CS 132 Notes

Yash Lala yashlala@gmail.com

2021 fall semester

First Half of the Course: Implementing the Basics

2021-09-23: Introduction + Lexical Analysis

This section covers Appel textbook chapters 1-2, big class PDF notes chapters 1-2.

- *Compiler*: Transforms executable code in one language into executable code in another language.
- *Interpreter*: Transforms executable code in one language into the *results* of running that code.

Chapter 1: Anatomy of a Compiler

Compilers aren't monoliths. Usually, separated into a front end and a back end. Front end turns source code into *IR* (intermediate representation), back end turns IR into machine specific code. Separation gives us adaptability; now different frontends (one per language) can use same backend, and different backends (one per architecture) can use the same frontend.

The Front End: Comprised of a *scanner* and a *parser*.

Scanner (also called *lexer* or *tokenizer*): Converts input into a list of *tokens*. Also strips out whitespace and comments.

Parser: Receives a list of tokens from the scanner. Given a context-free grammar (remember CS 181!), converts the list of tokens into a *parse tree* expansion. Using this tree, constructs IR. Parser also can do type checking and error correction.

The Back End: Comprised of an *instruction selector* and a *register allocator*. Goal is to translate the IR to machine code, choose what to keep in registers. Much harder to automate.

Instruction Selection: Given an efficient IR representation, produce compact and fast machine code. Use all available addressing modes, etc of the machine. This is mainly a pattern matching problem.

Register Allocation: We only have so many registers on the machine. Frequently used values should be stored in the registers; reduces to an optimization problem. Very hard.

Many compilers will also have some optimizing middle components (*middle end*). They make sets of passes over the IR, optimizing the code.

Chapter 2: Lexical Analysis

Scanners (= Lexers, Tokenizers) are straightforward; they perform regexp based matching. Remember CS 181 – we can recognize regexps with table-based finite automata.

What if the grammar cannot be recognized by a regexp (ie. nonregular language)? Eg. $\{ab, aabb, aaabbb, \ldots\}$ is not a regular language. People want to use the existing tools, so they avoid languages with nonregular identifiers.

There are tools to go from regexp to NFA/DFA based recognizer automatically. These are called *scanner generators*; given an input regexp like language (eg. "Numeric Literals match `(0|[1-9][0-9]+)`"), they output the source code for a scanner. The scanner takes an input string, and outputs a list of tokens (as C structs or whatever – passed directly to parser). Standard UNIX `lex` program does this.

2021-09-28: LL Parsers

Chapter 3: Parsing

Parsers receive strings of tokens (eg. `<Integer, 9>`) from the Tokenizer. Their job is to transform this into IR, while running some language-specific checks (types, etc).

They do so by building a *parse tree*. Eg: the list (`1`, `+`, `2`) should be interpreted as a tree. The `+` operator should have `1` and `2` as left and right children, respectively. This is equivalent to finding a CFG expansion that yields the list of input tokens.

As before, there's no real reason that a language has to be context fee. People do it so they can reuse the tools available to ease writing parsers. First, some review of grammars.

A CFG is $G := (V_t, V_n, S, P)$, where:

- $V_t :=$ List of terminal symbols.
- $V_b :=$ List of nonterminal symbols.
- $V := V_t \cup V_n =$ Vocabulary, list of symbols.
- $S :=$ Start symbol (nonterminal).
- $P :=$ A rule set of *productions*, specifying how nonterminals can be expanded.

And some terms:

- β is a *sentential form* of G if $S \implies {}^*\beta$.

We'll be using some notation conventions:

- $a, b, c \in V_t$
- $A, B, C \in V_n$
- $U, V, W \in V$
- $\alpha, \beta, \gamma \in V$
- $u, v, w \in V_t$

We're using the Kleene star notation above.

When expanding a list of nonterminals to try to match a given fragment (ie. we have $S \to E + T$ and want to match `3 + 2`), we choose who to expand first:

- Leftmost Derivation: always expand the leftmost symbol on our list of nonterminals.
- Rightmost Derivation: always expand the rightmost symbol on our list of nonterminals.

What do we do if we want an order of operations? We can enforce this in the grammar. Make a heirarchy of nonterminals, and put the higher precedence operator higher on the tree. For example,

```
Mult ::= AdditionClump "*" Mult
AdditionClump ::= Num + AdditionClump
Num ::= [0-9]+
```

Now once our Mult symbol "degrades" to an AdditionClump, it can never be promoted to a Mult again; in effect, enforcing an order of operations (expansion of parse tree will never have an add with 2 Mult terms as children).

What do we do if our grammar is ambiguous? Eg.

```
if x then if y then else z
                    ----
                      \_ who does this "else" belong to?
```

It's very annoying. Even figuring out if grammar is ambiguous is undecidable. Sometimes we just leave it for later in the compiler pipeline, although we can sometimes fix it in the grammar (eg. overloaded operations). See the notes for more.

2 broad approaches to parsing:

1. Top Down. Start at the root of the derivation tree, try possible expansions. If they choose a wrong expansion path, they hmay have to backtrack and try another option (although this may not happen depending if we're dealing with a *predictive* grammar).
2. Bottom Up. Start at the leaves of the derivation tree. Encode the possible trees that a symbol could be part of; as we keep consuming input, eliminate possibilities, then give the output. Look at slides.

Top down parsers have some problems. Given the following *left-recursive* grammar, a top-down leftmost parser will never terminate.

```
Expr ::= Expr + Num
```

We'll just keep expanding `Expr` forever. We don't want this.

Formally, a leftmost derivation parser can't handle left recursion. Left recursion is defined as $\exists A \in V_n \mid A \implies {}^*A\alpha$ for some $\alpha \in V$.

We can solve this by converting the left recursion to right recursion. Leftmost parsers handle right recursion just fine (they'll expand Expr, but the first term won't match, so they'll just give up). $A \to A\alpha \mid \beta$ matches "β, then any number of αs". Rewrite it as

$$\begin{aligned} A &\to \beta A' \\ A' &\to \alpha A' \mid \epsilon \end{aligned}$$

Great, now we know our parser will terminate.

We want parsing to be fast. We want it to operate in linear time, so we choose types of grammars that will permit this. Backtracking is a threat to linear time, so let's choose a grammar that will permit this; a $LL(1)$ grammar.

- $LL(1)$:= Left-to-right reading, Leftmost derivation, 1 character lookahead (we only need to look at the next character to determine the expansion we should take; no backtracking needed).
- $LR(1)$:= Left-to-right reading, Rightmost derivation, 1 character lookahead. These can be significantly more powerful and complicated, but we're not going into them yet. Read the book.

Let's try to convert our grammar to LL(1).

How do we avoid backtracking? We need it to be obvious at all points what the next possible expansion would be. Ie. if we see another symbol coming up, the expansion choice should be unambiguous. What if this isn't the case? Suppose:

$$A \to a\alpha_1 \mid a\alpha_2 \mid a\alpha_3 \mid a\alpha_4 \dots$$

We can prevent backtracking by *left factoring* (basically, create an unambiguous intermediate state). The above grammar becomes:

$$\begin{aligned} A &\to aA' \\ A' &\to \alpha_1 \\ A' &\to \alpha_2 \\ A' &\to \alpha_3 \\ &\dots \end{aligned}$$

Great. Now we have removed left recursion and left factoring. We do have a case to consider; if a nonterminal can go to ϵ, then it might get confusing how to parse it. Did it go to epsilon, or not? To avoid lookahead, we intuitively want it to be obvious when we should make a symbol disappear. More formally,

Suppose our symbol disappeared (expanded to ϵ. Then we would run into some terminals from the *next* nonterminal we have to expand (ie the "follow set" of our nonterminal). Suppose the symbol didn't disappear.

Then it would expand into some terminal symbol (eventually). We want these two sets to be disjoint. We have some math on paper and in the Lecture Notes, but idea is simple.

LL(1) Theorem: A grammar is LL(1) if for every set of productions $A- > \alpha_1 \mid \alpha_2 \mid \ldots \alpha_n$,

1. $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset \; \forall \; i \neq j$.
2. If $\text{NULLABLE}(\alpha_i)$, then $\forall 1 \leq j \leq n, j \neq i$, we have $\text{FOLLOW}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$.

No left recursive grammar is LL(1), and no ambiguous grammar is LL(1). It's undecidable if we can make a grammar LL(1) just by rearranging it using the tricks above.

How do we figure out the LL(1) parsing table for an operation by hand? You can try visiting LL(1) Academy. Use the first sets and follow sets to avoid a bunch of recursive problems. In order:

We calculate the first set of an expression as follows:

1. If `X` is terminal, `first(X)` is `X`.
2. If $X \rightarrow \epsilon$, add ϵ to first(x).
3. If $X \rightarrow Y_1, Y_2, \ldots$, then:
   - If $\epsilon \notin \text{first}(Y_1)$, then $\text{first}(X) = \text{first}(Y_1)$.
   - If Y_1 is nullable, add everything from $\text{first}(Y_1)$ except ϵ. Now add $\text{first}(Y_2)$ (recursion!).
   - If all of the Y_i are nullable, add ϵ.

And the follow set ... as follows:

2021-09-30: JavaCC + JTB

Not re-taking notes here. He goes over the visitor pattern, it's pretty straightforward.

2021-10-05: Semantic Analysis + Types

Chapter 5: Semantic Analysis

Once we have an abstract parse tree, we need to perform *Semantic Analysis* (verifying meaning). Connect variable definitions to their uses, make sure all types match up, etc.

Suppose an operation uses a variable (`a+b`). We need a way to look up the variable and it's type. This is accomplished through *symbol tables* (also called environments). When we declare a new variable, we add an entry to the symbol table. When we encounter an "identifier" token in our parse tree, we look up its value in the symbol table to find its type. Then we can proceed to type checking.

Symbols usually have a scope; eg: variables declared inside a block shouldn't be visible outside of the block. Symbol tables must be dynamic; at the end of a block, remove all entries added during the block. innermost definition takes precedence). Some nice algorithms exist for building tables efficiently, check the textbook. We can apply precedence rules while building symbol tables for nice variable shadowing rules.

Now we move on to the type checking itself. First, build the symbol table at all points of the program (traverse the parse tree, complain if you see duplicate variable definitions, etc). Now, traverse parse the tree. Every time you see an operation, check to make sure both of its arguments are of the same type (eg. `3 + 2`). Every method and operator should have a set of types that it allows and disallows (can store in symbol table). We can even use type variables; eg. the `+` requires both operations to be the same type.

We should also type check "statements", even though they don't have a type themselves. Eg. `if (a) then return b else return c` doesn't have a type itself; but we do need to assert that the types in the body work (`a` should become a boolean value, etc).

How much should the type checker do? It's a tradeoff. We test that `i` is an `int` when we type-check `arr[i]`, but we don't actually check whether `i` is positive until runtime in Java. Why do we bother type checking if we're going to have to wait to see the sign? Because knowing the type of `i` makes it very easy to check if it's positive at runtime (can check a single bit); no need to scramble and find out how to do it. These are the compile time <-> runtime tradeoffs.

2021-10-07: OOP + Types

Chapter 14: Object Oriented Languages

Consider a sketch of a method.

```
// _ is shorthand for an unknown type.
// `s` is a series of statements.
// `e` is an expression.
// Uppercase something to get its type (notation).
_ m(_ a) {
  _ b = 3;
  s;
  return e;
}
```

How do we make sure this function type checks? Obviously, if `e` is of type `E`, then the function return value should also be of type `E`.

```
E m(A a) {
  B b = 3;
  s;
  return e;
}
```

But now the question is – how do we typecheck `s` and `e`? They have a set of variables available to them; three sources of definitions to be added to the symbol table.

1. *Fields*; ie. the definitions that were in scope to begin with. Here, those could be global variables. If `m` were in a class (ie. if it was a method), they would be the fields of the class.
2. *Formal Parameters* of the function (ie. `a`).
3. *Local Variables* in the function (ie. `b` is of type `B`).

We should add them to the symbol tables in that order; fields have least priority, overwritten by formal parameters, which are overwritten by local variables. When we look up variables, look in the most recent symbol table entry first.

Note that Java doesn't make a distinction between formal parameters and local variables. The latter can't overwrite the former.

Now we know how to define and type check methods. How to we type check method calls (eg. `e1.m(e2)`)?

`e1` will evaluate to an object; the type will be the class name. Check that the class we've described exists in the program, and has the method `m`. Then call the type checker recursively on `e2`, find its type `E2`. Make sure the method `m` takes a parameter of type `E2`; if it does, the method call type checks successfully. Problems come in to play when we deal with inheritance and dynamic dispatch (need to dynamically figure out what type the object `e1` is, which method to call), but we seem to be avoiding them for now.

NOTE: Java doesn't have concept of an "uncalled method" (eg. `e1.m`). They are differentiated in the grammar, and are only included with an object. This is opposed to functional programming or Python, where we can pass functions as objects. Java does have lambda expressions, though they're not handled like you'd expect.

Java also gives us the `this` keyword. It refers to the implicit formal parameter passed in to every method – the object that the method is being called on. At least, that's the implementation's way of looking at it. More formally, we say that the type of the `this` statement is the enclosing class (in type form).

Now, suppose we want to complicate our type system by introducing inheritance. Inheritance originated (in a sense) farther back than C. Suppose we have the following:

```
int i = 3;
short s = 4;

x = y; // Valid
```

```
y = x; // Invalid
```

There exists an implicit hierarchy for conversions between the types; we notate this as short $\leq$ int. Every short can fit in an int, so we can always convert "down" along this heirarchy. We define the subclass relation, with some properties:

1. $x \leq x \; \forall \; x$
2. $a \leq b \leq c \implies a \leq c$
3. `B extends A` $\implies B \leq A$

Think of it as subset notation (which might have been more intuitive).

Now, we have a way to formally type check assignment statements. If we look through `x = e`, the statement typechecks iff

1. $A(x) = t$, where $A(x)$ looks up x in the current symbol table A.
2. `e` is of type `u`
3. $u \leq t$ (ie. `u` is a subclass of `t`).

We should also apply this to method calls. Suppose a method's signature is `(x1, x2, x3) -> x`. `m(y1, y2, y3)` should work if $y_i \leq x_i \; \forall \; i$.

We want a way to cast *up* the heirarchy, too. It shouldn't be automatic, because that would be too risky (downcasting is always safe, because an object implements at least what its superclass implements). However, there could be cases where the programmer *knows* that an object will implement everything required of it (type checker can't predict everything). Eg:

```
// B extends A
A a; = new B();
B b;

a = new B();
b = a; // As programmers, we know this will always work!
```

To solve this problem, we invent "type casts" – we treat the expression `(A) b` as an expression of type `A`, and insert a runtime assertion to make sure that the `b` expression really is of type `A`. It's not really worth making any more complicated rules, because those rules are easy to break anyways (eg. we can upcast up to `Object`, then downcast to anything, effectively converting anything to anything). Just add a runtime check, it's the simplest solution.

2021-10-12

An Introduction to the Sparrow Language

What does IR look like? We take a look at Sparrow.

Sparrow IR has no classes, just functions. We do have local variables. We can also store variables on the heap – given a heap address, we can store values to arbitrary offsets of the address space location. We have labels and jump functions. We'll later translate Sparrow IR to Sparrow5, the next IR.

Goal of today's lecture: give us a "sense" of the Sparrow IR, so we understand what's going on for the further homeworks.

First, dataflow.

- Sparrow is lower level than MiniJava, but it's still abstract. We have unbounded number of parameters to every function, unbounded number of local variables, etc.
- When we compile to Sparrow-V, it'll add in RISC-V registers, bounds, stack management, etc.
- When we compile Sparrow5 to RISC-V, it becomes longer. We add a header. All RISC-V opcodes are added. This is true assembly code.

Now, formal specification. What does Sparrow program look like?

- Programs (P) := List of (top-level function declarations, F)
- FunDecls (F) := `"func" funcname ( id_1, id_2, ...) B`
- Blocks (B) := List of instructions, `I`, followed by `"return" id`
- Instructions (I) :=
  - Can be a label `mylabel:`.
  - Can be a local variable `id = c` where c is an integer literal or a function pointer (`@f`).
  - Can be addition, multiplication, etc.
  - Can be load (`id = [id + c]`, c is the offset) or a store (`[id + c] = id`).
  - Can assign variables to other variables (`id = id`).
  - Can allocate memory (`id = alloc(c)`, where `c` is the number of bytes to allocate. Heap addr gets stored on LHS).
  - Can print (`print(id)`).
  - Can error with a string constant `error(s)`. Useful for out of bounds errors, etc.
  - Can unconditionally go to a label (`goto l`), or conditionally (`if0 id goto l`).
  - Can call functions (`id = call id(id, id,...)`). Notice we can't call a function directly; we have to execute the contents of a register. "'

So the potential types a variable can have are:

- Integer constants, `c`.
- Heap addresses, with offsets `(a, c)`.
- Function names `f`.

They're totally separate types. We shouldn't confuse them.

Also – note that when we remove the first statement from a block, the remainder is still a block. This is useful when we're thinking about a program's state. We'll use it in the next section.

Sparrow Operational Semantics

Let's define the "state" of a currently running Sparrow program, then define how each Sparrow instruction affects the current state. This will compactly define Sparrow's operational semantics (this approach is called *small step instructional semantics*).

Define the state as a 5-tuple:

- p, the entire program.
- H, the heap.
- b', the entire body of the currently executing function.
- E, the program environment. Maps from identifiers to values, represents parameters + local variables.
- b, the block that is executing right now.

Now, for instructions. Take a variable assignment:

- $(p, H, b', E, (\text{"}id = c\text{"}, b)) \rightarrow (p, H, b', E[id \mapsto c])$.

Notationally, putting square brackets to the right of the environment means that the new mapping overrides any previously existing ones. This rule says that an assignment instruction removes itself from the currently executing block, and adds a mapping to the environment.

Now to define addition:

- $(p, H, b', E, (\text{"}id = id_1 + id_2\text{"}b)) \rightarrow (p, H, b', E[id \mapsto (c_1 + c_2)])$, where $c_1 = E(id_1)$ and $c_2 = E(id_2)$. The plus represents literal integer addition; make sure the two variables refer to integers.

And addition of an object to the heap. First, what is the heap? It's a mapping from a heap address to a tuple

$$(p, H, b', E, (\text{"}id = [id_1 + c]\text{"}b)) \rightarrow (p, H, b', E[id \mapsto (c_1 + c_2)]),$$

TODO: Finish typesetting this.

QUESTION: Why does he keep using 4 when adding, dividing, etc? Shouldn't he use 8 because it's bytes? ANSWER: 4 bytes; everything is a 32 bit value in this computer.

Some more writing follows in the lecture. It is terrible to typeset, so I didn't include it in these notes. Nonetheless, it doesn't really say much; he's just rewriting our intuitions of "how does a function call work" in some alternate notation. If we want to learn more, go look at the spec for the compiler project; all of the notation is there, too. Not much ROI for this lecture, it seems.

One important note: Sparrow functions do have their own scope. An environment is only known in a single function; every time a method is called, we delete all existing variable bindings when we move to the next function. There is no scoping for `if0` statements, though.

2021-10-14

This is the most important lecture of the course. Most important topic. We cover Appel textbook chapters 7-8.

Introduction to Translation

We now get to the meat of the course – to translate language A into language B. Here, we're translating MiniJava into Sparrow.

All of our work revolves around one big question: How do we manage our temporary variables?

In upper level languages, we have the luxury of "expressions". These allow us to implicitly hold intermediate results (`x = (a * b) + c;`). Lower languages don't have this luxury. We also have to manage implicit labels (for `if` statements, etc). This is very complicated.

Luckily, this is not the last stage in our compiler. We have another step, register allocation, where we optimize the code and trim unneeded temporary variables. So we can be generous when creating temporary variables right now – unneeded ones will be cleaned up later.

We set up temporary variables `w0`, `w1`, `w2`, etc. Ensure every temporary variable has a unique name by creating a global counter `k`, which refers to the first "unused" number. Every time we need to allocate a new variable name, just define `wk` and increment `k`. We can use this numbering scheme to recursively allocate temporary variables to expressions. Consider a state `(code, k)`. We want to evaluate the expression `e1 + e2`.

1. Create a variable; store `e1 + e2` in `wk`. Now recurse to `e1`, incrementing global counter `k` to `k+1`.
2. Expand expression `e1` starting at `k+1`. It'll expand to some code, and will increment `k+1` (global counter) to `k1`.
3. Expand expression `e2` starting at `k1`. It'll expand to more code, and will increment the global counter to `k2`.
4. We now return to the overall expression. Output the following (where .. represents concatenating code): ((code from `e1`) .. (code from `e2`) .. $w_k = w_{k+1} + w_{k1}#). Leave the global counter at `k2`.

This will effectively set the global variable names for all expressions. It takes care of recursion well. It's obvious ("just recurse while incrementing variable names"), and isn't the only naming technique you can do.

Statements work almost exactly the same. Run through them in order. If you have 2 statements in order, execute the first one, then execute the second one. The global counter will be updated accordingly.

What about MiniJava `if` statements? Do what's intuitive. Map the boolean values `true` to `1`, and `false` to `0`. Use the `if0 wk goto l` statement in Sparrow. The label names to `goto` will also need to be unique; so reuse the label number that stores the value of the expression inside the `if` statement (in this case, `if0 wk goto lk`).

2021-10-19

We continue from the last lecture. Now, we focus on arrays.

How should we translate `new int[e]`? Use the `alloc()` Sparrow instruction. We want the `arr.length` expression to make sense in MiniJava – so we'll use a trick. Store the length of the array (in MiniJava! The number of

bytes allocated via Sparrow's `alloc` may be different) in the first slot of the array. Store `arr[0]` in the first slot, `arr[1]` in the second slot, and so on. It's functionally equivalent to storing the length in `arr[-1]` (depending on when you choose to increment and decrement the heap addresses). Now, we can compile `arr.length` to "retrieve the first element of the array".

Now:

1. Decide you'll use variable number range `k` to `k+4` for your intermediate computations, because the counter starts at `k`. Increment the variable accordingly. Let's say that you'll put your final answer in `wk`.
2. Now, evaluate the expression `e` (which will increment the global counter to `k1`).
3. If every data element takes 4 bytes, we should allocate `4 *(wk1 + 1)` bytes on the heap. Use some of the extra variables you reserved before (`k+1` to `k+4`) to calculate this intermediate variable. Say the final result is stored in `wk4`.
4. Run `wk = alloc(wk4)`.
5. Use a for loop (in Sparrow) to set `[wk + i] = 0` and initialize all elements in the array. You can also hardcode the assignment instructions, if you know the size of the array in advance.

This gives you the code you need to translate array allocation to Sparrow. First, the code for calculating `e`. Then, the algebra. Then, the `alloc` call. Then, the array element initialization. This is the overall code – the result, (perfect array address) is in `wk`, just as planned.

Array retrieval is straightforward. Just calculate the byte offset you want, and retrieve the value. Ditto array assignments. We do have the chance to add some runtime checks on array bounds, which eliminates the chance that the code is inadvertently writing to an unallocated address in the heap. Remember, the hardware won't always complain – it's very possible we're writing over another data structure in our process. To avoid this, invoke the Sparrow `panic()` instruction whenever the index offset passes the bounds of our array. Program the check using the `t1 < t2` Sparrow code.

If you think about it, really that's equivalent to checking all array accesses in a library. That library code should be identical to the "automatic checks" code when compiled. Don't worry too much about the cost of the check either – with the proper hint to the branch predictor, it costs us a few cycles at most.

OOP and IR

We must consider a few things when converting class structures into IR.

First, we convert all methods into functions. Two methods in different classes may have the same name. We solve this by prepending the class name; eg: `String__length`. This is called *name mangling*. JavaC uses a similar strategy to disambiguate overloading (which can all be figured out at compile-time for Java). It prepends the types to the function name.

Now, how do we give the *objects* themselves Sparrow representations? Turn them into a struct, C style; in other words, just turn them into a giant heap allocation, and index into it array style. This is pretty easy for MiniJava; all types have width 4, so to find the value of the 4th field in a class we just go to byte offset 16. Turn every field identifier into a number; use this number to index. The object creation code is exactly like the array creation code.

All methods are equivalent to functions with the `this` object passed in as the first parameter. In other words, they're equivalent to struct-specific functions in C, where we pass in the struct to operate on as an argument. This is also how we deal with the MiniJava `this` – just replace it with a reference to the first implicit parameter. We already know this, so I've left the details off.

We also have to worry about *dynamic binding*. As a review: static binding is like this:

```
class A {
  public void m() {
    return this.p();
  }
  public void p() { ; }
}
```

It's not hard to compile this into Sparrow. Inside `m`, we have a call to the `A` class's `p` method. Can hardcode this into IR. Now we consider inheritance:

```
class B extends A {
  public void p() { ; }
}
```

If we have:

```
A a;
B b = new B();
a = b;
a.m();
```

Then the `m` method in the superclass is called – but it needs to recognize at runtime that it's acting on a `C` object, and should invoke `C.p()` in the body of `m`! This is dynamic binding, and it means we can't hardcode the function in IR anymore.

We need a *method table* attached to every object. Objects should contain information about their fields. The first (invisible) "field" is a pointer to a method table. Method table maps method names to the function labels. Here, `B`'s field table would look like ($p \to \texttt{B}_\texttt{p}, m \to \texttt{A}_\texttt{m}$)). At runtime, calling a method on an object looks it up in the `this` object's method table. Method table is a fixed size for an object, so we can access the method by jumping to a constant offset (`p`'s offset could be `4` bytes) in the table, then calling the function pointed to there.

"Load, Load, Call" is the mnemonic. It is the mantra of Object Oriented Programming. The two are synonymous in Jens's mind. LLC is a philosophy.

Set up this method table when we're compiling MiniJava to Sparrow. The process is similar to setting up the fields. Allocate the fields (+ 1 slot), allocate the method table, link the two. Look at all the methods in the current class, fill in slots into the method table. Jump to the superclass. Look through all the methods, add them to the method table if the table doesn't already have an entry for them. Continue upwards.

2021-10-21

A new topic – needed for Homework 4. We're moving on to register allocation, the very backend of the compiler.

We first compiled MiniJava to Sparrow, an IR. Now, we're going to move to another IR even closer to the hardware – almost RISC-V, so we're calling it Sparrow-V. Sparrow-V adds bounds to the code; it introduces a finite set of RISC-V registers.

Register Allocation

Our machine has a finite set of registers on the CPU. Accessing a register is much faster than accessing memory; so we need to decide *which* of our unboundedly-many program variables should be stored in which registers. We do this (*Register Allocation*) all at one time.

Register Allocation – Overview

Register Allocation consists of two main steps:

1. *Liveness Analysis*: Done by compile time. Learn what variables are being used where. Optimize, then produce an undirected *interference graph*.
2. *Graph Coloring*: We color the interference graph. This helps us understand how to allocate physical registers to variables.

Consider an example, where we want to run the following with only 2 registers.

```
a = 1
b = 2
c = a + 3
print b + c
```

First, we go through the code and define a *live range* (*liveness* zone, etc) for every variable. `a` is defined in line 1, and is used in line 3; we need space to store its value between those two lines, so its liveness range is $(1, 3)$. Variables with different liveness ranges should be able to share a register (since they never use it at the same time).

We need to improve our definition, though! Looking above, it's clear that `c` and `a` could be stored in the same register, even though their liveness ranges overlap; we need to differentiate between the RHS and the LHS of an assignment statement. So we define the live range as an interval that's "closed" on one end (when it's used as the target of an assignment statement, eg `c` is closed at `c = a + 3`), and "open" on the other end (when it's last used in an expression, eg `a + 3`). It's like real number intervals; if one live range ends in an open manner while the other starts in a closed manner, the two intervals don't overlap. Virtually all intervals are closed at the start and open at the end, so the logic doesn't get too complicated.

Now we move on to the graph coloring part. Create a graph where every vertex is a variable. Draw an edge between two vertices if their live ranges overlap (in other words, draw an edge between 2 vertices if they'd mess with each other if they were on the same register). Now, the problem of allocating variables to registers decomposes to a simple undirected graph coloring problem, where every register gets a color. Adjacent vertices don't have the same color $\implies$ we can allocate registers to variables efficiently. If we don't have enough colors, we don't have enough registers. Will have to allocate the extra variables on the stack.

Now, make another pass through the same code. Substitute the name of a variable with the name of its register. Your code is now register allocated.

```
r1 = 1
r2 = 2
r1 = r1 + 3
print r2 + r1
```

Unfortunately, graph coloring is NP-complete, and it's equivalent to register allocation. Our graph colorings will be subpar and slow.

How can we optimize this process? Every time a variable is assigned to, we're effectively creating a new variable. This new variable is live until its last use in the RHS of a statement. This optimization is pretty simple to implement, even we can do it for the projects (start an interval whenever a variable is assigned to, then work your way forwards and get the ends). It does become complicated when we add `goto`s and conditionals, so we defer optimization with control flow for a later lecture.

Register Allocation – Liveness Analysis

How actually do Step 1 for MiniJava? For every statement (`n`), we need:

- `def[n]`: What vars are assigned/defined in statement `n`?
- `use[n]`: What vars are used in statement `n`?
- `in[n]`: This is what we want. What vars are live coming in?
- `out[n]`: This is what we want. What vars are live coming out?

Now we can define an algorithm that calculates the latter 2 given the first 2. They'll act on the (directed) control flow graph (CFG) of the program.

- `out[n]` must be equal to the union of `in[s]` (where `s` is the successor for node `n`; ie `n` points to `s`. Of course; if a var is in scope coming out of a statement, it's in scope coming in to the next ones.
- `in[n]`: if a var is used in `n`, it must be live. Otherwise, anything coming out of `n` must have come in to `n` (not counting things defined in `n`.

See how we'll have to traverse the program backwards to find the liveness (with the eqns above)? It's because we'll encounter the *last* time a variable is used first (don't allocate registers for something we'll never use again).

Rewrite as 2 equations:

- $out(n) = \bigcup_{s \in succ(s)} in(s)$
- $in(n) = use(n) \cup (out(n) \setminus def(n))$.

Build the CFG, and build a big table. Keep applying these definitions iteratively, until we hit a fixed point. Then, our liveness analysis is complete. This process may take $O(n^4)$ to the number of statements (can optimize to $O(n^2)$); polynomial, so it's "efficient". Still faster than graph coloring, not a bottleneck in practice.

Register Allocation – Graph Coloring

Now for the second half of register allocation. He did his research on this for about a decade.

Will be working with an interference graph now. Nodes are variables, and edges are between variables with overlapping live ranges.

Graph coloring algorithms are allowed to fail sometimes (color a triangle with 2 colors). That can't really happen here, so we want an alg that will figure out which nodes need "new colors". They will be *spilled* to memory (stack) instead of being stored in a register. Not every spill is equal; we want variables accessed infrequently to be stored in memory.

QUESTION: Why do people say that adding more registers doesn't make machines much faster nowadays?

Graph Coloring + Spilling tips:

- Clique := set of strongly connected components. The max size of a clique is a lower bound on how many registers we'll need.

Linear Scan Register Allocation: a fast greedy algorithm that does a surprisingly good job at performing graph coloring and spilling. Quite a few register allocators use this. How does it work? It doesn't bother with the interference graph at all (at least, it's not as deeply graph theoretical). Instead, it runs through the orders

TODO October 26th, 2021, 00:34:59

2021-10-28: Activation Records

How do we convert Sparrow-V to RISC-V?

Let's review some notation, so we can describe tricky parts of the Sparrow-V semantics. Function calls are the main "tricky part". Define a Sparrow-V program as a tuple:

- p, the Program.
- H, the Heap.
- R, the registers.
- b', the full block of code we're currently inside (from the start).
- E, the environment. Not everything fits in the registers, so this is the "stack".
- b the remainder of the block of code we're executing now.

Suppose we make a function call. When we transfer control to the callee, p, H, R are all the same. What should happen to E?

Clear E. Bind the name of the callee's formal parameter to the value of the corresponding argument (which is always on the stack, remember) at the call site.

Goal of the lecture is 2 goals:

- Manage the stack environments.
- Avoid registers clobbering each other over calls.

Second Half of the Course: Advanced Topics

2021-11-04: LR Parsing

LR parsing is a different method of parsing languages, more powerful than the LL methods. As a reminder, LR means: receive input left to right, and try the rightmost derivation (examine the rightmost nonterminal at all times). LR parsers tend to be *bottom up* parsers as opposed to top down parsers. In a nutshell:

- LL maintains a queue of symbols to expand (LHS -> RHS). Its problem: "Do I choose $A \to b$ or $A \to c$?".
- LR maintains a *stack* of symbols it has to contract (RHS -> LHS). Its problem: "Do I choose $A \to b$, or $B \to b$?".

Shift Reduce Parser: bottom-up implementation of an LR parser. Maintains a stack, sorta like "input I have yet to catch up to". Let's see how it works, given a sentence `abbcde` and ruleset:

- $S \to something$
- $A \to Abc \mid b$
- $B \to d$

We start with an empty stack.

1. Accept `a`, put on stack.
2. Look at stack. We can't unexpand it.
3. Accept `b`, put on stack.
4. Look at stack. We *can* unexpand it! $A \to b$, so pop `b` off the stack, replace with `A`. Stack is now `(a, A)`.
5. Accept `b`. Now – we could unexpand it back to `A` as before. But our parse table (which contains the entire state of the stack and also maybe lookahead) tells us that's not the right decision. We could have chosen a simpler grammar so we could always take the greedy approach... but not now. Pass. Stack is `aAb`.
6. Accept `c`. Put on stack. Stack is `aAbc`
7. Look on stack. We can unexpand! $A \to Abc$, so pop `Abc` and replace with `A`. Stack is now `aA`.
8. Etc etc.

LR parsers are great, because they are:

1. More powerful than LL parsers. Intuition: they can see the whole stack, so their parse tables are larger and more powerful. Theoretically the most general parser for deterministic, bottom up, N lookahead.
2. Better error messages – they detect problems as soon as they come up.
3. Almost all CFGs can be converted to LR.

They suck, because:

1. Lookahead table is *enormous*.

Tables must have k units of lookahead into the input, and need a row entry for every possible stack prefix. This is too big for any common language. Instead, we defined the weaker LALR parsers (add extra constraints on LR grammars in exchange for smaller table size).

2021-11-09: Type Checking Generic Classes

2021-11-16: Compiling Lambda Functions in Java

We want to compile lambda functions to something simple; don't convert to classes etc, just to labels and jumps.

How compile lambda functions? They're often recursive – we don't want that kind of stack growth, so we want to compile into an imperative version of the function that we can express in IR easily. This is the theme of the lecture; how do we get rid of the stack?

We'll do this by going in this order:

lambda -> tail -> first order -> imperative

tail: "functions never return". NOT tail recursive form (yet). others: TODO download the slides.

First, we have to transform a generally recursive program into a *Tail Form* program (ie. one that uses *Continuation Passing Style (CPS)*).

Conversion: - worse, because instead of having one top level lambda expression, we have multiple lambdas on every level, and you pass around a snowballing lambda that keeps getting bigger. We wanted to compile one lambda, now we have more lambdas. - better, because they're in tail forms. can do cool things with tail forms.

Q

When we evaluate tail form expr, we make one call, which is always the last operation.

2021-11-18: Control Flow Analysis: Devirtualization

Say that we have:

```
A a = new A();
a.m();
```

Why should we do the usual Load, Load, Call when we could just do a Load here? We know that `a` must always be an `A` here... so we should use *Static Program Analysis*: look through the program at compile time, and optimize what we have to do at runtime. Of course, we don't want to spend too much time analyzing the program, otherwise compiles will take forever. It's a tradeoff, and we have several quick heuristics.

That's the theme of the lecture: optimizing LLC (Load Load Call) to just a Load using static optimization.

Tangent:

Life was even worse in the old days, when languages didn't have static typing.

Back in 1984, there was a language called Smalltalk, and they wanted polymorphism like we had. But there were no types! It was duck typed. Compilers would take an object, ask it what methods it had at runtime, ask its parents what they had at runtime, and traverse the tree until things worked. This was abysmally slow, so they had some heuristics (Deutsch-Schiffman compiler) where they'd reuse type decisions when they came back to a statement.

Then ObjectiveC came along in 1986 (a guy named Brad Cox). He tried the approach of adding some knockoff method tables at runtime – a giant sparse table of `(object, method, correctfunction)`. They reused the type decisions, so now if you got `x.m`, you could quickly look up the "method table" (cache) and find that `x.m` should resolve to `A__m`.

Then Java came along, and people noticed the static types. They realized that static class info can be used to perform the above analysis really quickly; rather than do expensive lookups at runtime, we could build Method Tables at compile time. This was the birth of Method Tables and the LLC paradigm. Even better, they realized that we could use some heuristics at compile time to make the runtime code more efficient, like what we were discussing before the tangent.

End Tangent.

Static Program Analysis : CHA

In 1990, the Java guys created a simple static optimization technique that's fast ($O(n)$ time), surprisingly effective, and still used today: Class Hierarchy Analysis (CHA)!

Rather than looking through control flow graphs (which is hard), CHA does some very coarse grained optimization by looking through the class hierarchy. Consider the below code:

```
A {
  m()
  p()
}

B extends A {
  p()
}

C extends B {
  p()
}

D extends C { }
```

Suppose we had to evaluate `b.m()`, where `b` was stored in a variable of type `B`. Just by looking through the class hierarchy, we see that regardless of what `b` is at runtime, it'll call `A__m`. We don't need a method table LLC, we can just replace the code with a static call to `A__m`. The same is not true if we have `b.p`; if the `b` was of type `D`, then we'd need `D__p` instead of `A__p`, so we can't replace that LLC. This technique (CHA) is surprisingly good. Turns out we can replace most LLC lookups with direct calls using CHA (although that statistic is a bit old).

Static Program Analysis : RTA

In the 1990s, Java compiler writers made a small, obvious, extension to CHA called Rapid Type Analysis (RTA). Still $O(n)$, just a bit smarter.

- Use CHA. Make one pass through the program, scrape the class hierarchy. Given an object's static type and a method, we've defined a narrower "set of all functions that the given method could call".
- Make a quick scan through the whole program. Grep for instances of `new A()`. It may be the case that some objects are never instantiated in the program at all – if so, further narrow down the set of possibilities.
- We can optionally improve this by doing some primitive reachability analysis. Build a graph that shows us how methods call each other (don't bother looking through the `if` statements, that's too complicated – if we see the words `a.m()` in a method, assume that the `m()` method will be called on `a`). Use the graph to further narrow down the possibilities (eg: "`m()` will never receive an `a`, so remove the `a` handling code).

RTA = CHA + "All `new` statements in the program" + "reachability analysis".

The last (optional) step can be slower, but the first two are $O(n)$.

Static Program Analysis : 0-CFA

Now for more advanced analysis: *0-Control Flow Analysis (0-CFA)*. Runtime $O(n^3)$.

It's a type of *Constraint Based Program Analysis*. Start with what you know about the program's control flow, write out the data type constraints you discover, then use those new constraints + control flow to add more data type constraints.

Central idea: For every variable and method parameter, maintain a set of "possible types the value could have" at a point. No need to go trawling through while loops inside a method; it's too expensive, and not usually worth it. Just assume that any write to a variable will contribute to the set of its potential types. Eg:

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

Now the set of all types `x` can be is $\{A, B\}$.

Now, we can iteratively propagate these constraints! If we know that `x` can be $\{A, B\}$, and we see `a.foo(x)`, then we know that the first parameter of `foo` can be at least $\{A, B\}$. Continue iteratively adding constraints, then optimize out LLC code using the constraints.

Static Program Analysis : TSMI

Unfortunately, 0-CFA can cause problems when compilers inline code. If you have a class that satisfies an interface and you try to inline code, you're now running a class method on an interface, which doesn't type check (it needs to type check because apparently the JVM itself does type checking at runtime).

So we use a dumbed down, conservative, method of 0-CFA called Type Safe Method Inlining (TSMI). It's still $O(n^3)$.

Static Program Analysis : 1-CFA

What's the weakness of 0-CFA? There's only one "set of possibilities" for a given variable. Eg:

```
class C {
  _ id (_ arg1) {
    return arg1;
  }
}

x.m(new A()) // What's the type of this expression?
x.m(new B()) // What's the type of this expression?
```

0-CFA will look at the whole program, and decide that arg1's "set of possibilities" is $\{A, B\}$. So when we return, we'll get either an `A` or a `B`. But of course, we see that the first call will return an `A` and the second will return a `B`.

In effect, what 1-CFA does is that it creates one "copy" of a method for every single call site, then runs CFA as usual. So it'll check the first expression, and realize it only returns an `A`. It'll look at the second, and realize it only returns a `B`.

Can go even further beyond. Say `id` called another method inside – we can create a copy of *that* method too, then do type evaluation. Now we're doing 2-CFA. In effect, N-CFA manually recomputes (or stores copies of, however you see it) the N outermost expressions.

This copying is damn expensive. 1-CFA is $O(2^N)$, 2-CFA is $O(2^{2^N})$.

Data Flow Analysis: Copy Propagation

Suppose we have the following:

```
b = a
c = b
d = b
```

We can equivalently write:

```
c = a
d = a
```

This is *copy propagation*, the theme of the lecture. We want to perform this optimization automatically.

How do we do this conversion automatically? Consider a given assignment. We need to know two conditions hold:

1. `b = a` should be the only definition of `b` that can hold at `c = b`. Ie: we can't go "around" our definition. This shows that we're "far enough" along the CFG.
2. Every path from `b = a` (our statement) to `c = b` doesn't touch `b` Ie: `b` isn't modified later in the control path. This shows that we're "as far as we can go" along the CFG.

These conditions are kind of the same if you think about it. Palsberg might not have taught this well.

This is more in the realm of data flow, as opposed to control flow (vertices are copy statements). The dataflow guys say *gen* and *kill* instead of *def* and *use*; other than that, it's very similar to our previous work on liveness analysis. We'll traverse the control flow graph, keeping track of which definitions are valid. Every variable definition corresponds to a single variable assignment in the program.

For every program point (`n`), we need:

- `gen[n]`: What data elements are greated in this statement?
- `kill[n]`: What previously existing data elements are disturbed (killed) during this statement?
- `in[n]`: This is what we want. What copy statements can reach `n`?
- `out[n]`: This is what we want.

Some informal conditions:

- If `n` is a copy statement, then $n \in gen(n)$.
- What statements does `n` kill? For every element (copy statement) `x = y` currently active, `n` kills it if it assigns to `x` (because it overwrites the previous value). `n` also kills it if it uses `y`! If we're using `y`, then we can't eliminate `y`, because it's needed somewhere else. It's not a *redundant* temporary variable yet. Think

about this. TODO: We misunderstood this. He answered this in the last few minutes of the last lecture, so listen to that. This isn't the smallest kill set, it's just a quick heuristic.
- Consider `in[n]`. What copy statements are valid coming in to n? If we can be jumped to from 2 places, and a copy statement is valid in *both*, only then is it valid for us (Condition 1).
- Consider `out[n]`. Unless we screw with something, all copy statements coming in should be valid coming out.

Write in data flow equation format. We define $pred(n)$ as the set of predecessors of n on the control flow graph.

- $in(n) = \bigcap_{p \in pred(n)} out(p)$.
- $out(n) = gen(n) \cup (in(n) \setminus kill(n))$.

Iteratively solve these equations, just as with liveness analysis. We do this "forwards" through the CFG, not backwards (based on what's coming in, deduce what's going out. Liveness Analysis was the other way round).

Interpreters

Why choose an interpreter vs compiler?

- Generally, compiled languages are waay more than 20x faster than interpreted ones. An interpreter that's only 10-20 times as slow as a compiled version is masterfully written. God tier interpreters (advanced register tricks, all in assembly, Google level geniuses) can lower this to about 4-6x, but this is superhuman and requires decades of investment even by Google. Python was supposed to be 100-300 times as slow as C back in the day. Ruby was 500-1000 times as slow as C. Eek.
- Compilers deal with 3 different languages: source, target, and implementational (what the compiler is written in). Interpreters only have a source and implementation.
- Interpreters are much easier to write. When prototyping a language, write interpreters. (No need to deal with a target language, etc).
- Interpreters are much easier to set up debugging / instrumentation in. Valgrind runs a C interpreter; the user can add profiling code, breakpoints, etc at any point during the interpretation process. Much harder to do with a compiler; you'd have to somehow insert the debug code into the target language and hope further passes wouldn't screw it up.

Intuition: Consider the three languages:

- Source: The language you get as input.
- Target: The language you output.
- Implementation: The language you write your compiler/interpreter in.

How do you eventually get results? A compiler takes the source, assumes that the target has some way of "executing" (semantically), and turns source into target. An interpreter chooses the other fork in the road; it realizes that it's implemented in a language which provides some semantics (eg: in Java, we can do `3 + 1`, and uses *that* language to turn the glyphs into meaning, instead of relying on some other "string -> meaning" map on the target language).

At the end, it's all interpreted – the CPU is a hardware interpreter for machine code. The real question is whether you translate your whole program directly to CPU via a series of compilers, or translate it here in the interpreter itself.

Why are interpreters slower? Because they have to keep track of the program's state explicitly as they translate, interpret, validate, parse the input program. The compilers offload the validation and decision making to compile time.

Interpreters usually evolve into virtual machines. Virtual machines include both interpreters and compilers. When receiving new code, they run the interpreter so there's no compilation overhead. After some runtime has passed, runtime profiling code lets us know which methods are used often. We spend the time to compile those, then run those compiled method versions for the speed boost. Google's JavaScript compiler does this; the interpreter is in x86_64, and it compiles code to x86_64 – the transition between the "compiled code" and "interpreted code" is very low friction.

VMs often have two compilers; a fast compiler (little optimization) and a slow compiler (lot of optimization). Based on profiling data, they can decide which compiler to use. They calculate: compiling code with the slow

compiler will take 3 seconds, the compiled version will run in 0.4 seconds, so if the interpreter takes 4 seconds then it's worthwile to compile it even if we'll only call the compiled version once. This is the bread and butter of interpreter optimizations; try to predict the future.

You can have an interpreter that's implemented in the language that it interprets. This is called a self-interpreter. TODO: How? Why?

QUESTIONS FOR PALSBERG:

- How on earth does the computer do the accumulation trick automatically? Inductive proof simple for Fibonacci; but how anything harder? Conditionals?
- Is the "replace with number" just notational abuse?
- "0-CFA is more powerful than CHA" – what quantifies power in this sense? Aren't there some CHA problems that 0-CFA can't solve?
- Have they ever changed the calling conventions? Eg: more caller saved registers would make sense depending on programming style (many callbacks, etc).
- Isn't definition of `kill[n]` overgeneral? Why can't we assign to the rhs variable later?