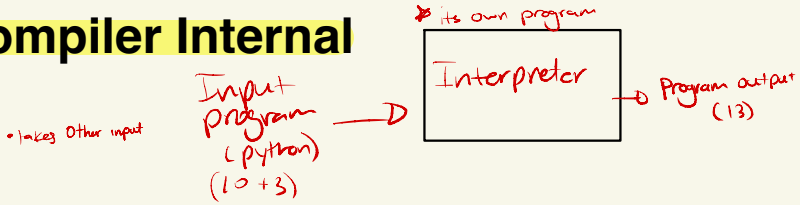
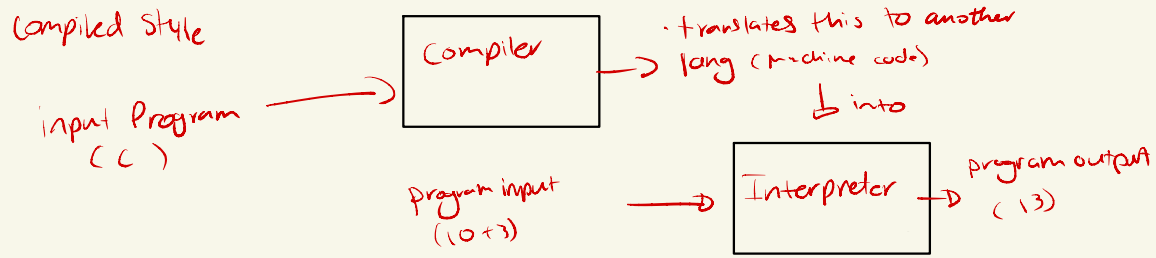
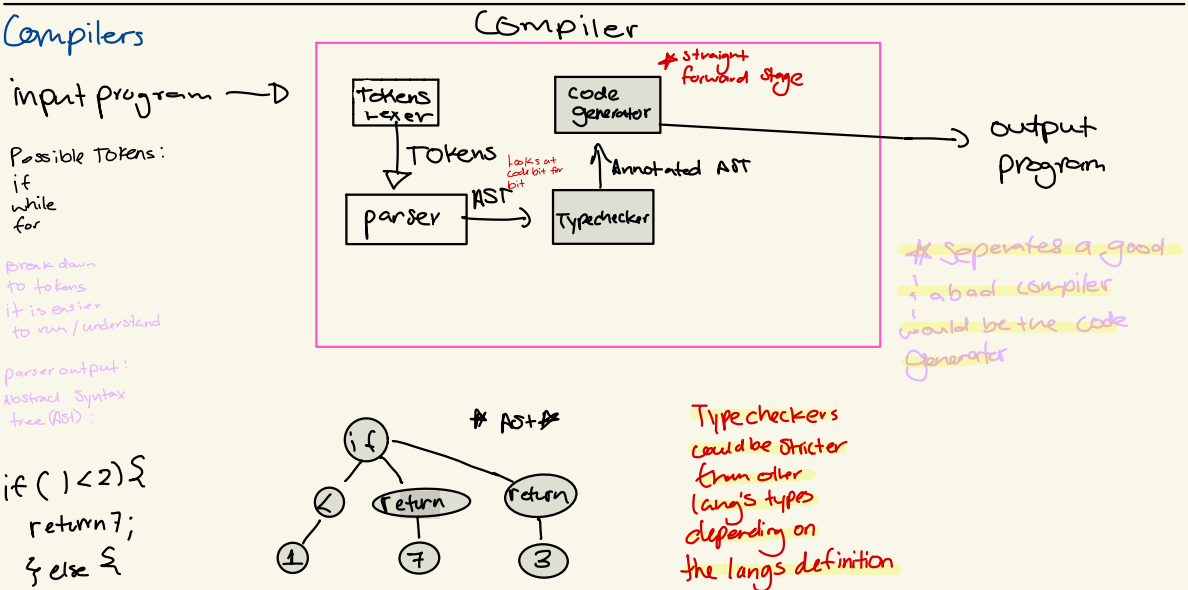
Part I Compiler Internal





transpiled / transpiler: does not differ; compiler generating high lvl lang to another high lvl lang

Compilers



// Grammer / BNF / Backus-Naur Form / context free grammer

digit ::= `0` | `1`
number ::= digit | digit number
number ::= digit* // EBF form
Expression ::= number | expression `+` expression

} symbols treats possible

` ` ← Backtext

::= is composed

* = means 0 or more of the thing

Example numbers:
0
1
01
101

Ex Expressions:
1101
101 + 110
(Expression) + (Expression) -D

1 + 111 + 01
(Expression + Expression) + (Expression)

Parsing grammer is difficult *

` ` = symbol must be here directly

X = refer to what ever production rule it is

epsilon = empty

Language design:
- Ints; Boolean
- Declare; Initialize
- Perform typical arithmatic; logical operations

var is a variable
num is a number

type ::= `int` | `bool` // Production ≈ line
vardec ::= `(` `vardec` type var expression `)` // (vardec int), (Vardec bool
expression ::= num | `true` | `false` | var |
`(` op expression expression `)`

Loop ::= `(` `while` expression statement `)`
assign ::= `(` `=` var expression `)`
Statement ::= vardec | loop | assign
op ::= `+` | `-` | `*` | `;` | `||` | `<`
// program ::= epsilon | vardec program
Program ::= Statement*

(vardec int x 7)
(vardec bool y true)
(vardec int a (+ 1 2)

In Java: int x = 7;
boolean y = true;
int a = 1+2;
boolean b = false ; ; true;

(vardec bool b (; ; false true))

```
int x=0;
while(x<10){
  x=x+1;
}
```

≈

```
(vardec int x 0)
while (< x 10
    (= x (+ x 1)))
```

Object Language (our language): MyLang

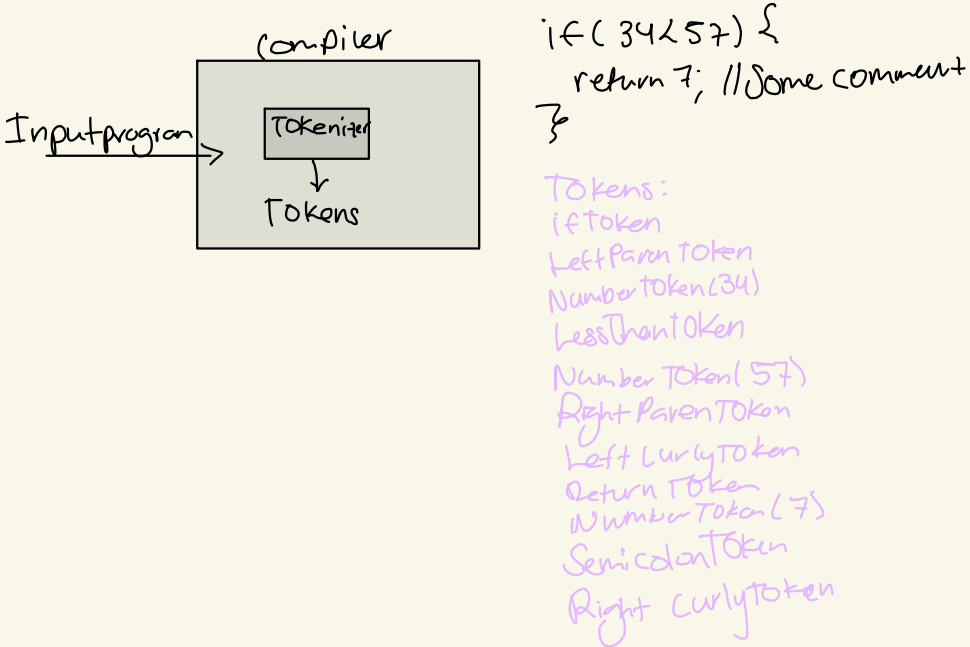
MetaLanguage (what we are writing the compiler in): Java

Target Language (what we are compiling to): JavaScript

```
package myLang;

public class Arithmetic {
    public static int add(int first, int second){
        return first + second;
}
public static int subtract(int first, int second) {
    return first - second;
```

```
package myLang;

public class Arthmetictest{


@Test
public void testAdd(){
    assertEquals(3, Arithmetic.add(1,2));
}
@Test
public void testSubtract(){
    assertEquals(2, Arithmetic.subtract(6,4)){
}
}
```

Part II Tokenization/Lexing



Keeping it simple so that we know what to expect for the tokens (important)

Possible Tokens:
IdentifierToken(String)
NumberToken(int)
IntToken
BoolToken
LeftParenToken
VardecToken
RightParenToken
TrueToken
FalseToken
WhileToken
SingleEqualsToken
MinusToken
LogicalAndToken
LogicalOrToken
LessThanToken

Example:

(vardec int x 7)

LeftParenToken
VardecToken
IntToken
IdentifierToken(“x”)
NumberToken(“7”)

##TOKENS#

Important to add/put all the tokens into the ReadMe file on github

‘ ‘ ‘ Java

```
public class MyClass{
    public static void main(String[] args){
 }
}
```

Tokens:

- PublicToken
- ClassToken
- IdentifierToken(“MyClass”)
- ….
- IdentifierToken(“main”)
- …..
- IdentifierToken(“String”)

Add all Tokenizer into its own branch of course don’t forget

- Also add *Import org.junit.Test; …*

When testing always remember the @Test to signify that your testing

TokenizerTest.java

```
import org.junit.Test;
import static org.junit.Assert.assertArrayEquals;
import static org.junit.Assert.assertEquals;

public class TokenizerTest{
    @Test
    public void testIdentiferEquals(){
        assertEquals(new IdentifierToken(“foo”),
        assertEquals(new IdentifierToken(“foo”));
}
@Test
public void testTokenizeIdentifier(){
```

```
final Token[] tokens = Tokenizer.tokenize(“vardectoken int x 7)”);
final Token[] expected = new Token[]{
    new LeftParenToken(), // all implies it is a base class token & subclasses for each token…
    new VardecToken(),
    new IntToken(),
    new IdentifierToken(“x”),
    new NumberToken(7),
    new RightParenToken()
};
assertArrayEquals(expected, tokens);
    }
} // always write the tests first before writing the whole program or deciding to run it
```

Tokenizer.java

```
public class Tokenizer{
    private final String input;
    private int position;
public Tokenizer(final int String input){
    this.input = input;
    position = 0;
}

//returns null if it is not a indentifier or reserved words (vardec while)
public  Token tokenizeIdentifierOrReservedWord(){
    String name = “”;
    if(Character.isLetter(input.charAt(position)){
    name += input.charAt(position);
    position++;
}
    while(Character.isLetterOrDigit(input.charAt(position))){
    name += input charAt(position);
    position++;
}
if(name.equals(“int”)){
    return new IntToken();
    }else if(name.equals(“bool”)){
    return new BoolToken();
}else if (name.equals(“vardec”)){
    return new VardecToken();
} else if(name.equals(“true”)){
    return new TrueToken();
}else if (name.equals(‘false)){
    return new FalseToken();
……
// returns null if  it not a number
```

```
    public static NumberToken tokenizeNumber(final String input,
                                                    int position) {
        String digits = “”; // holding the digits
        while (character.isDigit(input.charAt(position))){
            digits += input.charAt(position);
            position++;
}
    if(digits.length() > 0){
        return new  NumberToken(Integer.pareInt(digits));
    public static Token[] tokenize(final String input){
} else {
    return null;
     }
}
```

//once it returns we wont know where’s the token is we have to keep track of it this is important!!!

important and it’s difficult to do so since we have to always simplify it !!!!!!

// for every symbol we would assign its token its a lot of boilerplate but its a continuation of similar code like

new SymbolPair(“(“, new LeftParenToken())) // this is efficient and would make it better for memory and easier to code one benefit of boilerplate

we would do the same for “)”, “==”, “+”, every operator and we would have to iterate through all the symbols and add the length and position to it

We would need to implement a readToken() which would skip white space which is important to tokenize everything; order is not important!!

Important to simplify these with data structures as it gets annoying to do boiler-point

//Do not quickly go from a simple test to a difficult test, ways to avoid this

- Build smaller tests to test individual components
- Keep tests simple and always test before continuing your code so that you wont have a big error of tests and have to spend time debugging other problems

IdentifierToken.java

```
public class IdenfitierToken implements Token{
    public final String name;

    public IdentifierToken(final String name) {
        this.name = name;
}
@Override
public boolean equals(final Object other) {
    return (other instanceof IdentifierToken &&
            name.equals(((IdentifierToken)other).name));
}
@Override
public int hasCode(){
    return name.hasCode();
}
@Override
public String toString(){
    return “IdentifierToken(“ + name + “)”;
// you have to write the classes again and again for all the tokens you want and it will be a
hassle just remember to do so

example for this from vid ——>
public class NumberToken implements Token{
    public final int value;

    public NumberToken(final Object other){
        return (other instanceof NumberToken &&
                value == ((NumberToken)other).value);
}
```

```
@Override
public int hashCode(){
    return value;
}

@Override
public String toString(){
    return “NumberToken(“ + value + “);
}
```

// for the tokens that have no real value you can assign numbers to it so it can come back to us on the hasCode and we can see all int tokens return 0 for example and all boolean tokens return 2 and so fourth……

//should always read the file after each character at a time; never let it read the whole file as a whole as it can take up a lot of space/memory

//make sure everything is written properly since this can make methods and files to fail testing phase

//Test in error

- Throwing a exception and could not authorize the tests

- make more test under what your suspicious on and then it will help you figure out why it is failing also check the html file as it can also help you figure out what is red or green on the mvn file’s

when branches are missed;

- it could be that it the loop or method was never hit or made it to the return statement you ought to put for example if it has a false or null it means it never hit that point // could be ignored

- never hit the null which could just mean it did tokenize but it just couldn't return the null as it never saw a case where it could tokenize, should have a test if it did or didn't fail to tokenize which is important

Test example:

```
@Test

public void testTokenEmptyInput()( throws TokenizereException{
    assertArrayEquals(new Token[0], Tokenizer.tokenize(""));
}

public void testTokenOnlyWhiteSpaceInput()( throws TokenizereException{
    assertArrayEquals(new Token[0], Tokenizer.tokenize("  "));
}
@Test(excepted = TokeninzerExceptio.class)
public void testCannotTokenize(){
    Tokenizer.tokenizer("$");
}
```

RUNNNN COVERAGE REPORT ON JACOCO it'll give you proper reports for failures!

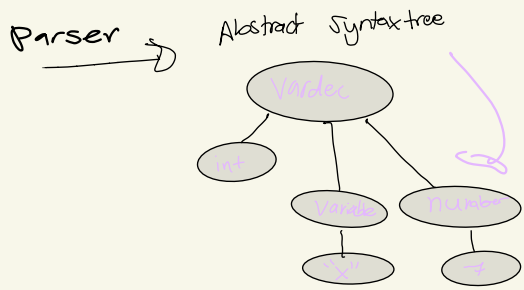
metal level it is ok to copy and paste code as it usually is test

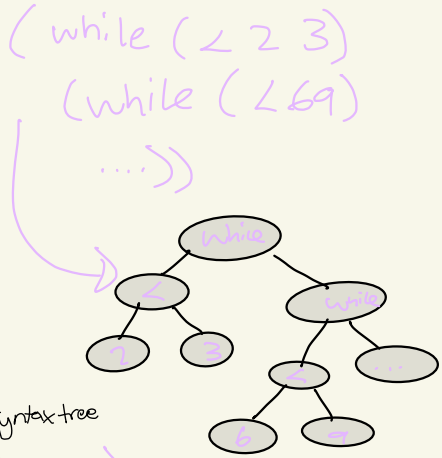
coverage is important and always have high demand of coverage is this is minimum bar for these type of stuff

Part III Parsing

(vardec int x 7)

Tokens
LeftParenToken
VardecToken
IntToken
IdentifierToken("x")
NumberToken(7)
RightParenToken





Easier & useful to work with

interface is only needed when you need to pick a choice between things

Var is a variable
Num is a number

type ::= `int` | `bool` // Production ≈ line
vardec ::= `(` `vardec` type Var expression `)`
expression ::= num | `true` | `false` | var
`(` `op` expression expression `)`

Loop ::= `(` `while` expression statement `)`
assign ::= `(` `=` var expression `)`
Statement ::= vardec | loop | assign
op ::= `+` | `-` | `*` | `/` | `||` | `<`
// program ::= epsilon | vardec program
Program ::= Statement*

(vardec int x 7)
(vardec bool y true)
(vardec int a (+ 1 2))

For every kind of production rule that we have it becomes a type of AST Node

Ex:

- **IntType**
- **BoolType**

Interface Statement

- **VardecStmt**
- **LoopStmt**
- **AssignStmt**

Interface Exp

- **NumberLiteralExp**
- **BooelanLiteralExp**
- **VariableExp**
- **BinaryOperatorExp**

Interface OP

- **PlusOp**
- **MinusOp**
- **LogicalAndOp**
- **LogicalOrOp**
- **LessThanOp**

Class Program

//

Littorals would be their own values like
7
"foo"
true

Interface AST

- **implementation of all classes as interface**
- **this is bad since you can keep many errors ongoing**
- **this can be avoidable if you just make methods and keep them under check in their own interface types!!!!**

Complex Hierarchy is better……

```
public class VardecStmt implements Stmt{
      public final Type type;
      public final Variable variable;
      public final Exp exp;
….
```

^^^^^^ Better

```
public classs VardecStmt implements AST{
      public final AST type; // has to cast which causes problems!!!!!! Type check!!! Java is difficult
      public final AST variable;
      public final AST exp;
```

^^^^^^ worst

Documentation is key for the Readme file yet again do not forgot david!!!

Intuition for Doing Tests first

- **key info can be discovered before jumping into the code**
- **Want to also confirm they fail its good if it does not always is passing important for tests**

Tokenizer and parsing should feel similar as we have similar problems which needs to be also tackled with the same solution

when hashing numbers dont matter it just keeping them on track and seeing if there's a failure we know that this is the issue and we can tackle it head on

instanceof: *A keyword used for checking if a reference variable contains a given type of object reference or not*

If a tests fails it will call the toString on the object to show what object it expected but ended up receiving

Parsing: the solved problems that isn't

O(n^3): memory usage // bad also number of tokens

Recursive Decent Parsing
- When developing everything is one to one
- The grammar becomes law and tidy
- Best as many tools aren't as tidy

Infamous parsing generator
- ANTLR

cons
- define it as AST
- make it generate your AST

Parser Combinators
- poor look on them and complexity

// S-expressions
(while(< 7 4))
while(< 3 4)….))
- Originated in LIST (List Processing
- everything in this family of LIST uses it

Languages that dont use S-expressions have issues like
Left Recursion
- exp ::= num | exp + exp
- uses its self recursively on the left and starts the production on itself and is a bad problem for parsing
- you will get lost in recursive cases and will forget when to use the base case

When it sees the left parentheses it will know that yes it has to do a recursive call!!!

Abstract Grammar
can't use it as specification

Concrete Grammar
you could use it for specification

conversions between the two takes a lot of time as it has a learning curve

cons for CG:
- Precedence….
- now were required to write the parentheses rather than inferring them

For each production rules were going to define a function or method for them in the parser

it will have a bunch of methods in the parser and tackle it with the greedy approach

important to think about a parsing exception so when it fails we can throw an exception

whenever parsing fails when retrieving parsing result we have to update and restore the stack and reset it because if we dont we wont be on the same position on the stack which would be a big issue

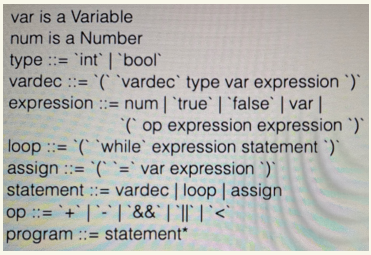
statement*

- **star is similar to its either empty or followed by another program**
- **read that statement and add it to the list and as you keep adding it then you return whatever you were able to read**

Generic types must be exactly that statement referred to in java sadly

Part IIII Type-checking

Types: used to describe data and the operations applicable to that data
Type error
Program with typed errors: ill typed
Program without typed errors: well-typed



- **vardec Puts a variable in scope with a type**
- **need to remember the variable and type**
- **need to ensure the expression is of the type**

- **num should be an int**
- **true or false should be bools**
- **var is whatever the type of the variable is**
- **while's expressions is a boolean**
- **assign:**
- **var should be in scope**
- **bars type should match expressions type**

- **exp1 + exp2 => int, exp1: int exp2: int**
- **exp 1 - exp2 => int, exp1: int exp2: int**
- **exp1 && exp2 => bool, exp1: bool exp2: bool**
- **exp1 ll exp2 => bool, exp1: bool exp2: bool**
- **exp1 < exp2 => bool, exp1: int exp2: int**
- **program: all statements are well typed**

rules similar to what java uses

When doing your typeChecking you should always put your AST expressions and expressions as a whole under the readMe file that we continue to talk about.

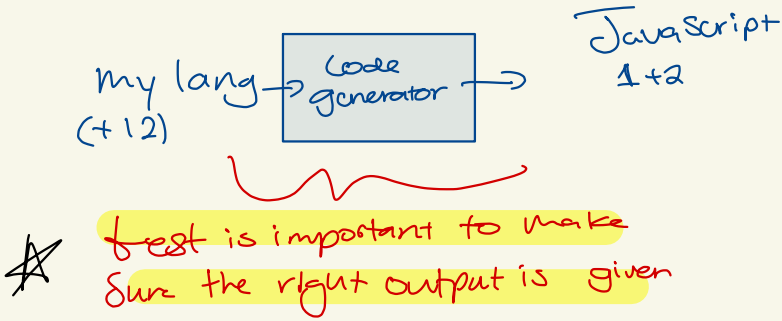
- **helps you stay focus as you have all your important coding info there that needs to be coded**
- **instanceof is a good tool to use do not forget this**

Make sure each variable or method your using is in scope if it isn't then we would have issues when running tests and going up to jacoco code coverage report

statements themselves don't give/return anything

Tests in operators its small details that can cause bigger issues later down the road just get it over with!

Part V Code Generator



Code generator needs a AST:

- **Takes a program ; list of statements**
- **Takes this program and writes it to an output files**

When you write a code generator you have to close the writer so dont forget to close it !!!

Basic Code Generator

- **For everything you have you do a traversal over the entire AST**
- **After being transversed you'll write it to the output destination is**

Read your documentation !!!!!

Indentation and format is also important when debugging as it can be an issue

Maps: are in order so you have to preserve order