CSC 330 - Test 1

Team 1

1. History

Background

On September 21, 2007, a team of Google programmers, Robert Griesemer, Rob Pike, and Ken Thompson, identified the need for a scalable programming language to help simplify some of the problems they were facing while programming complex networks. To resolve these issues, they developed the language Go that focused on three key needs; fast compilation, efficient garbage collection, and the support for concurrent execution and communication [1.1].

By November 10, 2009, Go was released as an open source project [1.1]. This open source model allowed Go's creators to better understand and facilitate the needs of a diverse user base. As the project grew, it became clear that more standardization was needed to improve portability and to make the language accessible to a larger audience.

By March 28, 2012, the first stable release, Go 1, was introduced along with binary distributions for common operating systems [1.2]. The version intended primarily to focus on stability. This meant that any documentation written about the language would be applicable to all users, and any program compiled in Go 1 would continue to compile even as new versions were released [1.2]. Unfortunately, standardizing Go years after its initial 2009 release meant that some older programs were suddenly incompatible with the new standard. The Go creators released a tool called *go fix* that automates the process of updating old code to address this issue.

In addition to *go fix*, a whole suite of tools were developed to be used with *go command*, "that automates the downloading, building, installation, and testing of Go packages and commands" [1.3].

Since Go 1, there have been 7 major releases, from Go 1.1 to Go 1.7. Each one focuses on incremental changes like expanding the Go library, improving compiler and garbage collection efficiency, and bug fixes. One of the most significant changes came in version 1.5 when a new compiler was introduced [1.4].

Motivations

The main motivations behind Go were to specifically address software engineering difficulties. It serves not to radically change established ideas, but rather to alleviate some point points. Go is designed so that programmers can quickly become proficient and productive if they are already familiar with languages like Java, JavaScript, and C [1.5]. As such, its semantics draw strong influence from those languages, although there are numerous differences. These changes are made in favour of robustness, efficient compilation, fast execution, and ease of programming. Many powerful modern features are also included.

Significant Feature Overview

Created by Google for use in large concurrent systems [1.6], Go discourages the use of traditional "synchronization primitives" like semaphores and mutexes, and instead encourages "message passing" as a primary means of communication between threads. It is a garbage-collected language and it does not have a traditional class-based object-oriented programming model. Go instead allows the programmer to define new types as aliases of other types (including any struct), and then to define methods that operate on that type. In this way, Go can be object-oriented, but it has no inheritance.

Historic Influences

Go attempts to improve on the failures of existing languages while maintaining their favourable characteristics. The ultimate goal is to create a language that is readable, safe, and efficient. Some of Go's major ancestors include ALGOL 60, C, and CSP[1.7].

Syntactically, Go is primarily influenced by C and related languages, however it takes semantic cues from many diverse languages. Go's type inference is similar to Python's, although Go also provides the option to explicitly specify types. Go's concurrency model is largely similar to Limbo's, which is also based on CSP [1.3].

Go inherited its expression syntax, control-flow, statements, basic data types, and call-by-value parameter passing from C. C compiles programs into efficient machine code, however, it can easily become confusing as it grows more complex [1.7]. Go solves this by adapting Pascal style declaration, declaring variables and type from left to right. This ensures the code readable at any point [1.8]. Also, the regular syntax does not require a symbol table to parse. Semicolons, parentheses, exceptions, and conditions are removed. Furthermore, Go includes features not present in C, such as slices, maps, composite literals, expressions at the top level of the file, and reflection [1.9].

Go is heavily influenced by languages created by Niklaus Wirth, starting with Pascal [1.7]. Next, Modula-2 inspired Go's package system, which speeds up debugging and compilation. Oberon eliminated the difference between module interface files and module implementation files. Lastly, Oberon-2 influenced the syntax for packages, imports, and declaration.

Go vs. C

Go has many notable minor differences from C in favour of safety and ease of use. For example, there is no support for pointer arithmetic or implicit numeric conversions, and the bounds of arrays are always checked. There are also more significant differences like Go's type inference [1.5].

Concurrency

Due to the rising ubiquity of multicore computers and distributed systems, Go features built-in language support for concurrency. This allows web servers written in Go to serve huge quantities of clients, a common requirement in today's software engineering industry. Conversely, languages like C++, Python, JavaScript and Java struggle in this regard due to their lack of native concurrency.

Go's concurrency model, essentially CSP with first-class channels, was influenced by one of the original developer's experience creating a language that was built on the ideas of CSP [1.5]. Further, CSP was relatively straightforward to include in a procedural programming model without major changes. Since few changes were necessary for this compatibility, concurrency was added at no expense to other parts of the language. This approach of independently concurrently running code allows powerful versatility. This makes Go extremely practical for web servers. For example, for a web server requiring related complex calculations, clients can be independently managed without compromising efficient execution.

With respect to its intended use cases, Go's concurrent features are simple and effective. This is largely due to its familiarity to those with procedural programming experience.

Garbage Collection

Go's inventors believe that manually maintaining program memory is tedious and cumbersome [1.5]. Thus, in contrast to many systems languages, Go has full garbage collection support to simplify development. In fact, it is impossible for allocated memory to be freed through any other means. Furthermore, since it can be difficult to maintain ownership of memory in the context of concurrency, Go attempts to separate program logic from managing resources. With regards garbage collection, Go is more similar to Java or C# than C or C++. Historically, this creates a large overhead and latency, but Go features certain properties and tools available to help mitigate these issues [1.7].

2. Syntax and BNF Grammar

Grammar of Go

Go’s syntax and grammar are specified “using Extended Backus-Naur Form (EBNF)” [2.1]. When translating Go’s EBNF grammar to produce corresponding railroad diagrams, we found that Go’s syntax required tweaking to conform to the tool's EBNF rules. There were six discrepancies between what Google claimed was EBNF and what the railroad diagram tool required:

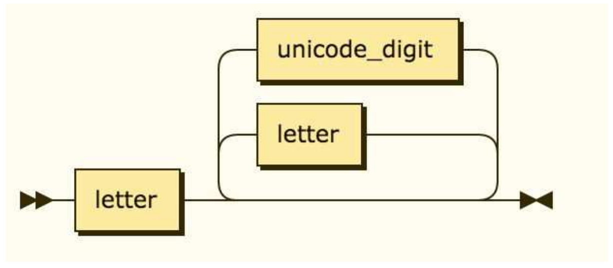
1. Google uses '=' rather than '::='
2. Google terminates each declaration with '.'
3. Google uses `backticks` instead of 'single quotes'
4. Google defines '{' and '}' to mean "repetition", rather than using the Kleene Star '*'
5. Google defines '[' and ']' to mean "option", rather than using the question mark '?'
6. Google uses ellipses '…' to denote a range of characters, rather than "A - Z"

These syntax differences lead us to conclude that Google’s "EBNF" is closer to a different metalanguage, Wirth Syntax Notation.

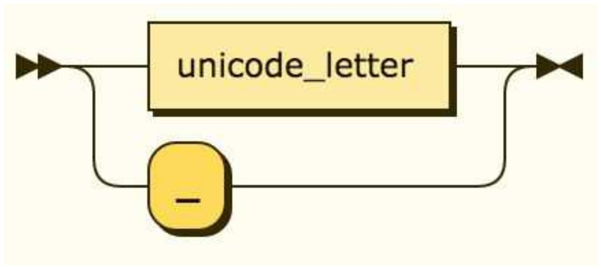
Translating Grammar to Rail Diagrams

The complete grammar and accompanying rail diagrams are included as an appendix. Note that the "newline", "unicode_char", "unicode_letter", and "unicode_digit" terms are only defined in the specification by comments, as there are simply too many Unicode characters to write out, even using ranges. Below are two snippets of EBNF grammar and rail diagrams generated from that grammar.

```
identifier ::= letter ( letter | unicode_digit )*
```



```
letter ::= unicode_letter | '_'
```



A Go "identifier" is used to name things like variables and new types. An identifier must start with one "letter" (alphabetic character) followed by any number of letters or digits. In other words, an identifier is made up of alphanumeric characters, and can not begin with a digit. The rail diagram for "identifier" reflects this: the rail passes through the "letter" block, then can enter either of the "unicode_digit" or "letter" blocks any number of times before finishing.

The expression and diagram for "letter" has also been included. Note how it explicitly allows the character '_' in addition to alphabetic Unicode characters. This second example also demonstrates the difference between terminals and nonterminals in EBNF and rail diagrams, with terminals surrounded in quotation marks in EBNF and in rounded boxes (rather than the square boxes of nonterminals) in a rail diagram.

Go vs. C Syntax

While Go is semantically similar to C, it was developed with two goals in mind to differentiate them [2.2]. Go was developed to have a “light” [2.3] feeling syntax, free of mandatory keywords and unnecessary repetition. As a result, Go eschews parentheses and semicolons wherever possible. Secondly, Go was designed to be far easier to analyze syntactically making it easier to parse and compile.

To achieve the two primary design goals, the following design choices were made:

1. Semicolons are not necessary for line termination in Go
2. Braces *must* go on the same line as the control structure that they belong to
3. Indentation is always done with tabs
4. No “brace-free” forms of if’s and loop statements
5. Go can be written to have implicit declarations and automatic typing
6. If statements do not require parentheses around their boolean expressions
7. Function declarations are “backwards” compared to C, with the return type coming last

As briefly discussed above, Go does not feature pointer arithmetic. With this change, the postfix and prefix increment (++) and decrement (--) operators are replaced with a single postfix statement. With this removal, “expression syntax is simplified” [2.3] and the common issues of evaluating expressions involving the ++ and -- operators are eliminated.

3. Compilers and Interpreters

The Development Toolchain

The Go toolchain originated from another compiler toolchain which came from an operating system called Plan 9. Plan 9 was capable of merging multiple computers into a single system. It was possible for this system to contain nodes with different processor architectures (e.g. ARM, PowerPC, x86). Thus, Plan 9 was able to support many linkers written in C [3.1]. Plan 9’s assemblers, compilers and linkers were adopted by the Go toolchain and were left mostly untouched during its development. Currently, Go’s toolchain is made up of a Go compiler, a C compiler, an assembler, and a linker [3.2].

How a Program is Executed

Note: The following process is described in the context of using the "gc" Go compiler, rather than "gccgo".

In order to execute and build programs in Go, it is necessary to go through the two main processes: compiling and linking; both are moderated by the `go` command/tool within the compiler. The Go compiler requires a specific directory structure. This comprises three main requirements [3.3]:

1. Go’s bin directory must be within the file path
2. There needs to be a `src` directory that stores the local programs and packages
3. The directory above `src` must be within the GOPATH environment variable

For example (assume the path includes the bin directory, `%GOROOT%\bin`):

```
$ export GOPATH=$HOME/gopath
$ cd $GOPATH/src/hello
$ go build hello.go
$ ./hello
```

Once the `go` tool has successfully built the program, the binary is ready to be executed. The go `build` command compiles the program and stores the result for future use; this also create another executable file which can be run without re-processing the program, allowing Go to compile quickly [3.4].

The Go Compiler

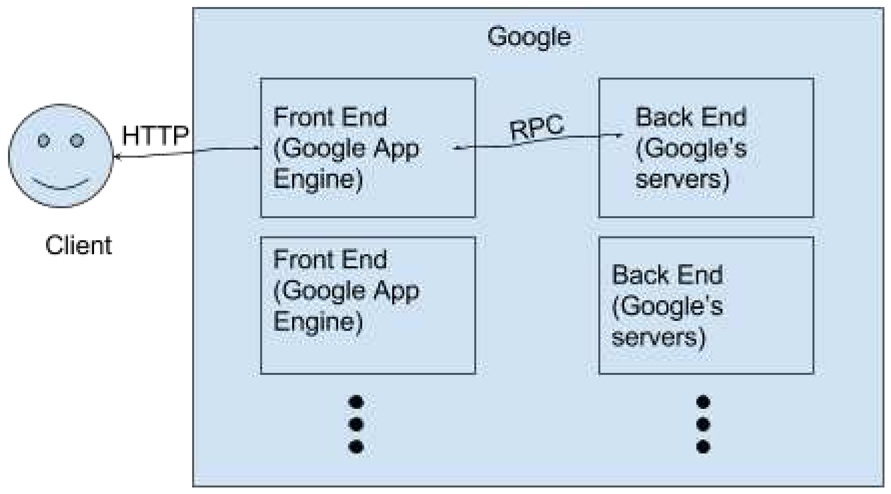
Originally, the Go compiler was written in C. It was later rewritten in Go for version 1.5. The compiler was first written in C due to bootstrapping issues: since Go was not yet implemented, it could not yet be used to build its own compiler [3.5]. Furthermore, Go was not designed to be a compiler implementation language [3.6]. The compiler implementation switched from C to Go for practical reasons: Go was easier to write correctly than C, Go makes parallel execution trivial, and Go has better support for testing and modularity [3.5].

When the compiler was converted from C to Go, custom auto-translation tools were repeatedly run over the code until it succeeded. Success was measured by “bit identical” output and then code was cleaned up by machine and by hand [3.7]. Once the compiler was translated into Go, it was self-hosting. The plan for bootstrapping was to have a functional Go compiler in place before creating the next compiler version. In other words, the Go 1.2 toolchain would be used to write the Go 1.3 toolchain, the Go 1.4 toolchain would be written using Go 1.3, and so on [3.5].

Go Playground

The Go Playground is an online web service launched in September 2010 that allows execution of arbitrary Go code and returns the program output [3.8]. The Go Playground compiles, links, and executes Go code on Google's servers, then returns the result to the user. The Go Playground is composed of three parts: the front end that runs on Google App Engine, the back end which runs on Google’s servers, and a Javascript client that implements the playground’s user interface [3.9].

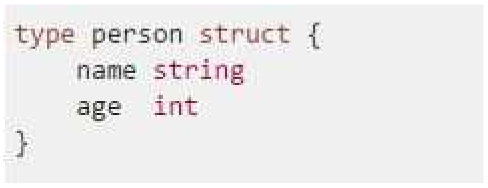
To safely execute a program in the browser, the back end runs the user’s program under Native Client (“NaCl”) using a custom version of the Go toolchain that generates NaCl executables [3.9]. Upon receiving a compilation request, the front end checks if there is already a copy of the program in its cache [3.9]. If no cached version exists, the front end makes a request to the back end, saves the response, parses the playback events, and returns a JSON object as an HTTP response [3.9]. The infrastructure of the Go Playground service is depicted below:


Google
Client
HTTP
Front End (Google App Engine)
RPC
Back End (Google's servers)
Front End (Google App Engine)
Back End (Google's servers)
The infrastructure of the Go Playground [3.9]

4. Type Construction and Type-Checking

Types available in Golang:

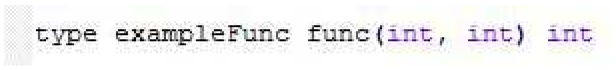
Go is equipped with all of the standard primitive types such as Booleans, Integers, Floats, Strings and Chars. The numeric types in Go are size-specific, and different types are used for the required number of bytes (e.g. int8 and float64). Composite types in Go language include: array, struct, pointer, slice, function, interface, map and channel types. New types are defined in Go by using “struct” as is shown in figure 4.1.


```
type person struct {
    name string
    age  int
}
```
Figure 4.1: A constructed type

Go is statically typed and is known for its immutable data types. This means that each piece of data is unable to change types once assigned; attempting to change a datatype will throw an error.

Creating new Types in Golang:

As stated earlier, new types can be constructed in Go as in C, by using a “struct”. In figure 4.1, a new type called “person” is defined and given two properties: a string “name” and integer “age”. Newly created types in Go are constructed through the union of other data types. For example, the type “person”, from figure 4.1, is defined as a union between a “string” and an “int” or `person = string + int`. The abstract concept of sequences can also be found in Go through its Arrays. For example, an array of the “person” type could be written formally as `[]person = person*`. Function types can also be created like shown in figure 4.2. In this example, `exampleFunc = function -> int`.



```
type exampleFunc func(int, int) int
```

Figure 4.2: Example of function type

Go’s Type System as Compared to C

A type system is defined by a set of basic types, a mechanism for defining new types, and rules for type equivalence, type compatibility, type inference and type checking [4.5].

Go’s and C’s type systems share some basic types such as int and float, but Go relies more on primitives than C. C features only int, double, float and char, while Go has basic types for booleans and strings. C has no string type and thus uses char arrays to implement strings.

Basic Types	
GO	C
bool	double
string	char
int int8 int16 int32 int64	int
uint uint8 uint16 uint32 uint64 uintptr	
byte // alias for uint8	
rune // alias for int32	
// represents a Unicode code point	
float32 float64	float
complex64 complex128	

Figure 4.3 Basic type comparison between Go and C [4.1][4.2]

Syntax	
GO	C
x int	Int x;
p *int	Int *p;
a [3]int	Int a[3];
func main(argc int, argv []string) int	int main(int argc, char *argv[]) { /* ... */ }

Figure 4.4 *Syntax comparison between Go and C* [4.3]

Type Equivalence

In Go all named types follow a "named equivalence" structure. That is, a named type is never equivalent to differently named or unnamed type. Unnamed types are compared through structural equivalence. Some examples of types that would be equivalent are:

- []int and []int
- struct{ x, y, z} and struct{ x, y, z}

Now say that x = []string and y = []string. Although x and y are similar in structure, they will not be considered equivalent due to their naming.

Type Inference

Type inference refers to the automatic deduction of a data type of an expression at compile time. In Go, type inference has a limitation differentiating it from other languages. When declaring a variable without specifying an explicit type, the type of the variable is automatically inferred. Variables are declared without given an explicit type by using either the (:=) or (var =) notation. In Go, inferring the type occurs from left to right. In other words, the type on the left of the notation is inferred by the type on the right. The example below demonstrates this [4.4].

```
var i int;
k := i;
```

Clearly k is not explicitly given a type, but at compile time, the compiler can infer the type of k from the type of i. In this example, k is inferred to be of type int. Type inference is limited in Go in that it only happens in variable declaration, and that the type on the right hand side is inferred to be the type of the variable on the left hand side of the :=. This also means that type information can only flow from the right hand side to the left, not vice versa. This differs from other languages with type inference which allow type information to flow freely in all directions and across a project.

References

Q1

[1.1] "The Go Programming Language - Frequently Asked Questions," [Online]. Available: https://golang.org/doc/faq#Origins.

[1.2] A. Gerrand, "The Go Programming Language - Go version 1 is released," 28 March 2012. [Online]. Available: https://blog.golang.org/go-version-1-is-released.

[1.3] "The Go Programming Language - About the go command," [Online]. Available: https://golang.org/doc/articles/go_command.html.

[1.4] "The Go Programming Language - Go 1.5 Release Notes," [Online]. Available: https://golang.org/doc/go1.5.

[1.5] "Go at Google: Language Design in the Service of Software Engineering", Talks.golang.org, 2016. [Online]. Available: https://talks.golang.org/2012/splash.article#TOC_13. [Accessed: 05- Oct- 2016].

[1.6] "Frequently Asked Questions (FAQ) - The Go Programming Language", *Golang.org*, 2016. [Online]. Available: https://golang.org/doc/faq#Origins.

[1.3] "Bell Labs and CSP Threads", *Swtch.com*, 2016. [Online]. Available: https://swtch.com/~rsc/thread/.

[1.7] B. Kernighan and A. Donovan. *The Go Programming Language*. Boston, MA: Addison-Wesley Professional, 2015.

[1.8] R. Griesemer, "Go for C programmers," [Online]. Available: https://talks.golang.org/2012/goforc.slide#1.

[1.9] Rob, "Less is exponentially more," 2012. [Online]. Available: https://commandcenter.blogspot.ca/2012/06/less-is-exponentially-more.html.

Q2

[2.1] "The Go Programming Language Specification". *Golang.org*. 2016. Web. https://golang.org/ref/spec.

[2.2] J. Walter. "The Go Programming Language, Or: Why All C-Like Languages Except One Suck.". *Syntax-K*. 2016. Web. https://www.syntax-k.de/projekte/go-review.

[2.3] "Frequently Asked Questions (FAQ)". *Golang.org*. 2016. Web. https://golang.org/doc/faq.

Q3

[3.1] C. McGee. "Origin of the Go toolchain". *Blogspot.ca.* 2014. Web. http://adventgo.blogspot.ca/2014/05/origin-of-go-toolchain.html

[3.2] D. Cheney. "How does the go build command work?" 2013. Web. http://dave.cheney.net/tag/toolchain

[3.3] M. Summerfield. "Getting Going with Go" 2012. Web. http://www.drdobbs.com/open-source/getting-going-with-go/240004971?pgno=2

[3.4] A. Donovan, B. Kernighan. "The Go Programming Language". *Gopl.io*. 2016. Web. http://gopl.io

[3.5] R. Cox. "Go 1.3+ Compiler Overhaul". *Golang.org*. 2013. Web. Available: golang.org/s/go13compiler

[3.6] R. Pike. "Go in Go". *Golang.org*. 2015. Web. Available: https://talks.golang.org/2015/gogo.slide#3

[3.7] —. "Go in Go". *Golang.org*. 2015. Web. Available: https://talks.golang.org/2015/gogo.slide#10

[3.8] A. Gerrand. "Introducing the Go Playground". *Golang.org*. 2010. Web. Available: https://blog.golang.org/introducing-go-playground

[3.9] —. "Inside the Go Playground". *Golang.org*. 2013. Web. Available: https://blog.golang.org/playground

Q4

[4.1] "A Tour of Go, Basic types". *Golang.org*. 2016. Web. https://tour.golang.org/basics/11

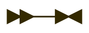
[4.2] "C Basic Datatypes". *Tutorialspoint.com.* 2014 .Web http://www.tutorialspoint.com/ansi_c/c_basic_datatypes.htm

[4.3] "Go's Declaration Syntax". *Golang.org*. 2010. Web.https://blog.golang.org/gos-declaration-syntax

[4.4] "A Tour of Go, Type Inference". *Golang.org.* 2016. [Online]. Available: https://tour.golang.org/basics/14

[4.5] “Types, Why Types?” . lmu.edu. 2016. Web http://cs.lmu.edu/~ray/notes/types/

newline:

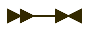


```
newline  ::=
```

referenced by:

- raw_string_lit

unicode_char:

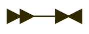


```
unicode_char
         ::=
```

referenced by:

- raw_string_lit
- unicode_value

unicode_letter:

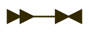


```
unicode_letter
         ::=
```

referenced by:

- letter

unicode_digit:

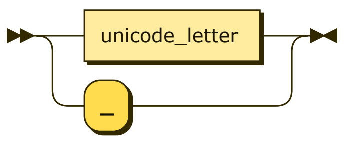


```
unicode_digit
         ::=
```

referenced by:

- identifier

letter:

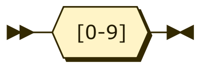


```
letter   ::= unicode_letter
           | '_'
```

referenced by:

- identifier

decimal_digit:

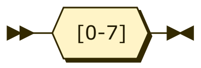


```
decimal_digit
         ::= [0-9]
```

referenced by:

- decimal_lit
- decimals

octal_digit:

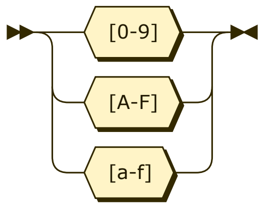


```
octal_digit
         ::= [0-7]
```

referenced by:

- octal_byte_value
- octal_lit

hex_digit:

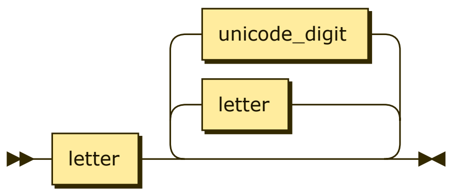


```
hex_digit
         ::= [0-9A-Fa-f]
```

referenced by:

- big_u_value
- hex_byte_value
- hex_lit
- little_u_value

identifier:

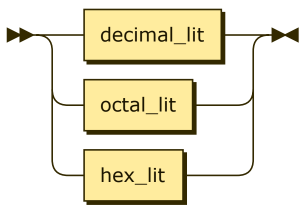


```
identifier
         ::= letter ( letter | unicode_digit )*
```

referenced by:

- FieldName
- FunctionName
- IdentifierList
- Label
- MethodName
- OperandName
- PackageName
- QualifiedIdent
- Selector
- TypeName
- TypeSpec
- TypeSwitchGuard

int_lit:

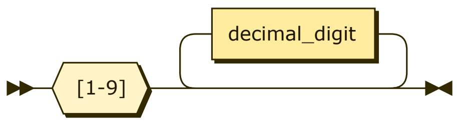


```
int_lit  ::= decimal_lit
           | octal_lit
           | hex_lit
```

referenced by:

- BasicLit

decimal_lit:

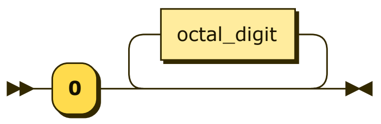


```
decimal_lit
         ::= [1-9] decimal_digit*
```

referenced by:

- int_lit

octal_lit:

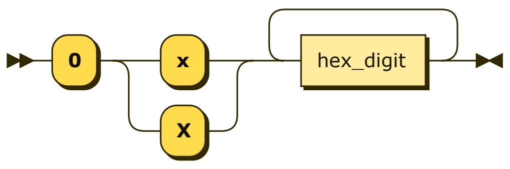


```
octal_lit
         ::= '0' octal_digit*
```

referenced by:

- int_lit

hex_lit:

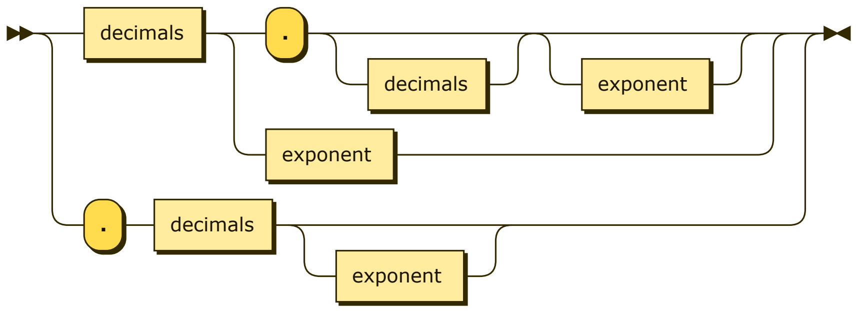


```
hex_lit  ::= '0' ( 'x' | 'X' ) hex_digit+
```

referenced by:

- int_lit

float_lit:

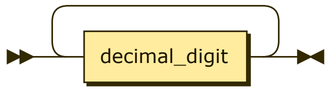


```
float_lit
         ::= decimals ( '.' decimals? exponent? | exponent )
           | '.' decimals exponent?
```

referenced by:

- BasicLit
- imaginary_lit

decimals:

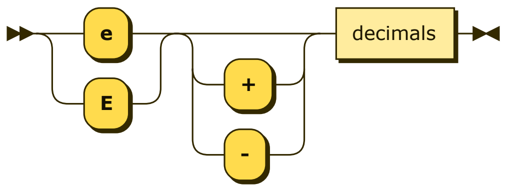


```
decimals ::= decimal_digit+
```

referenced by:

- exponent
- float_lit
- imaginary_lit

exponent:

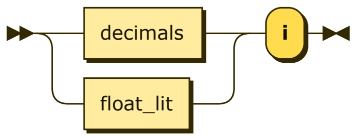


```
exponent ::= ( 'e' | 'E' ) ( '+' | '-' )? decimals
```

referenced by:

- float_lit

imaginary_lit:

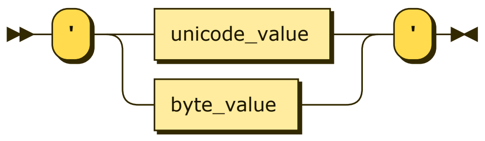


```
imaginary_lit
         ::= ( decimals | float_lit ) 'i'
```

referenced by:

- BasicLit

rune_lit:

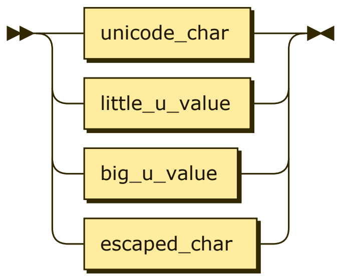


```
rune_lit ::= "'" ( unicode_value | byte_value ) "'"
```

referenced by:

- BasicLit

unicode_value:

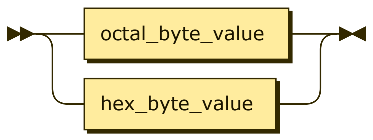


```
unicode_value
         ::= unicode_char
           | little_u_value
           | big_u_value
           | escaped_char
```

referenced by:

- interpreted_string_lit
- rune_lit

byte_value:

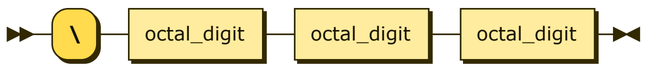


```
byte_value
         ::= octal_byte_value
           | hex_byte_value
```

referenced by:

- interpreted_string_lit
- rune_lit

octal_byte_value:

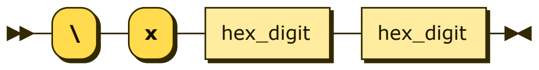


```
octal_byte_value
         ::= '\' octal_digit octal_digit octal_digit
```

referenced by:

- byte_value

hex_byte_value:

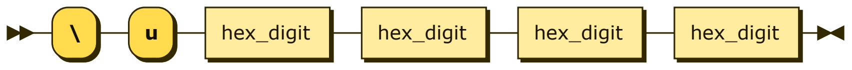


```
hex_byte_value
         ::= '\' 'x' hex_digit hex_digit
```

referenced by:

- byte_value

little_u_value:

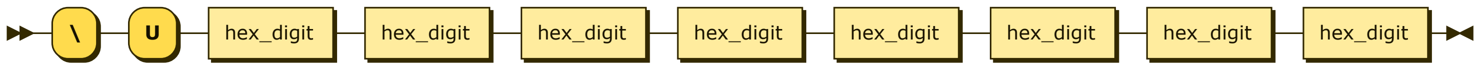


```
little_u_value
         ::= '\' 'u' hex_digit hex_digit hex_digit hex_digit
```

referenced by:

- unicode_value

big_u_value:

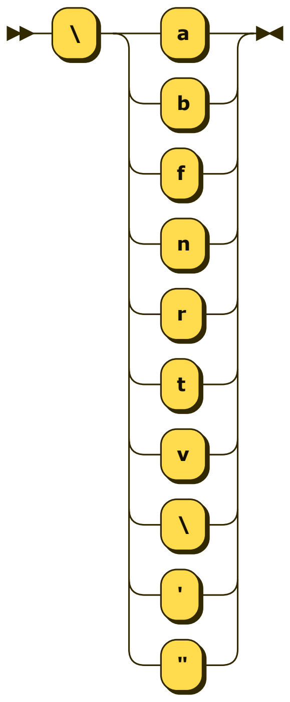


```
big_u_value
         ::= '\' 'U' hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit
```

referenced by:

- unicode_value

escaped_char:

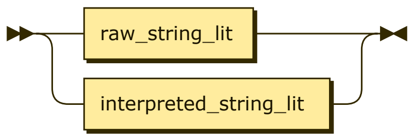


```
escaped_char
         ::= '\' ( 'a' | 'b' | 'f' | 'n' | 'r' | 't' | 'v' | '\' | "'" | '"' )
```

referenced by:

- unicode_value

string_lit:

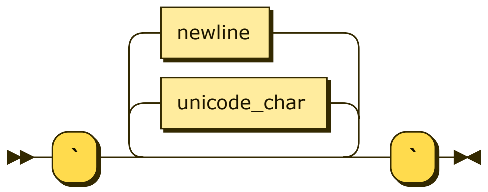


```
string_lit
         ::= raw_string_lit
           | interpreted_string_lit
```

referenced by:

- BasicLit
- ImportPath
- Tag

raw_string_lit:

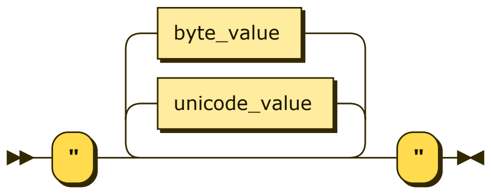


```
raw_string_lit
         ::= '`' ( unicode_char | newline )* '`'
```

referenced by:

- string_lit

interpreted_string_lit:

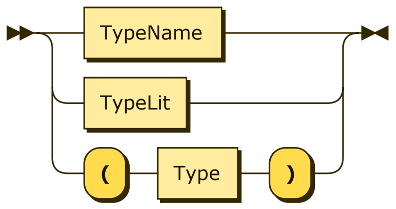


```
interpreted_string_lit
         ::= '"' ( unicode_value | byte_value )* '"'
```

referenced by:

- string_lit

Type:

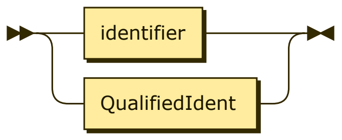


```
Type     ::= TypeName
           | TypeLit
           | '(' Type ')'
```

referenced by:

- Arguments
- BaseType
- ConstSpec
- Conversion
- ElementType
- FieldDecl
- KeyType
- ParameterDecl
- Result
- Type
- TypeAssertion
- TypeList
- TypeSpec
- VarSpec

TypeName:

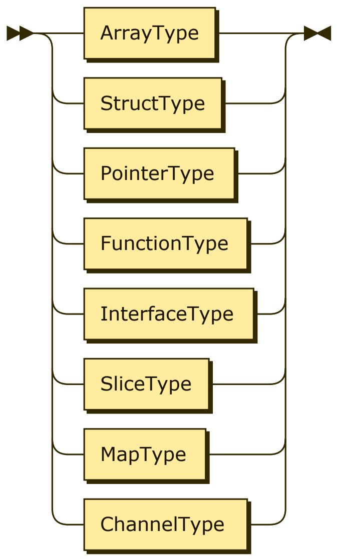


```
TypeName ::= identifier
           | QualifiedIdent
```

referenced by:

- AnonymousField
- InterfaceTypeName
- LiteralType
- ReceiverType
- Type

TypeLit:

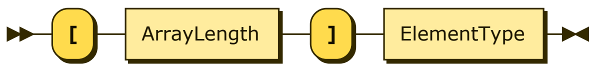


```
TypeLit  ::= ArrayType
           | StructType
           | PointerType
           | FunctionType
           | InterfaceType
           | SliceType
           | MapType
           | ChannelType
```

referenced by:

- Type

ArrayType:



```
ArrayType
         ::= '[' ArrayLength ']' ElementType
```

referenced by:

- LiteralType
- TypeLit

ArrayLength:

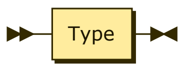
Expression

```
ArrayLength
         ::= Expression
```

referenced by:

- ArrayType

ElementType:

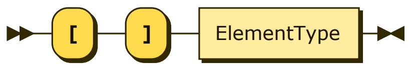


```
ElementType
         ::= Type
```

referenced by:

- ArrayType
- ChannelType
- LiteralType
- MapType
- SliceType

SliceType:

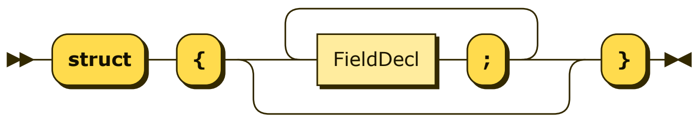


```
SliceType
         ::= '[' ']' ElementType
```

referenced by:

- LiteralType
- TypeLit

StructType:

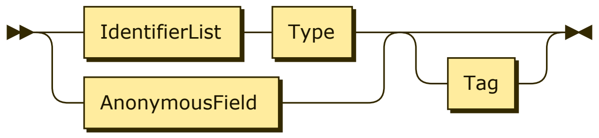


```
StructType
         ::= 'struct' '{' ( FieldDecl ';' )* '}'
```

referenced by:

- LiteralType
- TypeLit

FieldDecl:

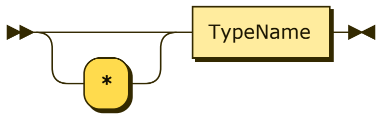


```
FieldDecl
         ::= ( IdentifierList Type | AnonymousField ) Tag?
```

referenced by:

- StructType

AnonymousField:



```
AnonymousField
         ::= '*'? TypeName
```

referenced by:

- FieldDecl

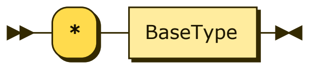
Tag:

```
Tag      ::= string_lit
```

referenced by:

- FieldDecl

PointerType:

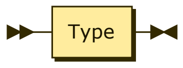


```
PointerType
         ::= '*' BaseType
```

referenced by:

- TypeLit

BaseType:

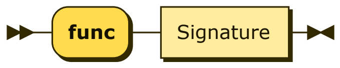


```
BaseType ::= Type
```

referenced by:

- PointerType

FunctionType:

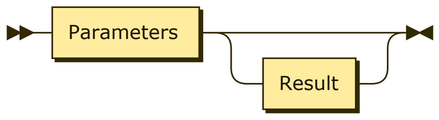


```
FunctionType
         ::= 'func' Signature
```

referenced by:

- TypeLit

Signature:

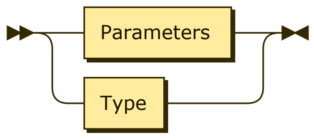


```
Signature
         ::= Parameters Result?
```

referenced by:

- Function
- FunctionDecl
- FunctionType
- MethodDecl
- MethodSpec

Result:

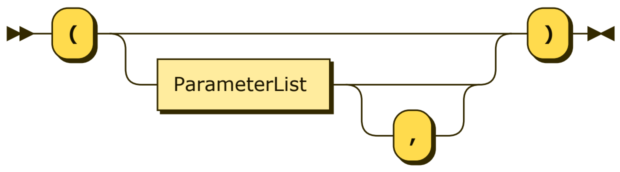


```
Result   ::= Parameters
           | Type
```

referenced by:

- Signature

Parameters:



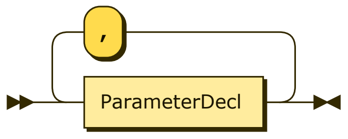
```
Parameters
         ::= '(' ( ParameterList ','? )? ')'
```

referenced by:

- Receiver
- Result

- Signature

ParameterList:

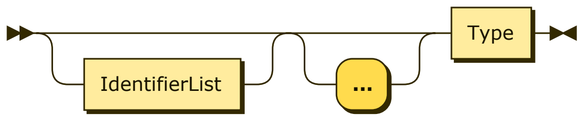


```
ParameterList
         ::= ParameterDecl ( ',' ParameterDecl )*
```

referenced by:

- Parameters

ParameterDecl:

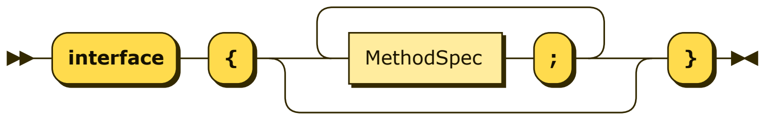


```
ParameterDecl
         ::= IdentifierList? '...'? Type
```

referenced by:

- ParameterList

InterfaceType:

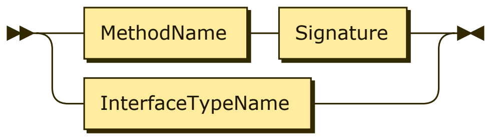


```
InterfaceType
         ::= 'interface' '{' ( MethodSpec ';' )* '}'
```

referenced by:

- TypeLit

MethodSpec:

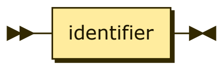


```
MethodSpec
         ::= MethodName Signature
           | InterfaceTypeName
```

referenced by:

- InterfaceType

MethodName:

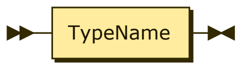


```
MethodName
         ::= identifier
```

referenced by:

- MethodDecl
- MethodExpr
- MethodSpec

InterfaceTypeName:

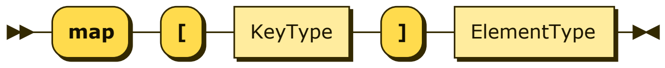


```
InterfaceTypeName
         ::= TypeName
```

referenced by:

- MethodSpec

MapType:

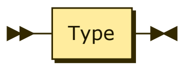


```
MapType  ::= 'map' '[' KeyType ']' ElementType
```

referenced by:

- LiteralType
- TypeLit

KeyType:

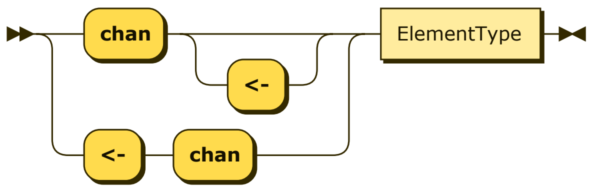


```
KeyType  ::= Type
```

referenced by:

- MapType

ChannelType:

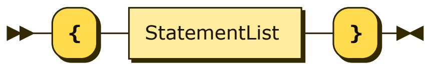


```
ChannelType
         ::= ( 'chan' '<-'? | '<-' 'chan' ) ElementType
```

referenced by:

- TypeLit

Block:

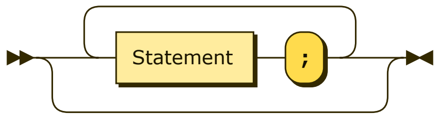


```
Block    ::= '{' StatementList '}'
```

referenced by:

- ForStmt
- FunctionBody
- IfStmt
- Statement

StatementList:

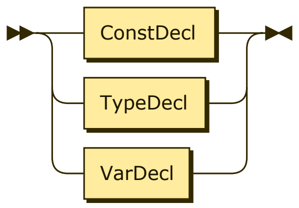


```
StatementList
         ::= ( Statement ';' )*
```

referenced by:

- Block
- CommClause
- ExprCaseClause
- TypeCaseClause

Declaration:

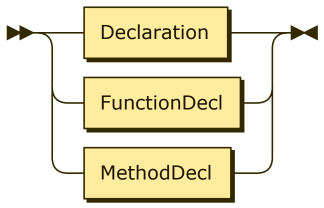


```
Declaration
         ::= ConstDecl
           | TypeDecl
           | VarDecl
```

referenced by:

- Statement
- TopLevelDecl

TopLevelDecl:

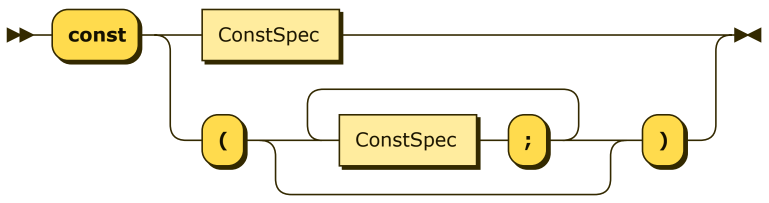


```
TopLevelDecl
         ::= Declaration
           | FunctionDecl
           | MethodDecl
```

referenced by:

- SourceFile

ConstDecl:

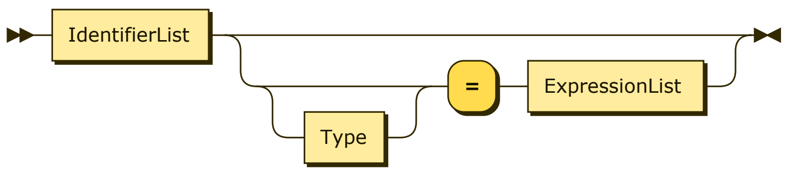


```
ConstDecl
         ::= 'const' ( ConstSpec | '(' ( ConstSpec ';' )* ')' )
```

referenced by:

- Declaration

ConstSpec:

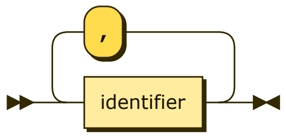


```
ConstSpec
         ::= IdentifierList ( Type? '=' ExpressionList )?
```

referenced by:

- ConstDecl

IdentifierList:

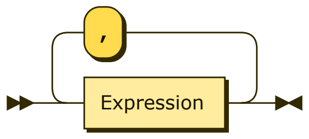


```
IdentifierList
         ::= identifier ( ',' identifier )*
```

referenced by:

- ConstSpec
- FieldDecl
- ParameterDecl
- RangeClause
- RecvStmt
- ShortVarDecl
- VarSpec

ExpressionList:

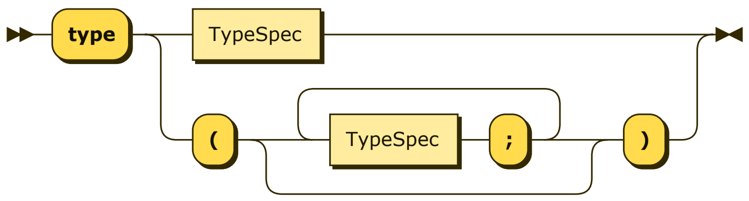


```
ExpressionList
         ::= Expression ( ',' Expression )*
```

referenced by:

- Arguments
- Assignment
- ConstSpec
- ExprSwitchCase
- RangeClause
- RecvStmt
- ReturnStmt
- ShortVarDecl
- VarSpec

TypeDecl:

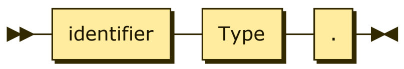


```
TypeDecl ::= 'type' ( TypeSpec | '(' ( TypeSpec ';' )* ')' )
```

referenced by:

- Declaration

TypeSpec:

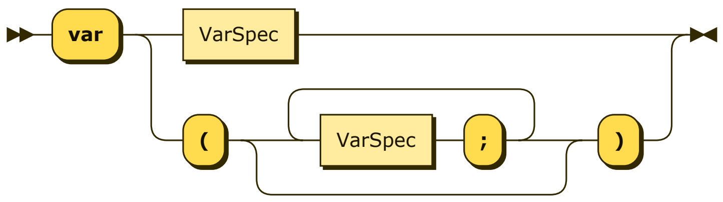


```
TypeSpec ::= identifier Type .
```

referenced by:

- TypeDecl

VarDecl:

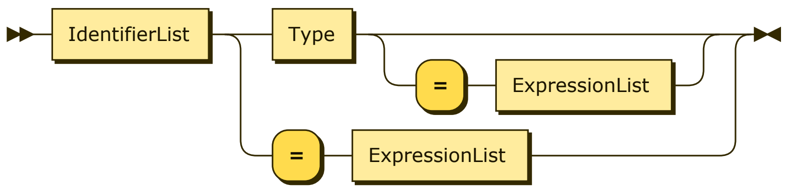


```
VarDecl  ::= 'var' ( VarSpec | '(' ( VarSpec ';' )* ')' )
```

referenced by:

- Declaration

VarSpec:

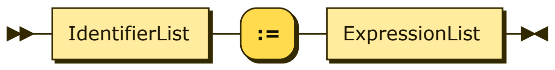


```
VarSpec  ::= IdentifierList ( Type ( '=' ExpressionList )? | '=' ExpressionList )
```

referenced by:

- VarDecl

ShortVarDecl:

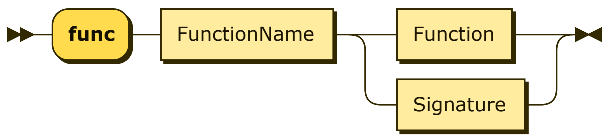


```
ShortVarDecl
         ::= IdentifierList ':=' ExpressionList
```

referenced by:

- SimpleStmt

FunctionDecl:



```
FunctionDecl
         ::= 'func' FunctionName ( Function | Signature )
```

referenced by:

- TopLevelDecl

FunctionName:

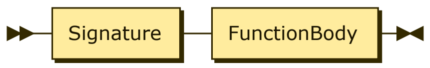
identifier

```
FunctionName
         ::= identifier
```

referenced by:

- FunctionDecl

Function:

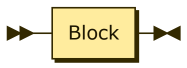


```
Function ::= Signature FunctionBody
```

referenced by:

- FunctionDecl
- FunctionLit
- MethodDecl

FunctionBody:

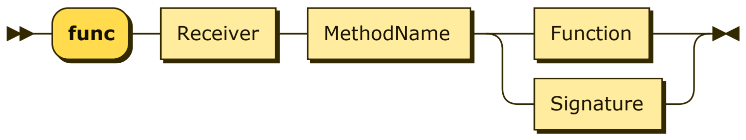


```
FunctionBody
         ::= Block
```

referenced by:

- Function

MethodDecl:

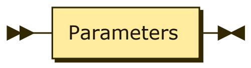


```
MethodDecl
         ::= 'func' Receiver MethodName ( Function | Signature )
```

referenced by:

- TopLevelDecl

Receiver:

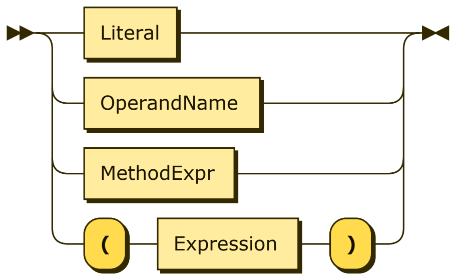


```
Receiver ::= Parameters
```

referenced by:

- MethodDecl

Operand:

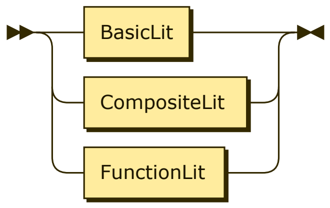


```
Operand  ::= Literal
           | OperandName
           | MethodExpr
           | '(' Expression ')'
```

referenced by:

- PrimaryExpr

Literal:

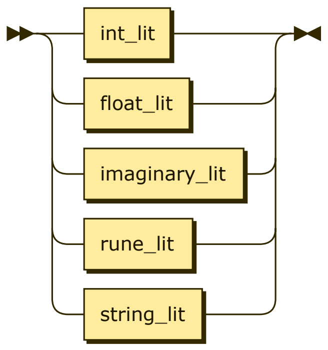


```
Literal  ::= BasicLit
           | CompositeLit
           | FunctionLit
```

referenced by:

- Operand

BasicLit:

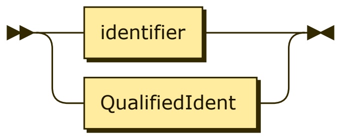


```
BasicLit ::= int_lit
           | float_lit
           | imaginary_lit
           | rune_lit
           | string_lit
```

referenced by:

- Literal

OperandName:

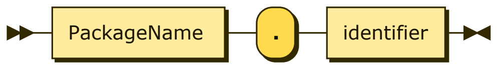


```
OperandName
         ::= identifier
           | QualifiedIdent
```

referenced by:

- Operand

QualifiedIdent:

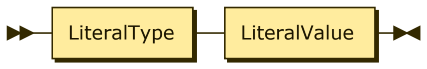


```
QualifiedIdent
         ::= PackageName '.' identifier
```

referenced by:

- OperandName
- TypeName

CompositeLit:

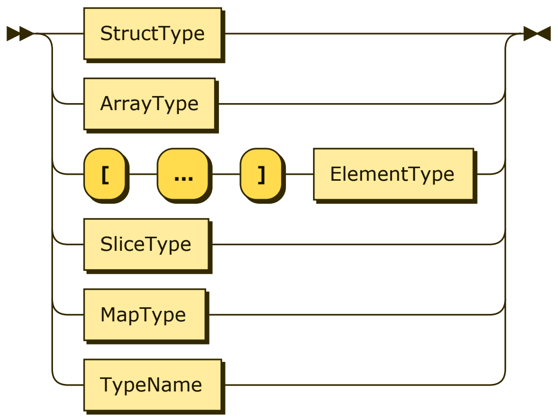


```
CompositeLit
         ::= LiteralType LiteralValue
```

referenced by:

- Literal

LiteralType:

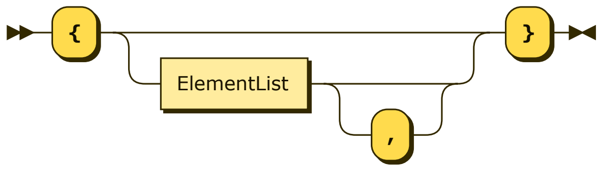


```
LiteralType
         ::= StructType
           | ArrayType
           | '[' '...' ']' ElementType
           | SliceType
           | MapType
           | TypeName
```

referenced by:

- CompositeLit

LiteralValue:

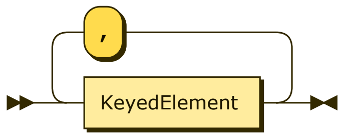


```
LiteralValue
         ::= '{' ( ElementList ','? )? '}'
```

referenced by:

- CompositeLit
- Element
- Key

ElementList:

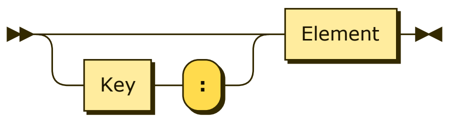


```
ElementList
         ::= KeyedElement ( ',' KeyedElement )*
```

referenced by:

- LiteralValue

KeyedElement:

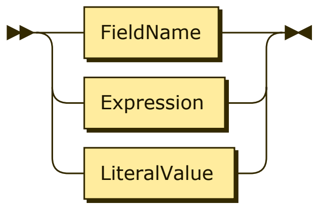


```
KeyedElement
         ::= ( Key ':' )? Element
```

referenced by:

- ElementList

Key:

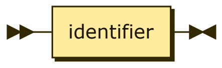


```
Key      ::= FieldName
           | Expression
           | LiteralValue
```

referenced by:

- KeyedElement

FieldName:

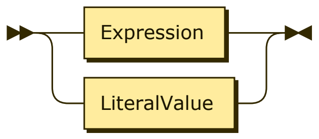


```
FieldName
         ::= identifier
```

referenced by:

- Key

Element:

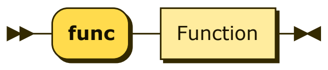


```
Element  ::= Expression
           | LiteralValue
```

referenced by:

- KeyedElement

FunctionLit:

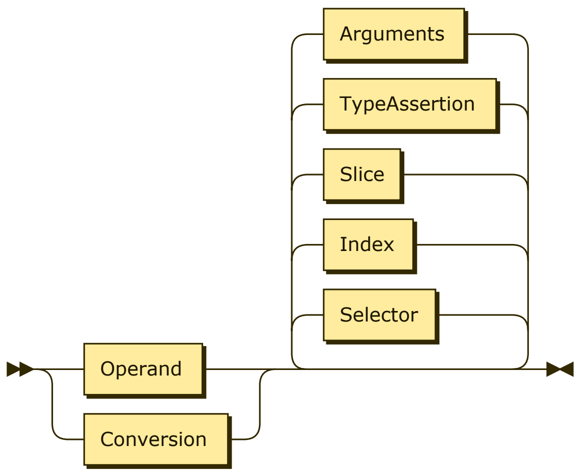


```
FunctionLit
         ::= 'func' Function
```

referenced by:

- Literal

PrimaryExpr:

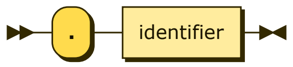


```
PrimaryExpr
         ::= ( Operand | Conversion ) ( Selector | Index | Slice | TypeAssertion | Arguments )*
```

referenced by:

- Expression
- TypeSwitchGuard

Selector:

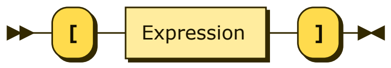


```
Selector ::= '.' identifier
```

referenced by:

- PrimaryExpr

Index:

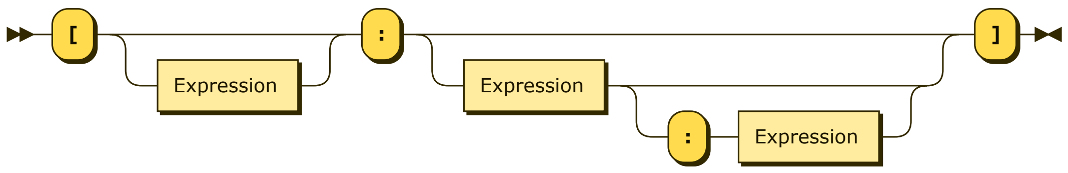


```
Index    ::= '[' Expression ']'
```

referenced by:

- PrimaryExpr

Slice:

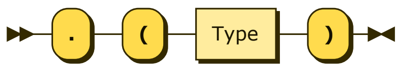


```
Slice    ::= '[' Expression? ':' ( Expression ( ':' Expression )? )? ']'
```

referenced by:

- PrimaryExpr

TypeAssertion:

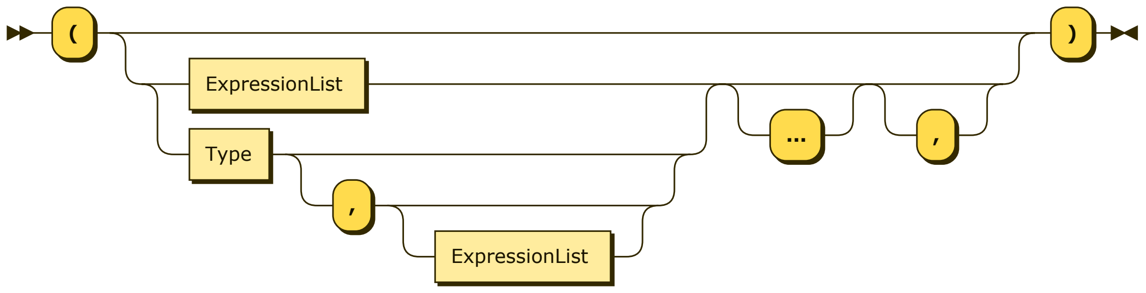


```
TypeAssertion
         ::= '.' '(' Type ')'
```

referenced by:

- PrimaryExpr

Arguments:

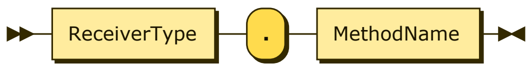


```
Arguments
         ::= '(' ( ( ExpressionList | Type ( ',' ExpressionList? )? ) '...'? ','? )? ')'
```

referenced by:

- PrimaryExpr

MethodExpr:

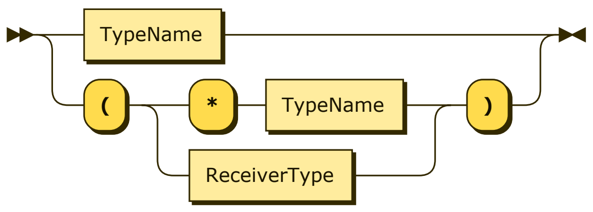


```
MethodExpr
         ::= ReceiverType '.' MethodName
```

referenced by:

- Operand

ReceiverType:

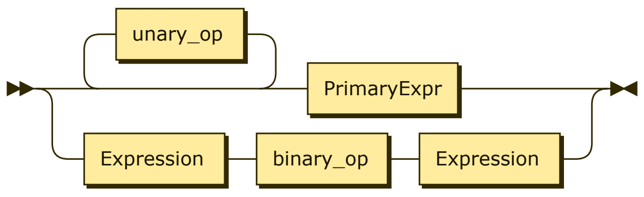


```
ReceiverType
         ::= TypeName
           | '(' ( '*' TypeName | ReceiverType ) ')'
```

referenced by:

- MethodExpr
- ReceiverType

Expression:

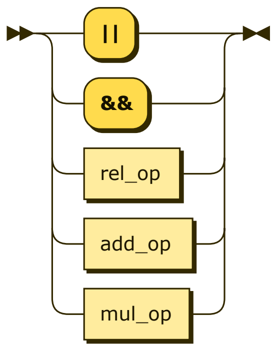


```
Expression
         ::= unary_op* PrimaryExpr
           | Expression binary_op Expression
```

referenced by:

- ArrayLength
- Channel
- Condition
- Conversion
- DeferStmt
- Element
- ExprSwitchStmt
- Expression
- ExpressionList
- ExpressionStmt
- GoStmt
- IfStmt
- IncDecStmt
- Index
- Key
- Operand
- RangeClause
- RecvExpr
- SendStmt
- Slice

binary_op:

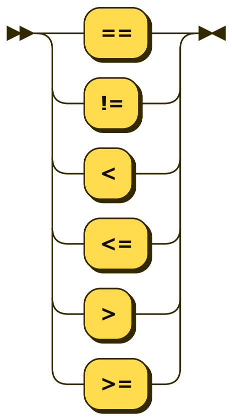


```
binary_op
         ::= '||'
           | '&&'
           | rel_op
           | add_op
           | mul_op
```

referenced by:

- Expression

rel_op:

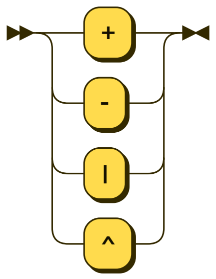


```
rel_op   ::= '=='
           | '!='
           | '<'
           | '<='
           | '>'
           | '>='
```

referenced by:

- binary_op

add_op:

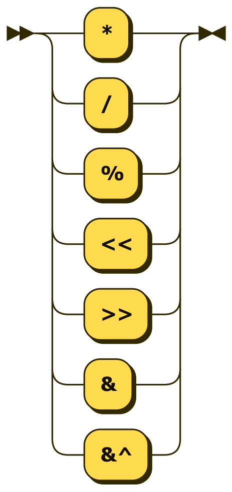


```
add_op   ::= '+'
           | '-'
           | '|'
           | '^'
```

referenced by:

- assign_op
- binary_op

mul_op:

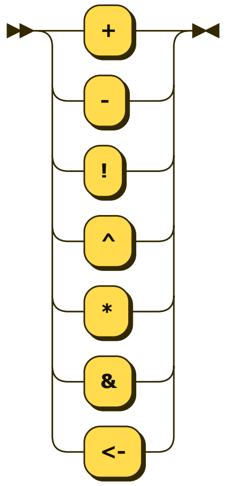


```
mul_op   ::= '*'
           | '/'
           | '%'
           | '<<'
           | '>>'
           | '&'
           | '&^'
```

referenced by:

- assign_op
- binary_op

unary_op:

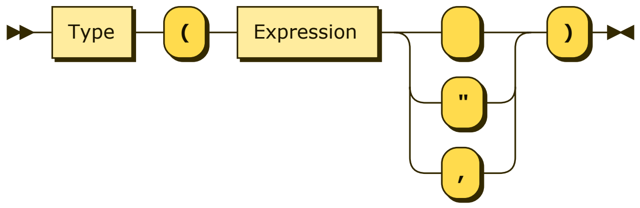


```
unary_op ::= '+'
           | '-'
           | '!'
           | '^'
           | '*'
           | '&'
           | '<-'
```

referenced by:

- Expression

Conversion:

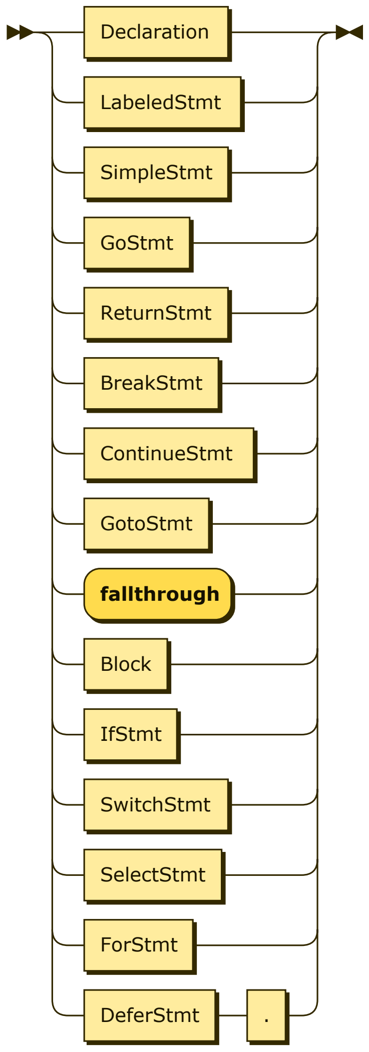


```
Conversion
         ::= Type '(' Expression [ ",] ')'
```

referenced by:

- PrimaryExpr

Statement:

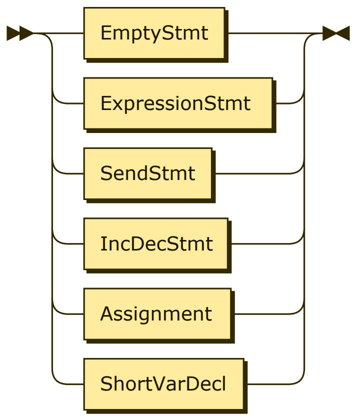


```
Statement
         ::= Declaration
           | LabeledStmt
           | SimpleStmt
           | GoStmt
           | ReturnStmt
           | BreakStmt
           | ContinueStmt
           | GotoStmt
           | 'fallthrough'
           | Block
           | IfStmt
           | SwitchStmt
           | SelectStmt
           | ForStmt
           | DeferStmt .
```

referenced by:

- LabeledStmt
- StatementList

SimpleStmt:

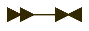


```
SimpleStmt
         ::= EmptyStmt
           | ExpressionStmt
           | SendStmt
           | IncDecStmt
           | Assignment
           | ShortVarDecl
```

referenced by:

- ExprSwitchStmt
- IfStmt
- InitStmt
- PostStmt
- Statement
- TypeSwitchStmt

EmptyStmt:

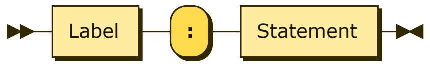


```
EmptyStmt
         ::=
```

referenced by:

- SimpleStmt

LabeledStmt:



```
LabeledStmt
         ::= Label ':' Statement
```

referenced by:

- Statement

Label:

identifier

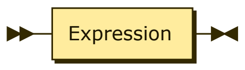
```
Label    ::= identifier
```

referenced by:

- BreakStmt
- ContinueStmt
- GotoStmt

- LabeledStmt

ExpressionStmt:

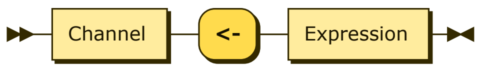


```
ExpressionStmt
         ::= Expression
```

referenced by:

- SimpleStmt

SendStmt:

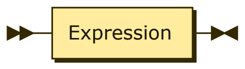


```
SendStmt ::= Channel '<-' Expression
```

referenced by:

- CommCase
- SimpleStmt

Channel:

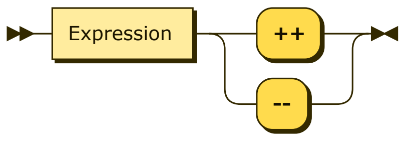


```
Channel  ::= Expression
```

referenced by:

- SendStmt

IncDecStmt:

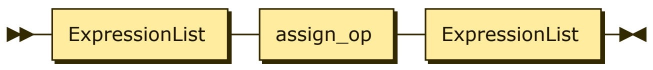


```
IncDecStmt
         ::= Expression ( '++' | '--' )
```

referenced by:

- SimpleStmt

Assignment:

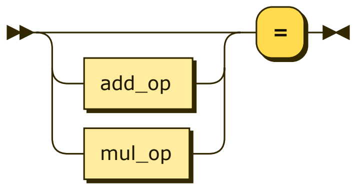


```
Assignment
         ::= ExpressionList assign_op ExpressionList
```

referenced by:

- SimpleStmt

assign_op:

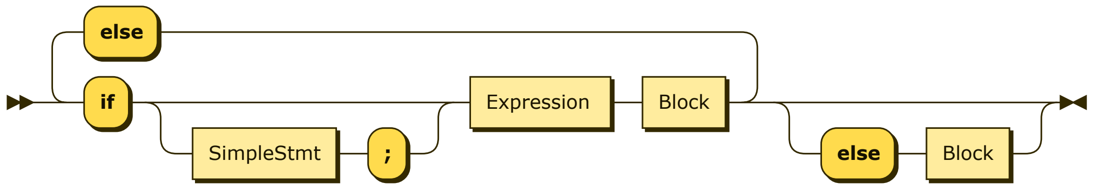


```
assign_op
         ::= ( add_op | mul_op )? '='
```

referenced by:

- Assignment

IfStmt:

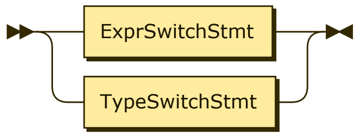


```
IfStmt   ::= 'if' ( SimpleStmt ';' )? Expression Block ( 'else' 'if' ( SimpleStmt ';' )? Expression Block )* ( 'else' Block )?
```

referenced by:

- Statement

SwitchStmt:

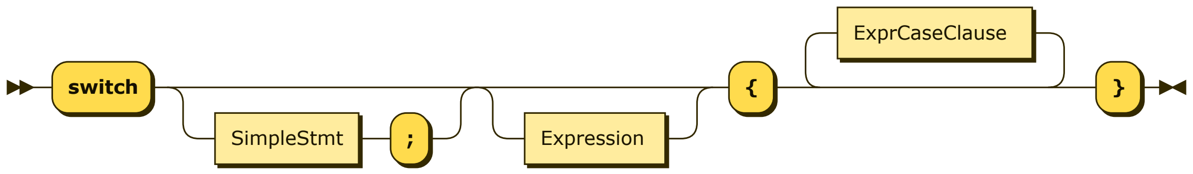


```
SwitchStmt
         ::= ExprSwitchStmt
           | TypeSwitchStmt
```

referenced by:

- Statement

ExprSwitchStmt:

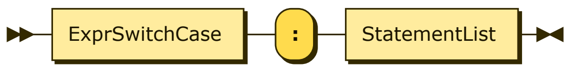


```
ExprSwitchStmt
         ::= 'switch' ( SimpleStmt ';' )? Expression? '{' ExprCaseClause* '}'
```

referenced by:

- SwitchStmt

ExprCaseClause:

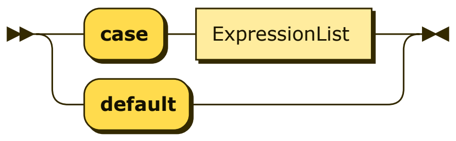


```
ExprCaseClause
         ::= ExprSwitchCase ':' StatementList
```

referenced by:

- ExprSwitchStmt

ExprSwitchCase:

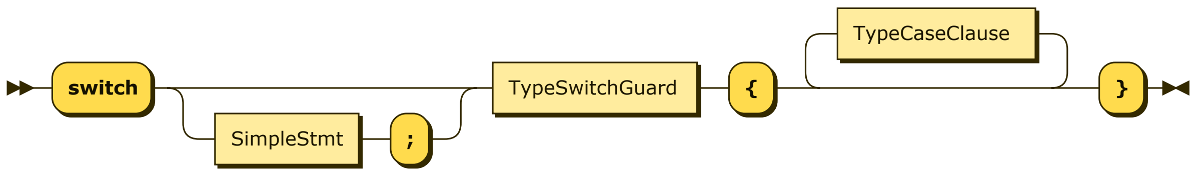


```
ExprSwitchCase
         ::= 'case' ExpressionList
           | 'default'
```

referenced by:

- ExprCaseClause

TypeSwitchStmt:

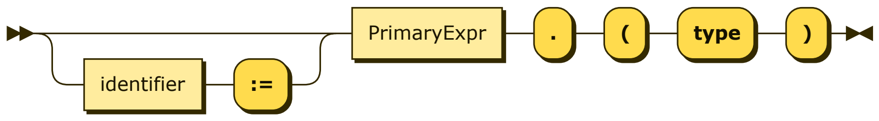


```
TypeSwitchStmt
         ::= 'switch' ( SimpleStmt ';' )? TypeSwitchGuard '{' TypeCaseClause* '}'
```

referenced by:

- SwitchStmt

TypeSwitchGuard:

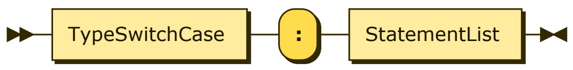


```
TypeSwitchGuard
         ::= ( identifier ':=' )? PrimaryExpr '.' '(' 'type' ')'
```

referenced by:

- TypeSwitchStmt

TypeCaseClause:

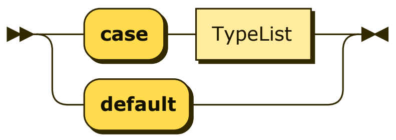


```
TypeCaseClause
         ::= TypeSwitchCase ':' StatementList
```

referenced by:

- TypeSwitchStmt

TypeSwitchCase:

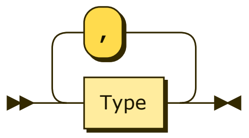


```
TypeSwitchCase
         ::= 'case' TypeList
           | 'default'
```

referenced by:

- TypeCaseClause

TypeList:

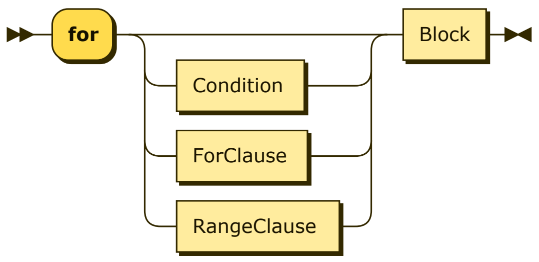


```
TypeList ::= Type ( ',' Type )*
```

referenced by:

- TypeSwitchCase

ForStmt:

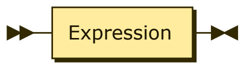


```
ForStmt  ::= 'for' ( Condition | ForClause | RangeClause )? Block
```

referenced by:

- Statement

Condition:

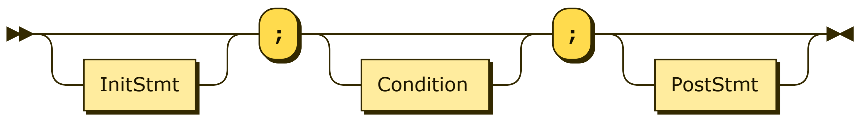


```
Condition
         ::= Expression
```

referenced by:

- ForClause
- ForStmt

ForClause:

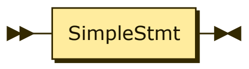


```
ForClause
         ::= InitStmt? ';' Condition? ';' PostStmt?
```

referenced by:

- ForStmt

InitStmt:

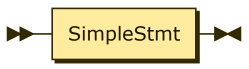


```
InitStmt ::= SimpleStmt
```

referenced by:

- ForClause

PostStmt:

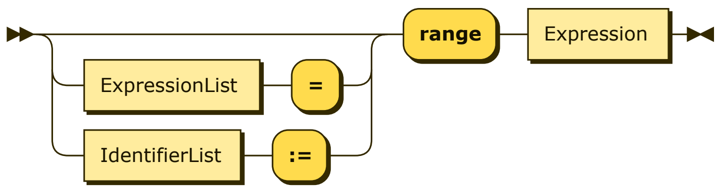


```
PostStmt ::= SimpleStmt
```

referenced by:

- ForClause

RangeClause:

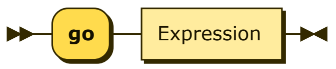


```
RangeClause
         ::= ( ExpressionList '=' | IdentifierList ':=' )? 'range' Expression
```

referenced by:

- ForStmt

GoStmt:

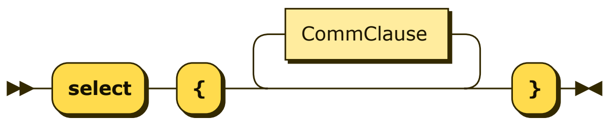


```
GoStmt   ::= 'go' Expression
```

referenced by:

- Statement

SelectStmt:

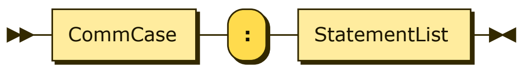


```
SelectStmt
         ::= 'select' '{' CommClause* '}'
```

referenced by:

- Statement

CommClause:

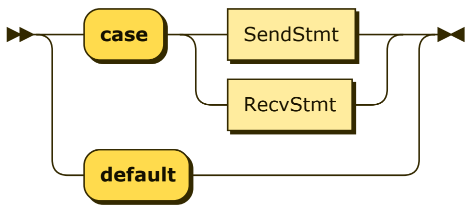


```
CommClause
         ::= CommCase ':' StatementList
```

referenced by:

- SelectStmt

CommCase:

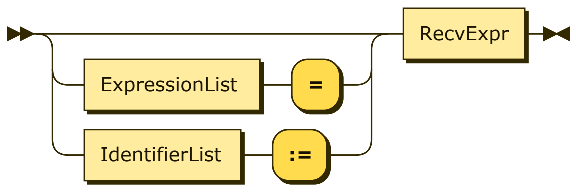


```
CommCase ::= 'case' ( SendStmt | RecvStmt )
           | 'default'
```

referenced by:

- CommClause

RecvStmt:

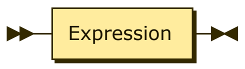


```
RecvStmt ::= ( ExpressionList '=' | IdentifierList ':=' )? RecvExpr
```

referenced by:

- CommCase

RecvExpr:

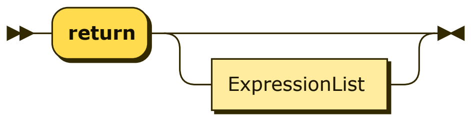


```
RecvExpr ::= Expression
```

referenced by:

- RecvStmt

ReturnStmt:

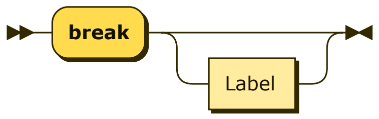


```
ReturnStmt
         ::= 'return' ExpressionList?
```

referenced by:

- Statement

BreakStmt:

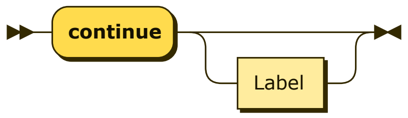


```
BreakStmt
         ::= 'break' Label?
```

referenced by:

- Statement

ContinueStmt:

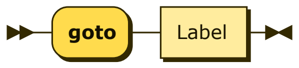


```
ContinueStmt
         ::= 'continue' Label?
```

referenced by:

- Statement

GotoStmt:

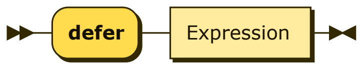


```
GotoStmt ::= 'goto' Label
```

referenced by:

- Statement

DeferStmt:

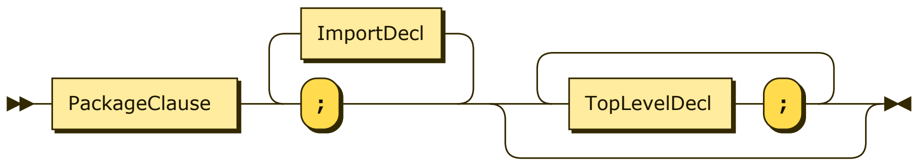


```
DeferStmt
         ::= 'defer' Expression
```

referenced by:

- Statement

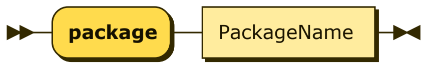
SourceFile:



```
SourceFile
         ::= PackageClause ';' ( ImportDecl ';' )* ( TopLevelDecl ';' )*
```

no references

PackageClause:

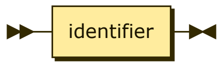


```
PackageClause
         ::= 'package' PackageName
```

referenced by:

- SourceFile

PackageName:

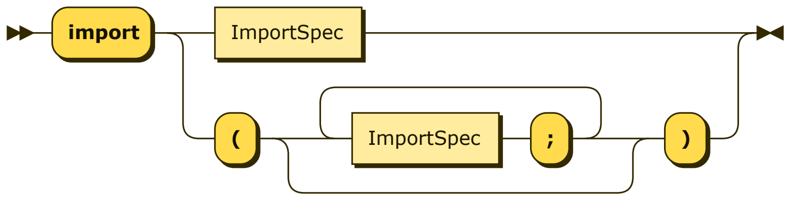


```
PackageName
         ::= identifier
```

referenced by:

- ImportSpec
- PackageClause
- QualifiedIdent

ImportDecl:

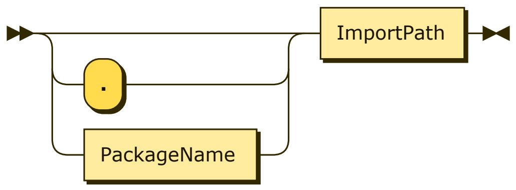


```
ImportDecl
         ::= 'import' ( ImportSpec | '(' ( ImportSpec ';' )* ')' )
```

referenced by:

- SourceFile

ImportSpec:

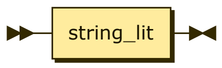


```
ImportSpec
         ::= ( '.' | PackageName )? ImportPath
```

referenced by:

- ImportDecl

ImportPath:



```
ImportPath
         ::= string_lit
```

referenced by:

- ImportSpec

... generated by Railroad Diagram Generator