Language Overview

This page provides an overview of the language you will be working with throughout the quarter. This quarter you will be building an optimizing compiler for a subset of the Go language that we are calling **Golite**, which will have very similar syntax and semantics to Go. However, certain aspects of the syntax and semantics will differ and be more restrictive than normal Go syntax so please make sure you read over the EBNF grammar carefully and fully understand the semantics of the language. Your first task is to make sure you understand the EBNF grammar for Golite.

All programs in this language are command-line applications that are coded in a single file (with any name). The file extension should be `.golite`.

Note

This document may be updated throughout the quarter. Please pay close attention to Ed for announcements about updates to this document.

GoLite Grammar

The following CFG grammar partially describes the Golite syntax. In the EBNF below, non-terminals are highlighted in blue with a `=` instead of an `->` as shown in class. You'll notice a series of semi-colons lined up at the end of each line in the EBNF. Please ignore those semi-colons because they are needed to signal the end of a production. The language does contain semicolons to signal the end of a statement but those are terminals and are indicated as `';'`. The terminals are highlighted in green with single quotes. Don't be frighten by the length of the grammar. We are just being very explicit about everything but the language is relatively small in comparison to the full Go language. Additional notation is provided in the below chart.

Usage	Notation	Meaning
Definition	`=`	The definition of a production
Repetition	`{...}`	Zero or more occurrences of the symbol(s) defined in the braces.
Optional	`[...]`	The symbol(s) defined in subscript brackets are not required to appear.
Grouping	`(...)`	Any terminal symbol from within this group is valid.

Please make sure to post on Ed if you do not understand the notation below. Do not just assume. We will help clarify.

```
Program = Types Declarations Functions 'eof'
Types = {TypeDeclaration}
TypeDeclaration = 'type' 'id' 'struct' '{' Fields '}' ';'
Fields = Decl ';' {Decl ';'}
Decl = 'id' Type
Type = 'int' | 'bool' | '*' 'id'
Declarations = {Declaration}
Declaration = 'var' Ids Type ';'
Ids = 'id' {',' 'id'}
Functions = {Function}
Function = 'func' 'id' Parameters [ReturnType] '{' Declarations Statements '}'
Parameters = '(' [ Decl {',' Decl}] ')'
ReturnType = type
Statements = {Statement}
Statement = Block | Assignment | Print | Delete | Read | Conditional | Loop | Return  | Ir
Read = 'scan' LValue ';'
Block = '{' Statements '}'
Delete = 'delete' Expression ';'
Assignment = LValue '=' Expression ';'
Print = 'printf' '(' 'string' { ',' Expression} ')'  ';'
Conditional = 'if' '(' Expression ')' Block ['else' Block]
Loop = 'for' '(' Expression ')' Block
Return = 'return' [Expression] ';'
Invocation = 'id' Arguments ';'
Arguments = '(' [Expression {',' Expression}] ')'
LValue = 'id' {'.' id}
Expression = BoolTerm {'||' BoolTerm}
BoolTerm = EqualTerm {'&&' EqualTerm}
EqualTerm =  RelationTerm {('==' | '!=') RelationTerm}
RelationTerm = SimpleTerm {('>' | '<' | '<=' | '>=') SimpleTerm}
SimpleTerm = Term {('+' | '-') Term}
Term = UnaryTerm {('*' | '/') UnaryTerm}
UnaryTerm = '!' SelectorTerm | '-' SelectorTerm | SelectorTerm
SelectorTerm = Factor {'.' 'id'}
Factor = '(' Expression ')' | 'id' [Arguments] | 'number' | 'new' 'id' | 'true' | 'false'
```

Lexical Analysis Information

The following rules provides additional information needed for lexical analysis:

- A valid program ends with an end-of-file indicator (EOF).
- An identifier (i.e., `'id'`) token must begin with a letter from the English alphabet. The letter can be capitalized or lowercase. Following the first character, an `'id'` token can contain zero or more alphanumeric characters (i.e., integer digits (0 to 9), lowercase or uppercase English letters).
- A `'number'` token is a positive/negative integer.
- A `'string'` token is a specific token needed for printing. It contains zero or more characters. A string is always embedded within double quotes (`"`).

- For Golite, A token is formed by taking the longest possible sequence of characters. Whitespace (i.e., one or more spaces, tabs, or newlines) may precede or follow any token. For example, `"a=1"` and `"a = 1"` are equivalent. Whitespace does delimit tokens. For example, `"abc"` is one token whereas `"a bc"` is two tokens.
- A comment begins with `//` and consists of all characters up to a new line. We do not have multiline comments in this language.

Language Semantics

The semantics for GoLite are provide informally. Please ask questions on Ed if you need clarification.

- **Scoping**: local declarations and parameters may hide global declarations (and functions), but a local declaration may not redefine a parameter.
- **Identifiers within the same scope can not be redeclared:**
  - Global variables with the same name cannot be defined (compiler produces an error)
  - Local variables of the same function cannot be defined (compiler produces an error)
  - Two `struct` definitions with same name cannot be defined (compiler produces an error)
  - Two fields within a struct cannot be defined (compiler produces an error)
  - Two functions with the same name cannot be defined (compiler produces an error)

- The entry point into a program begins in the a function named `main` that takes no arguments. All valid programs must define a `main` function.
- `struct` type may only include fields of the primitive types and `struct` types defined in the file. You must also allow fields of its own type.
- The language restricts Mutually-Recursive functions (https://en.wikipedia.org/wiki/Mutual_recursion); however, it does allow normal recursion.
- The semantics of `if` and `for` statements is the same as Go semantics. However, they both require a boolean expression and parentheses around the boolean expression.
- Assignment statements require the left-hand side and right-hand side to have same type. However, `nil` can be defined to any `struct` type.
- All `struct` values must be defined with the `new Type` syntax. `Type` must be equal to a struct type and cannot be an `int` or `bool` type. All `struct` values in the language are pointers to their types. You can not use the `&` to make a pointer to a struct in Golite. The `new` syntax evaluates to a reference (i.e., a pointer to a memory address) to store a value of the specific type.
- Unlike in Go where there is garbage collection, all allocated memory in Golite must be freed (i.e., deallocated) by the programmer. A programmer deallocates memory that was allocated with the `new` syntax by calling `delete Expression` , where `Expression` is a reference value (i.e., an pointer to an address). The `delete` function make take in a pointer to a memory address (i.e., it must take in a struct value). Make sure `Expression` evaluates to a a pointer to a struct type.
- The `.` operator is used to access a field in a `struct` value.
- All arguments in the language are passed by value (including pointers to `struct` values).
- GoLite only allows the reading and printing of integer values. `'scan'` reads in an integer from standard input. The `printf `` is used to print out the value of integer along with a static literal string. Strings are not actual values or types in Golite. However, you can define a literal strings in the format string argument to the ``'printf'` function. Use a `"%d"` to represent the placeholder for the integer in the literal string. There must be an equal number of placeholders to the number of variable expression arguments to `"printf"` . For example, `printf("%d=%d\n",4,5);` is correct but `printf("%d%d%d",4);` is incorrect. The only placeholder inside printf can only be `"%d"` and whitespace escape characters are allows (e.g., `"\n"` for newlines).

- All arithmetic and relational operators require integer operands.
- Equality operators require operands of integer or `struct` type. The operands must have the same type. `struct` values are compared by address.
- Boolean operators require boolean operands.
- The language does not require Short-Circuiting (https://en.wikipedia.org/wiki/Short-circuit_evaluation) via boolean expressions.
- Any function with a return type must return a valid value (of its return type) along all control flow paths. Each function with no return type must not return a value.

Back to top