CS132 (Compiler Construction)

TA: Akshay Utture (akshayutture@ucla.edu)

General Announcements

- Please ask as many questions as you want during the TA discussions.
- If you have a question outside of the discussion. Try these in this preference order:
  - Ask on Piazza
  - Ask in the office hours (add time)
  - Email me (akshayutture@ucla.edu)
- Do participate in discussion sessions. Either write in the comments or simply speak up. But I may not look at the comments as often.
- You do not need any libraries for the assignments apart from the std. lib.

Plan for the TA Sessions

Week 1: CS181 pre-reqs and setting up the homework framework
Week 2: HW1
Week 3: HW2
Week 4: HW2
Week 5: HW3
Week 6: HW3
Week 7: HW4
Week 8: HW4
Week 9: HW5
Week 10: Revision for the finals

Discussion Lec 1

Today's discussion

1. CS181 prerequisites (Regex, DFA, NFA, Context-free grammar)
2. Setting up the homework-framework

Part1: CS181 pre-reqs

Regular Expressions (Regex)

Regex is a:
- a symbol (a b c)
- a concatenation (ab bc ba)
- an alternation (a|b c|ab d|hl)
- a kleene closure (0* ab* (a|ds)*)
- epsilon empty (e') \eps, \epsilon
- combination of the above

eg. b (b)* a
=> accepts any string which starts with a 'b', ends with an 'a', and has any number of 'b's in between.

eg2. (a | b)* =
=> accepts any strings with any number of a's and b's

Regular-Language to Regex

What is the Regex for these languages:
1. {a, ab, abb, abbb, ….} => a(b)*
2. Whole Numbers => 0 | [1 - 9] [0 - 9]*
3. Any number of a's followed by the same number of b's => (not possible. not a regular language)

Q] Can you identify the language of these Regex?
(a|b)*ab(a|b)* => any string of As and Bs that has a B after an A

(b | a b* a b*)* => strings with an even number of A's

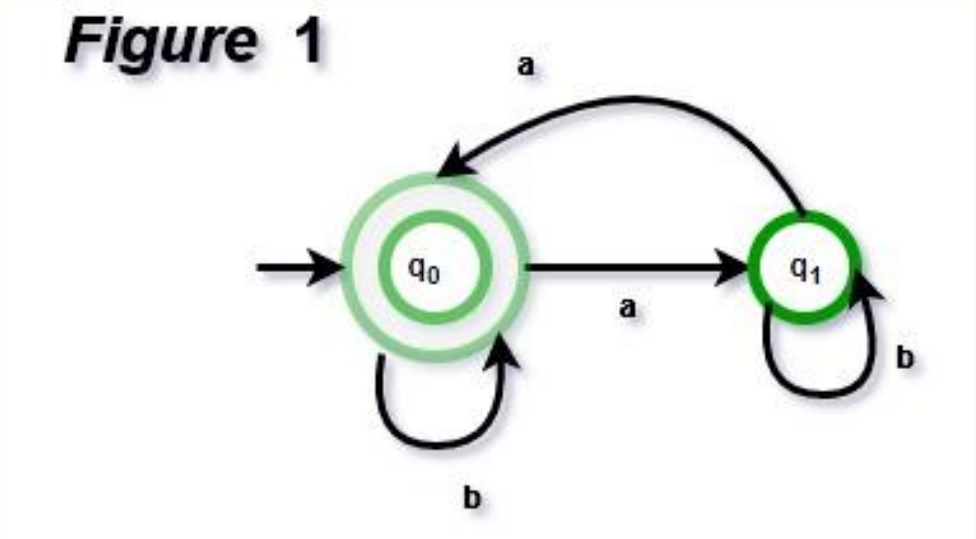
(b | a b* a)* (b | a b* a b*)* even number of a's

DFA (Deterministic Finite Automaton)

DFA = (Q,Σ,∆,q0,F)

Q: A set of states
Σ: set of input symbols,
∆: a transition function
q0: an initial state and
F: a set of Final or accepting states.

For Figure 1,
Q = { q0, q1}
Σ = { a, b}
q0 = { q0}
F = { q0 }
Language accepted = even number of a's



∆	a	b
q0	q1	q0
q1	q0	q1

NFA (Non-Deterministic Finite Automaton)

NFA is modification where a given input symbol can be on multiple out-edges.

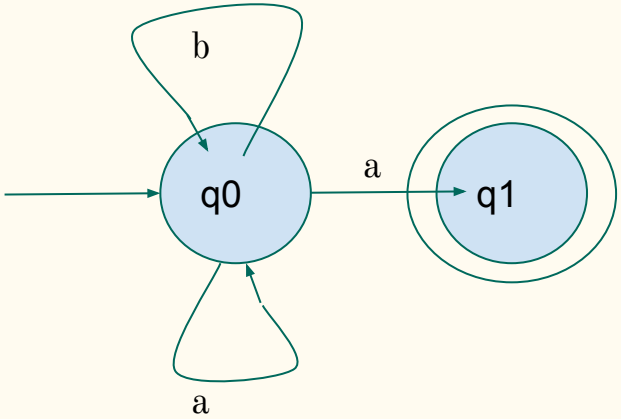
- Can convert any NFA to a DFA but with many more states
- Expressive power of Regex, NFA, DFA is the same.

(Q] What does expressive power mean?)

a

Regex to NFA

Ex. (a | b)* a (any string consisting of a,b which ends in an 'a')

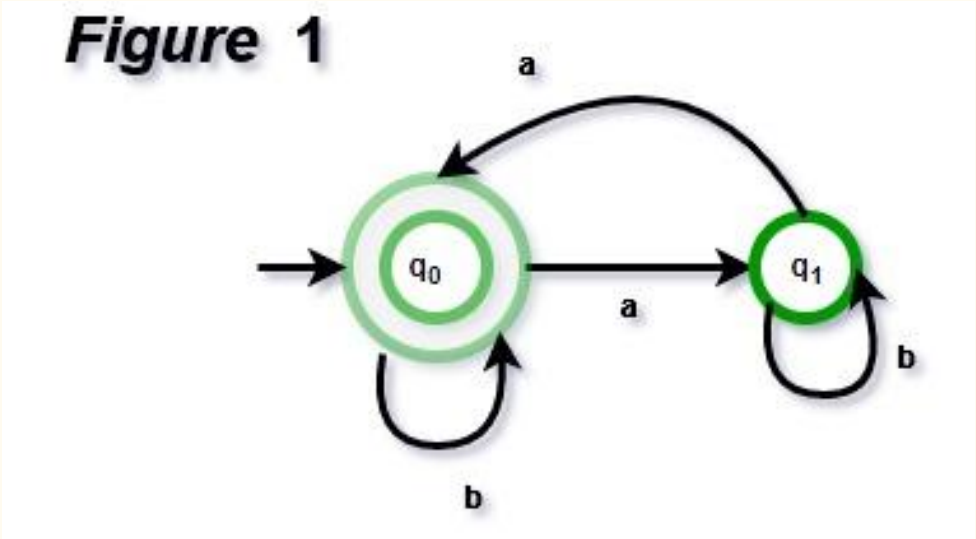


NFA to DFA

There exists a standard algorithm called **subset-construction** or **powerset-construction**

DFA to Recognizer

```
state = q0
transition_table = read_table()
accept_states = read_accept_states()

while(true){
        input = get_char()
        next_state = transition_table[state, input]
        state = next_state
        if input == EOF:
            if state in accept_states:
                return true
            else
                return false
}
```



Δ	a	b
q0	q1	q0
q1	q0	q1

The conversions we have learned

1. Regular-Language to Regex
2. Regex to NFA
3. NFA to DFA
4. DFA to Recognizer

So we have a method to go from Regular-Language to Recognizer.

Workflow for building a Scanner program

- Scanner
  - Input: Raw String (eg. "341 …..")
  - Internals: One Recognizer program (DFA) per token.
  - Output: Token corresponding to the Recognizer program which 'accepted' the string. (eg. <integer>)

- Now when we write the Parser, we can deal with tokens like <identifier>, <integer>, <for> instead of raw strings

What is the simplification if each of the strings in the language comes from a fixed, finite set?

eg. the only allowed tokens are: read, write, print.

Ans] Basically match the string character by character.

…..

Context Free Grammar

More expressive than Regular Expressions
G = (Vt, Vn, P, R)
Vt = terminals (lower-case)
Vn = non-terminals (upper-case)
P = Goal
R = productions

eg.
P ::= L $
L ::= S L | \epsilon
S ::= x = E; | read x; | write E;
E ::= true | false

generating an example string in the language
P ⟶ L $ ⟶ S L $ ⟶ S \eps $ ⟶ S $ ⟶ x = E; $ ⟶ x = true; $

Context Free Grammars

Q] Write a grammar for these languages

1) Any string consisting of a's and b's, and ending in an a

P ::= E a
E ::= a E | b E | \eps

2) any number of a's followed by an equal number of b's

P ::= a P b | \eps

Context-Free Grammars have the more expressive powers than a (DFA / NFA / Regex)

- The first language had a regex
- The second language did not have a corresponding regex.

Important Java concepts you should know

Class
Interface (class A implements B)
Inheritance (class A extends B)
Constructor
Class field
Encapsulation (public, private, protected)
Static methods
Overloading
Overriding
Dynamic Dispatch (Polymorphism)
Generics

Part 2: Setting up the framework.

You can leave if you have got the framework working.

Discussion Lec 2

Today's discussion

1. Parsing
2. LL(1) academy
3. Java recap

Remember: HW-1 based on Scanning+Parsing is due on Tuesday (Oct 5)

Part 1: Parsing

Q] What is top down parsing vs bottom-up parsing

Top down parsing:
Starting with start symbol and keep expanding the terminals until you match the non-terminals in the string

Bottom up parsing:
Start with the non-terminals in the string and we try to synthesize the start symbol from there
What kind is LL(1) parsing?: top-down parser

Q] What is an LL(1) grammar

It’s a grammar that we can parse in linear time with only 1 lookahead needed.
We can choose the next production by only looking at the following token.

A -> B | C

Steps to make a given grammar into an equivalent LL(1) grammar?

1. Remove left-recursion (A -> Ax)
2. Left Factoring (Ensure every leading symbol of a given nonterminal expansion is different)

Does this always work?
Ans] No this does not always work. So if we end up with a grammar where we can't apply the above 2 steps, we have an LL(1) grammar. But we may not terminate.

Steps in Building an LL(1) Parser

1. Compute the First, Follow and Nullable Sets
2. Build the Parse Table
3. Convert the Parse-table into a program

Building a Parser. Step1: Compute First, Follow, Nullable sets

Input Grammar

P ::= L

L ::= S L | \eps

S ::= <id> = E | read <id> | write E

E ::= true | false

For each of the non-terminal symbols (P, L, S, E) we need to find

First(A)

Follow(A)

Nullable(A)

Nullable(A) = there exists an expansion to \eps

2.1 If $a \in V_t$,

- NULLABLE(a) = false

2.2 If $A \to Y_1 \cdots Y_k$ is a production:

- [NULLABLE(Y_1) $\wedge \cdots \wedge$ NULLABLE(Y_k)] $\Longrightarrow$ NULLABLE(A)

Nullable(P) = Yes
Nullable(L) = Yes
Nullable(S) = No
Nullable(E) = No

P ::= L
L ::= S L | \epsilon
S ::= <id> = E | read <id> | write E
E ::= true | false

First(A) = Possible first token in expansion of A

2.1 If $a \in V_t$,

- FIRST(a) = { a }

2.2 If $A \rightarrow Y_1 Y_2 \cdots Y_k$ is a production:

- FIRST(Y_1) $\subseteq$ FIRST(A)
- $\forall i : 1 < i \leq k$, if NULLABLE($Y_1 \cdots Y_{i-1}$), then FIRST(Y_i) $\subseteq$ FIRST(A)

First(S) \subset First (L)

First(L) \subset First(P)

First(P) = {<id> , read, write}

First(L) = {<id>, read, write }

First(S) = {<id>, read, write }

First(E) = {true, false }

P ::= L

L ::= S L | \epsilon

S ::= <id> = E | read <id> | write E

E ::= true | false

Follow(A) = tokens appearing immediately after A

P ::= L
L ::= S L | \epsilon
S ::= <id> = E | read <id> | write E
E ::= true | false

Constraints:
FOLLOW(P) \subset FOLLOW(L)
FIRST(L) \subset FOLLOW(S)
FOLLOW(L) \subset FOLLOW(S)
FOLLOW(S) \subset FOLLOW(E)

2.1 If G is the start symbol and \$
- $\{ \$ \} \subseteq \text{FOLLOW}(G)$

2.2 If $A \to \alpha B \beta$ is a production:
- $\text{FIRST}(\beta) \subseteq \text{FOLLOW}(B)$
- if $\text{NULLABLE}(\beta)$, then $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$

Solution
Follow(P) = { $ } Follow(L) = { $ }
Follow(S) = { <id>, read, write, $ }
Follow(E) = { <id>, read, write, $ }

FIRST(\eps) \subset FOLLOW(E)

Step2: Follow the algorithm and build the Parse table

```
P ::= L
L ::= S L    |    \epsilon
S ::= <id>=E  |  read <id>  | write E
E ::= true    |    false
```

1. $\forall$ productions $A \to \alpha$:
   - 1.1 $\forall a \in \text{FIRST}(\alpha)$, add $A \to \alpha$ to $M[A, a]$
   - 1.2 If $\varepsilon \in \text{FIRST}(\alpha)$:
     - 1.2.1 $\forall b \in \text{FOLLOW}(A)$, add $A \to \alpha$ to $M[A, b]$
     - 1.2.2 If \$ $\in \text{FOLLOW}(A)$ then add $A \to \alpha$ to $M[A, \$]$
2. Set each undefined entry of M to `error`

If $\exists M[A, a]$ with multiple entries then grammar is not LL(1).

	<id>	read	write	true	false	$
P	P ::= L	P ::= L	P ::= L			P ::=L
L	L ::= SL	L ::= SL	L ::= SL			L ::= \eps
S	S ::= id=E	S ::= read<id>	S ::= write E			
E				E := true	E := false	

Step3: Convert parse table into a program

```
main() { P() ; }
P(){
    c = nextToken(); // lookahead
    if (c = id, read, write, $){
        L();
    } else if (c == true, c==false){
        Parse Error
    }
}

L() {

}
S()  {..}   E{..}
```

```
eat(token c){
    if (nextToken() == c){
        moveCursorToNextToken()
    } else
        parseError;
    }
}
```

Hashmap< Key =(Non-terminal,Terminal) , Value (RHS) >

HashMap<String,

s = Non-terminal + "%" terminal.
h.put (s, ..)

How do we decide that a grammar is not LL(1)?

If the parse table has more than 1 production rule in any cell, the grammar is not LL(1)

Part 2: LL(1) Academy

Part 3: Java Basics recap

1. Inheritance
2. Overloading and Overriding
3. Dynamic dispatch
4. Generics

Inheritance

```
class A{
    int f;
    int foo(){ … }
}

class B extends A{
    int g;
}
```

```
main(){
    B b = new B();
    y = b.f  // field A.f
    b.foo();  // A.foo()
}
```

Method Overloading

```
class A{
   void foo() { };
   void foo(int x) {} ;
   main() {
        foo(4);  // foo(int x)
  }
}
```

Operator overloading.
- define the operation '<' for non-integers.

Overriding

```
class A{
    foo() {.. }
}
class B extends A{
    foo() { … }  //overriding
}
B x = new B();
x.foo();  // B.foo
```

```
A y = new A();
y.foo();  // A.foo
```

Dynamic Dispatch

The target of a method call "a.foo()" is decided based on the run-time type of the **receiver-object "a"**.

```
class A{
   foo() {... }
   go(A a){ … }
   go(B b){ ... }
}
class B extends A {
   foo() { ... } //overriding
}
```

```
A x = new B() ;
x.foo(); // B.foo()
```

because
x was initialized
to be type B.

```
A x = new A() ;
A y = new B() ;
x.go(y); // A.go(A a)
```

because
we look at the dynamic type of 'x' and the static declared-type of 'y'

```
A x;
if (..){
   x = new A();
}
else ( .. ){
    x = new B();
}
x.foo();
```

Java Generics

Integer Max function

```
int max(int x, int y){
    if (x > y) return x;
    else return y;
}
```

Generic Max function

```
T max<T> (T x, T y){
    if (x > y) return x;
    else return y;
}
```

```
max<Double>(2.0, 3.0)
```

Java Generics

HashMap<Key-Type, Value-Type>

- put(key,value): O(1)
- get(key) : O(1)

```
HashMap<Integer, String> h = new HashMap<Integer, String> ();

h.put(3,"yes")
print(h.get(3)); //prints "yes"
if (h.containsKey(3)){ }..
```

Discussion Lec 3

Q] What is top down parsing vs bottom-up parsing (CORRECTION)

Top down parsing:
Starting with start symbol and keep expanding the non-terminals until you match the terminals in the string

Bottom up parsing:
Start with the terminals in the string and we try to synthesize the goal symbol from there
What kind is LL(1) parsing?: top-down parser

Today's discussion: HW2

Does MiniJava have interfaces?

Interface vs Superclass

```
interface A1{
    m();
}
class B1{
    n(){ .. }
}

class A2 implements A1 {
     m(){ … }
 }

class B2 extends B1 { … }
```

Q] What is the difference between a super-class and an interface?

1) You can't instantiate an interface.
2) Interfaces don't have instance fields. Fields have to be static and final.
3) Interfaces don't have method implementations typically.
4) Interface allows mutliplle inheritence

Java Generics

Operator overloading.
- define the operation '<' for non-integers.

Generic Max function

```
T max<T> (T x, T y){
    T a;
    if (x > y) return x;
    else return y;
}

double y = max<Double>(2.0, 3.0)
```

HashMap

- put(key,value): O(1)
- get(key) : O(1)

```
HashMap<Integer, String> h = new
HashMap<Integer, String> ();

h.put(3,"yes")
print(h.get(3));
```

Overloading

```
class A{
   void visit(String s) { };
   void visit(int x);
   int visit(int x);
   main() {
        int x = 4;
        visit(x);  // void visit(int x)
  }
}
```

A class can have methods with the same name but different signature.

Overriding

```
class A{
    foo() {.. }
}
class B extends A{
    foo() { … }  //overriding
}
B x = new B();
x.foo();  // B.foo()
```

If a subclass implements a method that was implemented in its superclass.

Dynamic Dispatch

The target of a method call "a.foo()" is decided based on the run-time type of the **receiver-object "a"**.

```
class A{
   foo() {... }
   go(A a){ … }
   go(B b){ ... }
}
class B extends A {
   foo() { ... } //overriding
}
```

```
A x = new B() ;
x.foo(); // B.foo()
```

because
x was initialized
to be type B.

```
A x = new A() ;
A y = new B() ;
x.go(y); // A.go(A a)
```

because
we look at the dynamic type of 'x' and the static declared-type of 'y'

HW-2 Problem Statement

Your main file should be called Typecheck.java, and if P.java contains a program to be type checked, then:

```
java Typecheck < P.java
```

should print either *"Program type checked successfully"* or *"Type error"*.

Javadoc

HW2 Main method

```
import minijava.MiniJavaParser;
public static void main(String[] args) {
      Goal root = new MiniJavaParser(System.in).Goal();
      if (root.accept(new TypecheckVisitor()))
          System.out.println("type-checks");
      else
          System.out.println("doesn't type check");
}
```

Simple strategy to deal with HW2

- Type check a single function. (20%)
- Single main class, and multiple. (40%)
- Type-checking for objects and classes (but no inheritance. no subtyping)
- Allow subtyping. (100%)

Why do we use the Visitor pattern?

Design1

```
class TypeCheckVisitor{
     boolean typeCheck(Goal root){
          Set<Node> nodesToVisit;
          nodesToVisit.add(root);
          while (!nodesToVisit.empty()){
               Node temp = nodesToVisit.get();
               if (temp instanceOf AssignmentStatement){
                    a = (AssignmentStatement) temp;  …   ;
               }
               else if (temp instanceOf Goal) { … }
               else {  error }
           }
```

Q] What are the issues with writing this way?
A]
1. Really untidy. Very unreadable.
2. a) It moves the error from compile time to runtime.
b) Casts have a performance cost

Design 2

```
class TypeCheckVisitor implements Visitor {
    visit (Goal n){
            visit(n.f0);
    }
   visit (MainClass n){ … }
   visit (AssignmentStatement n){
         rhsType = visit(n.f2);
    }
   visit(PlusExpression n){}
   visit (Expression n){
        if (n instanceOf PlusExpression){
             a = ( PlusExpression) n
             visit(a);
      }
```

Issue?

There is no dynamic dispatch for arguments.

So we have to resort to type casting.

x.foo()

Design3 (final design - Visitor Pattern)

```
class TypeCheckVisitor implements Visitor {
   visit (Goal n){
       n.f3.accept(this);
   }
   visit (MainClass n){ }
   visit(AssignmentStatement n){
         lhsType = n.f2.accept(this);
   }
   visit(PlusExpression n){ }
}
```

```
class MainClass{
     accept(Visitor v){
          v.visit(this);
     }
 }
```

```
class PlusExpression{
     accept(Visitor v){
          v.visit(this);
     }
 }
```

Visitor that just visits 2 kinds of nodes

```
class MyVisitor extends DepthFirstVisitor{
    @Overriding
     visit(MethodDeclaration m) { .. }
     @Overriding
     visit(VariableDeclaration) { .. }
}
```

SymbolTableVisitor extends DepthFirstVisitor

TypeCheckVisitor extend DepthFirstVisitor.

break. back at 5:04.

TypeChecking rule: PlusExpression

$$\frac{A,C \vdash p1 : int \quad A, C \vdash p2 : int}{A,C \vdash p1 + p2 : int}$$

$$\frac{p1 : int \ , \ p2 : int}{p1 + p2 : int}$$

TypeChecking Visitor Example

override vs extend

```
import cs132.minijava.syntaxtree.*;
import cs132.minijava.visitor.*;

class TypeCheckVisitor implements GJNoArguVisitor<String> {
     String visit(Goal n){ .. }
     String visit(MainClass n){ .. }
     String visit(PlusExpression n){ //n.f0 and n.f2 are ops
        if (n.f0.accept(this) == "int"){
            if (n.f2.accpet(this) == "int"){
                 return "int"
             }
         }
         error!;
```

Reading the MiniJava type system document

What simplifications in MiniJava help?

- There are no doubles, float, double arrays. Only int, bool, int arrays.
- No generics,
- No modulus, division operators
- No overloading
- No encapsulation for methods, variables. Everything is public\
- No type-casting
- No void return types.
- No static methods, no static fields, lambdas, reflection,
- Variable declarations in a methods are guaranteed to come before the statements.
-

Benefits of variable declarations before statements

1. We can populate the symbol table before checking.
2. We don't need to deal with block scoping.

(we still have method level scoping)

.

```
class A{
    int x;
    m(){
        int y;
    }
    go(){
        y = 3; // Type error in MiniJava
        x = 1; // Not a type error in MiniJava
    }
}
```

Symbol Table

What is it?

Map from variables to its type.

Why do we need it for type-checking?

To get the types for variables.

Think about the Symbol table design.

Discussion Lec 4

Proceeding with HW2

1. Make sure it works for a single function.
2. Make sure it works for multiple functions.
3. Make sure it works for multiple classes without inheritence.
4. Modify it to make it work with inheritence.

List of Possible Visitors

$$\text{parent}[\text{"A"}] = \text{"B"}$$
$$\text{parent}[\text{"B"}] = \text{"C"}$$
$$\text{parent}[\text{"C"}] = \text{null}$$

Assume we have a sequence of visitors to complete the type-checking.
If we have 1 visitor for type-checking statements and expressions. Will we need any visitors to run before that?

1. No overloading, acyclicity check in the 'extends' relationship.
2. Building class hierarchy (store information about super-classes)
3. Symbol table visitor (stores the types for all variables and fields)
4. Collect the list of methods, their formal parameters and return type.
5. No duplicates for classes, variables or methods.
6. Type-checking for statements and expressions.

Checking for acyclicity in the graph

$$parent["A"] = "B"$$
$$parent["B"] = "C"$$
$$parent["C"] = null$$

Design1 (O(n^2)):

```
getAllAncestors(){
}


for <class, parent> pair:
        ans = getAllAncestors()
        for a : ans
            a != parent
```

Design2 (DFS, O(n)):

Is this an example of Overloading, Overriding or none of them?

```
class A{
  void f (A a);
}
class B extends A{
  void f(); //type-error (overloading)
  void f(A a); //no error (overriding)
}
```

Answer: Yes,
y = newB()
y.f(new A());

```
class C{
    void f();
    void f(int
x);
}
/// Overloading
```

How to type-check for Overloading and Overriding?

For a given class, check its methods against the methods in all parent classes. When the method names match, they must have the same parameters and return type (overriding allowed) otherwise it is overloading (not allowed).

O(n^2)

Keeping track of current method/class in Visitor?

Why?
- For the symbol table (either storing or retrieving)
- to typecheck the 'this' variable

How?
- Whenever we are at visit(ClassDeclaration) we set a field called 'currentClass'
- You could pass it in the visit parameter

Side-note:
this.f, x.f - you can't access a field this way.

f - only way to access a field.

Symbol Table

What simplifications does the MiniJava grammar give us for the symbol-table?

1. The variable declarations come before the statements
2. No interfaces.
3. No nested classes

What would you have to do in real Java?

1. We would have to deal with scoping
2.
3. Deal with nested class scoping for fields.

Static vs dynamic symbol table.

Symbol Table Design - single main function

```
main(){
}
```

design?

hashmap<variable-name, variable-type>

Note: MiniJava doesn't have scoping.

What would you do if you had to deal with scoping? (assume there's only 1 function)

```
main(){
    int x;
    {
        int y;
        y = 2;
    }
    x = y;   // type-error

}
```

Use a stack of symbol tables/hashmaps, where the top of the stack is the closest/current scope and the bottom is the outermost scope. When we enter a new scope, add a hash map to the stack. When we leave a scope, pop the stack

Symbol Table Design - multiple functions, multiple classes

```
main(){
    int x;
}
foo(){
    bool x;
}
```

why won't our previous design work?

We need to distinguish between variables defined in different methods/classes.

new design?

Design 1:

hashmap<variable, type>
variable=(classname, methodname, varName)

Design 2: Multi-layered hashmap

level1: key=class, value=level2 hashmap

level2: key=method, value=level 3 hashmap

level3: key=varName, value=type

Symbol Table Design - with fields (assumption: single class with fields and methods)

Design: hashmap<fieldname, field-type>

What is the catch with fields?
- what if the field has the same name as a local variable or formal-parameter

field-names and method-names can overlap.

```
class A{
    int x;
    void x();
} // This is allowed. Not a type error.
```

Symbol Table Design - with fields (no inheritance)

```
class M{
    int x;
    int y;
    main(int z){
      bool x;
      y = x;
    //type-error
    }

    foo(){
     y =x;
    }
```

What is the catch with fields?
- what if the field has the same name as a local variable or formal-parameter

new design?

First check the symbol table for the local-variables.
If the variable is not there, then check the symbol table of the fields.

Note: child class could access the parent class's field.

Symbol Table Design - with fields and inheritance

```
class A extends C{
    int x;
}

class B extends A{
   boolean x;  // not a type error
    foo(){
       x = 3;
      //type error. x is boolean.
    }
}
```

What is the catch?

- we need to check the field-symbol -table of that class and all its superclasses.
- what if a field has the same name as its super-class.

A field can only be accessed within a class, or its subclasses.

At what points will we have to modify the symbol table

1. for the local-variable/formal-parameter hashmap
    - method-declarations and VariableDeclarations
2. For field-hashmap
    - field declarations

Any Questions? Any difficulties reading the doc?

- otherwise we can go over a few more rules.

Discussion Lec 5

Today's Lecture

1. Quick dynamic dispatch recap
2. Starting HW3
3. Discussing difficulties from HW2

Dynamic Dispatch

Only dynamic dispatch for the receiver object. No dynamic dispatch for fields, parameters.

```
class A{
   f ;
   foo() {}
   foo(A a) { }
}
class B extends A {
   f ;
   foo() {  }
   foo(B b) { … }
}
```

```
A x = new B() ;
x.foo(); //B.foo

because of
dynamic dispatch.
x is of type 'B' at
runtime.
```

```
A x = new A();
A y = new B();
y.foo(y);
// A.foo(A a)
// not foo(B b)
because
there's no
dynamic
dispatch for
parameters.
```

```
A x = new B();
x.f ; // A.f

because?
There's no
dynamic
dispatch for
fields.
```

Q] What are the targets? What features are involved?

```
class B{
    int z;
    f(){...}
    f(int x) {.... }
    g(){...}
}
class A extends B{
    int z;
    @Override
    f() {... }
    @Override
    h(A a)
    h(B b){...}
}
```

main(){
B x = new A()
x.g() // B.g() , inheritence
x.f() // A.f() , overriding
x.f(5) // B.f(int), inheritance, overloading
x.z // B.z, static type of the field-access
x.h(x) // A.h(B b), dynamic dispatch for receiver object, no dynamic dispatch for the parameter, overloading
}

- dynamic dispatch
- overloading
- overriding
- intheritence

Some Suggestions for HW3

- Start early. It is long.
- Test incrementally (First do it without Classes and Objects, then with, and so on)
- Use the Sparrow interpreter to run your translated Sparrow code.

What is an Interpreter?

- It executes statement line by line.

- The program is run on software, so it can run on any architecure

- Can be slower.

How is it different from a compiler?

- A compiler translates the entire program to machine-language, and this is run by hardware.

Running the Sparrow Interpreter

java -jar misc/sparrow.jar s < testcases/hw4/Factorial.sparrow

What are the differences between MiniJava and Sparrow?

Missing Constructs/Features in Sparrow

1) No classes, objects, fields
2) No loops
3) No expressions (in 3 address code)
4) No arrays
5) No types

New Constructs/Features in Sparrow

1) Labels
2) Goto
3) Heap

- error(s) is new
- explicit reference arithmetic (loads, stores)

(method becomes a function)

What Constructs/Features are still the same?

1) Variables are the same.
2) Functions are fairly similar.
3) Operations are still similar

Operational Semantics. What is it?

- Describes how a given instruction changes the state of a currently executing Sparrow program
- State = (program, current-instruction, heap, local-variables[environment])

Operational Semantics. Why do we need it?

- Mathematically precise (non-ambiguous) specification of what each instruction does.
- Write an interpreter
- Program Verification: You can reason about the execution and verify certain properties.

Is it necessary for translation?
- No

Is there just 1 possible translation?

No.
P(input) = output

Any translation, where
Translation(P)(input) = output

for every (input, output) pair.

Example: In a while loop we can jump if the condition is true or if the condition is false.

Starting the HW

```
public static void main(String[] args) {
      Goal root = new MiniJavaParser(System.in).Goal();
      Program p = root.accept(new TranslationVisitor()));
      System.out.println(p);
}

class TranslationVisitor implements GJVisitor<String>{
   visit(..){..}
    visit(..){..}
}
```

Visit Identifier

```
// Collect Program as String
String visit(Identifier n){
      return n.f0.tokenImage
}


// Collect Program as a Sparrow AST-Tree
cs132.sparrow.Node visit(Identifier n){
      return cs132.IR.token.Identifier(n.f0.tokenImage);
}
```

Translating a MultExpression by Hand

```
MiniJava: "y = e1 * e2"
Sparrow:

[c1]  // code for e1
t1 = … // stores the value of e1
[c2] // code for e2
t2 = .. // value of e2
t3 = t1 * t2
y = t3
```

lhs=t_3, op=t_4,

Visit MultExpression

```
global k // the next available temp-var
// Collect Program as String
String visit(MultExpression n){
   // operands in n.f0, n.f2 - PrimaryExp
   lhs = "t_" + k // k=3
   k++;
   op1 = "t_" + k // k=4
   c1 = n.f0.accept(this);
   k++;
   op2 = "t_" + k //
   c2 = n.f2.accept(this);
   code = c1 + c2 +
       lhs + " = "
       + op1 + " * "
       + op2 + "\n";
   return code;
}
```

Design options

St = cs132.IR.sparrow.Instruction

1) Pass down the next free temp variable.
2) Follow the 'k' numbering discipline strictly.

```
// Collect Program as AST
ArrayList<St> visit(MultExpression n){
   ArrayList<St> translation = new ArrayList<>;
   lhs = new Identifier("t_" + k);
   k++;
   op1 = new Identifier("t_" + k);
   c1 = n.f0.accept(this);
   k++;
   op2 = new Identifier("t_" + k);
   c2 = n.f2.accept(this);
   multStatement = new Multiply(lhs, op1, op2);
   translation.addAll(c1);
   translation.addAll(c2);
   translation.add(multStatement);
   return translation;
}
```

Translating a Conditional Statement by Hand

```
MiniJava: "if (e) s1 else s2"
Sparrow: ?

[c] // code for e
  // assume the return value is in t0
if0 t0 goto elseLabel4:
    [s1];
    goto end5
elseLabel4:
    [s2];
end5:
```

HW 2 Discussion.

Discussion Lec 6

Please start HW3 early

Today's class

- Translating array accesses
- Storing objects on the heap
- Translating field accesses
- Translating method calls

If you have requests for going over stuff from last time's discussion, do ask or type into chat. Each lecture here on out is timed to be 1.5 hours.

Translating Arrays

Mini Java	**Sparrow**
y = x.length	y = [x + 0]
x = new int[5]	y = alloc(24)
t = 1 y = x[t]	t0 = (t+1) t1 = t0*4 y = [t1 + 0]

Expanding on the definitions of compile-time (build-time) and run-time.

The compiler is also a program, and has to be built and run like all other programs.

Command	**Time**
gradle build	Compiler build-time
gradle run < InputProgram.java	Compiler run-time AND InputProgram build-time
java -jar sparrow.jar s < Input.s	InputProgram run-time

When we use the terms 'compile-time' and 'run-time', it usually refers to InputProgram's build and run time.

Array Bounds Checking

Q] At what time is the array bounds-checking actually done? i.e. If there is an out-of-bounds error, at what time will this error be thrown?
A] Run-time

Q] At what time is the code to do array bounds-checking written?
A] Compile-time

```
visit(ArrayLoad a){
// output code that does bounds check

// actual translation

}
```

What would happen if we skipped the Array Bounds Checking?

- We may get a segmentation fault
- Buffer overflow attack, buffer overread attack.

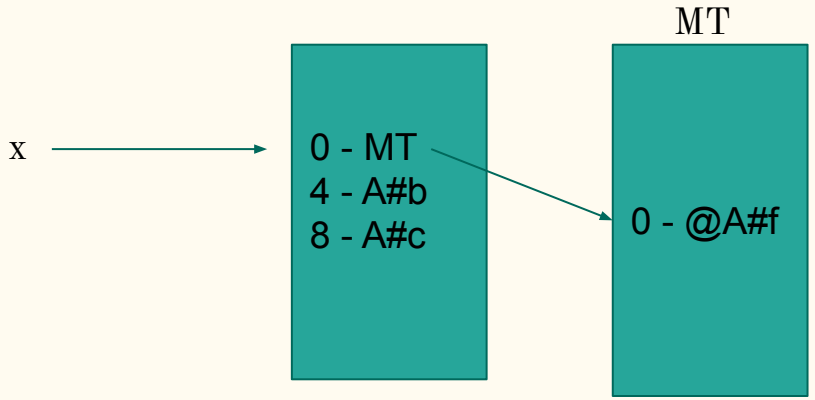
How do we translate a 'new' statement.

x = new A()

How an object is stored on the heap (no inheritance)

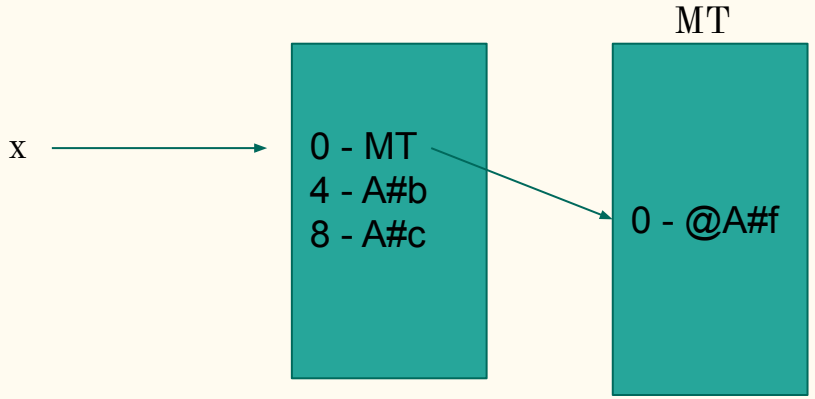
Mini Java

```
class A{
        // 0 - MT
         int b;  // 4
         int c;  // 8
         f(){
              c = 4;
         }
}
```



Translating "x = new A()" (no inheritance)

```
// Allocated space for x
t1 = 12
x = alloc (t1)
// Allocated space for MT
t2 = 4
mt_x = alloc(t2)
// Storing A#f at the right location.
[mt_x + 0] = @A#f
// Optional - (Fill in the initial value of 0 for b and c)
// Store a pointer to mt_x in x
[x + 0] = mt_x
```



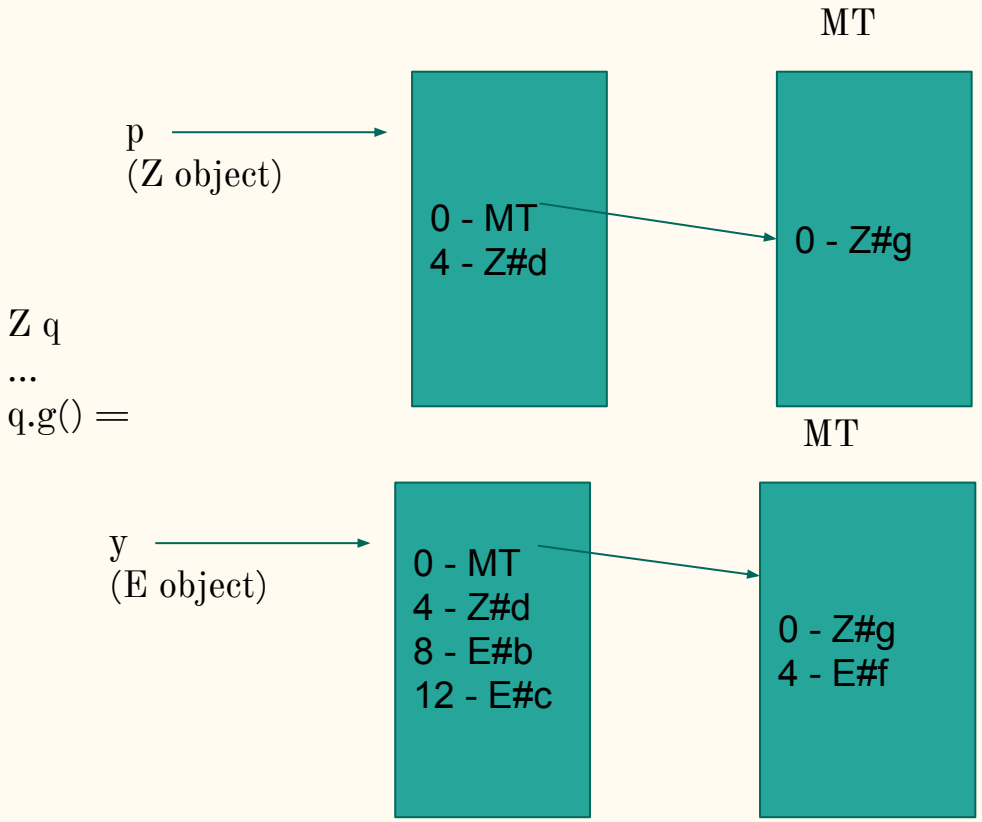
How do we translate a 'new' statement.

y = new E()

How an object of class E is stored on the heap (with inheritance)

Mini Java (with offsets for an object of class E)

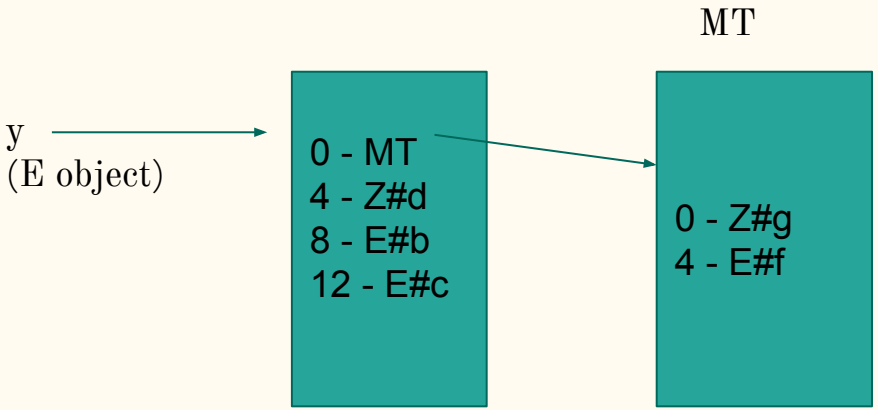
```
class Z {
      int d;
      g() {   d = 3;  }  // 0
}

class E extends Z{
          int b;
          int c;
           f(){
               d = 1;
               c = 4;
           }
}
```



Q] What would happen if we stored the fields of 'E' before the fields of 'Z'?

Translating "y = new E()" (with inheritance)

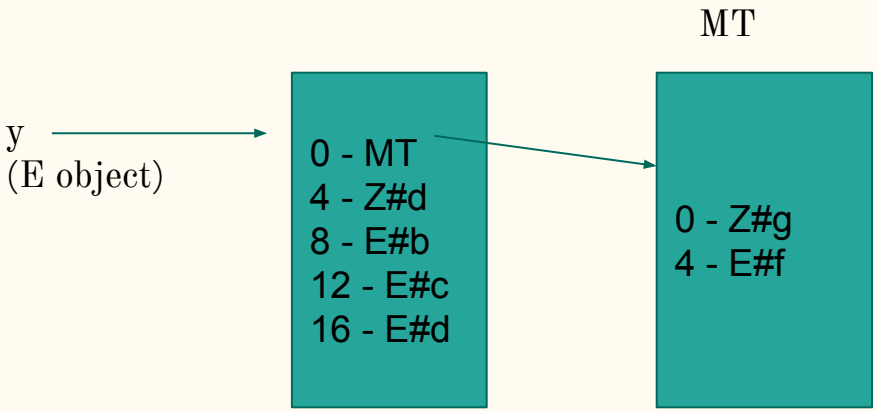
```
// Allocate space for the object
y = alloc(16)
// Allocate the method table
mt_y = alloc(8)
// Populate the method table
[mt + 0] = @E#g
[mt + 4] = @E#f
// Link the method table
[y + 0] = mt_y
```



What if we had a repeated field name (hard case)

Mini Java (with offsets for an object of class E)

```
class Z {
      int d;
      g() {   d = 3; // Z#d }
}

class E extends Z{
          int b;
          int c;
          bool d;
           f(){
               d = 1;  // E#d
               c = 4;
               g();
           }
}
main() {   y = new E() }
```



Translating Class Fields (with inheritence)

Mini Java	**Sparrow**

class Z {
  // 0 - MT
  int d; // 4
  g() { d = 3; }
}
 	
class E extends Z{
// 0 - MT
  int b; // 8
  int c; // 12
  f(){
  d = 1;
  c = 4;
  }
}
	// Translate d=3
	// object in pointer y
	[y + 4] = 3
	
	// Translate d =1
	// object in pointer y
	[y + 4] = 1
	
	// Translate c = 4
	[y + 12] = 4

Translate methods f() and g()

Translating Class Fields (with repeated field name) - tricky corner case

Mini Java	**Sparrow**

class Z {
  int d; // 4
  g() { d = 3; }
}
 	
class E extends Z{
// 0 - MT
  int b; // 8
  int c; // 12
  int d; // 16
  f(){ // [0,4]
  d = 1;
  c = 4;
  }
}	

'this' keyword and Method call (assuming you have a pointer to the method)

Mini Java	**Sparrow**
Method "int foo(int y)" in class A{	func <classname>#foo(thisPointer y)
= x.foo(z) // Assume that a pointer to method 'foo' is stored in variable 'w'	= call w (x z)
y = this	y = thisPointer

w = @f

Class A object

<expression>.foo()

Getting a pointer to the Method call (x.foo())

Case1: assume there is only 1 class (without method tables).

w = @foo
call w (..)

Case2: Multiple classes and no inheritence. (try without method tables). Why would the case1 solution not work?

// class = type returned by your type-checker
w = @<class>#foo

We still need the type-checker to know what offsets to use.

Getting a pointer to the Method call (x.f())

Case 3: We have inheritence. Why would the case2 solution not work?

```
// LOAD: Load the method table
mt = [x + 0]
// LOAD: Load the function from the correct offset
w = [mt + 0]   // we know to use offset 0 because x is of static type A
// CALL:
call w(x y)
```

Load, Load, Call

Some supplementary questions

Q] Do you have to copy the method table for an assignment statement?

Say the statement to translate is

E x;

Z y;

...

"y = x" // y is of static-type Z, x is of static-type E

Answer] y = x

Mini Java (with offsets for an object of class E)

```
class Z {  // 0 - MT
      int d; // 4
      g() {   d = 3;  }
}

class E extends Z{
          int b;  // 8
          int c;  // 4
           f(){
               d = 1;
               c = 4;
           }
}
```

Q] Why don't we need the equivalent of method tables for fields? Why are they treated differently?

1. We don't want to override fields
2. We don't want dynamic dispatch. - we already know exactly offset the field is at.

Discussion Lec 7

Plan for today

- Wrap up HW3
- Beginning HW4

Due Dates:
HW-3 - Nov 14
HW-4 - Nov 28
HW-5 - Dec 5 (Sunday before finals week)

HW3 Discussion

Null Pointer Exceptions

Q] Can field accesses throw Null Pointer Exceptions in Java?
A] Yes. z.x.x would be an example

Q] Can field accesses throw Null Pointer Exceptions in MiniJava?
A] No. Because the object of a field access is always implicitly 'this' which can never be null.

```
A x;
..
x.foo(); // can be NPE : need to do a check, and throw an exception if it is null.

int [] x;
..
y = x[3]; // can be a NPE. We still need to make a check.

// Uninitialized field read
class A{
   A x;
}
z = new A()
z.x.foo(); // can be a NPE. x is uninitialized.
```

Making the null pointer check

```
// MiniJava
x.foo();
```

```
// Sparrow

if0 x goto null_error
// Load MT
mt = [x +0]
// Load method from MT
foo_method = [mt + 0]
// Call it
call foo_method(x)

null_error:
        error("null error")
```

What happens if we do not perform the null-pointer check?

- We would end up accessing memory that we are not supposed to.

Are there languages that do not perform the null-pointer check?

Q] What all information do we need in the type-checker?

Why do we need a type-checker

```
e.foo();
// we need to know the type of
expression 'e'
```

Type-checker contents

- types of all local variables and parameters.
- type-checker to evaluate expressions. More specifically we only need to evaluate expressions which result in a user-defined type (eg. new A(), method-call)

If you pass pregrade but are unable to reach 100%

- Think about what constructs are not adequately tested by the current test-suite.
- Create your own test cases with complicated stuff - use lots of inheritance, use large expressions with multiple operations, etc. (f().g(h(3));
- Luckily, since you are using Java, you can just run the program to get the expected output.

HW4 Discussion

(any questions on HW-3?)

How is SparrowV different from Sparrow?

1) Register variables
    - a2-a7: Pass first 6 parameters
    - s1-s11: callee save registers
    - t0-t5: caller save registers

2) In Sparrow we just had variables as is from MiniJava. But in SparrowV, everything in a variable is implicitly on the stack.

3) You can't directly perform operations on variables. You need to move variables into registers to perform operations on them.

Calling functions: pass arguments in a2-a7

```
// sparrow
x = @foo
call x(x1,x2,x3,..,x7)
```

```
//sparrowV
t0 = @foo
a2 = x1
a3 = x2
a4 = x3
a5 = x4
a6 = x5
a7 = x6
call t0 (x7)
```

```
func foo(p):
    a2 //first parameters
```

If we have more than 6 parameters, we explicitly write the parameter. The understanding is that those are on the stack.

Addition in SparrowV (Assume no register allocation)

```
 //sparrow
x = y + z
...
// sparrowV  (t0, t1, t2)
t0 = y
t1 = z
t2 = t1 + t2
x = t2
```

Why do Register Allocation

- Stack is in memory, and it is much slower than accessing registers.

```
x = @foo
call x
```

Caller Save and Callee Save registers

- caller save means that the function calling is responsible for saving that register (t0 - t5)
- callee save means that the function being called is responsible for saving. (s1 - s11)

Q] How do we caller save?

```
stack_save_t0 = t0
stack_save_t1 = t1
...
foo()
t0 = stack_save_t0
```

```
foo()
    stack_save_s0 = s0
    ...
    stack_save_s11 = s11
    <code>
    s0 = stack_save_s0
    ..
    s11
    ret x
```

Save and restore parameter registers

```
// Sparrow
foo(x)
    .. = x
    go(y)
    ..
    ret
```

```
// SparrowV
foo()  // x is in a2
    .. = a2

    stack_save_a2 = a2
    a2 = y
    go()
    a2 = stack_save_a2
    ..
    ret
```

Register Optimizations discussed in class

1. Do some register allocation (even random is better than none)
2. Pass arguments in registers
3. Don't save registers that you don't use. (both for the 't' and 's' registers)
4. Don't save and restore 't' registers which are not live across the call.
5. Don't save and restore 'a' registers which are not live across
6. We can assign 's' and 't registers to stack parameters and return values.
7. Linear Scan Register allocation

You only need 2 temporary registers, not 3.

You don't need temporary registers, if nothing is on the stack.

save argument registers to another 's' or 't'

t0, t1, t2 are temporary registers. We are not going to use them for register allocation.

```
// SparrowV code
foo(x7 x8)
    s0 = x7 // load
    s1 = x8  // store

    s0 = s0 + s1
    s0 = s0 + s1
```

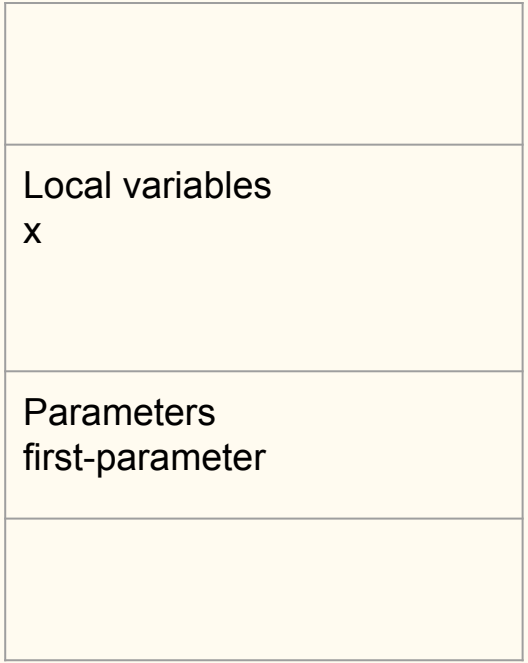
```
// Sparrow
foo(x7 x8)
    x7 = x7 + x8
    x7 = x7 + x8
```

Counting the Loads and Stores

Normally
x = s3 // 1 load
s3 = x // 1 store

In a method call
stack_save_t0 = t0 // 1 store
s5 = @f
s4 = s5 (x1,.. xn) // n load + n store (to copy
// x1,..xn into the location of the parameters)

t0 = stack_save_to // 1 load



Show use of 'stats'

Grading for HW4

- 20% for correct outputs (like passing all private test-cases)
- A SparrowV program that doesn't give the correct output gets 0
- Remaining 80% allotted based on how few loads+stores you do.
- Modifying Heap loads+stores gives you 0 for that test case

What about fields? Should they be allocated to registers?

- No. Fields are in the heap.
- Only local variables and parameters get registers

Discussion Lec 8

Null Pointer Exceptions (correction)

Q] Can field accesses throw Null Pointer Exceptions in Java?

A] Yes. The example for this is 'z.x.x'. The second .x will be a field access throwing a NPE.

Q] Can field accesses throw Null Pointer Exceptions in MiniJava?

A] No. Because the object of a field access is always implicitly 'this' which can never be null.

```
A x;
..
x.foo(); // can be NPE : need to do a check, and throw an exception if it is null.

int [] x;
..
y = x[3]; // can be a NPE. We still need to make a check.

// Uninitialized field read
class A{
   A x;
}
z = new A()
z.x.foo(); // can be a NPE. x is uninitialized.
```

Question about uninitialized local variables.

```
int x;
System.out.println(x)
```

```
A x;
x.foo()
```

Q] Should these throw null-pointer errors at runtime?
A] 'javac' throws a compile time error for both these cases.

To catch such errors at compile time, one needs to perform an analysis to check that all uses of a variable have a valid initialization. Since we did not discuss this in class, we do not have any test-cases in which you will have to deal with this.

But you still need to do null pointer checks

```
a[3] = 4
x = a[3]
```

```
x.foo()
```

Plan for today

- Two designs for outputting code.
- Fast Live Interval Analysis
- Linear Scan Implementation

Starter Code

```
Registers.SetRiscVregs();
try {
    cs132.IR.syntaxtree.Node root = new SparrowParser(System.in).Program();
    SparrowConstructor sc = new SparrowConstructor();
    root.accept(sc);
    Program p = sc.getProgram();
    p.accept(MyVisitor()); // then your code goes into the MyVisitor class
} catch (ParseException e) {
    e.printStackTrace();
}
```

Two Possible Designs for Outputting code

```
// Raw output
String visit (Add n){
    return (n.lhs + "=" + n.arg1 + "+" +
n.arg2 + "\n")
}
```

```
// SparrowV AST tree
List<cs132.IR.sparrowv.Instruction> visit
(Add n){
    x = new cs132.IR.sparrowv.Add(.. );
    y = new ArrayList<..>
    y.add(x)
    return y;
}
```

The repeated class-name problem for Sparrow, SparrowV

```
import cs132.IR.sparrow.Add;

void visit(Add n){
      cs132.IR.sparrowv. Instruction i = new cs132.IR.sparrowv.Add(new
Register("s1"), new Register("s2"), new Register("s3"));  // use fully qualified
name.
    statementList.add(i)
}
```

Translating from Sparrow to SparrowV

```
// Sparrow
f = @foo
x = 3
x = x + z
call f()
```

```
// SparrowV w/o Reg-alloc
t0 = @foo
f = t0
t0 = 3
x = t0
t0 = x
t1 = z
t2 = t0 + t1
x = t2
t0 = f
call t0()
```

```
// SparrowV with Reg-alloc
// f - t3
// x - t4
// z - t5
t3 = @foo
t4 = 3
t4 = t4 + t5
call t3()
```

Caller Save and Callee Save registers

- caller save means that the function calling is responsible for saving that register (t0 - t5)
- callee save means that the function being called is responsible for saving. (s1 - s11)

Q] What happens if we caller save all 's' and 't' registers? Is that okay?
A] There is an issue. Your code won't be interoperable with code compiled with another compiler

```
// Your library compiled with your compiler
libcall(){
    //save s0
     s0 = ..
     // restore s0
}
```

```
// Code compiled by someone else's
compiler
// not save s0
x.libcall()
// not restore s0
```

Temporary Registers

Q] Why can't we use all 17 registers for Register Allocation?
A] We need at least 2 temporary registers to operate on variables that are in the stack. So we have 15 registers for register allocation

// to perform (y+z), if both y and z are on the stack.
t0 = y
t1 = z
t0 = t1 + t0

Q] Which registers should we skip for Register Allocation? Does it matter?
A] Ideally want to use t0, t1 and t2 for temporaries.

Example translation for parameters

```
// Sparrow code
// f points to foo
call f (x)


func foo(thisPointer)
    y = thisPointer
```

```
// SparrowV code
// f points to foo
t0 = f
x = a2
call t0 ()


func foo()
    y = a2
```

Temporary register: one that you use to operate on stuff which is on the stack.

"proper" register: one that is bound to a single variable in that function, and it simply replaces all instances of that variable with this register.

List of passes for HW4?

1) Liveness Intervals
2) Linear Scan Register allocation (map: local-variable -> reg)
3) Translation

Live-Ranges vs Live-Intervals

line1: x =1
line2: y = x
line3: z = 3
line 4: x = 3
line 5: z = x

LiveRanges(x) = {(1,2) , (4,5) } - computed in the iterative Liveness Analysis
LiveInterval(x) = (1,5)

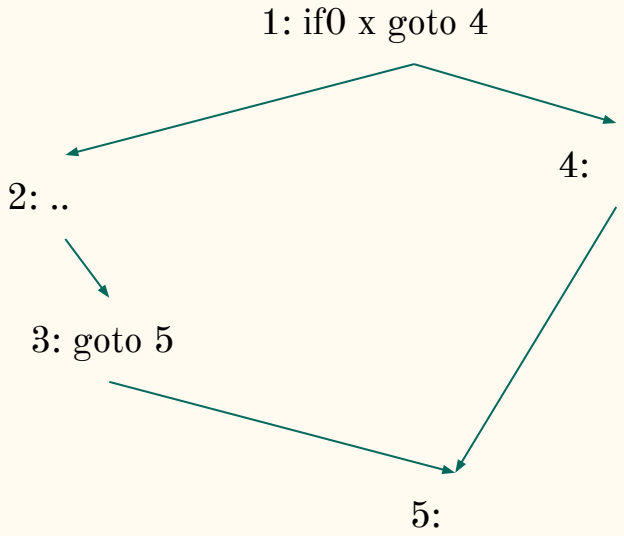
Overapproximation: less precise, more conservative(safe)

List of Steps required to do the Iterative Liveness Analysis? Any extra steps to use it for Linear Scan?

1) Build Control-flow graph
2) Initialize our In, Out, Use, Def
3) Generate constraints
4) Solve the constraints
5) Collapse the Live Ranges in Live Intervals

Control-flow graph

1: if0 x goto 4:
2: ..
3: goto 5
4: ..
5.:



Fast Live-Interval Analysis (from the paper)

- An over-approximation of the liveness-analysis, but for intervals instead of ranges
- Simpler, linear time. (Liveness analysis was O(n^4). Possible to show that it is O(n^2)
- Is sufficient to get 100% percent for HW4.

Algorithm

1. Live-interval = First definition to the last use.
2. If the last 'use' is in a loop, then modify the last 'use' to be the last statement of the outermost-loop that it is a part of.

Example 1

```
line1: x =1
line2: y = x
line3: y = foo(1)
line4: x = 3
line5: z = x
line 6: for () {
            for ( ) {
                = x
             }
Line9:       ...
Line10: } // x stops being live here
```

Example2

```
x = 1
y = 2
l1:
if0 y goto end; //while(y) {
z = x
t = z
y = t
jump l1:   // }
end:
```

Live-interval(x) : Line1- Line8

Example3 (2 nested loops)

```
loop1_start:
if0 .. goto loop1_end:
x = 1
..
        loop2_start:
        if0 .. goto loop2_end;
        ..   = x
        goto loop2_start
        loop2_end:
...
goto loop1_start:
loop1_end:
```

How to find the end of the loop where a variable is used?

1. Record instruction indices of all the labels. (map: label -> instr.index)
2. Find all backward gotos, and record the instruction-index that they jump to.
3. Make a list of all loops (eg. loopList = [(2, 4), (1,6)])
4. When you come across a variable - find the outermost loop that its instruction index is a part of by going through the list of loops.

Caller and callee save regs

```
// save caller-save regs
y = foo()
// restore caller-save regs
..
```

```
foo(){
// save callee-save regs
...

// restore callee-save regs
}
```

Register Allocation basics

Q] How many registers do you have for register allocation using Linear-scan or whatever?
A] 15. Save 2 for temporary registers.

Q] What local variables/parameters can you allot using register allocation? Which can you not?
A] yes. Allot registers to everything.
Even return values and parameters which are passed through the stack can be assigned a register.

```
foo( x7)
 s1 = x7
 // use s1 wherever x7 is used.
```

Linear Scan Register Allocation

1. Order registers left to right by liveness start time
2. Start tentatively assigning registers by start time (if there are enough registers). If a live range ends, that assignment becomes final and you can reuse the register.
3. If you don't have enough registers, push the longest lived (latest liveness end time) "tentative" variable to the stack.

(For simple examples, go over the lecture again)
Are there any non-trivial statements?

Dealing with If0 and Goto statements

line1: x =1
line2: if0 x goto line6
line3: y = x
line4: x = x + 1
line5: goto line2
line6: z = x

What modification do we need?

No modifications are needed. So we are already operating at the level of intervals - we don't have to worry about if0 and goto statements.

Q] Should we save argument registers also when we make a function call?

A] Yes.

```
go()
  // first parameter of go is in a2
// save argument registers
a2 = x
t0 = call foo()
// restore argument registers

z = a2 // need to access first parameter of go
```

2 uses of Liveness intervals

1. Linear scan
2. Optimization to save only live variables across a function call. ('t' and 'a')

Q] We saved only those caller-save registers which are live after the function call. Can we do the same for callee-save registers when we are in the callee?

Can the Callee not save some callee-save register (s1-s11) which is not used by the caller, but is used by the callee?

```
g(){
// doesn't use s1
foo()

}
```

```
foo(){
// can we choose to not save s1?
  = s1

}
```

- Why won't this optimization work?
1.
2.

Q] Some Optimizations we should avoid?

1. Consider all registers as caller-save.
2. Use unused argument registers as caller-save registers.

Q] Can you think of any other register optimizations?

Discussion Lec 9

Plan for today

- HW5 (first 1.5hrs)
- HW4 questions (last 0.5 hour)

System Calls

Q] What are they?
A] Function calls that defers control to the OS, and runs something in privileged mode.

Q] Examples?
A] File I/O, allocating memory, multi-threading, printing to the screen

What is RiscV?

(Is an assembly language) It is an ISA (Instruction Set Architecture). It is an interface between the software and hardware.

RISC : Reduced instruction set architecture. Only memory operations are loads and stores.

Q] Other languages in this set?
A] (other ISAs) x86 (CISC), ARM (RISC)

Starter code

```
main(){
        try{
                Registers.SetRiscVregs();
                Node root = new SparrowParser(System.in).Program();
                SparrowVConstructor svc = new SparrowVConstructor();
                root.accept(svc);
                Program p = svc.getProgram();
                p.accept(new MyVisitor());
        } catch ( .. ) { .. }
}
```

Running the Venus simulator

java -jar misc/venus.jar < a.riscv

https://github.com/kvakil/venus/wiki

Differences between SparrowV and RiscV?

1. We manually have to manage the stack memory (no variables anymore)
2. We have these "mv" and "sw" instructions. All RISCV
3. Function calls are made through JAL instructions (explicit jumps)
4. We have these Assembler directives.
5. System calls

RiscV Reference Card and RiscV Specification

See CCLE

RiscV directives

See examples in Factorial.riscv

https://github.com/kvakil/venus/wiki/Assembler-Directives

ecall = system call (see venus documentation)

System calls that we are concerned with:

error
print
alloc

Q] Are the in[0], in[1], etc. same as a2,a3,.. ? A] NO

Stack management (stack grows downward)

NORMAL OPERATION

```
          ..
        4(fp) in[1]
fp -> 0(fp) in[0]
        -4(fp) return-address
        -8(fp) previous frame pointer
        -12(fp) local-var1
        -16(fp) local-var2
sp -> ..
```

function foo(arg7, arg8){ ... v1 v2 v3}

Stack management (stack grows downward)

JUST BEFORE FUNCTION-CALL

..
4(fp) in[1]
fp -> 0(fp) in[0]
-4(fp) return-address
-8(fp) previous frame pointer
-12(fp) local-var1
-16(fp) local-var2
-20(fp) out[1]
sp ->-24(fp) out[0]

function foo(arg7, arg8){ ... v1 v2 v3}

Stack management (stack grows downward)

JUST AFTER FUNCTION-CALL

..
4(fp) in[1]
fp -> 0(fp) in[0]
-4(fp) return-address
-8(fp) previous frame pointer
-12(fp) local-var1
sp ->-16(fp) local-var2

out, out, ret, call

ret, out

$t0 = \text{call foo}()$

Function Call - caller

Before the call

1a. move sp to make space for args
1b. store the args there

2. store the return address (in ra) (done automatically)

3. jump to the callee's label.

After the call

1. move the return value from a0 to wherever you need it.
2. Reduce the space for the out arguments

Function Call - callee

At the start of a function

1. Store the old frame-pointer
2. set your fp = old sp
3. Set sp with enough space for locals, ret-addr, parent-fp
4. Store return-address of your parent.

1 before 2 (otherwise we lose the parent's fp)
2 before 3 (otherwise we lose the parent's sp)

At the end of the function

1. Store the ret-value in a0
2. Set sp back to fp
3. Set fp to the parent-fp

4a. put return-addr back in 'ra'
4b. jump to 'ra'

Q] What about saving and restoring registers?

A] No we don't need to explicitly save and restore registers for HW-5. You already did this in HW-4. So you can assume that the input program already does this for you.

The distant-branch issue

ERROR: branch to distant_branch too far on line 236

Q]What does this error mean?
A] "too far" would be caused by distance more than what the immediate field can encode. Maximum branch distance that can be specified in the immediate operand for a BEQ instruction is '2^12' (4096 bytes = 1024 instructions)

if0 s3 goto label

beqz s3 label

Fixing the distant-branch issue

beqz s3 label

(Solution: We will make use of the JAL instruction, which can jump to 2^20 distance)

```
beqz s3 l1
jal l2
l1: jal label
l2:
```

Discussion Lec 10