GOtham - compiles GOlang to x86

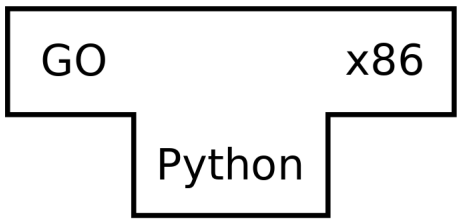
January 28, 2018

CS - 335A

1 Group Members

Akash Kumar Dutta - 150071 - akashdut@iitk.ac.in
Prakhar Agarwal - 150499 - pkhrag@iitk.ac.in
Swarnadeep Mandal - 150754 - swarnam@iitk.ac.in

2 T-diagram of the compiler



3 BNF of the source language

```
Notation-
Production  = production_name "=" [ Expression ] "." .
Expression  = Alternative { "|" Alternative } .
Alternative = Term { Term } .
Term        = production_name | token [ "..." token ] | Group | Option | Repetition .
Group       = "(" Expression ")" .
Option      = "[" Expression "]" .
Repetition  = "{" Expression "}" .

|   alternation
()  grouping
[]  option (0 or 1 times)
{}  repetition (0 to n times)
```

```
Characters-
newline        = /* the Unicode code point U+000A */ .
unicode_char   = /* an arbitrary Unicode code point except newline */ .
unicode_letter = /* a Unicode code point classified as "Letter" */ .
unicode_digit  = /* a Unicode code point classified as "Number, decimal digit" */ .


Letters and digits-
letter        = unicode_letter | "_" .
decimal_digit = "0" ... "9" .
octal_digit   = "0" ... "7" .
hex_digit     = "0" ... "9" | "A" ... "F" | "a" ... "f" .


Identifiers-
identifier = letter { letter | unicode_digit } .


Integer Literals-
int_lit     = decimal_lit | octal_lit | hex_lit .
decimal_lit = ( "1" ... "9" ) { decimal_digit } .
octal_lit   = "0" { octal_digit } .
hex_lit     = "0" ( "x" | "X" ) hex_digit { hex_digit } .


Floating-point literals-
float_lit = decimals "." [ decimals ] [ exponent ] |
            decimals exponent |
            "." decimals [ exponent ] .
decimals  = decimal_digit { decimal_digit } .
exponent  = ( "e" | "E" ) [ "+" | "-" ] decimals .


Imaginary literals-
imaginary_lit = (decimals | float_lit) "i" .


Rune Literals-
rune_lit         = "'" ( unicode_value | byte_value ) "'" .
unicode_value    = unicode_char | little_u_value | big_u_value | escaped_char .
byte_value       = octal_byte_value | hex_byte_value .
octal_byte_value = `\` octal_digit octal_digit octal_digit .
hex_byte_value   = `\` "x" hex_digit hex_digit .
little_u_value   = `\` "u" hex_digit hex_digit hex_digit hex_digit .
big_u_value      = `\` "U" hex_digit hex_digit hex_digit hex_digit
```

```
                          hex_digit hex_digit hex_digit hex_digit .
escaped_char     = `\` ( "a" | "b" | "f" | "n" | "r" | "t" | "v" | `\` | "'" | `"` ) .


String Literals-
string_lit             = raw_string_lit | interpreted_string_lit .
raw_string_lit         = "`" { unicode_char | newline } "`" .
interpreted_string_lit = `"` { unicode_value | byte_value } `"` .


Types-
Type      = TypeName | TypeLit | "(" Type ")" .
TypeName  = identifier | QualifiedIdent .
TypeLit   = ArrayType | StructType | PointerType .


Array Types-
ArrayType   = "[" ArrayLength "]" ElementType .
ArrayLength = Expression .
ElementType = Type .


Struct Types-
StructType    = "struct" "{" { FieldDecl ";" } "}" .
FieldDecl     = (IdentifierList Type) [ Tag ] .
Tag           = string_lit .


Pointer Types-
PointerType = "*" BaseType .
BaseType    = Type .


Function Types-
Signature      = Parameters [ Result ] .
Result         = Parameters | Type .
Parameters     = "(" [ ParameterList [ "," ] ] ")" .
ParameterList  = ParameterDecl { "," ParameterDecl } .
ParameterDecl  = [ IdentifierList ] Type .
MethodName         = identifier .



Blocks-
Block = "{" StatementList "}" .
StatementList = { Statement ";" } .
```

```
Declarations and Scope-
Declaration   = ConstDecl | TypeDecl | VarDecl .
TopLevelDecl  = Declaration | FunctionDecl .


Constant Declrations-
ConstDecl      = "const" ( ConstSpec | "(" { ConstSpec ";" } ")" ) .
ConstSpec      = IdentifierList [ [ Type ] "=" ExpressionList ] .
IdentifierList = identifier { "," identifier } .
ExpressionList = Expression { "," Expression } .


Type Declarations-
TypeDecl = "type" ( TypeSpec | "(" { TypeSpec ";" } ")" ) .
TypeSpec = AliasDecl | TypeDef .
AliasDecl = identifier "=" Type .


Type Definitions-
TypeDef = identifier Type .


Variable declaration-
VarDecl     = "var" ( VarSpec | "(" { VarSpec ";" } ")" ) .
VarSpec     = IdentifierList ( Type [ "=" ExpressionList ] | "=" ExpressionList ) .


Short Variable Declarations-
ShortVarDecl = IdentifierList ":=" ExpressionList .


Function Declaration-
FunctionDecl = "func" FunctionName ( Function |  Signature ) .
FunctionName = identifier .
Function     = Signature FunctionBody .
FunctionBody = Block .


Operands-
Operand     = Literal | OperandName | "(" Expression ")" .
Literal     = BasicLit | CompositeLit.
BasicLit    = int_lit | float_lit | imaginary_lit | rune_lit | string_lit .
OperandName = identifier | QualifiedIdent.
```

```
Qualified Identifier-
QualifiedIdent = PackageName "." identifier .


Composite literals-
CompositeLit  = LiteralType LiteralValue .
LiteralType   = StructType | ArrayType | "[" "..." "]" ElementType |
                TypeName .
LiteralValue  = "{" [ ElementList [ "," ] ] "}" .
ElementList   = KeyedElement { "," KeyedElement } .
KeyedElement  = [ Key ":" ] Element .
Key           = FieldName | Expression | LiteralValue .
FieldName     = identifier .
Element       = Expression | LiteralValue .


Primary Expressions-
PrimaryExpr =
    Operand |
    Conversion |
    PrimaryExpr Selector |
    PrimaryExpr Index |
    PrimaryExpr Slice |
    PrimaryExpr TypeAssertion |
    PrimaryExpr Arguments .

Selector       = "." identifier .
Index          = "[" Expression "]" .
Slice          = "[" [ Expression ] ":" [ Expression ] "]" |
                 "[" [ Expression ] ":" Expression ":" Expression "]" .
TypeAssertion  = "." "(" Type ")" .
Arguments      = "(" [ ( ExpressionList | Type [ "," ExpressionList ] ) [ "," ] ] ")" .


Operators-
Expression = UnaryExpr | Expression binary_op Expression .
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr .
binary_op  = "||" | "&&" | rel_op | add_op | mul_op .
rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op     = "+" | "-" | "|" | "^" .
mul_op     = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .
unary_op   = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .


Conversions-
```

```
Conversion = Type "(" Expression [ "," ] ")" .


Statements-
Statement =
    Declaration | LabeledStmt | SimpleStmt | ReturnStmt | BreakStmt | ContinueStmt
    | GotoStmt | Block | IfStmt | SwitchStmt | ForStmt .

SimpleStmt = EmptyStmt | ExpressionStmt | IncDecStmt | Assignment | ShortVarDecl .


Empty Statement-
EmptyStmt = .


Labeled Statements-
LabeledStmt = Label ":" Statement .
Label       = identifier .


Expression statements-
ExpressionStmt = Expression .


IncDec Statements-
IncDecStmt = Expression ( "++" | "--" ) .


Assignments-
Assignment = ExpressionList assign_op ExpressionList .

assign_op = [ add_op | mul_op ] "=" .


If Statements-
IfStmt = "if" [ SimpleStmt ";" ] Expression Block [ "else" ( IfStmt | Block ) ] .


Switch Statements-
SwitchStmt = ExprSwitchStmt | TypeSwitchStmt .
ExprSwitchStmt = "switch" [ SimpleStmt ";" ] [ Expression ] "{" { ExprCaseClause } "}" .
ExprCaseClause = ExprSwitchCase ":" StatementList .
ExprSwitchCase = "case" ExpressionList | "default" .
TypeSwitchStmt  = "switch" [ SimpleStmt ";" ] TypeSwitchGuard "{" { TypeCaseClause } "}"
TypeSwitchGuard = [ identifier ":=" ] PrimaryExpr "." "(" "type" ")" .
TypeCaseClause  = TypeSwitchCase ":" StatementList .
```

```
TypeSwitchCase  = "case" TypeList | "default" .
TypeList        = Type { "," Type } .


For Statement-
ForStmt = "for" [ Condition | ForClause | RangeClause ] Block .
Condition = Expression .
ForClause = [ InitStmt ] ";" [ Condition ] ";" [ PostStmt ] .
InitStmt = SimpleStmt .
PostStmt = SimpleStmt .
RangeClause = [ ExpressionList "=" | IdentifierList ":=" ] "range" Expression .


Return Statements-
ReturnStmt = "return" [ ExpressionList ] .


Break Statements-
BreakStmt = "break" [ Label ] .


Continue Statements-
ContinueStmt = "continue" [ Label ] .


Goto statements-
GotoStmt = "goto" Label .


Source File Organization-
SourceFile       = PackageClause ";" { ImportDecl ";" } { TopLevelDecl ";" } .


Package Clause-
PackageClause  = "package" PackageName .
PackageName    = identifier .


Import Declarations-
ImportDecl       = "import" ( ImportSpec | "(" { ImportSpec ";" } ")" ) .
ImportSpec       = [ "." | PackageName ] ImportPath .
ImportPath       = string_lit .
```

4 Deleted Rules from the source language

We have deleted the following rules from the source language -

```
Channel Type-
ChannelType = ( "chan" | "chan" "<-" | "<-" "chan" ) ElementType .


Go command-
GoStmt = "go" Expression .


Select Statement-
SelectStmt = "select" "{" { CommClause } "}" .
CommClause = CommCase ":" StatementList .
CommCase   = "case" ( SendStmt | RecvStmt ) | "default" .
RecvStmt   = [ ExpressionList "=" | IdentifierList ":=" ] RecvExpr .
RecvExpr   = Expression .


Send Statement-
SendStmt = Channel "<-" Expression .
Channel  = Expression .


Fallthrough Statement-
FallthroughStmt = "fallthrough" .


Defer statement-
DeferStmt = "defer" Expression .


Slice type-
SliceType = "[" "]" ElementType .

Map type-
MapType     = "map" "[" KeyType "]" ElementType .
KeyType     = Type .


Interface type-
InterfaceType      = "interface" "{" { MethodSpec ";" } "}" .
MethodSpec         = MethodName Signature | InterfaceTypeName .
MethodName         = identifier .
InterfaceTypeName  = TypeName .


Embedded field-
EmbeddedField = [ "*" ] TypeName .
```

```
Function definition inside args-
By removing function types.


Method Types and Expressions-
MethodExpr    = ReceiverType "." MethodName .
ReceiverType  = TypeName | "(" "*" TypeName ")" | "(" ReceiverType ")" .


Variable argument list-
Removing \..." from function arguments


Inline function declaration-
Removed inline method declaration.
```

5 Semantic description of the constructs added

We will use Polymorphism as an OOP construct in our language. The particular part of polymorphism that we are targeting is operator overload.
The example of operator overload can be seen from the following example:
a : type INT
b : type INT
c : type STRING
d : type STRING
Now in our language the code :- $a + b$ would lead to simple addition of integer values, while the same + operator for strings, i.e. $c + d$ would concatenate the strings.

6 Tools to be used

We would use the following tools at different stages as follows:-

- lexer generator - PLY for python
- parser
- Semantic Analyzer
- Code Generator