**How Compilers Work**

Structured in multiple phases. First there is an input program, then at the end is an output program.

**First, Tokenize**: Program is fed into a Tokenizer/Lexer where the code is broken down into tokens

**Second, Parsing**: Tokens are passed into a Parser and the program is then represented into an Abstract Syntax Tree (AST).

**Third, Typechecked**: The AST is passed into the Typechecker which annotates the AST.

**Fourth, Code Generated**: The annotated AST is passed into a Code Generator and outputs the program.

**Typechecker**

**Tokenizer/Lexer**

Input Program

Annotated AST

Tokens

Output Program

AST

**Typechecker**

**Parser**

**Tokenizer/Lexer**

An input program is taken in and broken down into tokens. **Tokens** are sections of a program divided up.

Example:

if(1 < 2) return 7;

else return 3;

Tokens would be: if\_token, else\_token, (\_token, )\_token, <\_token, number\_token(1), number\_token(2), number\_token(3), number\_token(7), return\_token, semi-colon\_token

**Parser**

A parser is what groups tokens together and represents the program as a data structure rather than a program, code, or bytes. The main output of the parser is an **Abstract Syntax Tree (AST)** which is an original representation of the program represented in a tree structure.

Example:

if(1 < 2) return 7;

else return 3;

**Typechecker**

It looks at all the types involved and check if each type is used correctly. If they are not, a compile error is returned, and the program terminates. The AST is then annotated and passed along.

Example: int i = 2; //Type int is paired with an int. Types are properly paired

int s = “dog”; //Type int is paired with a string. Types not a proper pair

string s = “dog”;

int i = 10;

if(s < i) return s + i;

// The comparison between variables s and i are not correctly used

Since this code’s syntax is correct, it will get through the Tokenizer/Lexer

and parser. This is why we need a Typerchecker.

**Code Generator**

This is where the actual code to low-level language translation is done. This is what returns the output program. What separates a good compiler from a great compiler is the code generator, so a good code generator is very important. The design and breakup of the code generator is strictly language and architecture dependent.

**Exercise**: **Building a Context-Free Grammar Binary Compiler**

This compiler inputs a string of 1’s and 0’s, combines them into the number variable, then goes through an expression to get the final “compiled” result.

Each line is referred to as a **Production**. Everything inside a production is referred to as **Productions**. and whenever a ‘’ is present around a variable, it means this specific quoted expression or input strictly needs to happen

digit ::= ‘1’ **|** ‘0’ // Input variable

number ::= digit **|** digit & number // Recursive Variable

expression ::= number **|** expression ‘+’ expression // Recursive Statement

Input:

1

10

10 + 1

Output:

1:

digit = 1, number = digit, expression = 1

10:

digit = 1, number = 1, expression = 1. Then digit = 0, number = 10, expression = 10

10 + 1:

digit = 1, number = digit, expression = 1, Then digit = 0, number = 10, expression = 10, first expression = 10. Recursive call is made, digit = 1, number = digit, next expression = 1. Since there are two expressions inputted, the recursion stops and does “expression + expression” which in this case = 11 as our “compiled” output.

**Language Design**

**Simplistic Very Basic Language Design Layout**:

* Has Integers and Booleans
* Variable can be declared and initialized and assigned
* Performs standard arithmetic and logical operations
* Able to loop

**Grammar Syntax Components For this Simplistic Very Basic Language**:

var ::= Variable name

num ::= Integer

type ::= ‘int’ **|** ‘bool’

variable\_decloration ::= ‘(‘ ‘vardec’ type var expression ‘)’

expression ::= num **|** ‘true’ **|** ‘false’ **|** var **|** ‘(‘ operator expression expression ‘)’

loop ::= ‘while’ ‘(‘ expression statement ‘)’ ‘END\_LOOP’

assign ::= ‘(‘ ‘=’ var expression ‘)’

statement ::= variable\_decloration **|** loop **|** assignment

operator ::= ‘+’ **|** ‘-‘ **|** ‘&&’ **|** ‘||’ **|** ‘<’

program ::= statments\*

{*NOTE: \* means you can have as many of these as needed*}

~**Using this as if it were code**~

(vardec int i 7) // int i = 7

(vardec bool b true) // bool b = true

(vardec int j (+ 15 22) ) // int j = 15 + 22

(vardec bool c (&& true false) ) // bool c = true && false

while( < i j) // while(i < j){ i += j}

( i = (+ i j) )

END\_LOOP

{NOTE: Having code in this format compared to the C/C++/Java format is good for us because it will make the parsing stage a lot easier}

**Tokens**:

* IdentifierToken(String) // Holds identifier type, and string used for operators
* NumberToken(int) // Holds the integer
* IntToken // For the string ‘int’
* BoolToken // For the string ‘bool’
* LeftParenthesesToken // For the char (
* RightParenthesesToken // For the char )

[NOTE – These simple character tokens are needed cause the start and ending of anything inside parentheses is an important operation and detection]

* VardecToken // For the string ‘vardec’
* TrueToken // For the string ‘true’
* FalseToken // For the string ‘false’
* WhileToken // Looks for a loop
* *END\_LOOPToken // Maybe needed maybe not, for string*
* EqualsToken // Looks for the equals operator
* PlusToken // Looks for the addition operator
* MinusToken // Looks for the Minus operator
* LogicalAndToken // Looks for the && operator
* LogicalOrToken // Looks for the || operator
* LogicalLessThanToken // Looks for the < operator

**Parsing’s**:

[Note, each specific parsing is a class. But for simplicity is broken down into categories]

* **Interface Type**
  + IntType
  + BoolType
* **Interface Statement** 
  + vardecStatment
  + LoopStatment
  + AssignmentStatment
* **Interface Expression** 
  + NumberExpression
  + BoolExpression
  + VariableExpression
  + BinaryOperatorExpression //For the expression production where there’s ‘expression expression’
* **Interface Operator** 
  + PlusOperator
  + MinusOperator
  + AndOperator
  + OrOperator
  + LessThanOperator
* **Interface AST**
  + IntType
  + BoolType
  + vardecStatment
  + LoopStatment
* **Class Program**

**Typechecks**:

* Varcec puts a Variable in a scope with a type
  + Needs to remember the variable and type
  + Needs to ensure the expression of the type
* Num should be an int
* Bool should be true or false
* Var is whatever the type of the variable is
* While expression is a Boolean
* Assignments
  + Var should be in scope
  + Vars type should match the expression type
* Expression1 + Expression2, Expression1 – Expression2, Expression1 < Expression2
  + int, expression1: int, expression2: int
* Expression1 && Expression2, Expression1 || Expression2
  + Bool, expression1: bool, expression2: bool: bool
* Program
  + All statements are correctly types

**Code Generation**: Need to watch the video

**Go to 54:15 in** [**Lecture 1's Video**](https://mycsun.app.box.com/s/h77o6g57vmct7c8cm2o5de9iehvzro5z/file/1131029573979) **to start implementing this as an actual compiler. The rest of the lectures are implementing this in code with each compiler Phase.**

[**Tokenizing**](https://mycsun.app.box.com/s/h77o6g57vmct7c8cm2o5de9iehvzro5z/file/1131044882559)

[**Parsing**](https://mycsun.app.box.com/s/h77o6g57vmct7c8cm2o5de9iehvzro5z/file/1131045425225)

[**Typechecking**](https://mycsun.app.box.com/s/h77o6g57vmct7c8cm2o5de9iehvzro5z/file/1131032697911)

[**Code Generator**](https://mycsun.app.box.com/s/h77o6g57vmct7c8cm2o5de9iehvzro5z/file/1131043171563)