COMPILERS: CS335

EBNF FOR GO LANGUAGE
ASSIGNMENT 0

Rohit Gupta	150594	rohitgpt
Pranav Sao	150504	prnvsao
Romal Jaiswal	150299	romalrj

GO EBNF

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

NOTE : Red colored rules are those which we are not implementing.

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" */ .

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 } .

Keywords

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	if	range	type
continue	for	import	return	var

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 .

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_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 | FunctionType | InterfaceType |
            SliceType | MapType | ChannelType .
```

```
ArrayType   = "[" ArrayLength "]" ElementType .
ArrayLength = Expression .
ElementType = Type .
```

Slice types

```
SliceType = "[" "]" ElementType .
```

Struct types

```
StructType    = "struct" "{" { FieldDecl ";" } "}" .
FieldDecl     = (IdentifierList Type | EmbeddedField) [ Tag ] .
EmbeddedField = [ "*" ] TypeName .
Tag           = string_lit .
```

Pointer types

```
PointerType = "*" BaseType .
BaseType    = Type .
```

Function types

```
FunctionType   = "func" Signature .
Signature      = Parameters [ Result ] .
Result         = Parameters | Type .
Parameters     = "(" [ ParameterList [ "," ] ] ")" .
```

ParameterList = ParameterDecl { "," ParameterDecl } .
ParameterDecl = [IdentifierList] ["..."] Type .

Interface types

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

Map types

MapType = "map" "[" KeyType "]" ElementType .
KeyType = Type .

Blocks

A *block* is a possibly empty sequence of declarations and statements within matching brace brackets.
Block = "{" StatementList "}" .
StatementList = { Statement ";" } .

Declaration = ConstDecl | TypeDecl | VarDecl .
TopLevelDecl = Declaration | FunctionDecl | MethodDecl .

Predeclared identifiers

The following identifiers are implicitly declared in the universe block:
Types:

bool byte complex64 complex128 error float32 float64
int int8 int16 int32 int64 rune string
uint uint8 uint16 uint32 uint64 uintptr

Constants:

true false iota

Zero value:
nil

Constant declarations

ConstDecl = "const" (ConstSpec | "(" { ConstSpec ";" } ")") .
ConstSpec = IdentifierList [[Type] "=" ExpressionList] .

IdentifierList = identifier { "," identifier } .
ExpressionList = Expression { "," Expression } .

Type declarations

TypeDecl = "type" (TypeSpec | "(" { TypeSpec ";" } ")") .
TypeSpec = AliasDecl | TypeDef .

Alias declarations

AliasDecl = identifier "=" Type .

Type definitions

TypeDef = identifier Type .

Variable declarations

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

Short variable declarations

ShortVarDecl = IdentifierList ":=" ExpressionList .

Function declarations

A function declaration binds an identifier, the *function name*, to a function.
FunctionDecl = "func" FunctionName (Function | Signature) .
FunctionName = identifier .
Function = Signature FunctionBody .

FunctionBody = Block .

Method declarations

MethodDecl = "func" Receiver MethodName (Function | Signature) .
Receiver = Parameters .

Expressions

Operands

Operand = Literal | OperandName | MethodExpr | "(" Expression ")" .
Literal = BasicLit | CompositeLit | FunctionLit .
BasicLit = int_lit | float_lit | imaginary_lit | rune_lit | string_lit .
OperandName = identifier | QualifiedIdent.

Qualified identifiers

QualifiedIdent = PackageName "." identifier .

Composite literals

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

Function literals

FunctionLit = "func" Function .

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 ] ) [ "..." ] [ "," ] ] ")" .
```

Selectors

Method expressions

```
MethodExpr    = ReceiverType "." MethodName .
ReceiverType  = TypeName | "(" "*" TypeName ")" | "(" ReceiverType ")" .
```

Method values

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 |
    GoStmt | ReturnStmt | BreakStmt | ContinueStmt | GotoStmt |
    FallthroughStmt | Block | IfStmt | SwitchStmt | SelectStmt | ForStmt |
    DeferStmt .

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

Empty statements

EmptyStmt = .

Labeled statements

LabeledStmt = Label ":" Statement .
Label   = identifier .

Expression statements

ExpressionStmt = Expression .

Send statements

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

Expression switches

ExprSwitchStmt = "switch" [SimpleStmt ";"] [Expression] "{" { ExprCaseClause } "}" .
ExprCaseClause = ExprSwitchCase ":" StatementList .
ExprSwitchCase = "case" ExpressionList | "default" .

Type switches

TypeSwitchStmt = "switch" [SimpleStmt ";"] TypeSwitchGuard "{" { TypeCaseClause } "}" .
TypeSwitchGuard = [identifier ":="] PrimaryExpr "." "(" "type" ")" .
TypeCaseClause = TypeSwitchCase ":" StatementList .
TypeSwitchCase = "case" TypeList | "default" .
TypeList   = Type { "," Type } .

For statements

ForStmt = "for" [Condition | ForClause | RangeClause] Block .
Condition = Expression .

For statements with for clause

ForClause = [InitStmt] ";" [Condition] ";" [PostStmt] .
InitStmt = SimpleStmt .
PostStmt = SimpleStmt .

For statements with range clause

RangeClause = [ExpressionList "=" | IdentifierList ":="] "range" Expression .

Go statements

GoStmt = "go" Expression .

Select statements

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

Return statements

ReturnStmt = "return" [ExpressionList] .

Break statements

BreakStmt = "break" [Label] .

Continue statements

ContinueStmt = "continue" [Label] .

Goto statements

GotoStmt = "goto" Label .

Fallthrough statements

FallthroughStmt = "fallthrough" .

Defer statements

DeferStmt = "defer" Expression .

Packages

Source file organization

SourceFile = PackageClause ";" { ImportDecl ";" } { TopLevelDecl ";" } .

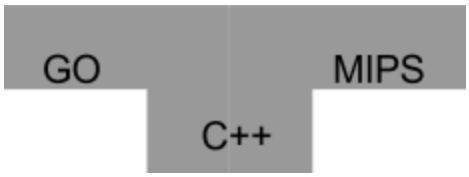
Package clause

PackageClause = "package" PackageName .
PackageName = identifier .

Import declarations

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

T-diagram



Tools

1. Flex
2. Bison
3. SPIM MIPS Simulator

References

The Go Programming Language Specification