورینگ هستند، منطق کلی زبان های تابعی محسابات لامبدا که توسط Alonzo Church معرفی شد. در این نحوه نوشتار، توابعی تک ورودی تعریف می‌شوند که خود می‌توانند توابع دیگری را به عنوان خروجی تولید کنند. به عنوان مثال نحوه پیاده سازی مقادیر بولی، و عملگر های AND و OR نشان داده شده است:

```
TRUE  = λx.λy.x
False = λx.λy.y

AND   = λa.λb.a b a
OR    = λa.λb.a a b
```

طراحی زبان برنامه نویسی هستی

زبان برنامه نویسی هستی نیز با توجه به همین رویکرد، و با استفاده تمرکز بر زبان های برنامه نویسی تابعی خانواده ML که شامل زبان هایی چون F#, OCaml و... هستند طراحی شد، چرا که این خانواده از زبان ها دارای خوانایی بهتر بوده، و همچنین سازگاری بسیار خوبی با زبان فارسی دارند.

همچنین سعی شده که در این زبان تمامی ساختار ها به فرم (نام : نوع = بدنه) و به کمک فاصله گذاری برای بلوک بندی ها استفاده شود.

به عنوان مثال از نمونه کد های این زبان (توجه کنید که در نسخه فعلی کامپایلر ممکن است همه این موارد پیاده سازی نشده باشند):

۱- تعریف متغیر

```
نام : متن = "آرین"
سن : عدد = 21
آیاپاسخدرستاست : بولی = درست
```

۲- توابع

در این فرمت، اولین شناساگر نام تابع، و باقی شناساگرها پارامتر های تابع به فرمت محاسبات لامبدا و اعمال جزئی هستند.

```
محاسبه‌گر سن سال_تولد : عدد -> عدد -> عدد = سن + سال_تولد
نویسنده نام سن آیا_کچل : رشته -> عدد -> بولی -> واحد =
    بنویس نام
    بنویس سن
    اگه آیا_کچل اونوقت بنویس "کچله!"
```

۳- عملگر های شرطی

```
اگه 10 == (5 + 5) اونوقت بنویس "بله" وگرنه بنویس "خیر"
متغیر : عدد = اگه درست اونوقت 3 وگرنه 5

اگه نام == "آر" + "ین" اونوقت
    بنویس نام
    بنویس "درسته"
وگرنه
    بنویس نام
    بنویس "غلطه"
```

٤- حلقه ها

```
برای الف = ۰ تا ۱۰ اونوقت
    اگه ۰ == (الف ٪ ۲) اونوقت
        بنویس "زوج"
    وگرنه
        بنویس "فرد"
```

۵- تعریف انواع

```
شکل : نوع =
    | گشنیز
    | پیک
    | خشت
    | دل

مقدار : نوع =
    | شاه
    | ملکه
    | سرباز
    | عددی از عدد

کارت : نوع = شکل * مقدار
انسان : نوع =
    { نام : رشته
    نام‌خانوادگی : رشته
    سن : عدد }
```

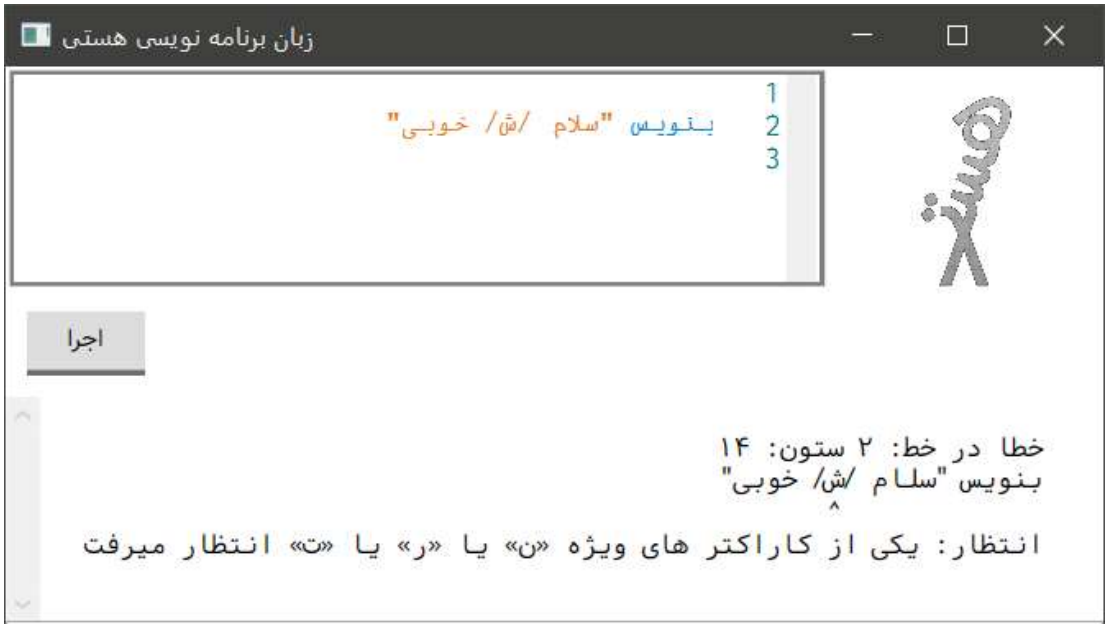
پیاده سازی زبان برنامه نویسی هستی

این زبان برنامه نویسی بدون استفاده از Parser Generator ها (مانند ANTLR) پیاده سازی شده است. به جای آن، با کمک FParsec که یک Parser Combinator برای زبان #F است بهره گرفته شده.

درParser Combinator ها با استفاده از تعریف تجزیه‌گر های بسیار ساده که هر کدام وظیفه تجزیه بخش کوچکی از زبان را بر عهده دارند، و همچنین عمل ترکیب این تجزیه‌گر ها به کمک Composition توابع، تجزیه گر های بزرگ و بزرگتری ساخته می‌شود که در نهایت میتواند کل زبان را تجزیه کند. در این روش انواعی برای ساختار زبان تعریف میگردد و سپس تجزیه گر نمونه هایی از این انواع را ساخته و در نهایت درخت تجزیه نحو زبان یعنی AST تشکیل می‌شود. انواع ساختار زبان در این زبان به شکل زیر است:

```
module BaseTypes

type HstAtomicTypeName = TypeName of string
and HstIdentifier = HstIdentifier of string
and HstUnit = HstUnit of unit

and HstType =
    | TypeAlias of string * HstTypeSig
    | RecordType of HstAtomicTypeName *
(HstIdentifier * HstType) list
    | SumType of (HstAtomicTypeName *
HstType) list

and HstTypeSig =
    | Atomic of HstAtomicTypeName
    | Chain of HstAtomicTypeName list

and HstBinding =
    { identifier : HstIdentifier
      args : HstIdentifier list option
      bindingType : HstTypeSig
      exps : HstStatement list }

and HstIfStmt =
    { cond : HstExpression
      thendo : HstStatement list }

and HstIfElseStmt =
    { cond : HstExpression
      thendo: HstStatement list
      elsedo : HstStatement list }

and HstExpression =
    | ExpValue of HstValue
    | ExpOpCall of HstOperatorCall
    | BindingExpr of HstIdentifier

and HstStatement =
     | IfStmt of HstIfStmt
     | IfElseStmt of HstIfElseStmt
     | TypeDef of HstType
     | Binding of HstBinding
     | Value of HstValue
     | OperatorCall of HstOperatorCall
     | PrintStatement of HstExpression
     | Loop of HstLoop
     | FuncCall of HstFuncCall

and HstFuncCall =
   { id : HstIdentifier
     callArgs : HstCallArg list }

and HstLoop =
    { fromIndex : HstNumber
      toIndex : HstNumber
      boundIdf : HstIdentifier
      body : HstStatement list }

and HstValue =
    | Number of HstNumber
    | String of HstString
    | RecordInstance of HstRecordInstance
    | Boolean of HstBoolean
    | Unit

and HstNumber =
    | Float of float
    | Integer of int

and HstCallArg =
    | CallValue of HstValue
    | CallIdf of HstIdentifier

and HstString = string

and HstRecordInstance = HstAtomicTypeName *
Map<HstIdentifier, HstValue>

and HstBoolean = HstTrue | HstFalse

and HstOperator = Operator of HstString

and HstOperatorCall =
    { op : HstOperator
      exp1 : HstExpression
   exp2 : HstExpression  }
```

شایان ذکر است که با این روش، به راحتی می‌توان خطاهای دقیقی را تولید کرد. به عنوان مثل به چند خطای تولید شده در محیط توسعه دقت کنید:




زبان برنامه نویسی هستی
1
2
3
اگه 0 == 0 بنویس "خوب"
اجرا
خطا در خط: ۲ ستون: ۱۲
اگه ۰ == ۰ بنویس "خوب"
^
انتظار: 'اونوقت'

همچنین توابع تجزیه گر به عنوان نمونه به شکل زیر توسعه داده شدند:

```
let ifElseStmt =
    pipe3 (pkw_if >>? pexpr .>>? ws .>>? pkw_then .>>? wsEol)
          (indentedStatements .>>? wsEol)
          (indented pkw_else >>? indentedStatements .>>? wsEol)

          (fun cond thn els -> IfElseStmt { cond = cond; thendo = thn; elsedo = els }) <?>
"عبارت شرطی"
```

تابع بالا به نام ifElseStms یک تجزیه‌گر برای عبارات شرطی را نشان میدهد. مطابق این تابع، ابتدا یک خط لوله 3 تایی تشکیل شده که سه تجزیه‌گر را به ترتیب روی کد اعمال کرده، و نتیجه خروجی آن‌ها را با استفاده از تابع نهایی در خط چهارم به شکل یک تجزیه‌گر دیگر برمیگرداند.

جدول زیر جزئیات توابع را نشان میدهد.

Parsers		**Return Types**
pkw_if	Parses “اگه” keyword	Parser<string>
pexpr	Parses an expression	Parser<HstStatement>
ws	Parses more than 0 white-spaces	Parser<unit>
pkw_then	Parses “اونوقت” keyword	Parser<string>
wsEol	Parses ws to the end of line and comments (can be empty)	Parser<unit>
pkw_else	Parses “وگرنه” keyword	Parser<string>
indentedStatements	Parses indented statements	Parser<HstStatement list>
indented	Parses indentation on parsers	Parser<’a> → Parser<’b>

Parser Operators		
`.>>`	Applies a then b, returns result of a	**P a → P b → P a**
`>>.`	Applies a then b, returns result of b	**P a → P b → P b**
`.>>.`	Applies a then b, returns result of (a * b)	**P a → P b → P (a*b)**
`.>>?`	Applies a then b, returns result of a	+ used for backtracking
`>>?`	Applies a then b, returns result of b	+ used for backtracking
`.>>.?`	Applies a then b, returns result of (a * b)	+ used for backtracking
`<?>`	Adds label to Parser	**P a → string → P a**

برخی دیگر از تجزیه‌گر های زبان:

Parsers		**Return Types**
pstring	Parses string	**string → Parser<string>**

notReserved	Used in Identifier parser to check if identifier is not a reserved keyword	**Parser<unit>**
idf	Parses identifier	**Parser<string>**
stringLiteral	Parses Hasti-lang customized strings	**Parser<string>**
numLiteral	Parses Int or Float	**Parser<HstNumber>**
pbool	Parses “درست” or “غلط”	**Parser<HstBoolean>**
punit	Parses “()”	**Parser<HstValue>**
pliteral	Parses literals in language, including Strings, Numbers, Units and Booleans	**Parser<HstValue>**
poperator	Parses operators, strings that are created using one of these characters ['+'; '='; '$'; '|'; '>'; '<'; '*'; '/'; '%']	**Parser<HstOperator>**
Comment	Parses comments	**Parser<unit>**
opCall	Parses an operation call like: 3 + 2	**Parses<HstOperationCall>**
ifStmt	Parses an indented if statement	**Parser<HstStatement>**
ifStmtInline	Parses an inline if statement	**Parser<HstStatement>**
ifElseStmt	Parses an indented if-else statement	**Parser<HstStatement>**
ifElseStmtInline	Parses an inline if-else statement	**Parser<HstStatement>**
binding	Parses a binding	**Parser<HstStatement>**
basicBinding	Parses an inline binding	**Parser<HstStatement>**
ploop	Parses a loop	**Parser<HstStatement>**
ploopInline	Parses an inline loop	**Parser<HstStatement>**
typedef	Parses a type definition	**Parser<HstStatement>**
pcallArg	Parses a call argument	**Parser<HstCallArg>**
funcCall	Parses a function call	**Parser<HstStatement>**
printStmt	Parses a print statement	**Parser<HstStatement>**
document	Parses a whole code script written in Hasti-Lang	**Parser<HstStatement list>**

پیاده سازی تجزیه‌گر در زبان برنامه نویسی هستی

اگرچه هدف اصلی زبان کامپایل به MSIL (کد میانی .NET) است، اما برای تست کد در نسخه اولیه، قابلیت تفسیر کد و اجرای آن به شکل ساده پیاده سازی شده است. تابع اصلی اجرای کد به شکل زیر است:

```
let rec exec (stmts:HstStatement list) (prints:List<string>) (session:Session) =
    for s in stmts do
        match s with
        | PrintStatement p -> prints.Add (statementPrint p session)
        | Binding b -> session.bindings.Add (b.identifier, b)
        | IfStmt ifs ->
            match evalBool ifs.cond session with
            | Ok true -> (exec ifs.thendo prints session) |> ignore
            | Error x -> prints.Add x
            | _ -> ()
        | IfElseStmt ifs ->
            match evalBool ifs.cond session with
            | Ok true ->  (exec ifs.thendo prints session) |> ignore
            | Ok false -> (exec ifs.elsedo prints session) |> ignore
            | Error x -> prints.Add x
        | Loop x -> let lpFunc = loopExecutator x.body prints session x.boundIdf
                    match (x.fromIndex, x.toIndex) with
                    | (Integer a, Integer b) -> lpFunc a        b
                    | (Float a, Integer b)   -> lpFunc (int a) b
                    | (Integer a, Float b)   -> lpFunc a       (int b)
                    | (Float a, Float b)     -> lpFunc (int a) (int b)
        | FuncCall f -> let funcBinding = session.bindings.[f.id]
                        execFunc funcBinding prints session f.callArgs
        | _ -> ()

    prints
```

در تابع بازگشتی exec مجموعه از دستورات، لیستی از عبارات پرینت (هنوز استریم پیاده سازی نشده است) و نشست فعلی شامل متغیر ها و نوع های تعریف شده به تابع داده شده، و با استفاده از تطبیق الگوها، مطابق با هر نوع عبارت آن را اجرا می‌کنیم.

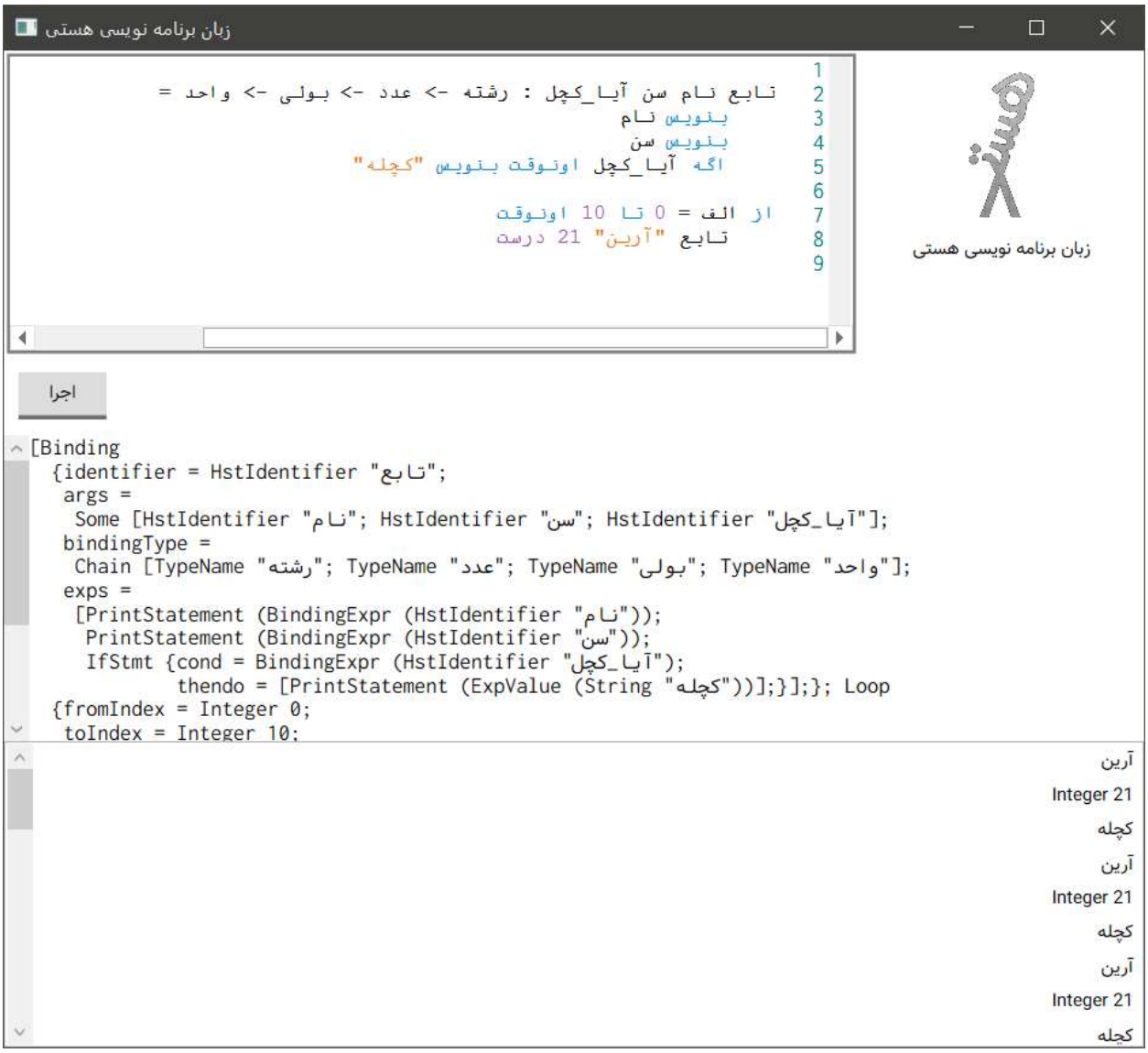
پیاده سازی محیط توسعه زبان برنامه نویسی هستی

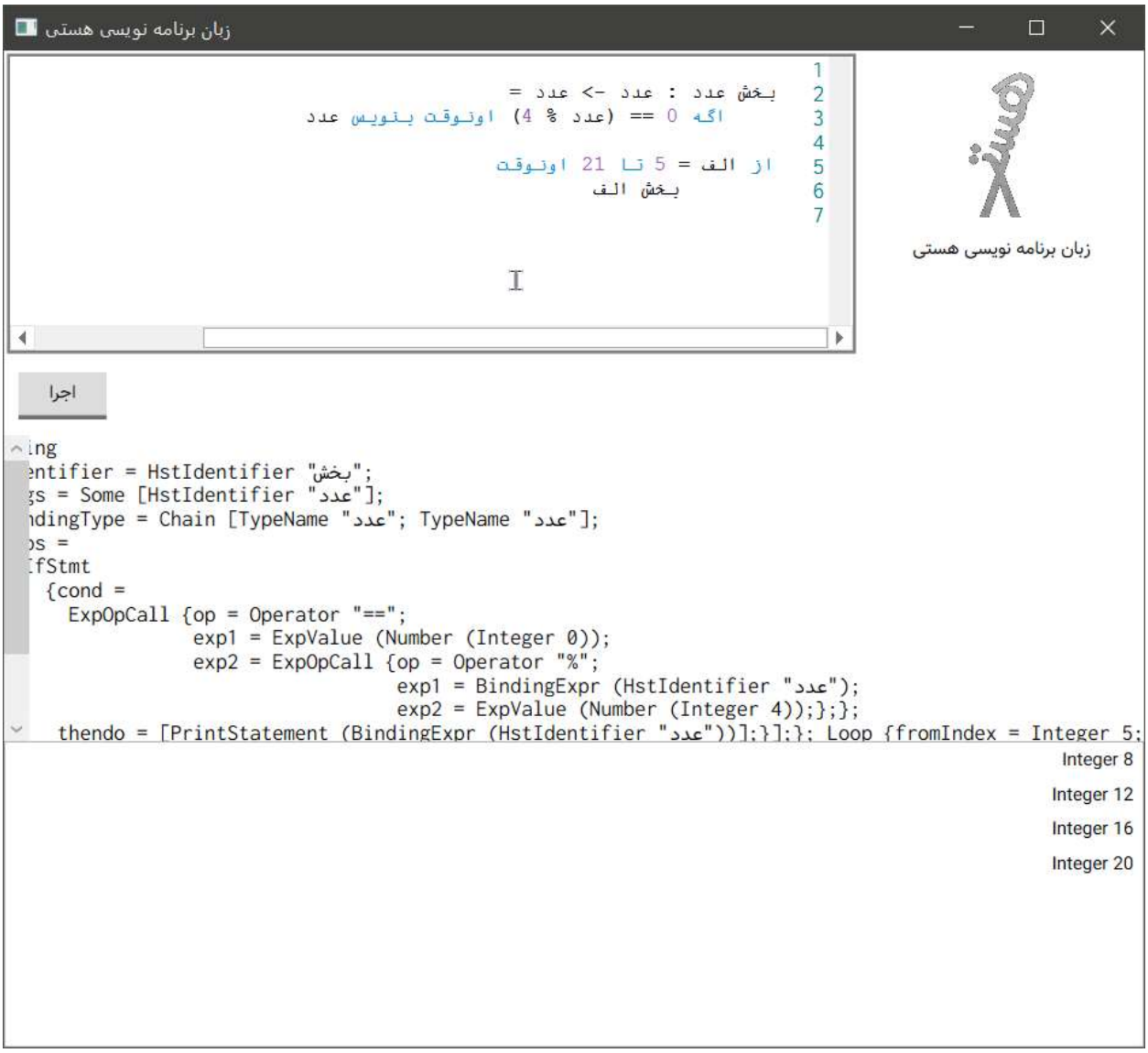
محیط توسعه این زبان، یک نرم افزار Desktop ساده، نوشته شده با زبان C#، تکنولوژی WPF و الگوی MVVM است. از آنجایی که هر دو زبان C# و F# زبان های دات‌نت هستند، امکان ارتباط این دو زبان به راحتی امکان پذیر است، و در یک Solution میتوان هر دو پروژه از نوع F# و C# ساخت که میتوانند هم را صدا بزنند.

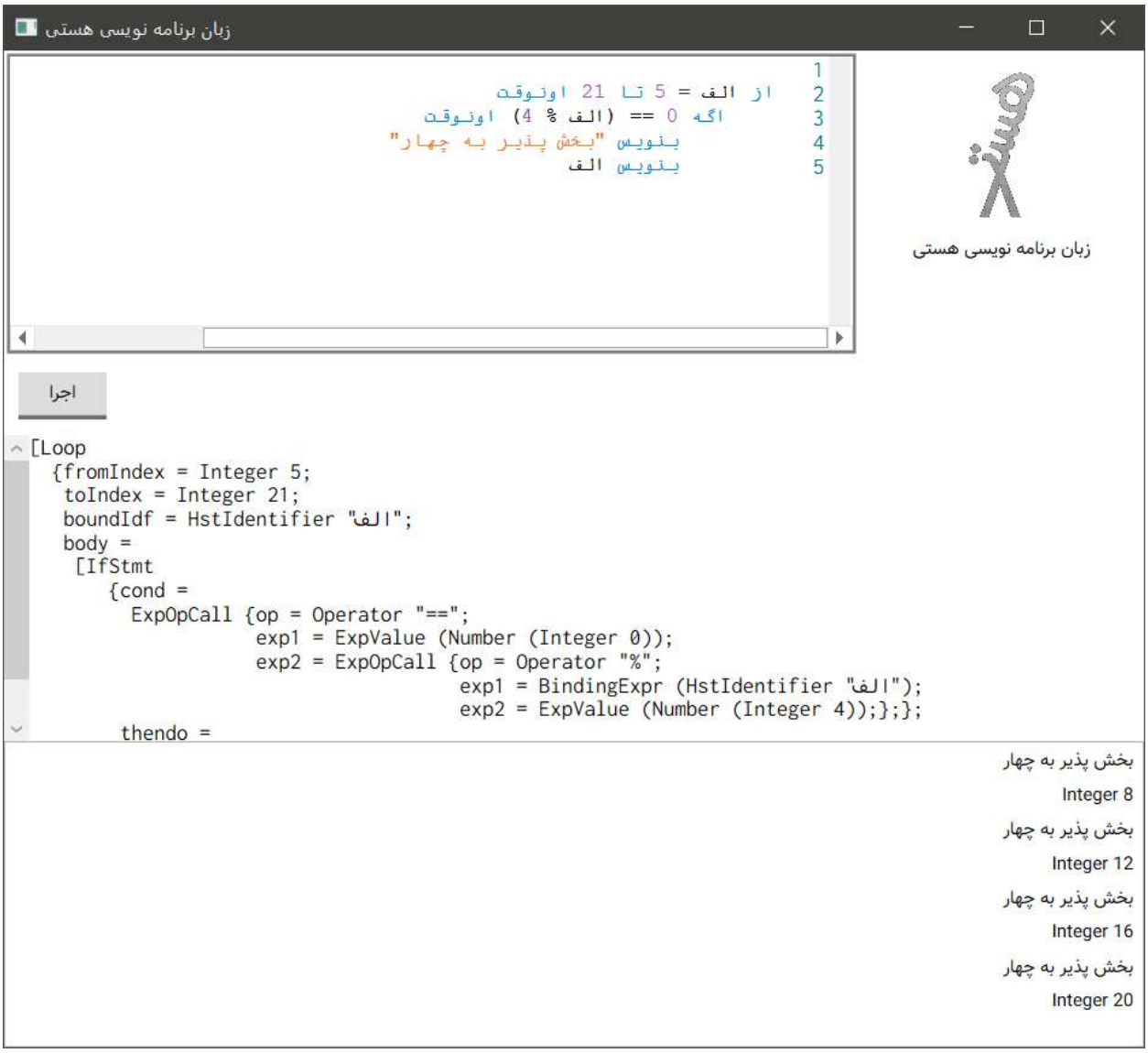
```
var x = HastiLang.Core.Ide.IdeUtils.RunParser.Invoke(EditControl.Text);
if(x is CompilerSuccess a)
{
    result.Text = a.Result.ToString();
    result.FlowDirection = FlowDirection.LeftToRight;
    resultList.Items.Clear();
    var run = CodeRunner.exec(a.Result, new List<string>(),
CodeRunner.createSession());

    run.ForEach(x => resultList.Items.Add(x));

}
else if(x is CompilerFailure b){
    result.Text = b.Error;
    resultList.Items.Clear();
    result.FlowDirection = FlowDirection.RightToLeft;
}
```

در تابع بالا RunParser از #F صدا زده می‌شود، و نتیجه آن در صورت برگردانده شدن یک CompilerSuccess اجرا شده و به همراه AST نمایش داده می‌شود، در غیر این صورت خطا روی صفحه مشاهده می‌گردد.

نمونه ای از کد های اجرا شده در این محیط توسعه:




زبان برنامه نویسی هستی
بخش عدد : عدد -> عدد =
اگه 0 == (عدد % 4) اونوقت بنویس عدد
از الف = 5 تا 21 اونوقت
بخش الف
زبان برنامه نویسی هستی
اجرا
ing
entifier = HstIdentifier "بخش";
gs = Some [HstIdentifier "عدد"];
ndingType = Chain [TypeName "عدد"; TypeName "عدد"];
os =
IfStmt
{cond =
ExpOpCall {op = Operator "==";
exp1 = ExpValue (Number (Integer 0));
exp2 = ExpOpCall {op = Operator "%";
exp1 = BindingExpr (HstIdentifier "عدد");
exp2 = ExpValue (Number (Integer 4));};};
thendo = [PrintStatement (BindingExpr (HstIdentifier "عدد"))];};]; Loop {fromIndex = Integer 5;
Integer 8
Integer 12
Integer 16
Integer 20


زبان برنامه نویسی هستی
1 از الف = 5 تا 21 اونوقت
2 اگه 0 == (الف % 4) اونوقت
3 بنویس "بخش پذیر به چهار"
4 بنویس الف
5
زبان برنامه نویسی هستی
اجرا
[Loop
{fromIndex = Integer 5;
toIndex = Integer 21;
boundIdf = HstIdentifier "الف";
body =
[IfStmt
{cond =
ExpOpCall {op = Operator "==";
exp1 = ExpValue (Number (Integer 0));
exp2 = ExpOpCall {op = Operator "%";
exp1 = BindingExpr (HstIdentifier "الف");
exp2 = ExpValue (Number (Integer 4));};};
thendo =
بخش پذیر به چهار
Integer 8
بخش پذیر به چهار
Integer 12
بخش پذیر به چهار
Integer 16
بخش پذیر به چهار
Integer 20