CS132: Compiler Construction

Professor Palsberg

Thilan Tran

Fall 2020

Contents

CS132: Compiler Construction

Introduction

- a **compiler** is a program that *translates* an executable program in one language to an executable program in another language
- an **interpreter** is a program that *reads* an executable program and produces the results of running that program
  - usually involves executing the source program in some fashion, ie. *portions* at a time
- compiler construction is a *microcosm* of CS fields:
  - AI and algorithms
  - theory
  - systems
  - architecture
- in addition, the field is not a solved problem:
  - changes in architecture lead to changes in compilers
    - new concerns, re-engineering, etc.
  - compiler changes then prompt new architecture changes, eg. new languages and features
- some compiler motivations:
  1. correct output
  2. fast output
  3. fast translation (proportional to program size)
  4. support separate compilation
  5. diagnostics for errors
  6. works well with debugger
  7. cross language calls
  8. optimization
- for new languages, how are compilers written for them?
  - eg. early compilers for Java were written in C
  - eg. for early, low level languages like C, **bootstrapping** is done:
    - a little subset of C is written and compiled in machine code
    - then a larger subset of C is compiled using that smaller subset, etc.

Compiler Overview

- abstract compiler system overview:
  - *input*: source code
  - *output*: machine code or errors
  - recognizes illegal programs, and outputs associated errors
- *two-pass* compiler overview:
  - source code eg. Java compiles through a frontend to an **intermediate representation (IR)** like Sparrow
    - the **frontend** part of the compiler maps legal code into IR:
      - language *dependent*, but machine *independent*
      - allows for swappable front ends for different source languages
  - IR then compiles through a backend to machine code
    - the **backend** part maps IR onto target machine:
      - language *independent*, but machine / architecture *dependent*
- frontend overview:
  - *input*: source code
  - *output*: IR
  - *responsibilities*:
    - recognize legality syntactically
    - produce meaningful error messages
    - shape the code for the backend
  1. the scanner produces a stream of tokens from source code:
  - ie. *lexing* source file into tokens
  2. the parser produces the IR:
  - recognizes context free grammars, while guiding context sensitive analysis
  - both steps can be automated to some degree
- backend overview:
  - *input*: IR
  - *output*: target machine code
  - *responsibilities*:
    - translate to machine code
    - **instruction selection**:
      - choose specific instructions for each IR operation
      - produce compact, fast code
    - **register allocation**:
      - decide what to keep in registers at each points
      - can move loads and stores
      - optimal allocation is difficult
  - more difficult to automate
- specific frontends or backends can be swapped
  - eg. use special backend that targets ARM instead of RISC, etc.
- middleend overview:
  - *responsibilities*:

    * optimize code and perform code improvement by analyzing and changing IR
    * must preserve values while reducing runtime
  - optimizations are usually designed as a set of iterative passes through the compiler
  - eg. eliminating redundant stores or dead code, storing common subexpressions, etc.
  - eg. GCC has 150 optimizations built in

Lexical Analysis

- the role of the **scanner** is to map characters into **tokens**, the basic unit of syntax:
  - while eliminating whitespace, comments, etc.
  - the character string value for a token is a **lexeme**
  - eg. `x = x + y;` becomes `<id,x> = <id,x> + <id,y> ;`
- a scanner must recognize language syntax
  - how to define what the syntax for integers, decimals, etc.

1. use regular expressions to specify syntax patterns:
   - eg. the syntax pattern for an integer may be `<integer> ::= (+ | -) <digit>*`
2. regular expressions can then be constructed into a **deterministic finite automaton (DFA)**:
   - a series of states and transitions for accepting or rejecting characters
   - this step also handles state minimization
3. the DFA can be easily converted into code using a while loop and states:
   - by using a table that categorizes characters into their language specific identifier types or classes, this code can be language *independent*
     - as long as the underlying DFA is the same
   - a linear operation, considers each character once

- this process can be automated using **scanner generators**:
  - emit scanner code that may be direct code, or table driven
  - eg. `lex` is a UNIX scanner generator that emits C code

Parsing

- the role of the **parser** is to recognize whether a stream of tokens forms a program defined by some grammar:
  - performs context-free syntax analysis
  - usually constructs an IR
  - produces meaningful error messages
  - generally want to achieve *linear* time when parsing:
    - need to impose some restrictions to achieve this, eg. the **LL** restriction
- context-free syntax is defined by a **context-free grammar (CFG)**:
  - formally, a 4-tuple $G = (V_t, V_n, S, P)$ where:
    - V_t is the set of **terminal** symbols, ie. tokens returned by the scanner
    - V_n is the set of **nonterminal** symbols, ie. syntactic variables that denote substrings in the language
    - S is a distinguished nonterminal representing the **start symbol** or goal
    - P is a finite set of **productions** specifying how terminals and nonterminals can be combined
      - each production has a single nonterminal on the LHS
    - the **vocabulary** of a grammar is $V = V_t \cup V_n$
    - the motivation for using CFGs instead of simple REs for grammars is that REs are not powerful enough:
      - REs are used to classify tokens such as identifiers, numbers, keywords
      - while grammars are useful for counting brackets, or imparting structure eg. expressions
      - factoring out lexical analysis simplifies the CFG dramatically
  - general CFG notation:
    - $a, b, c, ... \in V_t$
    - $A, B, C, ... \in V_n$
    - $U, V, W, ... \in V$
    - $\alpha, \beta, \gamma, ... \in V^*$, where V^* is a sequence of symbols
    - $u, v, w, ... \in V_t^*$, where V_t^* is a sequence of terminals
    - $A \to \gamma$ is a production
    - $\Rightarrow, \Rightarrow^*, \Rightarrow^+$ represent derivations of 1, ≥ 0, and ≥ 1 steps
    - if $S \Rightarrow^* \beta$ then β is a **sentential form** of G
    - if $L(G) = \{\beta \in V^* | S \Rightarrow^* \beta\} \cap V_t^*$, then $L(G)$ is a **sentence** of G, ie. a derivation with all nonterminals
- grammars are often written in **Backus-Naur form (BNF)**:
  - non-terminals are represented with angle brackets

    - terminals are represented in monospace font or underlined
    - productions follow the form `<nont> ::= ...expr...`
- the productions of a CFG can be viewed as rewriting rules:
    - by repeatedly rewriting rules by replacing nonterminals (starting from goal symbol), we can **derive** a sentence of a programming language
        - **leftmost derivation** occurs when the *leftmost* nonterminal is replaced at each step
        - **rightmost derivation** occurs when the *rightmost* nonterminal is replaced at each step
    - this sequence of rewrites is a **derivation** or **parse**
    - *discovering* a derivation (ie. going backwards) is called **parsing**
- can also visualize the derivation process as construction a tree:
    - the goal symbol is the root of tree
    - the children of a node represents replacing a nonterminal with the RHS of its production
    - note that the ordering of the tree dictates how the program would be *executed*
        - can multiple syntaxes lead to different parse trees depending on the CFG used?
    - parsing can be done **top-down**, from the root of the deriviation tree:
        - picks a production to try and match input using backtracking
        - some grammars are backtrack-free, ie. *predictive*
    - parsing can also be done **bottom-up**:
        - start in a state valid for legal first tokens, ie. start at the leaves and fill in
        - as input is consumed, change state to encode popssibilities, ie. recognize valid prefixes
        - use a stack to store state and sentential forms

Top-Down Parsing

- try and find a linear parsing algorithm using top-down parsing
- general top-down parsing approach:
    1. select a production corresponding to the current node, and construct the appropriate children
        - want to select the right production, somehow guided by input string
    2. when a terminal is added to the *fringe* that doesn't match the input string, backtrack
    3. find the next nonterminal to expand
- problems that will make the algorithm run worse than linear:

- too much backtracking
- if the parser makes the wrong choices, expansion doesn't even terminate
  - ie. top-down parsers *cannot* handle left-recursion
- top-down parsers may backtrack when they select the wrong production:
  - do we need arbitrary **lookahead** to parse CFGs? Generally, yes.
  - however, large subclasses of CFGs *can* be parsed with *limited* lookahead:
    - **LL(1)**: left to right scan, left-most derivation, 1-token lookahead
    - **LR(1)**: left to right scan, right-most derivation, 1-token lookahead
- to achieve LL(1) we roughly want to have the following initial properties:
  - no left recursion
  - some sort of *predictive* parsing in order to minimize backtracking with a lookahead of only one symbol

Grammar Hacking

Consider the following simple grammar for mathematical operations:

```
<goal> ::= <expr>
<expr> ::= <expr> <op> <expr> | num | id
<op> ::= + | - | * | /
```

- there are multiple ways to rewrite the same grammar:
  - but each of these ways may build *different* trees, which lead to *different* executions
  - want to avoid possible grammar issues such as precendence, infinite recursion, etc. by rewriting the grammar
  - eg. classic precedence issue of parsing `x + y * z` as `(x+y) * z` vs. `x + (y*z)`
- to address **precedence**:
  - additional machinery is required in the grammar
  - introduce extra **levels**
  - eg. introduce new nonterminals that group higher precedence ops like multiplication, and ones that group lower precedence ops like addition
    - the higher precedence nonterminal cannot reduce down to the lower precedence nonterminal
    - forces the *correct* tree

Example of fixing precedence in our grammar:

```
<expr> ::= <expr> + <term> | <expr> - <term> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= num | id
```

- **ambiguity** occurs when a grammar has more than one derivation for a single sequential form:
  - eg. the classic **dangling-else** ambiguity `if A then if B then C else D`
  - to address ambiguity:
    - rearrange the grammar to select one of the derivations, eg. matching each `else` with the closest unmatched `then`
  - another possible ambiguity arises from the context-free specification:
    - eg. **overloading** such as `f(17)` , could be a function or a variable subscript
    - requires context to disambiguate, really an issue of type
    - rather than complicate parsing, this should be handled separately

Example of fixing the dangling-else ambiguity:

```
<stmt> ::= <matched> | <unmatched>
<matched> ::= if <expr> then <matched> else <matched> | ...
<unmatched> ::= if <expr> then <stmt> | if <expr> then <matched> else <unmatched>
```

- a grammar is **left-recursive** if $\exists A \in V_n s.t. A \Rightarrow^* A\alpha$ for some string α:
  - top-down parsers fail with left-recursive grammars
  - to address left-recursion:
    - transform the grammar to become right-recursive by introducing new nonterminals
  - eg. in grammar notation, replace the productions $A \rightarrow A\alpha|\beta|\gamma$ with:
    - $A \rightarrow NA'$
    - $N \rightarrow \beta|\gamma$
    - $A' \rightarrow \alpha A'|\varepsilon$

Example of fixing left-recursion (for `<expr>, <term>`) in our grammar:

```
<expr> ::= <term> <expr'>
<expr'> ::= + <term> <expr'> | - <term> <expr'> | E // epsilon

<term> ::= <factor> <term'>
<term'> ::= * <factor> <term'> | / <factor> <term'> | E
```

- to perform **left-factoring** on a grammar, we want to do repeated prefix factoring until no two alternaties for a single non-terminal have a common prefix:
  - an important property for LL(1) grammars
  - eg. in grammar notation, replace the productions $A \rightarrow \alpha\beta|\alpha\gamma$ with:
    - $A \rightarrow \alpha A'$
    - $A' \rightarrow \beta|\gamma$
  - note that our example grammar after removing left-recursion is now properly left-factored

Achieving LL(1) Parsing

Predictive Parsing

- for multiple productions, we would like a *distinct* way of choosing the *correct* production to expand:
  - for some RHS $\alpha \in G$, define *FIRST*(α) as the set of tokens that can appear *first* in some string derived from α
  - key property: whenever two productions $A \to \alpha$ and $A \to \beta$ both appear in the grammar, we would like:
    - *FIRST*$(\alpha) \cap$ *FIRST*$(\beta) = \emptyset$, ie. the two token sets are disjoint
  - this property of **left-factoring** would allow the parser to make a correct choice with a lookahead of only *one* symbol
  - if the grammar does not have this property, we can hack the grammar
- by left factoring and eliminating left-recursion can we transform an *arbitrary* CFG to a form where it can be predictively parsed with a single token lookahead?
  - no, it is undecidabe whether an arbitrary equivalent grammar exists that satisfies the conditions
  - eg. the grammar $\{a^n 0 b^n\} \cup \{a^n 1 b^{2n}\}$ does not have a satisfying form, since would have to look past an arbitrary number of a to discover the terminal
- idea to translate parsing logic to code:
  1. for all terminal symbols, call an `eat` function that *consumes* the next char in the input stream
  2. for all nonterminal symbols, call the corresponding function corresponding to the production of that nonterminal
     - perform predictive parsing by looking *ahead* to the next character and handling it accordingly
       - there is only one valid way to handle the character in this step due to the left-factoring property
     - how do we handle epsilon?
       - just do nothing ie. consume nothing, and let recursion handle the rest
  - creates a mutually recursive set of functions for each production
    - the name is the LHS of production, and body corresponds to RHS of production

Example simple recursive descent parser:

```
Token token;
void eat(char a) {
  if (token == a) token = next_token();
  else error();
}

void goal() { token = next_token(); expr(); eat(EOF); }
void expr() { term(); expr_prime(); }
void expr_prime() {
  if (token == PLUS) { eat(PLUS); expr(); }
  else if (token == MINUS) { eat(MINUS); expr(); }
  else { /* noop for epsilon */ }
}
void term() { factor(); term_prime(); }
void term_prime() {
  if (token == MULT) { eat(MULT); term(); }
  else if (token == DIV) { eat(DIV); term(); }
  else { }
}
void factor() {
  if (token == NUM) eat(NUM);
  else if (token == ID) eat(ID);
  else error(); // not epsilon here
}
```

Handling Epsilon

- handling epsilon is not as simple as just ignoring it in the descent parser
- for a string of grammar symbols α, $\mathit{NULLABLE}(\alpha)$ means α can go to ε:
  - ie. $\mathit{NULLABLE}(\alpha) \iff \alpha \Rightarrow^* \varepsilon$
- to compute *NULLABLE*:
  1. if a symbol a is terminal, it cannot be nullable
  2. otherwise if $a \to Y_1 ... Y_n$ is a production:
     - $\mathit{NULLABLE}(Y_1) \wedge ... \wedge \mathit{NULLABLE}(Y_k) \Rightarrow \mathit{NULLABLE}(A)$
  3. solve the constraints
- again, for a string of grammar symbols α, $\mathit{FIRST}(\alpha)$ is the set of terminal symbols that begin strings derived from α:
  - ie. $\mathit{FIRST}(\alpha) = \{a \in V_t | \alpha \Rightarrow^* a\mathrm{B}\}$
- to compute *FIRST*:
  1. if a symbol a is nonterminal, $\mathit{FIRST}(a) = \{a\}$

2. otherwise if $a \rightarrow Y_1 ... Y_n$ is a production:
    - $FIRST(Y_1) \subseteq FIRST(A)$
    - $\forall i \in 2...n$, if $NULLABLE(Y_1 ... Y_{i-1})$:
        * $FIRST(Y_i) \subseteq FIRST(A)$
3. solve the constraints, going for the $\subseteq$-least solution

- for a nonterminal B, $FOLLOW(B)$ is the set of terminals that can appear immediately to the *right* of B in some sentential form:
    - ie. $FOLLOW(B) = \{a \in V_t | G \Rightarrow^* \alpha B \beta \wedge a \in FIRST(\beta \$)\}$
- to compute *FOLLOW*:
    1. $\{\$\} \subseteq FOLLOW(G)$ where G is the goal
    2. if $A \rightarrow \alpha B \beta$ is a production:
        - $FIRST(\beta) \subseteq FOLLOW(B)$
        - if $NULLABLE(\beta)$, then $FOLLOW(A) \subseteq FOLLOW(B)$
    3. solve the constraints, going for the $\subseteq$-least solution

Formal Definition

- a grammar G is **LL(1)** iff. for each production $A \rightarrow \alpha_1 | \alpha_2 | ... | \alpha_n$:
    1. $FIRST(\alpha_1), ..., FIRST(\alpha_n)$ are pairwise *disjoint*
    2. if $NULLABLE(\alpha_i)$, then for all $j \in 1...n \wedge j \neq i$:
        - $FIRST(\alpha_j) \cap FOLLOW(A) = \emptyset$
    - if G is ε-free, the first condition is sufficient
    - eg. $S \rightarrow aS | a$ is not LL(1)
        * while $S \rightarrow aS', S' \rightarrow aS' | \varepsilon$ accepts the same language and is LL(1)
- provable facts about LL(1) grammars:
    1. no left-recursive grammar is LL(1)
    2. no ambiguous grammar is LL(1)
    3. some languages have no LL(1) grammar
    4. an ε-free grammar where each alternative expansion for A begins with a distinct terminal is a simple LL(1) grammar
- an LL(1) **parse table** M can be constructed from a grammar G as follows:
    1. $\forall$ productions $A \rightarrow \alpha$:
        - $\forall a \in FIRST(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
        - if $\varepsilon \in FIRST(\alpha)$:
            * $\forall b \in FOLLOW(A)$, add $A \rightarrow \alpha$ to $M[A, b]$ (including EOF)
    2. set each undefined entry of M to an error state
    - if $\exists M[A, a]$ with multiple entries, then the grammar is *not* LL(1)

LR Parsing

- recalling definitions:
  - for a grammar G with start symbols S, any string α such that $S \Rightarrow^* \alpha$ is a **sentential form**
  - if $\alpha \in V_t^*$, then α is a **sentence** in $L(G)$
  - a **left-sentential** form is one that occurs in the *leftmost* derivation of some sentence
  - a **right-sentential** form is one that occurs in the *rightmost* derivation of some sentence
- **bottom-up parsing**, ie. **LR parsing**, is an alternative to parsing top-down:
  - want to construct a parse tree by starting at the leaves and working to the root
  - repeatedly, match a right-sentential form from the language against the tree
    - ie. for each match, apply some sort of reduction to build on the *frontier* of the tree
    - conversely to LL parsing, LR parsing prefers left recursion
  - creates a rightmost derivation, in reverse
  - eg. given the grammar `S → aABe, A → Abc|b, B → d` :
    - parse `abbcde` by replacing terminals with nonterminals repetaedly, ie. applying productions backwards
    - `abbcde` $\longrightarrow$ `aAde` $\longrightarrow$ `aABe` $\longrightarrow$ `S`
- must scan input and find some kind of a *valid* sentential form:
  - this valid form is called a **handle**:
    - a handle α is a substring that matches the production $A \to \alpha$ where reducing α to A is one step in the reverse of a rightmost derivation
    - formally, if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$ then $A \to \beta$ can be used as a handle
    - right sentential, so all symbols in w are terminals
  - the process of reducing through handles is called **handle pruning**
  - how can we find *which* unique handle to apply?
    - recognizing handles is out of the scope of this course
- important theorem involving handles:
  - if G is unambiguous, them every right-sentential form has a unique handle
  - this is because an unambiguous grammar has a *unique* rightmost derivation
- general LR algorithm:
  1. repeatedly, working backwards, find the handle $A_i \to B_i$ in the input
  2. replace ie. prune the handle to generate γ_{i-1}

Stack Implementation

- one way to implement a handle-pruning, bottom-up parser is called a **shift-reduce parser**:
  - uses a stack (handles the shift operation) and an input buffer
  - shift-reduce vs. reduce-descent
- algorithm:
  1. initialize stack with EOF $
  2. repeat until the top of the stack is the goal symbol and input token is $:
     - find the handle:
       - if we don't have a handle on top of the stack, *shift* an input symbol onto the stack
       - ie. shift until top of stack is the right end of a handle
     - prune the handle:
       - if we have a handle $A \to \beta$ on the stack, reduce by:
         - popping β symbols off the stack
         - pushing A onto the stack

Simple left-recursive expression grammar:

```
<goal> ::= <expr>
<expr> ::= <expr> + <term> | <expr> - <term> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= num | id
// note that left-recursion is fine with LR
```

Running example input `id - num * id` through a shift-reduce parser:

```
// stack
$    // shift next input onto stack
$ id // can reduce, id is a handle
$ <factor>
$ <term>
$ <expr> // out of handles, shift next input
$ <expr> -
$ <expr> - num // can reduce num at top of stack
$ <expr> - <factor>
$ <expr> - <term> // reducing here would disregard the rest of the input stream
$ <expr> - <term> *
$ <expr> - <term> * id
$ <expr> - <term> * <factor> // by looking at input, we should reduce
                      // <factor> → <term>*<factor> instead of <factor> → <term>
$ <expr> - <term>
$ <expr>
$ <goal> // this would be accepted
```

- shift-reduce parsers support four actions:
  1. shift next input symbol onto stack
  2. reduce by popping handle off stack and pushing production LHS
  3. accept and end parsing
  4. error
  - the key problem is recognizing handles
- LR vs. LL grammars and parsing:
  - almost all CFG languages can be expressed in LR(1)
  - LR parsers detect an error as soon as possible
  - naturally, right recursion is used in top-down parsers
    - while left recursion is used in bottom-up parsers
  - LL dilemma is to pick between different alternate production rules:
    - $A \to b$ vs. $A \to c$
    - ie. LL(k) parser must recognize the use of of a production after seeing only k symbols of its RHS
  - LR dilemma is to pick between different matching productions:
    - $A \to b$ vs. $B \to b$
    - ie. LR(k) parser must recognize the RHS of a production after having seen all that is derived from RHS with k symbols of lookhead
  - in an LL parsing table:
    - only need to store nonterminals vs. terminals
  - in an LR parsing table:
    - need to store all the possible entries at the top of the stack vs. terminals
    - top of stack can contain *every* alternate in a production rule
      - usually much greater than number of nonterminals
    - more complicated and space consuming, but more powerful as well
      - in addition, $LL(k) \subseteq LR(k)$

JavaCC

- the **Java Compiler Compiler (JCC)** generates a parser automatically for a given grammar:
  - based on LL(k) vs. LL(1)
  - transforms an EGBNF grammar into a parser
  - can have embedded (additional) action code written in Java
  - `javacc fortran.jj` $\longrightarrow$ `javac Main.java` $\longrightarrow$ `java Main < prog.f`

JavaCC input format:

```
TOKEN :
{
  < INTEGER_LITERAL: ( ["1"-"9"] (["0"-"9"])* | "0" ) >
}

void StatementListReturn() :
{}
{
  ( Statement() )* "return" Expression() ";"
}
```

Handling Syntax Trees

Visitor Pattern

- parsers generate a syntax tree from an input file:
  - this is an aside on design patterns in order to facilitate using the generated tree
  - see Gamma's *Design Patterns* from 1995
- for OOP, the **visitor pattern** enables the definition of a *new* operation of an object structure *without* changing the classes of the objects:
  - ie. new operation *without* recompiling
  - set of classes must be fixed in advance, and each class must have a hook called the `accept` method

Consider the problem of summing up lists using the following list implementation:

```
interface List {}

class Nil implements List {}

class Cons implements List {
  int head;
  List tail;
}
```

First approach using type casts:

```
List l;
int sum = 0;
while (true) {
  if (l instanceof Nil)
    break;
  else if (l instanceof Cons) {
    sum += ((Cons) l).head;
    l = ((Cons) l).tail;
  }
}
```

- *pros*:
  - code is written without touching the classes
- *cons*:

  - code constantly uses type casts and `instanceof` to determine classes

Second approach using dedicated methods (OO version):

```
interface List { int sum(); }

class Nil implements List {
  public int sum() { return 0; }
}

class Cons implements List {
  int head;
  List tail;
  public int sum() { return head + tail.sum(); }
}
```

- *pros*:
  - code can be written more systematically, without casts
- *cons*:
  - for each new operation, need to write new dedicated methods and recompile
- **visitor pattern** approach:
  - divide the code into an object structure and a **visitor** (akin to functional programming)
  - insert an `accept` method in each class, which takes a Visitor as an argument
  - a visitor contains a `visit` method for each class (using *overloading*)
    - defines both actions and access of *subobjects*
  - *pros*:
    - new methods without recompilation
    - no frequent type casts
  - *cons*:
    - all classes need a hook in the `accept` method
  - used by tools such as JJTree, Java Tree Builder, JCC
  - summary, visitors:
    - make adding new operations easily
    - gather *related* operations
    - can accumulate state
    - can break encapsulation, since it needs access to internal operations

Third approach with visitor pattern:

```
interface List {
  // door open to let in a visitor into class internals
  void accept(Visitor v);
}
interface Visitor {
  void visit(Nil x); // code is packaged into a visitor
  void visit(Cons x);
}

class Nil implements List {
  // `this` is statically defined by the *enclosing* class
  public void accept(Visitor v) { v.visit(this); }
}
class Cons implements List {
  int head;
  List tail;
  public void accept(Visitor v) { v.visit(this); }
}

class SumVisitor implements Visitor {
  int sum = 0;
  public void visit(Nil x) {}
  public void visit(Cons x) {
    // take an action:
    sum += x.head;
    // handle subojects:
    x.tail.accept(this); // process tail *indirectly* recursively

    // The accept call will in turn call visit...
    // This pattern is called *double dispatching*.
    // Why not just visit(x.tail) ?
    // This *fails*, since x.tail is type List.
  }
}
```

Using `SumVisitor` :

```
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```

Java Tree Builder

- the produced JavaCC grammar can be processed by the JCC to give a parser

that produces syntax trees:
- the produced syntax trees can be traversed by a Java program by writing subclasses of the default visitor
- JavaCC grammar feeds into the **Java Tree Builder (JTB)**
- JTB creates JavaCC grammar with embedded Java code, syntax-tree-node classes, and a default visitor
- the new JavaCC grammar feeds into the JCC, which creates a parser
- `jtb fortran.jj` → `javacc jtb.out.jj` → `javac Main.java` → `java Main < prog.f`

Translating a grammar production with JTB:

```
// .jj grammar
void assignment() :
{}
{ PrimaryExpression() AssignmentOperator() Expression() }

// jtb.out.jj with embedded java code that builds syntax tree
Assignment Assignment () :
{
  PrimaryExpression n0;
  AssignmentOperator n1;
  Expression n2; {}
}
{
  n0 = PrimaryExpression()
  n1 = AssignmentOperator()
  n2 = Expression()
  { return new Assignment(n0, n1, n2); }
}
```

JTB creates this syntax-tree-node class representing `Assignment` :

```
public class Assignment implements Node {
  PrimaryExpression f0;
  AssignmentOperator f1;
  Expression f2;

  public Assignment(PrimaryExpression n0,
    AssignmentOperator n1, Expression n2) {
    f0 = n0; f1 = n1; f2 = n2;
  }

  public void accept(visitor.Visitor v) {
```

```java
    v.visit(this)
  }
}
```

Default DFS visitor:

```java
public class DepthFirstVisitor implements Visitor {
  ...
  // f0 -> PrimaryExpression()
  // f1 -> AssignmentExpression()
  // f2 => Expression()
  public void visit(Assignment n) {
    // no action taken on current node,
    // then recurse on subobjects
    n.f0.accept(this);
    n.f1.accept(this);
    n.f2.accept(this);
  }
}
```

Example visitor to print LHS of assignments:

```java
public class PrinterVisitor extends DepthFirstVisitor {
  public void visit(Assignment n) {
    // printing identifer on LHS
    System.out.println(n.f0.f0.toString());
    // no need to recurse into subobjects since assignments cannot be nested
  }
}
```

Type Checking

- a program may follow a grammar and parse properly, but other problems may remain
  - eg. type errors, use of undeclared variables, cyclical inheritance, etc.

Simple Expressions

- eg. in Java, `5 + true` gives a type error:
  - this type rule could be expressed as:
    - if two expressions have type `int`, then their addition will also be of type `int`
  - in typical notation:

$$\frac{a : \texttt{int} \quad b : \texttt{int}}{\texttt{a+b} : \texttt{int}}$$

    - in this notation, the **conclusion** appears under the bar, with multiple **hypotheses** above the bar
      - ie. if hypotheses are true, than conclusion is true
    - to *check* this rule, recursively check if e_1 is type `int`, and then if e_2 is also of type `int`
  - when given `5: int` and `true: boolean`, type checker will see that the types don't obey the rule, and should throw an error

Implementing a simple type checker:

```
class TypeChecker extends Visitor {
  // f1: Expression (e1)
  // f2: Expression (e2)
  Type visit(Plus n) { // should either return Plus or throw an error
    // recursive calls for subexpressions that *could* individually fail
    Type t1 = n.f1.accept(this);
    Type t2 = n.f2.accept(this);

    // making a decision, ie. actual type checking
    if (t1 == "int" && t2 == "int") { // need some mechanic to check type equality
      return "int";
    } else {
      throw new RuntimeException();
    }
  }
}
```

- what about for more complex compound type rules, eg. for parsing `3 + (5 + 7)` :
  - simply by following the recursive calls, the previous type checker would still successfully check the type
    - ie. type checking in a DFS manner

$$\frac{5 : \texttt{int} \quad 7 : \texttt{int}}{\texttt{5+7} : \texttt{int}} \rightarrow \frac{3 : \texttt{int} \quad 5 + 7 : \texttt{int}}{\texttt{3+(5+7)} : \texttt{int}}$$

- handling the simple nonterminal `true` :

$$\frac{}{\texttt{true} : \texttt{boolean}}$$

- handling boolean negation:

$$\frac{e : \texttt{boolean}}{\texttt{!e} : \texttt{boolean}}$$

- handling ternary expressions:

$$\frac{a : \texttt{boolean} \quad b : t \quad c : t}{\texttt{(a ? b : c)} : t}$$

  - t is a type variable since b and c should have the same type

Statements

- different types of statements in MiniJava:
  - `System.out.println(e)` , assignments, `if` and `while` statements
- unlike expressions, statements don't *return* anything:
  - they may have side effects, but do not have their *own* types
  - type checkers would only either return silently or throw an error
  - no overall type value to return
  - in typical notation $\vdash$ means that a sentence type-checks
    - not necessarily that it *is* a particular type
- handling `System.out.println` :

$$\frac{\vdash e : \texttt{int}}{\vdash \texttt{System.out.println(e)}}$$

- handling `if` statements:

$$\frac{\vdash e : \texttt{boolean} \qquad \vdash a \qquad \vdash b}{\vdash \texttt{if (e) a else b}}$$

- handling `while` statements:

$$\frac{\vdash e : \texttt{boolean} \qquad \vdash s}{\vdash \texttt{while (e) s}}$$

Declarations

- for declared variables, how can we track what specific identifiers represent?
  - create and add to a **symbol table** that caches the declaration of variables with types
  - maps identifiers to types
- type checking rule for an assignment statement, given the symbol table A:

$$\frac{A(x) = t \quad A \vdash e : t}{A \vdash \texttt{x=e}}$$

  - by convention, A should go before each $\vdash$ in all type checking rules
    - this is because any subexpressions may contain variables
  - ie. $A \vdash e : t$ can be read as expression e *given* program context from table A can *lead* to conclusion t when type checking

Symbol table example:

```
class C {
  boolean f;
  t m(int a) {
    int x,
    ...
    x = a + 5;
    ...
  }
}

// symbol table contains:
// id | type
// ---------
// f  | boolean
// a  | int
// x  | int
```

- to type check a variable, just look it up in the symbol table:

$$\frac{A(x) = t}{A \vdash \texttt{x} : t}$$

  - rewriting our symbol table lookup using the previous notation
- to type check `x = a + 5` :

$$\frac{A \vdash a : \texttt{int} \quad A \vdash 5 : \texttt{int}}{A \vdash \texttt{a+5} : \texttt{int}} \rightarrow \frac{A \vdash x : \texttt{int} \quad A \vdash a + 5 : \texttt{int}}{A \vdash \texttt{x=a+5}}$$

Example of variable shadowing:

```
class a {
  int a;
  int a(int a) {
    // int a;     // compiler error, can't redeclare parameter
    ... a + 5 ... // checks closest `a`, ie. read from bottom of symbol table
    return 0;
  }
}
```

Arrays

- type checking arrays:
  - expressions like `arr[idx], new int[len], arr.length`
  - statements like `arr[idx] = val`
  - note that it is *unreasonable* for the type checker to check out of bound indices
    - would require the type checker to do some arithmetic in this stage
- handling array indexing:

$$\frac{A \vdash a : \texttt{int[]} \quad A \vdash b : \texttt{int}}{A \vdash \texttt{a[b]} : \texttt{int}}$$

  - in MiniJava, the only array type is `int` , but arrays can generally hold any type t
  - note that the `int` in the conclusion refers to the array type while the other refers to the index type
  - note that this rule is an **elimination rule** since the `int[]` type is *consumed* into an `int` in the conclusion

- handling array assignments:

$$\frac{A \vdash x : \texttt{int[]} \quad A \vdash i : \texttt{int} \quad A \vdash v : \texttt{int}}{A \vdash \texttt{x[i] = v}}$$

- handling array constructions:

$$\frac{A \vdash e : \texttt{int}}{A \vdash \texttt{new int[e]} : \texttt{int[]}}$$

- handling the array length property:

$$\frac{A \vdash e : \texttt{int[]}}{A \vdash \texttt{e.length} : \texttt{int}}$$

Methods

- what needs to be checked in a method call in Java?
  - the method being called needs to *exist*
  - the type of the actual parameter should match the formal parameter

Declaring and calling methods in Java:

```
u2 m(u a) { // declaration
  s
  return e2;
}

m(e); // call
```

- when type checking the method body, need to ensure:

$$A \vdash s, A \vdash e_2 : u_2$$

- to type check the method call:

$$\frac{A \vdash e : t \quad A \vdash m : u \to u_2 \quad t = u}{A \vdash \texttt{m(e)} : u_2}$$

  - note that we need a mechanism to find the types of the method, eg. `m` takes parameter of type `u` and returns type `u2`

Classes

- expressions such as `new C(), this, (C) e` refer to classes:
  - need a new type C, representing some class:
    - C is another contextual record similar to the symbol table A
    - $A, C \vdash e : t$
    - ie. expression e given program context from table A and class C can lead to conclusion t when type checking
  - `this` refers to the lexically enclosing class type
- handling `new C()` :
  - need to check that C exists
- handling casts:

$$\frac{A, C \vdash e : t}{A, C \vdash \texttt{(D)e} : D}$$

  - again, may need a runtime check (like array indexing) to actually *perform* the type cast
  - in Java, *upcasts* will always succeed, while *downcasts* need a runtime check
- handling subtyping:
  - the notation $C \leq D$ indicates the class `C` is a subclass of the class `D` , perhaps transitively or reflexively
    - eg. for primitives as well, `char ≤ short ≤ int ≤ long ≤ float ≤ double`
  - note that:

$$\frac{}{C \leq C}, \frac{C \leq D \quad D \leq E}{C \leq E}$$

  - when $u \leq t$:
    - for polymorphic assignments:

$$\frac{A, C \vdash x : t \quad A, C \vdash e : u}{A, C \vdash \texttt{x=e}}$$

    - for method parameters:

$$\frac{A, C \vdash a : D \quad A, C \vdash e : u \quad \texttt{D.m} : t \to t_2}{A, C \vdash \texttt{a.m(e)} : t_2}$$

  - note that to check subclassing relationships, we can either use visitors to traverse the `extends` relationship on the fly, or cache a previous result

Polymorphism in Java:

```
// class B and C both extend class A
A x = (e ? new B() : new C());
// Compiler checks *both* B and C have correct subclassing relationship to A.
// At runtime, the type of x will be related to either B or C.
```

Extended method example:

```
class D {
  t1 f1
  t2 f2
  u3 m(u1 a1, u2 a2) {
    t1_1 x1
    t2_2 x2
    s
    return e;
  }
}
```

- hypotheses for type checking the above method:
  1. `a1, a2, x1, x2` are all different
  2. $A, C \vdash s$, where $C = D$:
     - but what should the symbol table A hold?
       * `f1:t1, f2:t2, a1:u1, a2:u2, x1:t1_1, x2: t2_2`
       * when using A, we should search from *bottom* of table to handle variable shadowing
  3. $A, C \vdash e : u_3'$, where $u_3' \leq u_3$
- conclusion from the above hypotheses:
  - $A, C \vdash$ `D.m` ie. the method `D.m` type-checks

Entire Java Program

Sample Java program:

```
class Main {...}

class C1 extends D1 {...}
...
class Cn extends Dn {...}
```

- type checking responsibilities:
  1. `main` must exist
  2. `C1...Cn` need to all be different

3. `D1...Dn` need to all exist
4. no `extends` cycle in classes

Generic Types

MiniJava generic type syntax:

```
class C <X*> extends N {
  S* f*; // class fields
  k // class constructor method

  // zero or more methods of the following pattern:
  <Y*> U m(U* x*) {
    ...
    return e;
  }
  ...
}

// additional new supported expressions:
e.<V*>m(e*)
new N (e*)

// Where N ::= C<T*>
// and S, T ::= N | X
// and U, V ::= N | Y
// and X, Y are *type parameters*.
// The asterisk indicates a vector of possibly multiple values.
```

Example with generic types:

```
class List<A> extends Object {
//         ^declaration of A

  <R>     R accept(ListVisitor<R,   A> f) {
// ^decl ^use                  ^use ^use

// The R,A in ListVisitor<R,A> are used as type vars
// in an instantiation of *another* class ListVisitor.

    return this.accept(f);
  }
}

class ListNull<B> extends List<B> {
//             ^decl           ^instantiates List with B as type var A

  <W>    W accept(ListVisitor<W,   B> f) {
// ^decl ^use                 ^use ^use
    return this.accept(f);
  }
}
```

- statements to type check in the above example:
  - `return this.accept(f)` trivially type checks due to recursion
  - `extends List<B>` establishes the inheritance relationship to `List`
    - while linking together `B` with the type variable `A` in the declaration of `List`
- to typecheck the entire generic class declaration:

$$\frac{\vdash \texttt{M*}\ \textit{in}\ \texttt{C<X*>}}{\vdash \texttt{class C<X*> extends N \{ S* f*; k M* \}}}$$

- to typecheck a generic method declaration:

$$\frac{\vdash x^* : U^* \quad \vdash \texttt{this} : \texttt{C<X*>} \quad \vdash e : S, S \le U \quad \textit{override}(m, N, \texttt{<Y*>U*} \to U)}{\vdash \texttt{<Y*> U m(U* x*) \{ return e; \}}\ \textit{in}\ \texttt{C<X*>}}$$

- to check method overriding:

$$\frac{\textit{mtype}(m, N) = \texttt{<Z*>U*} \to U \implies T^* \le U^*[Y^*/Z^*] \wedge T \le U[Y^*/Z^*]}{\textit{override}(m, N, \texttt{<Y*>T*} \to T)}$$

  - where $U[Y/Z]$ indicates the type U with all instances of type Y replaced with Z
    - this operation is used to *instantiate* a type variable declaration with a different name, ie. replacing a declaration with a usage
- to implement `mtype` :

$$\frac{\texttt{class C <X*> ... \{ ... M* \}} \quad m \in M^*}{\textit{mtype}(m, \texttt{C<T*>}) = (\texttt{<Y*>U*} \to U)[X^*/T^*]}$$

- and if the method is not found:

$$\frac{\texttt{class C <X*> extends N \{ ... M* \}} \quad m \notin M^*}{\textit{mtype}(m, \texttt{C<T*>}) = \textit{mtype}(m, N[X^*/T^*])}$$

- to type check a method call expression:

$$\frac{\vdash e : T \quad \textit{mtype}(m, T) = \texttt{<Y*>U*} \to U \quad \vdash e^* : S^* \quad S^* \leq U^*[V^*/Y^*]}{\vdash \texttt{e.<V>m(e*)} : [V^*/Y^*]U}$$

Sparrow

- **Sparrow** is the intermediate language used in CS132
- characteristics:
  - no classes
    - methods in classes have concatenated names, eg. `Fac.ComputeFac` becomes `FacComputeFac`
  - uses `goto` to implement `if else`
  - uses brackets to indicate heap loads or stores
    - no global variables, only heap and functions have global visibility
  - functions may have extra parameters added

Grammar

- a program `p` is a series of functions, `p ::= F1...Fm`
- a function declaration `F` has syntax `F ::= func f(id1...idf) b`
- a block `b` is a series of instructions, `b ::= i1...in return id`
- an instruction can be:
  - a label `l:`
  - an assignment `id = c` , where `c` is an integer literal:
    - or `id = @f` , where `f` is a function
  - an operation `id = id + id` , `id = id - id` , `id = id * id`
  - a less-than test `id = id < id`
  - a heap load or store:
    - `id = [id + c]` , `[id + c] = id` , where `c` is the offset and heap addresses are valid
  - an allocation `id = alloc(id)` :
    - eg. `v0 = alloc(12)` allocates 12 bytes
    - creates a 3-tuple of addresses to values accessible by `[v0 + 0]` , `[v0 + 4]` , `[v0 + 8]`
  - a print `print(id)`
  - an error print `error(s)`
  - an unconditional goto `goto l` , where `l` is a label
  - a conditional goto `if0 id goto l` , jumps if `id` contains 0
  - a function call `id = call id(id1...idf)` , where `id` contains a function name
- identifiers can be any reasonable identifiers, except:
  - `a2...a7` , `s1...s11` , `t0...t5` which are all RISC register names

Translation to IR

- overall translation pipeline is:
  1. MiniJava:
     - mostly *unbounded*, eg. in the number of variables, parameters, methods, classes, etc.
  2. Sparrow:
     - still unbounded
     - may even generate *new* variables for simplicity / ease of translation
  3. Sparrow-V:
     - number of registers becomes bounded
     - want to minimize variables allocated on the stack
  4. RISC-V:
     - register count still bounded

State and Transitions

- program *state* consists of the tuple (p, H, b^*, E, b):
  - p is the program
  - H is the *heap* that maps from heap addresses to *tuples* of values
    * the tuple can be *indexed* into
  - b^* is the body of the function that is executing right now
    * can only perform a `goto` within this function block
  - E is the environment that maps from identifiers to values
  - b is the remaining part of the block that is executing right now
    * ie. b^* contains the entire function block, while b only contains the current and remaining statements in the block
- in a state transition, we want to *step* from one state to the next:

$$(p, H, b^*, E, b) \rightarrow (p, H', {b^*}', E', b')$$

- assignment state transition:

$$(p, H, b^*, E, \texttt{id=c}\ \ b) \rightarrow (p, H, b^*, E \cdot [id \mapsto c], b)$$

- arithmetic state transition:

$$(p, H, b^*, E, \texttt{id=id1+id2}\ \ b) \rightarrow (p, H, b^*, E \cdot [id \mapsto (c_1 + c_2)], b)$$

  - where $E(id_1) = c_1$ and $E(id_2) = c_2$

  - ie. this transition requires a runtime check in the environment
- heap load state transition:

$$(p, H, b^*, E, \texttt{id=[id1+c]}\ \ b) \to (p, H, b^*, E \cdot [id \mapsto (H(a_1))(c_1 + c)], b)$$

  - where $E(id_1) = (a_1, c_1)$ such that a_1 is a heap address and c_1 its offset, and $(c_1 + c) \in domain(H(a_1))$
  - ie. the new computed offset is a valid index into the tuple on the heap
- heap allocation state transition:

$$(p, H, b^*, E, \texttt{id=alloc(id1)}\ \ b) \to (p, H \cdot [a \mapsto t], b^*, E, b)$$

  - where $E(id_1) = c$ and c is divisible by 4, a is a *fresh* address, and $t = [0 \mapsto 0, 4 \mapsto 0, ...(c-4) \mapsto 0]$
- unconditional `goto` state transition:

$$(p, H, b^*, E, \texttt{goto l}\ \ b) \to (p, H, b^*, E, b')$$

  - where $find(b^*, l) = b'$
  - $find(b, l)$ is used to find the label l inside the block b
- conditional `goto` state transition:

$$(p, H, b^*, E, \texttt{if0 id goto l}\ \ b) \to (p, H, b^*, E, b')$$

  - where $E(id) = 0$ and $find(B^*, l) = b'$
- function call state transition:

$$(p, H, b^*, E, \texttt{id = call id0(id1...idf)}\ \ b) \to (p, H', b^*, E \cdot [id \mapsto E'[id']], b)$$

  - where $E(id_0) = f$, p contains `func f (id1'...idf') b'`, and $E' = [id'_1 \mapsto E(id_1), id'_2 \mapsto E(id_2), ...]$
  - the function f is then *called* through the following state transition:
    - $(p, H, b', E', b') \to (p, H', b', E', \texttt{return}\ \ id')$
  - note the intermediate transfer of control to the callee

Expressions

- want to translate $e, k \to c, k'$ where e is some expression in MiniJava and c is the output code in Sparrow, while managing:
  1. additional variables (that could have been added during translation)

2. labels used in jumps in IR

- k is a *"fresh"* or new integer number that has not yet been used
  - can be utilized to generate variable and label names
- after translation, the number k' should be the next *new* number
- by convention, the *result* of the expression e is stored in the variable t_k

- simple expression:

  `5` $, k \to$ `tk = 5` $, k+1$

- expression with a local variable:

  `x` $, k \to$ `tk = x` $, k+1$

- recursive translations:

  `e1+e2` $, k \to c_1 c_2$ `tk=tl+tk1` $, k_2$

  - given that $e_1, k+1 \to c_1, k_1$ and $e_2, k_1 \to c_2, k_2$, and where $t_l = t_{k+1}$
  - note that by convention, c_1 will be stored in t_{k+1} and c_2 will be stored in t_{k1}

- example addition translation:

  `7+9` $, 3 \to$ `t4=7 t5=9 t3=t4+t5` $, 7$

  - after `7` $, 4 \to$ `t4=7` $, 5$ and `9` $, 5 \to$ `t5=9` $, 6$

- example nested addition translation:

  `(7+9)+11` $, 3 \to$ `t5=7 t6=9 t4=t5+t6 t7=11 t3=t4+t7` $, 8$

  - `7` $, 5 \to$ `t5=7` $, 6$ and `9` $, 6 \to$ `t6=9` $, 7$
  - `7+9` $, 4 \to$ `t5=7 t6=9 t4=t5+t6` $, 7$
  - `11` $, 7 \to$ `t7=11` $, 8$
  - variables `t4...t6` handle `7+9` , `t7` handles `11` , and `t3` holds the entire expression

Statements

- want to translate $s, k \to c, k'$ where s is some statement in MiniJava and c is the output code in Sparrow
  - k' will differ from k when a subexpression is contained within the statement

- simple recursive statements:

$$s_1 s_2, k \to c_1 c_2, k_2$$

  - where $s_1, k \to c_1, k_1$ and $s_2, k_1 \to c_2, k_2$
  - no return value since statements only have a side effect, so only ordering of output code matters
- simple assignment statement:

$$\texttt{x=e}, k \to c\ \ \texttt{x=tk}, k'$$

  - where $e, k \to c, k'$ and the result of c is held in variable t_k by convention
- `if else` statement:

$$\texttt{if(e) s1 else s2}, k \to c_k, k_2$$

  - $e, k+1 \to c_e, k_e$
  - $s_1, k_e \to c_1, k_1$
  - $s_2, k_1 \to c_2, k_2$
  - to generate control code, need to *generate* labels and use jumps:
    - can use k in order to avoid label duplication
    - also need to *linearize* the code while allowing s_1 to execute without s_2, and vice versa
    - uses an unconditional jump and a conditional jump

c_k, the generated code from translating the `if else` statement:

```
c_e // result stored in temporary t_{k+1}
if0 t_{k+1} goto else_k
  c1
  goto end_k
else_k:
  c2
end_k:
```

- `while` statement:

$$\texttt{while(e) s}, k \to c_k, k_s$$

  - $e, k+1 \to c_e, k_e$
  - $s, k_e \to c_s, k_s$
  - again, need to generate control code
  - note that the compiler does not care about how many times the loop runs
    - thus the label subscript k simply depends on static code structure
  - conventially, `for` loops are usually reduced into `while` loops

c_k, the generated code from translating the `while` statement:

```
loop_k:
c_e // result stored in temporary t_{k+1}
if0 t_{k+1} goto end_k
  c_s
  goto loop_k
end_k:
```

Examples

To translate $s, k \rightarrow, c, k'$ for a simple sequence s of two substatements using visitors:

```
class Result {
  String code;
  int k1;
}

Result visit(Seq n, int k) {
  Result res1 = n.f1.accept(this, k);
  Result res2 = n.f2.accept(this, res1.k);
  return new Result(res1.code + res2.code, res2.k);
}
```

Translating the following code starting from $k = 0$:

```
while (true) {
  if (false)
    x = 5;
  else
    y = 7;
}
```

Translation process:

```
// handling `while (e) s`:
true, 1 → t1 = 1, 2 // e

// handling s, which has the form `if (e1) s1 else s2`:
false, 3 → t3 = 0, 4 // e1

5, 4   → t4 = 5, 5 // s1
x=5, 4 → x = t4, 5
```

```
7, 5   → t5 = 7, 6 // s2
y=7, 5 → x = t5, 6

if (false) x=5 else y=7, 2 → { // s
  t3 = 0
  if0 t3 goto else2
  t4 = 5
  x = t4
  goto end2
  else2:
    t5 = 7
    y = t5
  end2:
}, 6

while (true) {if (false) x=5 else y=7}, 0 → {
  loop0:
    t1 = 1
    if0 t1 goto end0
    // code generated from s, the if-else statement
    goto loop0
  end0:
}, 6
```

Arrays

- need to generate Sparrow IR for the following MiniJava code involving arrays:
  - expressions like `arr[idx], new int[len], arr.length`
  - statements like `arr[idx] = val`
- in Sparrow, arrays will be represented on the heap:
  - can create an array by allocating space on the heap
  - need to *track* and store the array length somewhere:
    - by *convention*, simply store the length of the array at position 0 on the heap, and shift array positions in the heap accordingly
    - thus in Sparrow, first array element is stored at offset 4, and last element is at offset $4n$ where n is array length
    - total amount to allocate is $4 \times (n+1)$ to store length
- array length:

  `e.length` $, k \to c$ `tk=[tl+0]` $, k'$

  - after evaluating $e, k+1 \to c, k'$ and where $t_l = t_{k+1}$

- array allocation:

  `new int[e]`, $k \to c$ `tk'=4*(tl+1) tk=alloc(tk') [tk+0]=tl`, $k' + 1$

  - after evaluating $e, k + 1 \to c, k'$ and where $t_l = t_{k+1}$
  - t'_k is a new temporary to store $4 * (e + 1)$ while saving the original array length to store back in array
  - in the Sparrow spec, `alloc` will initialize all heap entries to zero
- array indexing:

  `e1[e2]`, $k \to c_1 c_2$ `tk2=4*(tk1+1) tl=tkl+tk2 tk=[tl+0]`, $k_2 + 1$

  - after evaluating $e_1, k + 1 \to c_1, k_1$ and $e_2, k_1 \to c_2, k_2$, and where $t_l = t_{k+1}$
  - note that the second part of a heap lookup must be a constant so we build up the entire dynamic offset in the first parameter
  - to check bounds, need to ensure that $0 \leq e_2 < n$ where n is the length of the array in e_1
  - similar process for array asignments in the form `a[e1] = e2`

Implementing a bounds check:

```
t_{k2+1} = -1 < tk1
t_{k2+2} = [t_{k+1} + 0]
t_{k2+3} = tk1 < t_{k2+2}
t_{k2+4} = t_{k2+1} * t_{k2+3} // multiplication acts as logical AND
// including a bounds check uses up to k2+5 for temporaries

if0 t_{k2+4} goto error
... // loading or storing code here
goto end
error:
  error("out of bounds")
end:
```

Classes

MiniJava method example:

```
class C {
  Y m(T a) {
    S;
    return e;
  }
}
```

In Sparrow, there are no more classes:

```
func C.m(this a) { // name mangling
  c
  return x
}
```

- how to implement objects in Sparrow?
  - like arrays, represent objects on the heap
  - object fields will be given by offsets into the heap
    - eg. to access the field $f, k \rightarrow$ `tk=[this+L]`, $k+1$ where L is the static offset remembered by the compiler associated with the field f
  - but how can we handle `this` ie. the current object?
    - need to translate the method call `e1.m(e2)` into `C.m(t1 t2)`
    - where `t1, t2` hold the expression results of `e1, e2` respectively
    - note that we can *identity* the full mangled name for `e1.m` by finding the *type* `C` of `e1`

Inheritance

- when we allow for inheritance and `extends`, we no longer know exactly which methods we are calling
  - similarly, object fields can also be inherited and accessed in subclasses

MiniJava inheritance example:

```
class B {
  t p() { this.m(); }
  u m() { ... }
}

class C extends B {
  u m()  { ... } // overrides B.m
}
```

In Sparrow:

```
B.p(this) {
  B.m(...) // this is now *wrong*, could be called with C instead of B
}
B.m(this) { ... }
```

```
C.m(this) { ... }
```

- to handle inheritance, need to add an additional level of *indirection*:
  - add a **method table** for all objects:
    - by convention, method table stored at position 0, similarly to where length is stored in arrays
    - shift locations of object fields by 4
  - method table contains entries pertaining to all inherited visible methods
    - each entry holds the address of a function
  - for the above example:
    - a `B` object's method table should contains references to `B.p, B.m`
    - while a `C` object's method table should contains references to `B.p, C.m`
    - note that the offsets across related objects should line up
  - method tables are *shared* across objects of the same class, so each class can have a single method table allocated and all objects will point to that single copy
  - to call a class function in OOP:
    1. *load* method table
    2. *load* function name
       - additional indirection to handle inheritance
    3. *call*
- consider building method tables for the following inheritance relationship:
  - class `A` has methods `m, n`
    - method table holds refs. to `A.m, A.n`
  - class `B extends A` has no methods
    - method table holds refs. to `A.m, A.n`
  - class `C extends B` with methods `p, m`
    - method table holds refs. to `C.m, A.n, C.p` , regardless of method definition order
  - class `F extends C` with methods `q`
    - method table holds refs. to `C.m, A.n, C.p, F.q`
  - thus compiler needs to associate the method `n` with offset 4 into method tables, etc.

Control Flow Analysis

- generally, calling a method `e.m()` is implemented using a load, load, call on the method table:
  - in some *special* scenarios involving a **unique target**, we can replace this implementation with a single *direct* call

  - ie. statically replacing a general call sequence with a much faster, specialized one
- what are the characteristics that make a call have a unique target?
  1. what are the classes of the results of `e` in `e.m()` ?
     - eg. classes `A, B, C`
  2. which methods `m` can be called?
     - eg. two methods, `A.m` and `C.m`
  3. is the `m` that is called always the same?
     - if not, eg. `A.m` is different from `C.m` , cannot go to specialized sequence
  - but `e` may have nested callsites within it
    * thus answering question (1) requires cyclically answering (2)
- old approach used at *runtime* in Objective-C:
  - instead of having individual method tables for each object, have one giant method table at runtime representing the methods for all possible object types and their method names
  - at runtime, check this large method table to find which method to call
    * requires a lookup into the table, as well as storing a representation of the type in each object
- a *static* approach is to perform **Class Hierarchy Analysis (CHA)**:
  - revolves around knowing class hierarchies
  - consider the simple hierarchy `A -> B -> C` , with methods `A.m, C.m` :
    * the declaration `C x` and later call `x.m()` has multiple possibilities, according to the hierarchy
    * the code `x = new C()` whould lead to calling `C.m` , while `x = new B()` would call `A.m`
  - CHA approach:
    1. use type declarations
    2. use the class hierarchy
- another *static* approach called **Rapid Type Analysis (RTA)**:
  - again uses class hirearchies
  - consider the previous example from CHA:
    * RTA would notice that there is no `x = new C()` in the program
    * thus RTA will realize that `x.m()` has the unique target of `A.m`
  - requires just another linear scan to check for `new` statements
  - RTA approach:
    1. use type declarations
    2. use the class hierarchy
    3. use `new` statements
- another *static* approach called **Type-Safe Method Inlining (TSMI)**:
  - takes an entire method and inlines it in the call whenever it is a unique target

  - not as expensive or good as CFA

Illustrating issue with TSMI:

```java
interface I { void m(); }

class C implements I {
  C f;
  void m() {
    this.f = this;
  }
}

I x = new C();
x.m();
// becomes translated to ⟹
x.f = x; // an error since x has type I
         // TSMI needs to detect that I will only have type C
```

0CFA

- another *static* approach is called level 0 **Control Flow Analysis (0CFA)**
- goes from syntax, to constraints, to information
  - want to generate variables $[e]$ for each `e` that range over sets of possible class names `e` can take
- CFA approach:
  1. starting from syntax, *generate* constraints by performing a linear pass over the code
  2. *solve* constraints to get the desired sets of class names
  - runs in $O(n^3)$
- syntax of interest and the constraints they imply:
  - `new C()` :
    - implies the constraint $[$`new C()`$] = \{C\}$
  - `x = e` :
    - implies the constraint $[e] \subseteq [x]$
  - `e1.m(e2)` and `class C { _ m (_ a) { return e } }` :
    - imply the constraint:
      - if $C \subseteq [e1]$, then $[e2] \subseteq [a] \wedge [e] \subseteq [e1.m(e2)]$

Code example:

```java
A x = new A();
B y = new B();
x.m(new Q());
```

```
y.m(new S());

class A {
  void m(Q arg) {
    arg.p();
  }
}

class B extends A {
  void m(Q arg) {...}
}

class Q { ... }
class S extends Q { ... }
```

- CHA:
  1. `x.m` no unique target
  2. `y.m` unique target, nothing overrides `B.m`
  3. `arg.p` no unique target
- RTA:
  1. `x.m` still no unique target, both `new A()` and `new B()` occur
  2. `y.m` unique target, only one `new`
  3. `arg.p` no unique target
- CFA:
  - `A x = new A()` implies $[x] \subseteq$ [`new A()`] $\wedge$ [`new A()`] $= \{A\}$
  - `x.m(new Q())` and `class A { void m(Q arg) { return arg.p(); }}`
    - implies that if $A \in [x]$ then [`new Q()`] $\subseteq [arg]$
    - and [`arg.p()`] $\subseteq$ [`x.m(new Q())`]
  - etc.
  1. `x.m` unique target
  2. `y.m` unique target
  3. `arg.p` unique target
- 1CFA:
  - spends *exponential* time when generating constraints
  - but is able to catch inconsistencies in 0CFA that are more likely to detect unique targets
  - essentially, uses separate copies of constrained methods instead of just one

Lambda Expressions

- want to compile **lambda expressions** while moving from recursion (more expensive stack) to iteration
- translate lambda expressions to an intermediate form in three steps / approaches:
  1. tail form
     - functions never return
     - using continuations
  2. first-order form
     - functions are all top level
     - using data structures
  3. imperative form
     - functions take no arguments
     - using register allocation

Illustrating recursion vs. iteration:

```java
// recursion:
static Function<Integer,  Integer> // in class Test
//              ^param(s) ^return type
fact = n ->
  n == 0 ? 1 : n * Test.fact.apply(n-1); // builds up a stack

// iteration:
static int factIter(int n) {
  int a = 1;
  while (n != 0) {
    a = n * a;
    n = n - 1;
  }
  return a;
}

// low-level iteration with goto:
factIter: a = 1;
Loop:     if (n == 0) { }
          else { a = n*a; n = n-1; goto Loop }
```

Tail Form

- in **continuation passing style (CPS)**:
  - the **continuation** is the code representing the returning of the computation
  - instead of returning from a function, just call the continuation

General program to tail form:

```
// use CPS:
static BiFunction<Integer, Function<Integer, Intenger>, Integer>
factCPS = (n, k) →
  n == 0 ?
    k.apply(1)
  : Test.factCPS.apply(n-1, v → k.apply(n * v)); // builds up continuation

factCPS.apply(4, v → v); // how to actually call CPS function
// evaluation of a tail form has *one call* as the last operation
```

- evaluation of a **tail form** expression has *one call* as the last operation:
  - `Tail ::= Simple | Simple.apply(Simple*) | Simple ? Tail : Tail`
  - while evaluation of a **simple** expression has *no* calls
    - `Simple ::= Id | Constant | Simple PrimitiveOp Simple | Id → Tail`

CPS transformation rules:

```
static Function<...> foo = x → ...
// ⟹
static BiFunction<...'> fooCPS = (x, k) → k.apply(...)

k.apply(... (foo.apply(a, n-1)) ...) // can only have one call in CPS!
// ⟹
fooCPS.apply(a, n-1, v → k.apply(... v ...)) // extract out foo.apply call

k.apply(foo.apply(a, n-1)) // special case
// ⟹
fooCPS.apply(a, n-1, k);

k.apply(y ? ... : ...)
// ⟹
y ? k.apply(...) : k.apply(...)

k.apply(foo.apply(a) ? ... : ...)
// ⟹
fooCPS(x, v → k.apply(v ? ... : ...))
// ⟹
fooCPS(x, v → v ? k.apply(...) : k.apply(...))
```

Translating original `fact` into its tail form, `factCPS` :

```
fact = n → n == 0 ? 1 : n * Test.fact.apply(n-1);
// ⟹
```

```
factCPS = (n, k) → k.apply(
  n == 0 ? 1 : n * Test.fact.apply(n-1));
// ⟹
factCPS = (n, k) →
  n == 0 ? k.apply(1) : k.apply(n * Test.fact.apply(n-1));
// ⟹
factCPS = (n, k) →
  n == 0 ? k.apply(1) : Test.factCPS.apply(n-1, v → k.apply(n * v))
```

First-Order Form

- in CPS form, there are really *two* continuations at play, `k` and `v → k.apply(...)`:
  - `Cont ::= v → v | v → Cont.apply(n * v)`
  - if we can represent continuations without lambda functions, would only have functions at the top level

Implementing continuations as purely datatypes:

```
interface Continuation {
  Integer apply(Integer a);
}

class Identity implements Continuation {
  public Integer apply(Integer a) { return a; }
}

class FactRec implements Continuation {
  Integer n; Continuation k;
  public FactRec(Integer n, Continuation k) {
    this.n = n; this.k = k;
  }

  public Integer apply(Integer v) { return k.apply(n * v); }
}
```

CPS to first-order form:

```
static BiFunction<Integer, Continuation, Integer>
facCPSadt = (n, k) → // adt - abstract data type
  n == 0 ? k.apply(1) :
    Test.factCPSadt.apply(n-1, new FactRec(n, k));
```

Imperative Form

- from first-order, we can prove that it is possible to purely represent a continuation as a *number*:
  - every continuation (at least for `factCPS`) is of the form `v -> p * v`
  - eg. `k.apply(1)` is just `p * 1 = p`
  - eg. `v -> k.apply(n * v)` is just `v -> (p * n) * v` or just `(p * n)`
  - this simplification can only work for some primitive operations, but in general, all first-order forms can be expressed as iteration

First-order form to imperative form:

```
static BiFunction<Integer, Integer, Integer>
factCPSnum = (n, k) -> n == 0 ? k :
  Test.factCPSnum.apply(n-1, k * n);

// get rid of function parameters and use global registers:

static Integer n; static Integer k;
static void factCPSimp() {
  if (n == 0) { }
  else { k = k*n; n = n-1; factCPSimp(); }
}

// to low-level iteration with gotos:

static Integer n; static Integer k;
factCPSimp:
  if (n == 0) { }
  else { k = k*n; n = n-1; goto factCPSimp }
```

Register Allocation

- in Sparrow, all variables are on the stack, which forces relatively *slow* access:
  - in the Sparrow-V IR, *some* variables are held in registers for *faster* access
  - want to fit as many variables as possible into the registers in a process called **register allocation**
- register allocation can be broken down into two subproblems:
  1. liveness analysis
  2. graph coloring (such that no adjacent nodes are the same color)
  - interface betwen the two steps is an **interference graph**
  - an approach using heuristics
- incremental steps for saving stack variables:
  1. no registers used
  2. use registers for storing all subcomputations
  3. use registers to pass function parameters
     - have to save all registers before function calls in case they are *clobbered*
  4. avoid saving registers that are not utilized after function calls
  5. using liveness analysis to reuse registers

Liveness Analysis

Sparrow to Sparrow-V example:

```
// Sparrow
a = 1       // prog. pt. 1 (point lies just before instruction)
b = 2       // prog. pt. 2
c = a + 3   // prog. pt. 3
print b + c // prog. pt. 4

// Sparrow-V with two registers (s0, s1)
s0 = 1
s1 = 2     // cannot assign to s0, would clobber `a`
c = s0 + 3 // leave `c` on the stack
// but `a` is no longer used, can reuse a register
print s1 + c

// Sparrow-V attempt 2
s0 = 1
s1 = 2
```

```
s0 = s0 + 3 // load before store
print s1 + s0
```

- **program points** lie between lines of instructions:
  - ie. where labels can be introduced
  - first program point lies before the first instruction
- in liveness analysis, want to examine:
  - *when* is a variable *live*, or what is its **live range**?
    - `a` is live in the range $[1, 3]$, ie. from 1 *up to* 3
    - `b` is live in $[2, 4]$
    - `c` is live in $[3, 4]$
  - note that the live ranges for `a, b` and `b, c` overlap, while the live ranges for `a, c` do *not*
    - ie. overlapping live ranges *conflict*
- in an **inteference graph**:
  - nodes represents variables
  - edges represent conflicts
  - allocating registers is similar to coloring this graph

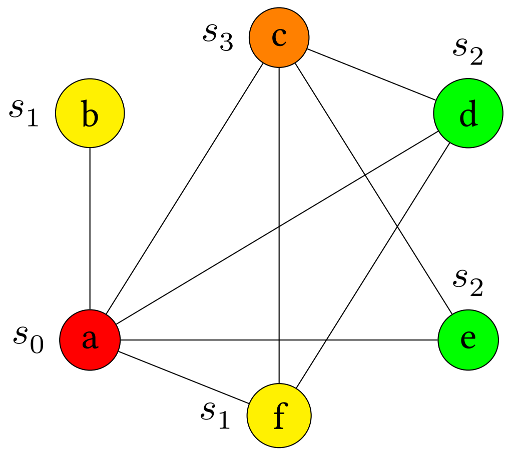
What is the fewest number of registers to entirely allocate the following code?

```
a = 1        // p.1
b = 10       // p.2
c = 8 + b    // p.3
d = a + c    // p.4
e = a + d    // p.5
f = c + e    // p.6
d = a + f    // p.7
c = d + f    // p.8
f = a + c    // p.9
return c + f // p.10
```

Live ranges, from inspection:

```
a : [1, 9]
b : [2, 3]
c : [3, 6] *and* [8, 10] (c can be reused before it is declared again)
d : [4, 5] and [7, 8]
e : [5, 6]
f : [6, 8] and [9, 10]
```

Corresponding inteference graph:



Sparrow-V output code:

```
s0 = 1
s1 = 10
s3 = 8 + s1
s2 = s0 + s3
s1 = s0 + s2
s1 = s3 + s1
s2 = s0 + s1
s3 = s2 + s1
s1 = s0 + s3
return s3 + s1
```

Algorithm

- given some statement n:
  - $def[n]$ includes the variables defined or assigned in n
  - $use[n]$ includes the variables used in n
  - $in[n]$ includes the variables live coming *in* to n
  - $out[n]$ includes the variables live coming *out* of n
- in a **control-flow graph**:
  - nodes are statements
  - directed edges are possible control flow directions
  - a node's successor(s) are the nodes reachable from it
- defining **liveness equations**:

$$out[n] = \bigcup_{s \in succ(n)} in[s]$$

$$in[n] = use[n] \bigcup (out[n] - def[n])$$

Example code:

```
a = 0              // 1
L1: b = a + 1      // 2
c = c + b          // 3
a = b * 2          // 4
if a < 10 goto L1  // 5
return c           // 6

// in control-flow graph:
// 1→2, 2→3, 3→4, 4→5, 5→6, 5→2
```

Building liveness equations for example code:

n	def	use	in_1	out_1	in_2	out_2	in_3	out_3
1	a			a		a, c	c	a, c
2	b	a	a	b, c	a, c	b, c	a, c	b, c
3	c	b, c	b, c	b	b, c	b	b, c	b, c
4	a	b	b	a	b	a, c	b, c	a, c
5		a	a	a, c	a, c	a, c	a, c	a, c
6		c	c		c		c	

- algorithm steps:
  1. initially, in and out are both $\emptyset$
  2. use liveness equations as update steps, iteratively
     - in the table, in_k represents in at iteration k
  3. keep iterating until *no* change
  - give the final in and out values, we can determine intefering variables:
    * every variable in the same box interferes with each other, pairwise
  - given n statements and $O(n)$ variables, there are $O(n^2)$ iterations:
    * ie. at each stage, at least one element will be added to out
    * at each iteration, n set-unions are performed each with a runtime of $O(n)$
  - thus, the algortihm has $O(n^4)$ runtime
    * can be reduced to $O(n^2)$ with more detailed analysis

Graph Coloring

- **graph coloring** ie. **liveness allocation** is the problem of allocating colors / registers given live ranges
- liveness analysis in *linear* time:

  - instead of the full $O(n^4)$ liveness analysis algorithm, is there a linear time approximation?
  - instead of dividing up live ranges, simply take the entire interval from a variable's first declaration to its last use:
    - intervals can be retrieved in linear time
    - ignores gaps in live ranges, and interpolates them
    - seems to be a reasonable approximation that still indicates intefering variables

- **linear scan** liveness allocation:

  - now that we have a linear liveness analysis algorithm, want to achieve liveness *allocation* in linear time as well:
    - will not be as thorough of an allocation as running the full $O(n^4)$ liveness analysis and a complete graph coloring in exponential time
    - but should be a good estimate, using heuristics

- linear scan algorithm:

  1. sort live range intervals by starting point
  2. perform a *greedy* coloring in a left-to-right scan:
     - tentatively assign different color ranges to colors ie. registers while the ranges overlap
     - once a previous interval *expires* before the start time of the current interval, *finalize* its coloring

  - when we have run out of registers, we have to **spill** a variable onto the stack:
    - ie. store it on the stack instead of in registers
    - as a heuristic, spill the one that extends the *furthest* into the future (based on end time)
    - note that we can *only* take over ie. spill tentatively assigned registers
  - thus, this algorithm has $O(n)$ time
    - there are only a constant number of registers to track tentative assignments for

Liveness analysis and allocation example using linear algorithms:

```
a = 1     // 1
b = 10    // 2
c = 9 + a // 3
d = a + c // 4
e = c + d // 5
f = b + 8 // 6
c = f + e // 7
```

```
f = e + c  // 8
b = c + 5  // 9
return f   // 10
```

Live range intervals (using linear approximation):

```
a : [1, 4]
b : [2, 9]
c : [3, 9]
d : [4, 5]
e : [5, 8]
f : [6, 10]
// note here that the intervals in this problem are already in sorted order
```

- given three registers `r1, r2, r3` to color with:
    1. assign `r1` to `a`
    2. assign `r2` to `b`
    3. assign `r3` to `c`
    4. `a` has expired, finalize `r1` for `a`
        - assign `r1` to `d`
    5. `d` immediately expires, finalize `r1` for `d` as well
        - assign `r1` to `e`
    6. no more registers, have to spill:
        - spill `f` onto the stack
    - coloring is complete

Allocation results using three registers:

```
a : r1
b : r2
c : r3
d : r1
e : r1
f : <mem>
// r1 can be reused multiple times without inteferences
```

- given two registers `r1, r2` to color with:
    1. assign `r1` to `a`
    2. assign `r2` to `b`
    3. no more registers, have to spill:
        - spill `b` onto the stack (could also spill `c`)
        - assign `r2` to `c`
    4. `a` has expired, finalize `r1` for `a`
        - assign `r1` to `d`
    5. `d` immediately expires, finalize `r1` for `d` as well

    - assign `r1` to `e`
6. no more registers, have to spill:
    - spill `f` onto the stack
- coloring is complete

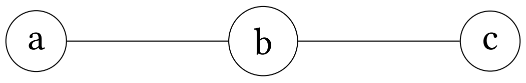
Allocation results using two registers:

```
a : r1
b : <mem>
c : r2
d : r1
e : r1
f : <mem>
// r1 can be reused multiple times without inteferences
```

NP-Completeness

- in a full inteference graph, generated from running the full liveness analysis algorithm:
    - $k \geq s$, where k is the minimum number of colors and s is the size of the max clique
    - *however*, if all the live ranges are presented as intervals, $k = s$
    - in general, graph coloring is NP-complete
        - by using intervals, graph coloring becomes a polynomial time operation
- liveness analysis transforms the register allocation problem into a graph coloring problem:
    - do we *lose* anything in this transformation?
        - ie. is the register allocation problem itself also an NP-complete problem?
        - can transform in the other direction, from a coloring to an allocation problem
        - eg. transform a certain graph into an equivalent code segment to run register allocation over, and run it through liveness allocation in order to color it

Graph coloring example:



From graph coloring to a representative program:

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

// consider the graph as an inteference graph
print(a + b) // a and b overlap, ie. both are live
print(b + c) // b and c overlap
// meanwhile, a and c are never live at the same time
```

- because this problem is transformable, register allocation is indeed an NP-complete problem:
  - thus, if the compiler should perform anything *smart* in its execution, it should do register allocation
  - register allocation by hand is unfeasible considering the very good linear approximation the compiler can quickly perform

Copy Propagation

Sample code illustrating when copy propagation can be used:

```
b = a // copying statement from a to b (without changes), can we get rid of it?
c = b
d = b
// ⟹
b = a // this is a useless statement
c = a
d = a
// ⟹
c = a
d = a
```

- when could this type of copy propagation be performed?
  1. formally, given the control flow graph of the code:
     - the top node (`b = a`) **dominates** the other lower node (`c = b`)
     - ie. every path from above one node that makes it to the other, must go through the first node
     - ie. we will always *“hit”* it
  2. there are no *updates* to `a` or `b` between `b = a` and `c = b`

  - then we can update `b = a` and `c = b` as just `c = a`
  - when translating Java to Sparrow, we generated a lot of extra copy statements for ease of development

    * this is handled by copy propagation

- comparing types of data-flow analysis:
  - in **liveness analysis**, we keep track of defines and uses in the sets in and out:
    * the variables range over sets of program *variables*
    * given constraints of equalities over sets, we work backwards
  - in **copy propagation**, we keep track of *"gens"* and *"kills"* again in the sets in and out:
    * the variables range over sets of copy *statements*
    * given constraints of equalities over sets, we work forwards, ie. push copy statements *forward*
- defining copy propagation equations:

$$in[n] = \bigcap_{p \in pred(n)} out(p)$$

$$out[n] = gen(n) \bigcup (in(n) - kill(n))$$

- for the statement n in the program, where:
  - $pred(n)$ contains the predecessors of the statement in the control flow graph
  - $gen(n) = \{n\}$ if n is a copy statement, and $\emptyset$ otherwise
  - $kill(n)$ contains the statement `b = a` anywhere in the program, or whenever n assigns to either `b` or `a`
  - very similar equations to the liveness analysis equations, but working *backwards* in the other direction
    * in addition, liveness analysis uses a large union ie. cares about all live variables
    * while copy propagation uses a large intersection ie. picking which statements to propagate
- algorithm:
  1. initialize all set variables to $\emptyset$
  2. repeatedly update in, until no change
  - with $O(n^4)$ complexity, or $O(n^2)$ complexity with more detailed analysis
- another similar optimization called **constant propagation** can also performed

Activation Records

- we have not yet addressed **calling conventions**, ie. implementing the procedure abstraction in the compiler
- need to *ensure* that every procedure:
  - inherits a valid run-time environment
  - restores an environment for its parent
- the **procedure abstraction** involves some caller p and callee q:
  - on entry of p, establish an environment
  - around a call of q, preserve the environment of p
  - on exit of p, tear down its environment
  - in between, handle addressability and proper lifetimes
- an **activation record** is a stack frame:
  - each procedure call will have an associated activation record at run time
  - two registers point to relevant activation records:
    - the **stack pointer** pointing to the end of the current stack
    - the **frame pointer** pointing to the end of the parent's stack
  - information stored on the stack:
    - *incoming* arguments from the caller
      - by convention, saved right before the frame pointer
    - return address
    - local variables
    - temporaries
    - saved registers
    - outgoing arguments
    - return value?
  - because functions may make different method calls dynamically, the number of outgoing arguments varies from call to call
    - thus frame sizes will vary as well
  - how can we pass return values *backwards* up the stack to the caller?
    - in addition to storing outgoing arguments just before its stack pointer, the caller will also allocate space for callee to place the return value in
      - ie. in the callee's frame pointer
    - thus this *shared* area between the two stack frames will hold arguments and return values
    - better to address values near the frame pointer, whose offsets are known, rather than the stack pointer

Procedure Linkages

- the procedure linkage convention divides responsibility between the caller and callee
  - caller pre-call → callee prologue → callee epilogue → caller post-call
- caller **pre-call**:
  1. allocate basic frame
  2. evaluate and store params
  3. store return address (may be a callee responsibility on different systems)
  4. jump to child
- callee **prologue**:
  1. save registers, state
  2. store `FP` ie. perform the *dynamic* link
  3. set new `FP`
  4. store *static* link
  5. extend basic frame for local data and initializations
  6. fall through to code
  - steps 1-5 may be done in varying orders
  - the **static link** is a feature used in languages that allow for nested functions:
    - the **dynamic link** points back up to the calling procedure's frame
      - following these links creates a **dynamic chain** of nested calls
    - while the static link will point up to the statically nested parent's frame:
      - specifically, the frame corresponding to the nearest parent call (since parent may have been called multiple times)
      - following these links creates a **static chain** of statically nested methods
    - ie. static links allow accessing of nonlocal values
- callee **epilogue**:
  1. store return value (in shared frame area)
  2. restore registers, state
  3. cut back to basic frame
  4. restore parent's `FP`
  5. jump to return address
- caller **post-call**:
  1. copy return value
  2. deallocate basic frame
  3. restore parameters (if copy out)
- possibilities for dividing up the work of saving and restoring registers:
  1. difficult: callee saves caller's registers
     - call must include a bitmap of caller's used registers

2. easy: caller saves and restores its own registers
3. easy: callee saves and restores its own registers
4. difficult: caller saves callee's registers
   - caller must use a bitmap held in callee's stack frame, or some method table
5. easy: callee saves and restores all registers
6. easy: caller saves and restores all registers

- in practice, approaches (2) and (3) are used

RISC-V Details

- RISC-V registers:
  - `x0` or `zero` holds hard-wired zero
  - `x1` or `ra` holds the return address (caller-saved)
  - `x2` or `sp` holds the stack pointer (callee-saved)
  - `x3` or `gp` holds the global pointer
  - `x4` or `tp` holds the thread pointer
  - `x5-7` or `t0-2` hold temporaries (caller-saved)
  - `x8` or `s0/fp` holds the frame pointer (callee-saved)
  - `x9` or `s1` is a callee-saved register
  - `x10-11` or `a0-1` hold function arguments and return values (caller-saved)
  - `x12-17` or `a2-7` hold function arguments (caller-saved)
  - `x18-27` or `s2-11` are callee-saved registers
  - `x28-31` or `t3-6` are caller-saved registers
- RISC-V linkage:
  - pre-call:
    1. pass arguments in registers `a0-7` or the stack
    2. save caller-saved registers
    3. execute a `jalr` that jumps to target address and saves the return address in `ra`
  - post-call:
    1. move return value from `a0`
    2. remove space for arguments from stack
  - prologue:
    1. allocate space for local variables and saved registers
    2. save registers eg. `ra` and callee-saved registers
  - epilogue:
    1. restore saved registers
    2. copy return value into `a0`
    3. clean up stack and return

Interpreters

- interpreters are *not* compilers:
  - an **interpreter** executes a program *"on-the-fly"*, without an explicit total translation to another language:
    - will still take in code *parsed* into a data structure, as a compiler does
    - ie. interpreter works with a single language, while a compiler works with at least two
    - typically incurs a performance overhead of ~10x
  - a Java program may be converted to Java bytecode using `javac` :
    - the bytecode is then often run using an interpreter
    - JavaVM has the choice of using an interpreter or a compiler
      - compiling the bytecode is an *investment* into the future, and if the time lost isn't regained, better to just run the interpreter
    - Java bytecode is more stable than Java source code, and was originally intended to be used for easy distribution
  - could we write an interpreter for source-level Java code?
    - eg. like interpreter in CS132 used for Sparrow, Sparrow-V, and RISC-V
    - all outputs / result states should be *comparable*
- given the simple grammar `e ::= c | e + e` :
  - our interpreter should take a single expression `e` as an input and give a numeric output
  - going from program syntax to a different type like a Java integer, rather than to another *language's* syntax

Implementing a simple interpreter:

```
class Interpreter implements Visitor {
  int visit(Nat n) { return n.f0; }
  int visit(Plus n) {
    return (n.f0).accept(this) + (n.f1).accept(this);
  }
}
```

Compared to a simple compiler for the same grammar:

```
class State { String p, int k }

class Compiler implements Visitor {
  String visit(Nat n, State s) {
    s.p += String.format("v%s = %s", s.k, n.f0);
    return String.format("v%s", s.k++);
  }
  String visit(Plus n, State s) {
    String v0 = n.f0.accept(this, s);
    String v1 = n.f1.accept(this, s);
    s.p += String.format("%s = %s + %s", s.k, v0, v1);
    return String.format("v%s", s.k++);
  }
}
```

- same grammer with boolean extension:
  - `e ::= true | !e` as well as previous rules
  - with more types, the interpreter has to work with many different types at once:
    - contrasted with compilers which only handle strings
  - thus, this is the fundamental reason why interpreters are so much slower:
    - types have to be *unpacked* and *repacked* together
    - ie. untagged and tagged
  - in addition, with loops and recursions, compilers compile to some code just *once*
    - while interpreters have to recursively re-visit and re-execute loops multiple times

Another interpreter for more types:

```
class Value {}
class Nat Extends Value {
  int i;
  Nat(int i) { this.i = i; }
  <A> A accept(ValueVisitor v) {
    return v.visit(this);
  }
}
class Boolean extends Value { ... }

class Interpreter implements Visitor {
  Value visit(Plus n) {
    // n.f0 -> check isNat -> unpack and getNat ->
    // n.f1 -> check isNat -> unpack and getNat ->
    // Pack both ints using addition (+) into a Value, and return it,
    // ie. unpack, and then repack into a value.
  }
  ...
}
```

Sparrow Interpreter

- interpreter must maintain *state* of the Sparrow program:
  1. program
  2. heap
  3. current block
  4. environment
  5. remaining executing block ie. program counter
  - must distinguish between local and global state:
    * program and heap are global
    * rest are local (can be represented in some kind of stack)
      - *stack* of *maps*!

Part of the Sparrow interpreter:

```
public class Interpreter extends Visitor() {
  Program prog;
  List<Value[]> heap = new ArrayList<Value[]>();      // Value tuples
  Stack<LocalState> state = new State<LocalState>(); // block, environment, and pc

  public void visit(Subtract n) {
    Value v1 = access(n.arg1); // unpacking to Value
    Value v2 = access(n.arg2);
    if ((v1 instanceof IntegerConstant) && (v2 instanceof IntegerConstant)) {
      int rhs = ((IntegerConstant) v1).i - ((IntegerConstant) v2).i;
      Value v = new IntegerConstant(rhs); // repack
      update(n.lhs, v);
      state.peek().pc = state.peek().pc + 1;
    }
  }

  public void visit(Store n) {
    Value vl = access(n.base);
    Value vr = access(n.rhs);
    if (vl instanceof HeapAddressWithOffset) {
      HeapAddressWithOffset hawo = (HeapAddressWithOffset) vl;
      heap.get(hawo.heapAddress)[(hawo.offset + n.offset)/4] = vr;
      state.peek().pc = state.peek().pc + 1;
    }
  }

  public void visit(Call n) {
    Value v = access(n.callee);
    if (v instanceof FunctionName) {
      FunctionName ce = (FunctionName) v;
      GetFunctionDecl gfd = new GetFunctionDecl(ce);
      prog.accept(gfd);
      FunctionDecl fd = gfd.result;

      List<Identifier> formalParameters = fd.formalParameters;
      List<Value> actualParameters = new ArrayList<Value>();

      for (Identifier s : n.args) {
        actualParameters.add(access(s));
      }

      // create and push new local environment
      Map<String, Value> m = new HashMap<String, Value>();
      for (int i = 0; i < formalParameters.size(); i++) {
        m.put(formalParameters.get(i).toString(), actualParameters.get(i));
      }
      state.push(new LocalState(fd.block, m, 0)); // func's block, new env, pc = 0

      stepUntilReturn();
      Value result = access(fd.block.return_id);
      state.pop();
      update(n.lhsm, result);
      state.peek().pc = state.peek().pc + 1;
    }
  }
}
```

Interpreter helper functions:

```
Value access(Identifier id) {
  return state.peek().env.get(id.toString());
}

void update(Identifier id, Value v) {
  state.peek().env.put(id.toString(), v);
}
```

```
void update(Register id, Value v) {
  registerFile.put(r.toString(), v);
}
```

More Compiler Optimizations

- **constant propagation**:
  - similar to copy propagations, use constants instead of additional variables
- **loop unrolling**:
  - loop is always run the same number of times, so we can get rid of the code used to maintain loop state and just duplicate body code
- **loop invariant code motion**:
  - move a part of the loop outside the loop since it performs the same thing each iteration
  - eg. some kind of loop initialization
- **common subexpression elimination**:
  - motivation of working with arrays
  - *caching* an array element where it would be accessed more expensively multiple times without *changing* its value
  - done similarly to copy propagations
- **polyhedral optimization**:
  - motivation of working with loops and nested loops
  - reorders or splits loops to allow for optimal parallelism
    - loops themselves can also be run in *any* order
  - tries many different loop orderings

Appendix

Example Lambda Translations to First-Order

Euclid's algorithm:

```java
public static int gcd(int x, int y) {
  return y == 0 ? x : gcd(y, (x % y));
} // already first-order form
```

Even-Odd deciders:

```java
static Function <Integer, Boolean>
even = n -> n == 0 ? true : Test.odd.apply(n-1);
static Function <Integer, Boolean>
odd  = n -> n == 0 ? false : Test.even.apply(n-1);
// already first-order form
```

Fibonacci to tail form:

```java
static Function<Integer, Integer>
fib = n -> n <= 2 ? 1 : Test.fib.apply(n-1) + Test.fib.apply(n-2);

static BiFunction<Integer, Function<Integer, Integer>, Integer>
fibCPS = (n, k) -> n <= 2
  ? k.apply(1)
  : Test.fibCPS.apply(n-1,
    v1 -> Test.fibCPS.apply(n-2,
      v2 -> k.apply(v1 + v2)));
```

Tail form Fibonacci to first-order:

```java
class FibRec1 implements Continuation {
  Integer n; Continuation k;
  public FibRec1(Integer n, Continuation k) {
    this.n = n; this.k = k;
  }
  public Integer apply(Integer v) {
    return Test.fibCPSadt.apply(n-2, new FibRec2(v, k));
  }
}

class FibRec2 implements Continuation {
  Integer v1; Continuation k;
  public FibRec1(Integer v1, Continuation k) {
    this.v1 = v1; this.k = k;
  }
  public Integer apply(Integer v) {
    return k.apply(v1 + v);
  }
}

static BiFunction<Integer, Continuation, Integer>
fibCPSadt = (n, k) -> n <= 2 ? k.apply(1) :
  Test.fibCPSadt.apply(n-1, new FibRec1(n, k));
```

Practice Questions

---

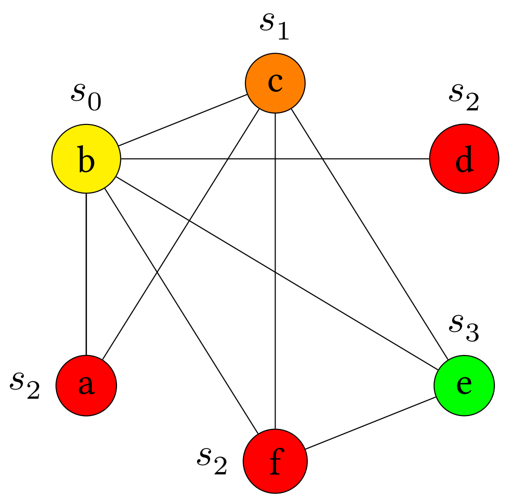
1. given the following grammar:
    - $A ::= \varepsilon | zCw$
    - $B ::= Ayx$
    - $C ::= ywz | \varepsilon | BAx$
    - then:
        - $\mathit{FIRST}(A) = \{z\}$
        - $\mathit{FIRST}(B) = \{y, z\}$
        - $\mathit{FIRST}(C) = \{y, z\}$
        - $\mathit{NULLABLE}(A) = \mathit{true}$
        - $\mathit{NULLABLE}(B) = \mathit{false}$
        - $\mathit{NULLABLE}(C) = \mathit{true}$
    - we can make the following observations for each nonterminal on the RHS:
        - $w \in \mathit{FOLLOW}(C)$
        - $y \in \mathit{FOLLOW}(A)$
        - $\mathit{FIRST}(A) \subseteq \mathit{FOLLOW}(B)$
        - $x \in \mathit{FOLLOW}(B)$
        - $x \in \mathit{FOLLOW}(A)$
    - thus:
        - $\mathit{FOLLOW}(A) = \{x, y\}$
        - $\mathit{FOLLOW}(B) = \{x, z\}$
        - $\mathit{FOLLOW}(C) = \{w\}$
    - therefore the grammar is *not* LL(1), since for C:
        - $\mathit{FIRST}(ywz) \cap \mathit{FIRST}(BAx) \neq \emptyset$
2. What is the fewest number of registers to entirely allocate the following code?

```
a = 1         // 1
b = 10        // 2
c = a + a     // 3
d = a + c     // 4
e = c + d     // 5
f = b + 8     // 6
c = f + e     // 7
f = e + c     // 8
b = c + 5     // 9
return b + f // 10
```

Live ranges, from inspection:

```
a : [1, 4]
b : [2, 6], [9, 10]
c : [2, 6], [7, 9]
d : [4, 5]
e : [5, 8]
f : [6, 7], [8, 10]
```

Corresponding inteference graph:



s_1
c
s_0
b
s_2
d
s_3
e
s_2
a
s_2
f