While Language Specification

Source Language Definition

**Language in Extended Backus Naur Form (ANTLR g4 file contents):**

|  |
| --- |
| grammar whileGrammar;  program  :codeblock  ;  codeblock  : statement+  ;  statement  : declaration SC  | assignment SC  | printcall SC  | function  | functioncall SC  | control  | rtrn  ;  declaration  : decs ID  | decs assignment  ;  assignment  :ID ASS exp  |ID ASS conditional  ;  exp  :pexp  |operand  |functioncall  |exp EXPO exp  |exp (DIV | MULT | MODULUS) exp  |exp (ADD | SUB) exp  ;  pexp  :LPARA exp RPARA  ;  operand  : INT  | SUB INT  | FLOAT  | SUB FLOAT  | ID  | BOOL  ;  decs  :DECINT  |DECFLOAT  |DECBOOL  ;  control  : ifst  | whilest  ;  ifst  :IF LPARA conditional RPARA then  |IF LPARA conditional RPARA then elsest  ;  then  :LBR codeblock RBR  ;  elsest  :ELSE LBR codeblock RBR  ;  whilest  :WHILE LPARA conditional RPARA LBR codeblock RBR  ;  conditional  : paraconditional  | ID  | NOT conditional  | conditional ( AND | OR ) conditional  | exp (GTHAN | GTHANEQ | LTHAN | LTHANEQ) exp  | conditional (EQUALITY | INEQUALITY) conditional  | exp (EQUALITY | INEQUALITY) exp  | functioncall  ;  paraconditional  : LPARA conditional RPARA  ;  function  : typefunction  | voidfunction  ;  typefunction  : funcdeclaration LBR codeblock RBR  ;  voidfunction  : '$void' ID LPARA funcarguments RPARA LBR codeblock RBR  ;  funcdeclaration  :declaration LPARA funcarguments RPARA  ;  funcarguments  :(declaration ',') + declaration  |declaration  |  ;  rtrn  : 'Return' (exp | conditional) SC  ;  functioncall  : ID LPARA arguments? RPARA  ;  arguments  : arguments ',' arguments  | conditional  | exp  |  ;  printcall  : 'print' LPARA (exp | conditional | STRING) RPARA  ;  ASS : '=' ;  LBR : '{' ;  RBR : '}' ;  LPARA : '(' ;  RPARA : ')' ;  SC : ';' ;  //Control flows  IF : 'if' ;  ELSE : 'else' ;  WHILE : 'while' ;  DO : 'do' ;  //Operators  ADD : '+' ;  SUB : '-' ;  MULT : '\*' ;  DIV : '/' ;  EXPO : '^' ;  MODULUS : '%' ;  //Conditional operators  LTHAN : '<' ;  GTHAN : '>' ;  LTHANEQ : '<=' ;  GTHANEQ : '>=' ;  EQUALITY : '==' ;  INEQUALITY : '!=' ;  AND : '&&' ;  OR : '||' ;  NOT : '!' ;  //Operands  FLOAT : INT FLTP [0-9]\*;  ID : [a-z] [a-zA-Z0-9\_-]\*;  INT : [0-9]+;  FLTP : '.';  BOOL : 'True' | 'False' ;  STRING : '"' .\*? '"';  //Declarations  DECINT : '$int' ;  DECFLOAT : '$float' ;  DECBOOL : '$boolean' ;  COMMENT : '/^' .\*? '^/' -> skip;  WS : [ \t\r\n]+ -> skip; |

**Terminal value summary:**

|  |  |  |
| --- | --- | --- |
| **Token** | **Value** | **Description (if applicable)** |
| x | Identifier operand | Used to recognise a variable or function name. |
| 1 | Integer operand |  |
| 3.14159 | Float operand |  |
| True | Boolean operand |  |
| “Hello, world” | String value |  |
| ^ | Exponent / power operator | Numeric operator |
| \* / | Multiplication and division operator | Numeric operator |
| + - | Addition and subtraction operator | Numeric operator |
| ! | Not / inversion operator |  |
| && || | “and” and “or” operator |  |
| > and >= and < and <= | Greater than (or equal to) and less than (or equal to) operators |  |
| == and != | Equality and inequality operator |  |
| $int | Integer declaration | Denotes a new declaration for an identifier of type int. |
| $float | Float declaration | Denotes a new declaration for an identifier of type floating point. |
| $boolean | Boolean declaration | Denotes a new declaration for an identifier of type Boolean. |
| $void | Void declaration | Denotes a new declaration for a function identifier with no return value. |
| if | “if” statement control flow | Denotes a new scope within a single code block dependant on a condition. |
| else | “else” statement control flow | Denotes a new scope within a single code block dependant on a previous if statement returning false. |
| while | “while” statement control flow | Denotes a conditional statement that repeats its corresponding “do” code block until it find the condition to be false. |
| do | “do” statement control flow | Denotes a new scope within a single code block that is repeatedly called until the corresponding while condition is found to be false. |
| = | Assignment operator | Assigns a value to a single identifier. |
| { and } | Left and right brackets. | Denote the start and finish of a new code block. |
| ( and ) | Left and right parentheses. | Denote the start and finish of parameters. |
| ; | Statement close. | Denotes the end of a single statement. |

Program Documentation

*Explanations of classes and methods are included with the commenting within the code. This section will address all features of the language and the compiler and the reasoning behind the decisions made.*

Language Features

The language supports many features. These include:

* Integer, floating point, boolean and string type variables
* Variable declarations and assignments
* Expressions and conditional expressions
* If, if-else and while conditional control flow structures with individual scopes and statement blocks.
* Return type and void functions with any number of arguments
* Nesting of functions

Program Timeline

From being given a character sequence from a text file to outputting the target language the program carries out many functions. This section will outline the main steps made that the program carries out to reach the final target language output.

1. Lexical analysis, parsing using grammar

This is all carried out through the ANTLR plug-in in IntelliJ. A parse tree is created with the root node as a “ProgramContext” object.

2. Translation of a parse tree to an abstract syntax tree

The ProgramContext object is used to create an AST. The nodes are traversed and redundant information is removed, leaving only the absolutely necessary information.

3. Static type checking

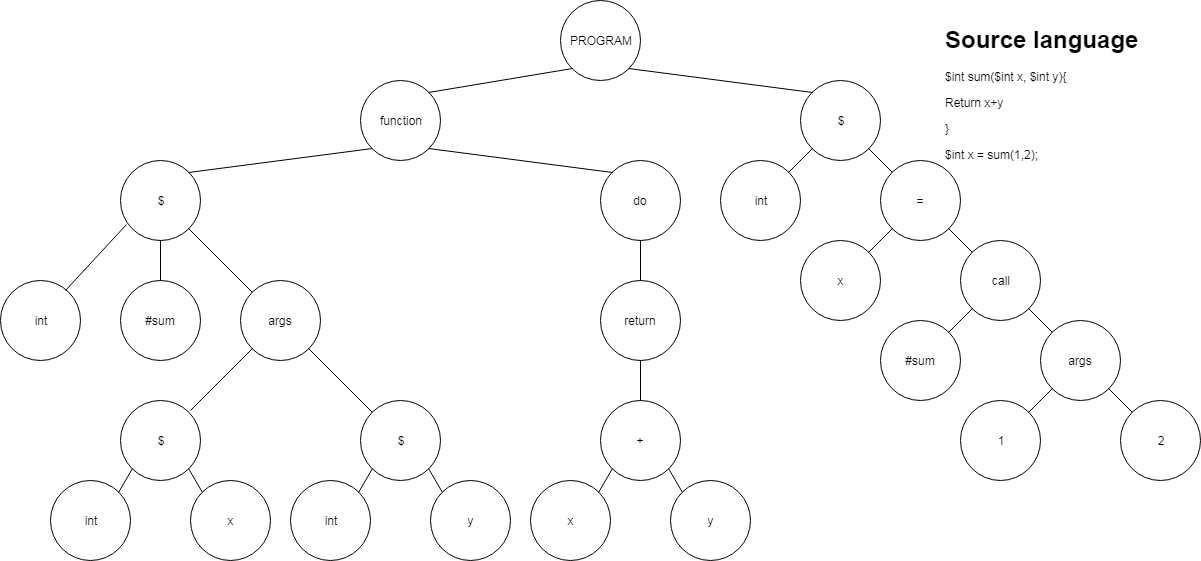
The newly generated AST is traversed, top to bottom, left to right (pre-order traversal). Each declaration is added to a simple lookup table and conflicting types throw exceptions.

4. Target language generation

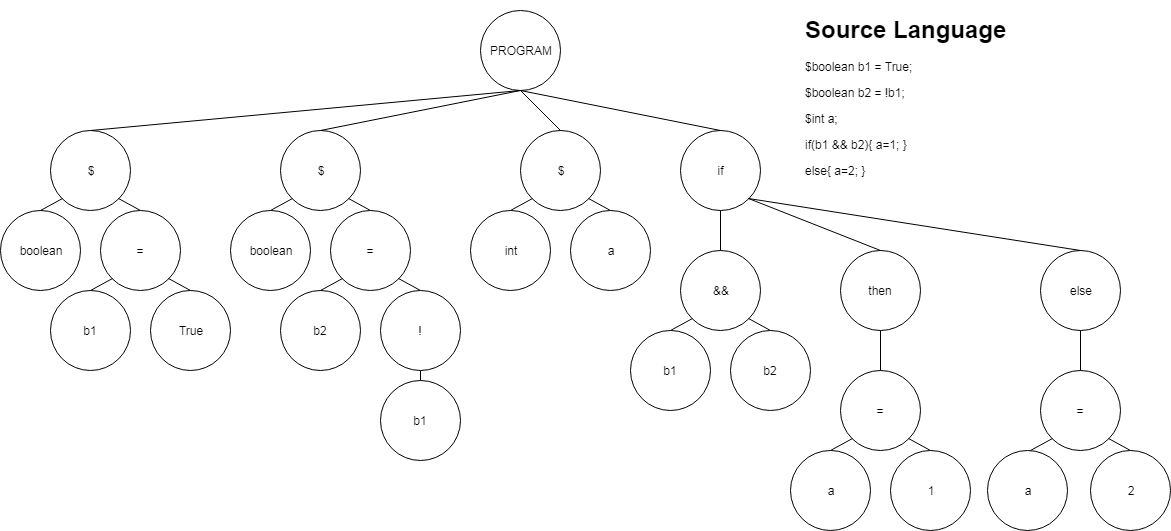
The AST is again traversed in pre-order. This time appending each statement found to a StringBuilder object in its correct formatting.

Abstract Syntax Tree Generation

This section will detail the generation of abstract syntax trees.



*Figure 1: Assigning a variable “x” with the value returned from calling a function “sum(1+2)”*

**

*Figure 2: Declaring a variable “a” and only assigning it a value based on a condition (if then else)*

The AST generated shows the minimum information required for the compiler to carry out static type checking and generate target language code. The original parse tree is traversed recursively in pre-order, returning AST objects up the method call chain. Redundant nodes are removed and others are formatted into AST objects to be appended to a parent node, eventually being added to a “PROGRAM” AST root node.

Whilst traversing the parse tree the compiler will throw exceptions if any abnormalities show.

The resulting AST is used for static type checking and python code generation.

Type Checking & Scope Management

Types

The language allows 5 primitive types. These are integer, floating point, Boolean, string, and void types.

String type is exclusive to the print function call. Void type is exclusive to functions.

Static Type Checking

Type checking is carried out by traversing the generated AST and monitoring symbol lookup values. Static type checking will ensure all of the following:

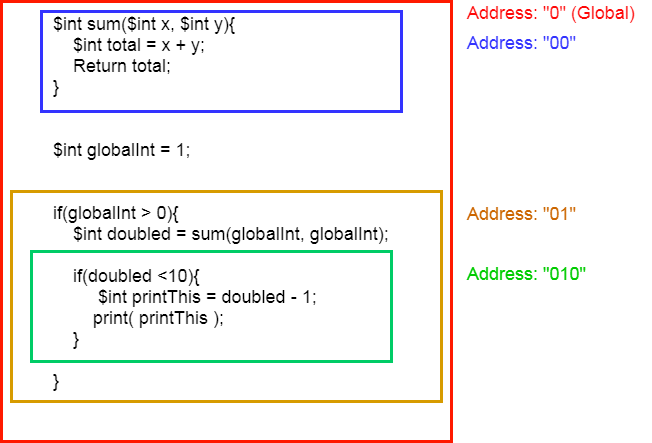
* Declaration and assignment statements are assigning the correct type to the identifier (includes function call types).
* Function return types are matching with a function’s type.
* A function call has the same number of arguments as the function signature specifies.
* A function call has the correct types of arguments given.
* Variables being declared are not already visible in the current scope.
* Variables being assigned values are visible within the current scope.
* Operators used in expressions make logical or numerical sense. (E.g. will not allow an expression or condition such as “True + False” or “5 && 6”.

The AST is traversed top to bottom, left to right, following the program execution direction.

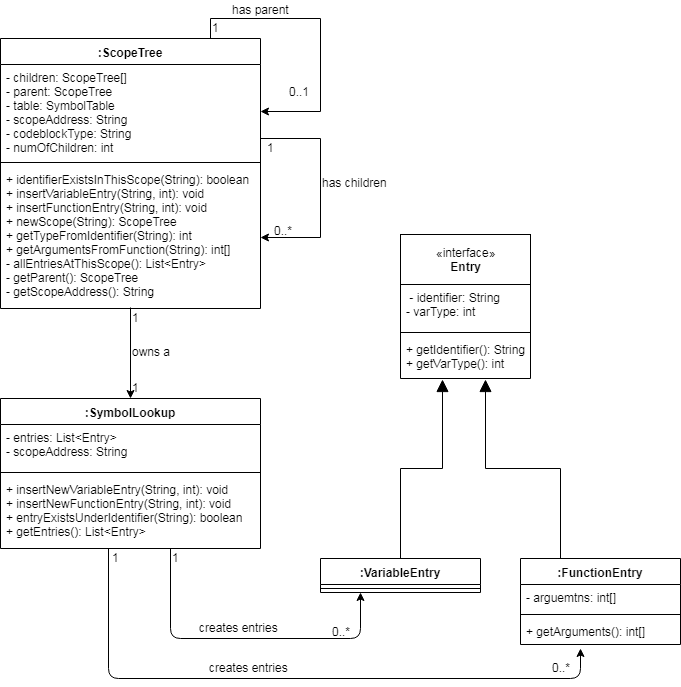
Scoping and Symbol Lookup

**Scope Addressing**

In this compiler scopes are addressed using a string of integers. The address is read from left to right. Each integer represents the child number of its parent, child numbers begin from 0. “0” is the global scope.



*Figure 3: Scope addressing visual representation*



*Figure 4: Scoping class diagram*

Scoping is managed through the use of ScopeTree objects. A ScopeTree object is a single scope with a SymbolLookup object field. A ScopeTree can have any number of child scopes that represent nested scopes.

All ScopeTree objects apart from the global ScopeTree object will have a reference to its parent object. This allows ScopeTree objects to look up identifiers at wider scopes that should still be visible to themselves. For example, in reference to *Figure 3* the “010” scope can access variables in “01” and “0”.

**SymbolLookup Objects**

SymbolLookup objects simply hold a list of Entry objects and offer methods to update and query the list.

**Entries**

Entry objects hold information about a single entry in the SymbolLookup list. The Entry class is abstract and extended to VariableEntry and FunctionEntry classes. They by default store identifier names as a string and the type as an integer value. FunctionEntry objects will also store argument information as an array of integer values.

Python Code Generation

Python code is generated and formatted in the PythonCodeGenerator inner-class of the ASTManager class. This class holds an integer value denoting the tabbing necessary.

The AST is traversed in pre-order and each statement is formatted into a python recognisable string, for example, generating a colon for control flow statements after the condition. If a new scope statement is found (function, if, while etc.) then the tabbing integer is incremented. Once the new scoped body has been traversed the tabbing integer is decremented. All static typing notations are removed, such as function and variable types being declared.

Using a StringBuilder object the strings are appended. If a new line is appended the appropriate number of tabs are added to the next line.