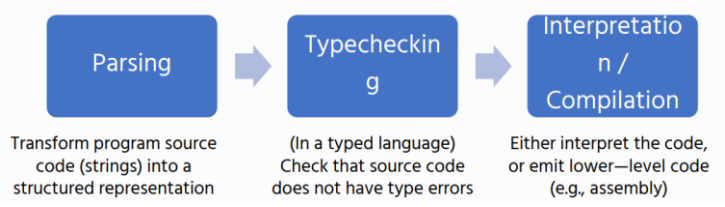
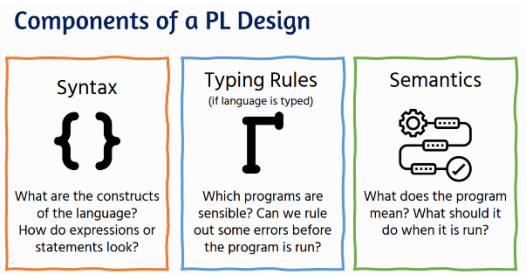
Week 1: Basics and Syntax

Code in a PL cannot be run directly by a processor

• Instead, it must be interpreted or compiled into a different (often more basic) language





Wadlers Law: In any language design, the total time spent discussing a feature in this list is proportional to two raised to the power of its position.

0. Semantics 1. Syntax 2. Lexical syntax 3. Lexical syntax of comments (That is, twice as much time is spent discussing syntax than semantics, twice as much time is spent discussing lexical syntax than syntax, and twice as much time is spent discussing syntax of comments than lexical syntax.)

Programming Paradigms

A programming paradigm is a style of programming that defines principles, techniques, and methods for problem-solving.

Choosing the right paradigm is crucial (e.g., SQL is not suited for systems programming).

Common paradigms:

- Imperative
- Functional
- Object-Oriented

paradigms include logic programming and concurrent programming.

languages (e.g., Scala) support multiple paradigms

Imperative Programming Languages

• Imperative language: Program Counter + Call Stack + State

• We record our current position in the program

• Statements can alter that position (and perform side effects)

• Variable assignments alter some store

Examples: JS, C, Python and Java (also object oriented)

Expression Vs Statement

Statement: An instruction/Step doesn't return anything
Expression: A term in the language that eventually reduces to a value (e.g., a string, integer, ...)
Can be contained within a statement

Functional Programming Languages

• Main difference: everything is an expression
• Often have first-class functions: can create, apply, and pass functions just like any other expression
• Evaluation is reduction of a complex expression to a value
You can program in IL in a functional style by using const rather than var or let in JS
Haskell/Idris are pure: separation of side effecting code

```
def addOne(x):
  return x + 1

xs = []
for x in range(1,3):
  xs.append(addOne(x))
return xs
```

```
def addOne(x):
  return x + 1

xs = []
for x in range(1,3):
  xs.append(addOne(x))
return xs
```

```
map addOne [1,2,3]
→ 2 :: (map addOne [2, 3])
→  2 :: 3 :: (map addOne [3])
→ 2 :: 3 :: 4 :: (map addOne [])
= [2, 3, 4]
```

Regular Expressions - Expressive Power
Regular expressions are useful for **basic string matching** but can only match **regular languages**.
They **cannot handle nested structures**, making them unsuitable for parsing languages like:
HTML/XML
Programming languages with nesting (e.g., Python, Java, etc.)
However, they are **useful for tokenisation**, converting strings into **streams of tokens**.

Regular Expressions: Definition

a	Matches character a	R^*	Matches 0 or more copies of R
R_1R_2	Matches concatenation of patterns R1 and R2	(R)	Matches R (brackets used for grouping)
$R_1 \mid R_2$	Matches either R1 or R2	ε	Empty string

Regular Expressions
• A regular expression is a pattern that can match strings. Regular expressions have many uses:

Regular Expressions: Derived Forms

$R? \triangleq R \mid \varepsilon$
Optional occurrence of R

$R+ \triangleq RR^*$
One or more occurrences of R

$[abc] \triangleq a \mid b \mid c$
Match any of a, b, or c

Concrete vs. Abstract Syntax

The grammar of our language contains **more syntax than is needed for computation**: in particular, brackets are only needed to show how to parse an expression.

```
(1 + 2 * 3) + (3 * 4)
```

Includes irrelevant syntactic information, has implicit information about operator precedence

```
Add(
  Add(Num(1),
    Mul(Num(2), Num(3))),
  Mul(Num(3), Num(4)))
```

Simplified representation of a parsed expression: unambiguous precedence, no irrelevant syntax

• A grammar is **ambiguous** if we can derive two or more different parse trees for a string
• If we were to allow arbitrary expressions on either side of an operator, we can show how one of our tedious Facebook puzzles is ambiguous...
Sytnax of PLs should be unambiguous.
For example the dangling if problem
if b then if c then print “hello” else print “hi”

Backus-Naur Form

- Our grammar for LArith is an example of a grammar in Backus-Naur Form (BNF)
- A BNF grammar consists of a series of productions takes the form $S ::= \alpha \mid \beta \mid \gamma$ where α, β, γ each stand for a sequence of terminal and nonterminal symbols
- Extended BNF (EBNF) allows us to use regular expression notation in productions (e.g. in int)

Example: L_{Arith}

We can now give a formal grammar for the **concrete syntax** of our expression language

This is a **production**, showing that an expr can either be a prim or a term of the form expr op prim

Strings occurring without quotes are **nonterminals**, standing for an occurrence of a member of the given symbol

Strings in quotes are **terminal symbols**: character or string literals

```
expr ::= prim
       | expr op prim
op   ::= '+' | '-' | '*' | '/'
prim ::= int | '(' expr ')'
int  ::= digit | digit int
digit ::= '1' | '2' | '3' | '4' | '5'
        | '6' | '7' | '8' | '9' | '0'
```

Grammars

Regular expressions help, but don't cut it for describing the syntax of a programming language.

A Grammar is a set of formal rules specifying how to construct a sentence in a language. It consists of:

A set of **terminal symbols**: symbols that occur in a sentence

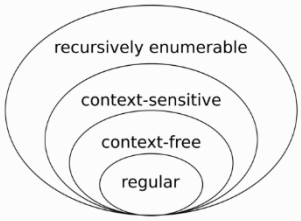
A distinguished **sentence symbol** that stands for a complete sentence. A set of **nonterminal symbols**, which each 'stand for' part of a phrase. A set of **production rules** that show how phrases can be made up from terminals and sub-phrases

```
expr ::= prim
       | expr op prim
op   ::= '+' | '-' | '*' | '/'
prim ::= int | '(' expr ')'
int  ::= digit+
digit ::= '1' | '2' | '3'
        |  '4' | '5' | '6'
        | '7' | '8' | '9' | '0'
```

How do we parse text into an AST?

- Generally, we do the following:
- Tokenise text into chunks using regular expressions (lexing)
- Match token streams and convert these into AST nodes (parsing)

Chomsky's Hierarchy of Language



- Regular expressions are **regular languages**
  - They can be recognised by a **finite state automaton**
- Context-free languages (the ones we will focus on) can be recognised by a **push-down automaton**
- Context-sensitive languages and recursively enumerable languages require more interesting recognisers (e.g. a Turing machine)

Abstract Syntax

- Abstract syntax allows us to **abstract away irrelevant syntactic noise**: we can concentrate on the important parts of the language
  - (i.e., we assume parsing has been done already)
- Apart from the lecture on parsing later in the course, **all examples from now on will use abstract rather than concrete syntax**

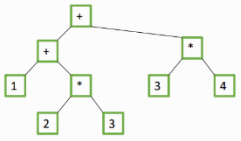
```
expr ::= prim
       | expr op prim
op   ::= '+' | '-' | '*' | '/'
prim ::= int | '(' expr ')'
int  ::= digit+
digit ::= '1' | '2' | '3'
        |  '4' | '5' | '6'
        | '7' | '8' | '9' | '0'
```

```
Integers n
Operators ⊙  ::= + | - | * | /
Terms L, M, N ::= n
                | L ⊙ M
```

Abstract Syntax Trees (ASTs)

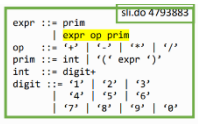
- We can also write abstract syntax as a tree (and indeed, this is how every parsed program is represented).
- Consider our example from earlier, the AST for:
  `(1 + 2 * 3) + (3 * 4)`
- All further operations (desugaring, typechecking, evaluation, compilation) are done on ASTs

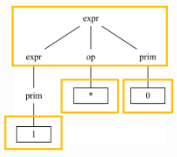
```
Add(
  Add(Num(1),
    Mul(Num(2), Num(3))),
  Mul(Num(3), Num(4)))
```



Parse Trees

We can represent how a string corresponds to a grammar G using a **parse tree** (or **syntax tree**)





Each **terminal node** is labelled by a terminal symbol (here, digits and operators)

Each **nonterminal node** is labelled by a nonterminal symbol of G and can have children nodes X, Y, Z **only if G has a production rule N ::= ... | X Y Z | ...**

Week 2: Semantics ANTLR SVM

Compilers

• But for general-purpose hardware, a compiler translates code into (often a series of) lower-level languages, such that eventually they can be executed on hardware

• Often, even compiled code needs to be supported by a runtime system (that provides things like garbage collection)

Interpreters

• An interpreter is a program that accepts a program written in a given programming language, and executes it directly (without generating a separate executable).

• For example, Perl is fully interpreted: the interpreter is a separate program written in C

• Interpreters work by:

• Fetching, analysing, and executing instructions (for imperative languages)

• Evaluating subexpressions (for expression-based / functional languages)

• Generally, interpreters are easier to write but slower than compiled code

• The first lab will involve writing an interpreter for LIf

Virtual Machines

• A physical machine runs machine code (e.g. x86 assembly) directly

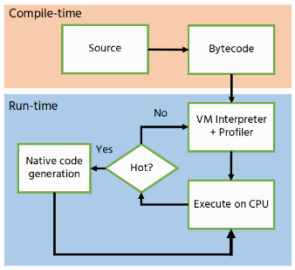
• In contrast, a virtual machine evaluates instructions (usually encoded in some sort of bytecode) by an interpreter

• Advantages of VMs:

• Platform independence: Can run compiled code on multiple platforms

• Common backend: multiple (quite different) languages can target the same backend – for example the .NET suite of languages target the .NET CLR, and all of { Java, Scala, Kotlin, Clojure } target JVM bytecode

Just-in-time (JIT) Compilers



- A JIT compiler is a middle-ground between compilers and interpreters, where code is compiled to native code **at run-time**
- JIT compilers operate **selectively**: they profile code and compile "hot" (i.e., frequently called) code
- An example is Java's HotSpot JIT compiler for Java

Approaches to Programming Language Semantics

sli.do 1238123

Operational	Denotational	Axiomatic
$\rightarrow^*$	$[\![-]\!]$	$\{P\}\ M\ \{Q\}$
Details **reduction** of an expression either to another expression (small-step) or a value (big-step)	Maps expressions to **semantic (mathematical) objects**.	Describes evaluation in terms of pre- and post-conditions on the program state

L_{Arith} Abstract Syntax

```
Integers n
Operators ⊙   ::= + | - | * | /
Values V, W   ::= n
Terms L, M, N ::= n
                | L ⊙ M
```

Textual Descriptions

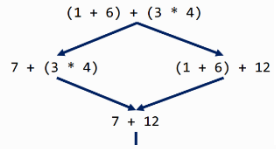
• Let's first write out textual descriptions for how to evaluate LArith expressions.

• We have two types of expression: integer literals n, and arithmetic operations L ⊙ n.

n	$L \odot M$
An integer n is a **value**: it is already in its simplest form	To evaluate an arithmetic operation, evaluate L to a value, evaluate M to a

Aside: Determinism and the Church-Rosser Theorem

- We can reduce summands in **either order** and arrive at the same result: we say that evaluation is **deterministic** or satisfies the **Church-Rosser property**
- This is not the case for all PLs – especially those that have **side-effects** (e.g. sending packets, printing to the console)
- The **Theory of Computation** course (running again next year, hopefully!) treats this in much more depth



$$M \Downarrow V \qquad \frac{P_1 \quad P_2 \quad \ldots \quad P_n}{Q}$$

Pitfalls of Textual Descriptions

• Textual descriptions can be imprecise: the language designer might mean something different to what is understood by the language implementer

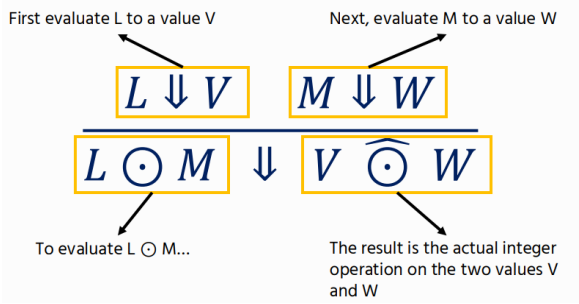
• We can't prove properties about textual descriptions: they are not mathematically defined

• Textual descriptions don't scale: more complex language features require lots of (confusing) text to describe

• Textual descriptions leave room for edge cases due to ambiguity

Expressions and Values

- A **value** is data, and is the final result of a computation. It cannot evaluate further.
- **Expressions** in LArith include binary operations, which may need to evaluate further. In our language, every expression should eventually evaluate to a value.



(Big-step) Operational Semantics

Big-step operational semantics involves showing how an expression M evaluates to a value V. You can think of the judgement on the left being like a function signature Expression -> Value.

We then need inference rules to show how expressions evaluate: remember that an inference rule says that if

$$\dfrac{\dfrac{\dfrac{}{1 \Downarrow 1} \quad \dfrac{\dfrac{}{2 \Downarrow 2} \quad \dfrac{}{3 \Downarrow 3}}{2 * 3 \Downarrow 6}}{1 + (2 * 3) \Downarrow 7} \quad \dfrac{\dfrac{}{3 \Downarrow 3} \quad \dfrac{}{4 \Downarrow 4}}{3 * 4 \Downarrow 12}}{(1 + (2 * 3)) + (3 * 4) \Downarrow 19}$$

L_{If} reduction rules

The rules from L_{Arith} are very similar. We generalise the value rule to arbitrary constants rather than just numbers. We don't need to change the binary operator rule.

$$\frac{}{c \Downarrow c} \qquad \frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \Downarrow V \mathbin{\widehat{\odot}} W}$$

We need **two** rules for conditional statements: one for if the predicate returns true (which evaluates the first branch), and another for if the predicate returns false (which evaluates the second branch).

$$\frac{L \Downarrow \texttt{true} \quad M \Downarrow V}{\texttt{if}\ L\ \texttt{then}\ M\ \texttt{else}\ N \Downarrow V} \qquad \frac{L \Downarrow \texttt{false} \quad N \Downarrow V}{\texttt{if}\ L\ \texttt{then}\ M\ \texttt{else}\ N \Downarrow V}$$

Finally we need **two** rules for equality: one if both expressions evaluate to two identical values, and one if not.

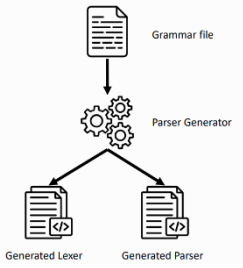
$$\frac{M \Downarrow V \quad N \Downarrow V}{M == N \Downarrow \texttt{true}} \qquad \frac{M \Downarrow V \quad N \Downarrow W \quad V \neq W}{M == N \Downarrow \texttt{false}}$$

L_{If} Abstract Syntax

L_{If} Abstract Syntax

```
Integers n
Booleans b     ::= true | false
Operators ⊙    ::= + | - | * | /
                 | < | > | && | ||
                 | ==
Values V, W    ::= n | b
Constants c    ::= n | b
Terms L, M, N  ::= c
                 | L ⊙ M
                 | if L then M else N
```

Parser Generators



- A **parser generator** is a program that takes a grammar and produces a lexer and parser
- This means that a language implementer can concentrate on the **design** of the grammar, rather than writing hand-crafted logic each time
- Many different parser generators exist for different languages (e.g. Yacc / Bison for C, Happy / Alex for Haskell, Menhir for OCaml)

$$\dfrac{\dfrac{\dfrac{}{5 \Downarrow 5} \quad \dfrac{}{6 \Downarrow 6}}{5 > 6 \Downarrow \texttt{false}} \quad \dfrac{\dfrac{\dfrac{}{4 \Downarrow 4} \quad \dfrac{}{5 \Downarrow 5}}{4 * 5 \Downarrow 20} \quad \dfrac{}{6 \Downarrow 6}}{(4 * 5) + 6 \Downarrow 26}}{\texttt{if}\ 5 > 6\ \texttt{then}\ 3\ \texttt{else}\ (4 * 5) + 6 \Downarrow 26}$$

L_{Arith} ANTLR Grammar

```
expr: expr DIV expr # Div
    | expr MUL expr # Mul
    | expr ADD expr # Add
    | expr SUB expr # Sub
    | INT # Int
    | '(' expr ')' # Parens
    ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
MUL: '*' ;
DIV: '/' ;
SUB: '-' ;
ADD: '+' ;
```

- **ANTLR** is an open-source parser generator for Java, originally written by Terence Parr, and widely used in industry
- The idea is that you can write a grammar using EBNF syntax, and it will generate a lexer, parser, and visitor code to traverse the parse tree
- We will use ANTLR in this course mainly for its parser generation component, as opposed to using visitors directly to implement interpretation / typechecking

Background: The Visitor Design Pattern

sli.do 1238123

Visitor Definition

```
public interface TreeVisitor {
  public void visitNode(Node n);
  public void visitLeaf(Leaf l);
}

public class BaseTreeVisitor
    implements TreeVisitor {
  public void visitNode(Node n) {
    n.getLeft().accept(this);
    n.getRight().accept(this);
  }
  public void visitLeaf(Leaf l) { }
}
```

Visitor Implementations

```
public class PrintLeafVisitor extends BaseTreeVisitor {
  @Override
  public void visitLeaf(Leaf l) {
    System.out.println(l.getValue());
  }
}

public class IncrementLeafVisitor extends BaseTreeVisitor {
  @Override
  public void visitLeaf(Leaf l) {
    l.setValue(l.getValue() + 1);
  }
}
```

- The **visitor pattern** separates the **operation** on each element of a structure from **the act of traversing it**.
- The code above shows two visitors: one that increments the leaves of a binary tree, the other that prints their values.
- We **do not** need to manually inspect the left and right subtrees, only writing the logic for the leaves. The traversal logic is contained in `BaseTreeVisitor`.
- Note that you **don't need to understand how to implement a visitor yourself**, just that it allows us to write the logic for each type of element independently.

- ANTLR generates a base Visitor class from the supplied grammar.
- We can either use this to create instances of our AST nodes (and then write our own code to traverse them), or work directly with the generated visitors (without needing to write our own AST)
- The labs and assignment (and code in my lectures) will use a custom AST.
- We will only use ANTLR visitors to create instances of our hand-written ASTs.
- This is because hand-written ASTs have more specific structure and types, are easier to conceptualise, and writing explicit recursive functions is closer to the formal definitions.

General pointers for using ANTLR Visitors

- Parameterise your visitor with the type you wish to return.
- Override the visit method for each generated expression, and implement the desired behaviour.
- The generated AST classes have accessor methods for subexpressions, based on the production names in the grammar. If there are multiple, you can supply the index.

Generating an AST using an ANTLR Visitor

```
public class LArithASTGenerator
  extends LArithBaseVisitor<Expr> {

  @Override
  public Expr visitAdd(LArithParser.AddContext ctx) {
    Expr e1 = visit(ctx.expr(0));
    Expr e2 = visit(ctx.expr(1));
    return new EBinOp(e1, EBinOp.Op.ADD, e2);
  }

  @Override
  public Expr visitInt(LArithParser.IntContext ctx) {
    int val = Integer.valueOf(ctx.INT().getText());
    return new EInt(val);
  }
  …
}
```

We can **parameterise** the visitor with the type we want each method to return, in this case an Expr

Each rule in the grammar will result in a method in the visitor that we can override (in this case, Add)

We visit both subexpressions (accessed using the `ctx.expr()` method) to get two Exprs, and we can use these to construct a new `EBinOp` expression

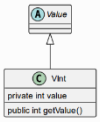
Terminals are always represented as strings, so we need to use the `getText()` method and convert to an integer

Implementing an Interpreter (Overview)

- Remember the form of our operational semantics judgement, which says that an expression M will evaluate to a value V.

$$M \Downarrow V$$

- It helps to write a class hierarchy representing Values (in L_{Arith}, this will only contain `VInt`). Values are **only introduced at runtime and are never parsed in**.



- Our interpreter is therefore a function from an Expr to a Value.

```
public Value interpExpr(Expr e) { … }
```

Implementing an Interpreter (Values)

- To write an interpreter, we need different cases based on each rule in the operational semantics
- The value rule just states that a number is already a value, so we just need to create a `VInt` with the same value.

$$\frac{}{n \Downarrow n}$$

Note: e is of type Expr. We need to do **instance tests** to check which kind of expression e is, and then **cast** it to an EInt

```
if (e instanceof EInt) {
  EInt exprInt = (EInt) e;
  return new VInt(exprInt.getValue());
}
```

Implementing an Interpreter (Binary Operators)

$$\frac{L \Downarrow V \quad M \Downarrow W}{L \odot M \Downarrow V \mathbin{\widehat{\odot}} W}$$

Key idea: recursively evaluate subexpressions, switch on the operator, and apply the concrete operation

Note: `unwrapInt` is a function that extracts an integer from a Value

```
…
} else if (e instanceof EBinOp) {
  EBinOp exprBinOp = (EBinOp) e;
  Value v1 = interpExpr(exprBinOp.getE1());
  Value v2 = interpExpr(exprBinOp.getE2());
  int i1 = unwrapInt(v1);
  int i2 = unwrapInt(v2);
  switch (exprBinOp.getOp()) {
    case ADD: return new VInt(i1 + i2);
    case SUB: return new VInt(i1 - i2);
    case MUL: return new VInt(i1 * i2);
    case DIV: return new VInt(i1 / i2);
    default:
      throw new RuntimeException("Invalid op");
  }
}
```

SVM is a simple(ish) VM used for teaching, and suitable for implementing simple imperative languages. It'll be used in Michele's part of the course.

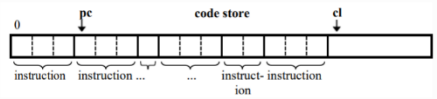
- Much like both the JVM and WebAssembly, SVM is stack-based: instructions use an operand stack rather than having explicit parameters.

Machine State (Data Store)

- The second part of the SVM state is a data store containing the program's data (simplified diagram on the right).
- Roughly speaking, there is a stack pointer (sp) that denotes the top of the stack, some data that occurs on the stack, and then global data.
- The full definition includes additional information to remember return addresses, etc.
- Finally, the machine contains a status register that indicates whether the program is running, halted, or failed.

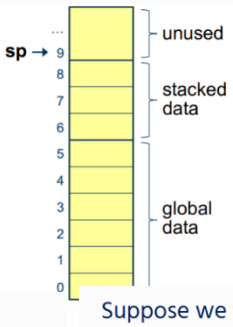
Machine State (Code Store)

The first part of the SVM state is a **code store** containing the machine's bytecode.



There are two registers associated with the code store:

- A **program counter** (pc) that tracks the current instruction
- A **code limit** (cl) that marks the end of the code store



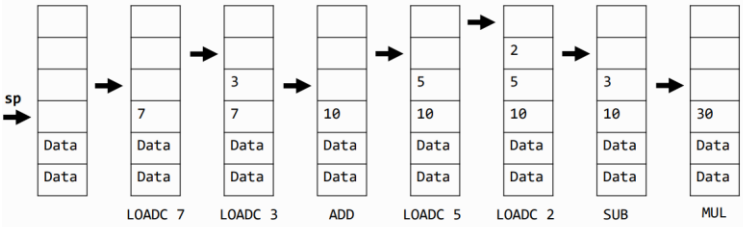
Simplified SVM Instruction Set (1)

Opcode	Mnemonic	Description
6	ADD	pop w_2; pop w_1; push (w_1 + w_2)
7	SUB	pop w_2; pop w_1; push (w_1 − w_2)
8	MUL	pop w_2; pop w_1; push (w_1 × w_2)
9	DIV	pop w_2; pop w_1; push (w_1 / w_2)
10	CMPEQ	pop w_2; pop w_1; push (if w_1 = w_2 then 1 else 0)
11	CMPLT	pop w_2; pop w_1; push (if w_1 < w_2 then 1 else 0)
14	INV	pop w; push (if w = 0 then 1 else 0)

Simplified SVM Instruction Set (2)

Opcode	Mnemonic	Description
0	LOADG d	w <- word at address d; push w
1	STOREG d	pop w; word at address d <- w
4	LOADC v	push v
…		
16	HALT	status <- halted
17	JUMP c	pc <- c
18	JUMPF c	pop w; if w = 0 then pc <- c
19	JUMPT c	pop w; if w =/= 0 then pc <- c

Suppose we want to evaluate `(7 + 3) * (5 - 2)` in SVM



SVM Implementation

```
public void interpret () {
  // Initialise state
  status = RUNNING;
  sp = 0; fp = 0; pc = 0;
  do {
    // Fetch the next instruction:
    byte opcode = code[pc++];
    // Inspect opcode and execute:
    switch (opcode) {
      case ADD:
        int w2 = data[--sp];
        int w1 = data[--sp];
        data[sp++] = w1 + w2;
        break;
      …
    }
  } while (status == RUNNING);
}
```

- There's a full specification of SVM on Moodle, along with an interpreter – take a look if you're interested!
- Virtual machines don't have to be complicated. It's essentially one big loop that continually fetches and executes each instruction until the machine is halted.

Week 3: Variables and Binding, Functions And Recursion

Recap: Predicate Logic Quantifiers

- In predicate logic we have two **quantifiers**: these allow us to allow a variable to 'stand for' an element of a set
  - Universal quantifiers require a property to hold for all elements of a set
  - Existential quantifiers require a property to hold for at least one element of a set

Quantifier | Formula body

$\exists c \in Cities.\ isFriday \rightarrow Rainy(c)$ — "There is some city where it rains every Friday"

Binding occurrence of c | **Bound** occurrence of c

- A variable is **free** if it is not in the scope of a quantifier. A quantifier $\forall x\ .P$ binds all free occurrences of x in its body P.
- For example, c is **free** in $isFriday \rightarrow Rainy(c)$, but **bound** in $\exists c \in Cities.\ isFriday \rightarrow Rainy(c)$.

Let-bindings

- We will begin by looking at a simple type of binding: a **let-binding**
- The key idea is that we can give a name to a subexpression in a given continuation, so we don't need to repeat it
  - You can think of a let-binding as an **abbreviation**
- For example, the following expression would evaluate to 225

```
let x = (5 + 10) in x * x
```

- We can also **nest** let-bindings: for example the expression on the right would evaluate to 15

```
let x = 5 in
let y = 10 in
x + y
```

L_{Let} Abstract Syntax

```
Integers n
Variables x, y, z
Operators ⊙   ::= + | - | *  | /
Values V, W   ::= n
Terms L, M, N ::= x | n
                | L ⊙ M
                | let x = M in N
```

Binding and Bound Occurrences in L_{Let}

- Much like with predicate logic formulae, **let** expressions act as a binder for a variable.

```
let x = (5 + 10) in x * x
```

Binding occurrence of x | **Bound** occurrences of x

- Formally, `let x = M in N` binds all **free occurrences** of x in the function body N.
- In the above example, the body of the let is `x * x`. Here, both occurrences of x are free, so become bound.

Free Variables in L_{Let}, formally

- We can write out a recursive function fv(M) to get the free variables of an expression

```
fv(x) = {x}                          x on its own is a free variable
fv(n) = ∅                            Integer literals don't contain variables
fv(let x = M in N) = fv(M) ∪ (fv(N) \ {x})
fv(M ⊙ N) = fv(M) ∪ fv(N)
```

The free variables of a binary operator are the union of the free variables of its operands

Any free occurrence of x would be **bound** by the `let`, so we need to remove it from the set of N's free variables

Name Shadowing

- Remember, a let-expression only binds **free** occurrences of the variable in the body. Consider the following:

```
let x = 5 in
let x = 10 in
x + x
```

- The body of the first let-binder is `let x = 10 in x + x`
  - There are **no free occurrences of** x – both occurrences of x are instead bound by the second let-binder
- Therefore, the first let-binder is redundant: we say that it has been **shadowed** by a more recent binder

Scope

- The **scope** of a variable is the collection of program locations in which occurrences of the variable refer to the same thing
  - (With some caveats – for example when we consider mutation later on)

Scope of x { `let x = 5 in let y = 10 in x + y` } Scope of y

Let-bindings are **not** imperative variable assignment!

- Remember that we are working in a language **without** mutation at the moment – we can shadow bindings, but data **cannot change**
- Consider the following:

```
let x = 5 in
  let y =
    let x = 100 in x - x
  in
  x + y
```

This occurrence of x refers to 5

Both occurrences of x here refer to 100

Substitution (Intuition)

- To define the operational semantics for L_{Let} we need to first define a **substitution** operation

$M\ \{V\ /\ x\}$ — "Replace all free occurrences of x in term M with value V"

Note: substitution is also written different ways in the literature, e.g. `M[V / x]`, `M[x ↦ V]` and `M[x := V]`
If you're interested, this talk by Guy Steele gives an interesting discussion of the variety of meta-notation used in PL papers

- Examples:
  - `(x + 10) { 100 / x } = 100 + 10`
  - `(y + 10) { 100 / x } = y + 10`
  - `(let x = 5 in x + y) { 10 / y } = let x = 5 in x + 10`
  - `(let x = 5 in x + y) { 10 / x } = let x = 5 in x + y`

Substitution (Formal Definition)

$$x\{V/y\} = \begin{cases} V & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

If x is the same variable as the one we're substituting for, replace it with the value V. Otherwise, leave it as it was.

Substitution doesn't affect integer literals.

$$n\{V/y\} = n$$

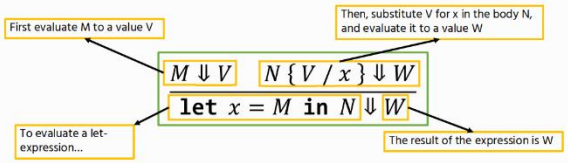
Substitution only affects **free** occurrences of x. If x = z, then the previous binding for y is **shadowed** and the body will contain no free occurrences of x so we stop

$$(\textbf{let}\ x = M\ \textbf{in}\ N)\{V/y\} = \begin{cases} \textbf{let}\ x = M\{V/y\}\ \textbf{in}\ N & \text{if } x = y \\ \textbf{let}\ x = M\{V/y\}\ \textbf{in}\ N\{V/y\} & \text{otherwise} \end{cases}$$

$$(M \odot N)\{V/x\} = (M\{V/x\}) \odot (N\{V/x\})$$

Propagate substitutions into the subexpressions of binary operators

L_{Let} Operational Semantics

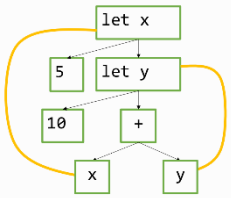


- There is no rule for variables : trying to evaluate a free variable is an error.
- Note that we're choosing to evaluate M **before** substitution. We call this approach **eager**, or **call-by-value**.
  - There are other potential choices, for example substituting the expression **before** evaluating it (called **call-by-name**). Alas we won't have time to go into that in more detail.

$$\dfrac{\dfrac{10 \Downarrow 10 \quad 15 \Downarrow 15}{10 + 15 \Downarrow 25} \qquad \dfrac{100 \Downarrow 100 \qquad \dfrac{25 \Downarrow 25 \quad 100 \Downarrow 100}{25 + 100 \Downarrow 125}}{\texttt{let } y = 100 \texttt{ in } 25 + y \Downarrow ?}}{\texttt{let } x = 10 + 15 \texttt{ in } (\texttt{let } y = 100 \texttt{ in } x + y) \Downarrow ?}$$

$$\dfrac{\dfrac{10 \Downarrow 10 \quad 15 \Downarrow 15}{10 + 15 \Downarrow 25} \qquad \dfrac{100 \Downarrow 100 \qquad \dfrac{25 \Downarrow 25 \quad 100 \Downarrow 100}{25 + 100 \Downarrow 125}}{\texttt{let } y = 100 \texttt{ in } 25 + y \Downarrow 125}}{\texttt{let } x = 10 + 15 \texttt{ in } (\texttt{let } y = 100 \texttt{ in } x + y) \Downarrow 125}$$

Binding Diagrams



- Sometimes it helps to visualise the binding structure for an expression
- On the left, we have a diagrammatic representation of the abstract syntax tree for the expression

```
let x = 5 in
let y = 10 in
x + y
```

- We can visualise the binding structure by drawing connecting lines between binding and bound occurrences of each variable

α-equivalence

- It is useful to treat two expressions as the same, as long as their **binding structure** is the same. For example, we can equate the following expressions:

```
let x = 5 in
let y = 10 in
x + y
```
$\approx_\alpha$
```
let a = 5 in
let b = 10 in
a + b
```

- This is because we **consistently** rename **bound variables** x to a, and y to b.

Non-examples of α-equivalence

`x`	$\not\approx_\alpha$	`y`	Only **bound** variables can be renamed – in this case, x and y are distinct **free** variables
`let x = 5 in let y = 10 in x + y`	$\not\approx_\alpha$	`let a = 5 in let a = 10 in a + a`	This example **changes the binding structure** since both occurrences of a refer to the second binder
`let x = 5 in let y = 10 in x + y`	$\not\approx_\alpha$	`let x = 5 in x + y`	The right expression is syntactically different – the second let-binder is removed
`let x = 5 in let y = 10 in x + y`	$\not\approx_\alpha$	`let x = 5 in let y = 10 in y + x`	Even though + is commutative, the structure has changed

Determining α-equivalence by renaming

- We can determine whether two terms are α-equivalent by consistently renaming all bound variables, and then checking syntactic equality

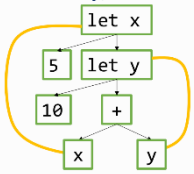
```
let x = 5 in
let y = 10 in
x + y
```
Rename x to c, and y to d
```
let c = 5 in
let d = 10 in
c + d
```
= Expressions are equal

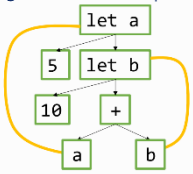
```
let a = 5 in
let b = 10 in
a + b
```
Rename a to c, and b to d
```
let c = 5 in
let d = 10 in
c + d
```

Determining α-equivalence by comparing binding diagrams

sli.do 1

- Alternatively, we can draw binding diagrams for both expressions





- The only place the diagrams are allowed to be different is the name of the let-binder, and the names of bound variables
  - In this case, the connecting lines must be identical

Function bindings

```
let fun square x =
  x * x
in
square 5 + square 10
```

```
let fun square x = x * x in
let fun double x = x * 2 in
let fun doubleAndSquare x =
  square (double x)
in
doubleAndSquare 100
```

- As a first attempt, let us extend our let binding from L_{Let}
- The `let fun f x = M in N` construct defines a function `f` with parameter `x` and body `M`, allowing it to be used in body `N`
- We also need a construct for **application**, in this case of the form `f M`

Anonymous Functions (Lambdas)

There's a simpler, more primitive way of representing functions: **anonymous** (or **lambda**) functions

$$\underbrace{\lambda n.}_{\text{Parameter}} \quad \underbrace{n + 5}_{\text{Body}}$$

We can then **apply** a function, meaning we replace the free occurrences of the parameter with an argument before evaluating

$$(\lambda n.\ n + 5)\ 10 \Downarrow 15$$

Multi-Argument Functions

Lambda expressions only have a **single** parameter, but we can define functions with multiple parameters by **nesting** multiple functions

$$((\lambda n1.\ (\lambda n2.\ n1 + n2))\ 5)\ 10$$
$$\Downarrow$$
$$(\lambda n2.\ 5 + n2)\ 10$$
$$\Downarrow$$
$$5 + 10$$
$$\Downarrow$$
$$15$$

This is known as **currying**, after the logician Haskell Curry

Let / Let-fun are Syntactic Sugar!

```
let fun f x = M in N
        ⇔
let f = (λx. M) in N
```

```
let x = M in N
        ⇔
   (λx. N) M
```

- In fact, as soon as we have anonymous functions and application, we can **encode** both `let fun` and `let`
  - They are much easier to write, though, so it's worth keeping them in the source language as **syntactic sugar**
  - We don't, however, need to write explicit reduction / substitution rules for them
- We would first need to desugar `let fun` into a `let` binding, and then desugar the `let` binding into lambda expressions and function applications

Desugaring Example

```
let fun add5 x = x + 5 in add5 10
```
(desugars to)
```
let add5 = λx. x + 5 in add5 10
```
(desugars to)
```
(λadd5. add5 10) (λx. x + 5)
```

L_{Lam} Abstract Syntax

L_{Lam} Abstract Syntax

```
Integers n
Variables x, y, z
Operators ⊙   ::= + | - | * | /
Values V, W   ::= n | λx . M
Terms L, M, N ::= x | n
                | λx . M
                | M N
                | L ⊙ M
```

- We extend the language again with variables, and also anonymous functions `λx. M` and function application `M N`
- We need not consider **`let`** or **`let fun`** expressions in the abstract syntax; we can assume that they have already been encoded
- We will write `\x. M` rather than `λx. M` in code

Semantics: Informal example

`(\x. x * x) (10 + 15)` — Begin by evaluating function (\x . x * x): it's already a value

`(\x. x * x) 25` — Evaluate 10 + 15 down to the value 25

`25 * 25` — Replace every occurrence of 'x' in the function body with the argument, 25

`625` — Evaluate the (now closed) function body to get the final result

L_{Lam} Operational Semantics

First evaluate L to a function $\lambda x.N$ · Next, evaluate argument M to a value V · Finally, substitute V for x in the body N, and evaluate it to a value W

$$\frac{L \Downarrow \lambda x.N \quad M \Downarrow V \quad N\{V/x\} \Downarrow W}{L\ M \Downarrow W}$$

To evaluate function L applied to argument M... · The result of the expression is W

- The rule follows the informal description, and is similar in spirit to the rule for evaluating let-bindings

Avoiding Variable Capture

Variable capture is **never desirable** and we need to find a way to avoid it. Fortunately there is a straightforward solution:

`(λf. λmyInt. f)  (λx. x + myInt)`

Whenever we need to substitute under a binder, if we pick fresh names (unused elsewhere) for each of the binders, to generate an α-equivalent expression:

`(λbob. λroger. bob)  (λx. x + myInt)`

Since we know roger is fresh, we know that it definitely **won't** be free in the argument, and we can substitute without fear of variable capture

`(λroger. (λx. x + myInt))`

sli.do 1275413

Implementing Capture-Avoiding Substitution (1)

- The previous definition is **not defined** when variable capture would occur.
- However, we can always make substitution safe when implementing it by applying the variable freshening trick we saw.
- Let $M(x \leftrightarrow y)$ be a **swapping** operation that renames x to y, and y to x, in M, for example:
  `(let x = 5 in x + y)(x ↔ y)  =  let y = 5 in y + x`
  - Full definition in the lab sheet
- We next define `subst(M, N, x)` as the operation to substitute N for x in M, freshening variables where required

Recursion with let-rec

```
let fun
let fun square x =
  x * x
in
square 5 + square 10
```

- Recall our 'square' example from the start of the lecture
- We can't use **`let fun`** (or lambdas) **recursive** functions: here we can't refer to square in the body of the function definition

```
let rec
let rec fac n =
  if n <= 1 then 1 else
  n * (fac (n - 1))
in fac 10
```

- We can instead make a **`let rec`** construct that allows us to refer to the function recursively in its definition.
  - We can therefore write, e.g., factorial, as on the left (with the assumption $n \geq 0$)

Semantics: Informal Description

- A function on its own shouldn't reduce further: we shouldn't be reducing the body of a function without being given an argument
- (Non-examinable caveat: there are times in PL theory where allowing reduction under binders is sometimes useful for reasoning – but it is impractical for real PL designs/implementations)
- To evaluate a function application M N, we:
- Evaluate the function down to a lambda expression (λx. M)
- Evaluate the argument down to a value V
- Replace all occurrences of x in M with V, and evaluate the result

This evaluation strategy of evaluating the argument before substituting into & running the function body is a design choice. Haskell (for example) instead uses lazy evaluation that only evaluates a term when it is needed.

We need to be careful with substitution…

Suppose we have some variable `myInt` (in practice, provided as a system environment variable or something similar). Then what happens if we evaluate the following?

`(λf. λmyInt. f)  (λx. x + myInt)`

⇓ Evaluate the application by calculating `(λmyInt. f) {(λx. x + myInt) / f}`

`λmyInt. (λx. x + myInt)`

Whereas `myInt` was free before, it has now been **captured** by the binder after substitution.

This changes the meaning of the program!

Capture-Avoiding Substitution (Function cases)

`(λx. M) { N / x } = λx. M`

We know there will be **no free occurrences** of x (as all will be bound by the lambda), so can stop.

`(λx. M) { N / y } = (λx . M {N / x})`
if $x \neq y$ and $x \notin fv(N)$

We restrict substitution to only be defined if variable capture doesn't occur

Implementing Capture-Avoiding Substitution (2)

- The full definition of `subst(M, N, x)` is again in the lab sheet, and most of the cases are straightforward:
  - If M is the variable x, then put N there instead
  - For binary operations and function application, substitute into both subterms
- The interesting cases are for functions:

As before, if we're substituting for the same variable as the binder, we know there will be **no free occurrences** of x, so can stop.

`subst(λx. M, N, x) = λx. M`

Anonymous recursive functions

`rec f(x). M`

```
rec fac(n).
  if n <= 1 then 1 else
  n * (fac (n - 1))
```

```
let rec f x = M in N
        ⇔
let f = rec f(x). M in N
```

- As with **`let fun`** and lambda expressions, we can also have **anonymous recursive functions**
- These are just like lambdas, except we give the function a name that is also bound in the function body
- **`let rec`** can also be desugared into anonymous recursive functions
- We refer to the language with **`rec`** functions as L_{Rec}

L_{Lam} Operational Semantics

First evaluate L to an anonymous recursive function · Next, evaluate argument M to a value V · Finally, substitute **a copy of the function** for f, and the argument for V, and evaluate to W

$$\frac{L \Downarrow \mathbf{rec}\ f(x).M \quad M \Downarrow V \quad N\{\mathbf{rec}\ f(x).M\ /\ f,\ V\ /\ x\} \Downarrow W}{L\ M \Downarrow W}$$

To evaluate function L applied to argument M... · The result of the expression is W

- The rule for evaluating an anonymous recursive function application is similar to evaluating a usual function application
- The main difference is that we substitute a **copy** of the **rec** function when evaluating the body

Week 4: Type checking, Semantics and Type Soundness

Types

Types classify terms. Let's begin by considering **base types**: Booleans and Integers

$$\text{Types}\quad A, B ::= \text{Int} \mid \text{Bool}$$

$\vdash M : A$ "Term M has type A"

$$\frac{}{\vdash n : \text{Int}}$$
"Integer literals have type Int"

$$\frac{}{\vdash b : \text{Bool}}$$
"Boolean literals have type Bool"

$$\frac{\vdash M : \text{Int} \quad \vdash N : \text{Int}}{\vdash M + N : \text{Int}}$$
"If M has type Int and N has type Int, then $M + N$ has type Int"

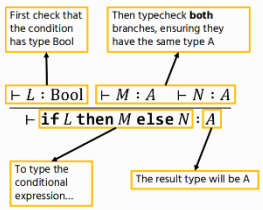
Example: Addition (Well-typed)

$$\frac{}{\vdash n : \text{Int}} \quad \frac{}{\vdash b : \text{Bool}} \quad \frac{\vdash M : \text{Int} \quad \vdash N : \text{Int}}{\vdash M + N : \text{Int}}$$

$$\frac{\dfrac{\vdash 10 : \text{Int} \quad \vdash 15 : \text{Int}}{\vdash 10 + 15 : \text{Int}} \quad \vdash 20 : \text{Int}}{\vdash (10 + 15) + 20 : \text{Int}}$$

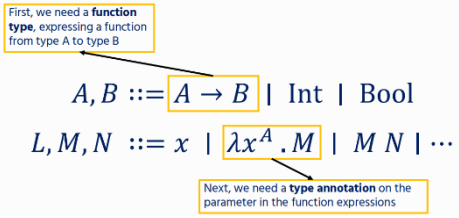
```
ty(+) = Int -> Int -> Int
ty(-) = Int -> Int -> Int
ty(*) = Int -> Int -> Int
ty(/) = Int -> Int -> Int
ty(&&) = Bool -> Bool -> Bool
ty(||) = Bool -> Bool -> Bool
ty(<) = Int -> Int -> Bool
ty(>) = Int -> Int -> Bool
ty(==) = A -> A -> Bool
  (if A = Int or A = Bool)
```

Typing Conditionals



- Note that both the then and the else branches must have the same type
- This is because if is an **expression** and must evaluate to a result, and we need to give the overall expression a type
- Some type systems (e.g. the one used for TypeScript) *do* allow different types in each branch, but this requires heavier machinery (set-theoretic types)

Functions: Additional Types and Syntax



Type Environments

- To handle variables, we need to store their types in a **type environment** (sometimes also called a **typing context**)
- A type environment (normally written as the Greek letter Γ), is a mapping from variables to types

$$\Gamma ::= \cdot \mid \Gamma, x : A$$

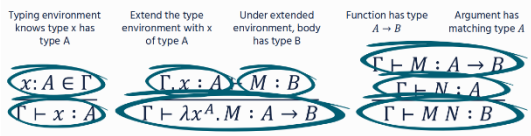
- Whenever we encounter a binding occurrence of a variable, we add it to the type environment so that we can look up the type later
- The other rules (e.g. binary operators and conditionals) also need to be

$$\frac{x : \text{Int} \vdash x : \text{Int} \quad x : \text{Int} \vdash 1 : \text{Int}}{x : \text{Int} \vdash x + 1 : \text{Int}}$$
$$\cdot \vdash \lambda x^{\text{Int}} . x + 1 : \text{Int} \to \text{Int}$$

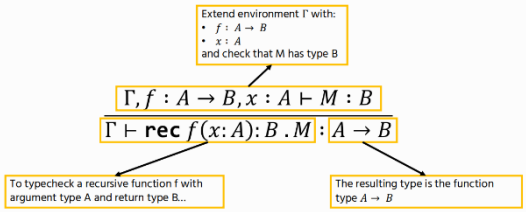
Typing Rules for Variables & Functions

$\Gamma \vdash M : A$ "Under typing environment Γ, term M has type A."

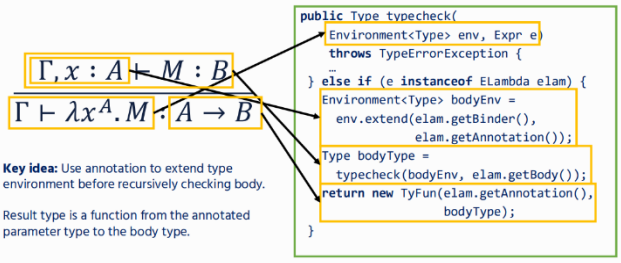
$\Gamma ::= \cdot \mid \Gamma, x : A$



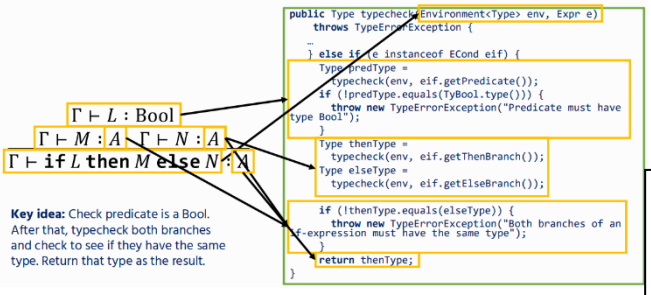
Recursive Functions



Example: Typechecking Lambdas



Example: Typechecking Conditionals



Static vs. Dynamic Typechecking

- So far, we've considered static typechecking, where all typechecking happens before a program is run
- Alternatively, dynamically-checked languages (e.g. Python) tag values with their types, and perform typechecking dynamically to avoid crashes
- The main benefits of static typechecking are early error detection, the ability to check all code paths without running them, and the lower memory / runtime due to absence of dynamic checks
- The main benefit of dynamic typechecking is flexibility, allowing e.g. branching control flow where branches have different types.

Proving Type Soundness for L_{Arith}: A Problem

- Our type soundness statement says that if an expression is well-typed (under an empty environment), then it can evaluate to a value of the same type
- However, what about the following case?

$$(1 + 2) \,/\, 0 \Downarrow ???$$

- Since division by zero is undefined, the expression **does not evaluate to a value** and is a **counterexample** (our interpreters would throw an exception)
- There are a few ways of fixing this (adding exceptions into the language, adding a special case with a default value) – but for simplicity we'll just

Proving Type Soundness: Overall Approach

Theorem: If $\cdot \vdash M : A$ then there exists some V such that $M \Downarrow V$ and $\cdot \vdash V : A$.

To prove this, our approach is as follows:

- Proceed by rule induction on the derivation of $\cdot \vdash M : A$
- For each case, we can **assume** that any subexpressions are typed according to the typing rule
- By the induction hypothesis, we can show that any subexpression will evaluate to a value
- Using this information, we'll use the corresponding big-step rule to show that the whole expression evaluates to a value

Proving Type Soundness for L_{Arith} without division

Theorem: If $\cdot \vdash M : A$ then there exists some V such that $M \Downarrow V$ and $\cdot \vdash V : A$.

Proof: By induction on the derivation of $\cdot \vdash M : A$.

Case $\cdot \vdash n : \text{Int}$

It follows immediately that $\dfrac{}{n \Downarrow n}$, as required

Proving Type Soundness for L_{Arith} without division

Case $\cdot \vdash M \odot N : \text{Int}$

Assumption: $\dfrac{\cdot \vdash M : \text{Int} \quad \cdot \vdash N : \text{Int}}{\cdot \vdash M \odot N : \text{Int}}$

By the induction hypothesis, there exist V, W such that $M \Downarrow V$ and $N \Downarrow W$ and both $\cdot \vdash V : \text{Int}$ and $\cdot \vdash W : \text{Int}$.

Consider the evaluation rule for binary operators: $\dfrac{M \Downarrow V \quad N \Downarrow W}{M \odot N \Downarrow V \mathbin{\widehat{\odot}} W}$

Since {+, -, *} are total functions on integers, $V \mathbin{\widehat{\odot}} W$ is defined, and therefore by the above evaluation rule, $M \odot N \Downarrow V \mathbin{\widehat{\odot}} W$ with

Limitations of Big-Step Semantics

The big-step type soundness property is very strong:

Theorem: If $\cdot \vdash M : A$ then there exists some V such that $M \Downarrow V$ and $\cdot \vdash V : A$.

This **does not** hold for L_{Rec} as it requires that every L_{Rec} term terminates – which is not the case, for example

```
(rec f(x). f x) true
```

But naturally we want to show some form of type soundness property holds for recursive languages!

The answer: reason about evaluation **step-by-step**

Small-Step Operational Semantics

$M \Downarrow V$

Big-step operational semantics

Expression M evaluates to value V

$M \longrightarrow N$

Small-step operational semantics

Expression M takes a reduction step to **expression** N

We write $M \longrightarrow^* N$ to mean M takes zero or more reduction steps to N

Small-Step Rules for L_{Arith}

$$\frac{}{V \odot W \longrightarrow V \mathbin{\widehat{\odot}} W}$$

If both arguments are values, then we can apply the actual operation

$$\frac{M \longrightarrow M'}{M \odot N \longrightarrow M' \odot N}$$

Otherwise, we need to use the **congruence rules**

The first says that if we have an expression $M \odot N$ and M can take a step to M', then the whole expression can take a step to $M' \odot N$.

$$\frac{M \longrightarrow M'}{V \odot M \longrightarrow V \odot M'}$$

The second allows reduction of the second subexpression, if the first is already a value.

The rules enforce a **left-to-right evaluation order**

Evaluating $(1 + (2 * 3)) + (4 * 5)$

$$\frac{\dfrac{2 * 3 \longrightarrow 6}{1 + (2 * 3) \longrightarrow 1 + 6}}{(1 + (2 * 3)) + (4 * 5) \longrightarrow (1 + 6) + (4 * 5)}$$

First step: We can't evaluate either addition, but we **can** evaluate multiplication

$$\frac{1 + 6 \longrightarrow 7}{(1 + 6) + (4 * 5) \longrightarrow 7 + (4 * 5)}$$

Second step: We can now evaluate 1 + 6 to 7

Evaluating $(1 + (2 * 3)) + (4 * 5)$

$$\frac{4 * 5 \longrightarrow 20}{7 + (4 * 5) \longrightarrow 7 + 20}$$

Third step: evaluate the multiplication

$$\frac{}{7 + 20 \longrightarrow 27}$$

Final step: both operands are values, so perform the addition, and we're done

Small-Step Rules for L_{If}

$$\frac{}{\texttt{if true then } M \texttt{ else } N \longrightarrow M}$$

We have **two** main small-step rules for conditionals

The first takes a step to the then branch if the test is the value `true`

$$\frac{}{\texttt{if false then } M \texttt{ else } N \longrightarrow N}$$

The second takes a step to the `else` branch if the test is the value `false`

$$\frac{L \to L'}{\texttt{if } L \texttt{ then } M \texttt{ else } N \longrightarrow \texttt{if } L' \texttt{ then } M \texttt{ else } N}$$

We need a congruence rule to evaluate the test if it isn't yet a value

Small-Step Rules for L_{Rec}

$$\frac{}{(\lambda x.\, M)\, V \longrightarrow M\,\{V \,/\, x\}}$$

When evaluating a function application where the function is a value, reduce to the function body (with argument substituted for parameter)

$$\frac{}{(\,\texttt{rec } f(x).\, M)\, V \longrightarrow M\,\{(\texttt{rec } f(x).\, M \,/\, f,\ V \,/\, x\}}$$

Recursion is similar, but we also need to substitute in a copy of the function

$$\frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N} \qquad \frac{M \longrightarrow M'}{V\ M \longrightarrow V\ M'}$$

Finally, the congruence rules allow us to reduce the function first, and then the argument

Equivalence of Big- and Small-Step Semantics (1)

- Big-step and small-step semantics are different ways of saying the same thing, each with advantages and disadvantages
- Big-step: Closer to how we write interpreters, but cannot reason easily about nonterminating expressions
- Small-step: More fine-grained reasoning power, but (usually) further away from how we would implement a language

Equivalence of Big- and Small-Step Semantics (2)

There's a corresponding small-step reduction sequence for every big-step derivation

If $M \Downarrow V$, then $M \longrightarrow^* V$.

Proof is by rule induction on the derivation of $M \Downarrow V$ (exercise if you're interested!)

There's a corresponding big-step derivation for every small-step reduction sequence from an expression to a value

If $M \longrightarrow^* V$, then $M \Downarrow V$.

Proof is by structural induction on M and inspection of the small-step rules (exercise if you're interested!)

Type Soundness for Small-Step Semantics

We can now specify a more general type soundness property that we can use on L_{Rec}!

If $\cdot \vdash M : A$, then either M is a value V, or there exists some N such that $M \longrightarrow N$ and $\cdot \vdash N : A$.

Preservation

If $\Gamma \vdash M : A$ and there exists some N such that $M \longrightarrow N$, then $\Gamma \vdash N : A$.

Progress

If $\cdot \vdash M : A$, then either M is a value V, or there exists some N such that $M \longrightarrow N$

Type Soundness for Small-Step Semantics

We can now specify a more general type soundness property that we can use on L_{Rec}!

If a program is well typed, then it is either already a value, or it can take a step while staying well typed

Preservation

Reduction doesn't change the result type or introduce type errors

Progress

Well typed processes don't get "stuck"

Preservation

If $\Gamma \vdash M : A$ and there exists some N such that $M \longrightarrow N$, then $\Gamma \vdash N : A$.

sli.do 3400724

$$\frac{}{V \odot W \longrightarrow V \mathbin{\widehat{\odot}} W}$$

Assumption: $\dfrac{\Gamma \vdash V : \text{Int} \quad \Gamma \vdash W : \text{Int}}{\Gamma \vdash V \odot W : \text{Int}}$

As $\odot \in \{+, -, *\}$, we know that $V \mathbin{\widehat{\odot}} W$ is of type Int, as required

$$\frac{M \longrightarrow M'}{M \odot N \longrightarrow M' \odot N}$$

Assumptions: $\dfrac{\Gamma \vdash M : \text{Int} \quad \Gamma \vdash N : \text{Int}}{\Gamma \vdash M \odot N : \text{Int}}$ **and** $M \longrightarrow M'$

By the induction hypothesis, $\Gamma \vdash M' : \text{Int}$, so we can show:

$\dfrac{\Gamma \vdash M' : \text{Int} \quad \Gamma \vdash N : \text{Int}}{\Gamma \vdash M' \odot N : \text{Int}}$ as required.

$$\frac{M \longrightarrow M'}{V \odot M \longrightarrow V \odot M'}$$

Assumptions: $\dfrac{\Gamma \vdash V : \text{Int} \quad \Gamma \vdash M : \text{Int}}{\Gamma \vdash V \odot M : \text{Int}}$ **and** $M \longrightarrow M'$

By the induction hypothesis, $\Gamma \vdash M' : \text{Int}$, so we can show:

$\dfrac{\Gamma \vdash V : \text{Int} \quad \Gamma \vdash M' : \text{Int}}{\Gamma \vdash V \odot M' : \text{Int}}$ as required.

Preservation for L_{If}

$$\frac{}{\texttt{if true then } M \texttt{ else } N \longrightarrow M}$$

(the case for 'false' is similar)

Assumption: $\dfrac{\Gamma \vdash \texttt{true} : \text{Bool} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash \texttt{if true then } M \texttt{ else } N : A}$

It follows immediately from our assumption that $\Gamma \vdash M : A$ as required.

$$\frac{L \longrightarrow L'}{\texttt{if } L \texttt{ then } M \texttt{ else } N \longrightarrow \texttt{if } L' \texttt{ then } M \texttt{ else } N}$$

Assumptions: $\dfrac{\Gamma \vdash L : \text{Bool} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash \texttt{if } L \texttt{ then } M \texttt{ else } N : A}$ **and** $L \longrightarrow L'$

By the induction hypothesis, $\Gamma \vdash L' : \text{Bool}$, so we can show:

$\dfrac{\Gamma \vdash L' : \text{Bool} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash \texttt{if } L' \texttt{ then } M \texttt{ else } N : A}$ as required.

Progress for L_{If}

(Omitting the value cases as they are similar to integer literals)

$$\frac{\Gamma \vdash L : \text{Bool} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash \texttt{if } L \texttt{ then } M \texttt{ else } N : A}$$

By the IH, we know that either L is a value, or $L \longrightarrow L'$

If L is a value V, then since $\cdot \vdash V : \text{Bool}$ it must be the case that either V = true or V = false and we can reduce by (1) or (2)

If $L \longrightarrow L'$ then we can reduce by (3)

1. $\dfrac{}{\texttt{if true then } M \texttt{ else } N \longrightarrow M}$
2. $\dfrac{}{\texttt{if false then } M \texttt{ else } N \longrightarrow N}$
3. $\dfrac{L \longrightarrow L'}{\texttt{if } L \texttt{ then } M \texttt{ else } N \longrightarrow \texttt{if } L' \texttt{ then } M \texttt{ else } N}$

Progress

If $\cdot \vdash M : A$, then either M is a value V, or there exists some N such that $M \longrightarrow N$

sli.do 3400724

$$\frac{}{\Gamma \vdash n : \text{Int}}$$

This case follows immediately, since n is already a value.

$$\frac{\Gamma \vdash M : \text{Int} \quad \Gamma \vdash N : \text{Int}}{\Gamma \vdash M \odot N : \text{Int}}$$

By the induction hypothesis:

- either M is a value, or there exists some N such that $M \longrightarrow M'$
- either N is a value, or there exists some N such that $N \longrightarrow N'$.

So we have three cases we need to consider:

- $M \odot N$, where $M \longrightarrow M'$ -- so we can reduce by rule 2
- $V \odot N$, where $N \longrightarrow N'$ -- so we can reduce by rule 3
- $V \odot W$, so we can reduce by rule 1

1. $\dfrac{}{V \odot W \longrightarrow V \mathbin{\widehat{\odot}} W}$
2. $\dfrac{M \longrightarrow M'}{M \odot N \longrightarrow M' \odot N}$
3. $\dfrac{M \longrightarrow M'}{V \odot M \longrightarrow V \odot M'}$

Week 5: Imperative Programming

L_{While}: A Core Imperative Language

```
n := 10
if n == 0 || n == 1 then
  fibN := n
else
  fibNTake2 := 0
  fibNTake1 := 1
  fibN := 1
  x := 2
  while (x <= n) {
    fibN := fibNTake2 + fibNTake1
    fibNTake2 := fibNTake1
    fibNTake1 := fibN
    x := x + 1
  }
```

- The code on the left is written in L_{While}, a core imperative language
- The program computes the n^{th} Fibonacci number (here we set n to 10)
- Unlike previous languages, we allow **variable assignment** `:=` to change the value pointed to by a variable
- After the program has finished executing, the result will be contained in the `fibN` variable
- Note that the program **doesn't return a result**: the result is contained in program state

L_{While} Syntax

```
Constants c ::= n | b
Values V, W ::= n | b
Expressions
  L, M, N ::= x | c | M ⊙ N
Statements
  s, t ::= s; t
         | skip
         | x := M
         | if M then s else t
         | while M { s }
```

We still need expressions, constants, and values – these will be used for variable assignment or for tests in `if`- or `while`-statements

Statement sequencing: run s then t

`skip` is a no-op: `skip; s = s = s; skip`

Variable assignment: set value of x to the result of evaluating M

Conditional statement: if M evaluates to true, evaluate s; otherwise evaluate t

Evaluate s until M evaluates to false

What does running a statement **mean**?

Given an input state σ, evaluating statement s...

$$\sigma, s \Downarrow \sigma'$$

...will produce updated state σ'.

We need to think about evaluation differently: we're not trying to **produce a value**, but instead **modify the program state**

Here, σ is a mapping from program variables to values

L_{While} Semantics (Expressions)

$$\sigma, M \Downarrow V$$

We might need to evaluate a variable, so we need to update our expression evaluation judgement to take in the current program state
"Under state σ, expression M evaluates to value V".

$$\frac{x \mapsto V \in \sigma}{\sigma, x \Downarrow V} \qquad \frac{}{\sigma, c \Downarrow c} \qquad \frac{\sigma, L \Downarrow V \quad \sigma, M \Downarrow W}{\sigma, L \odot M \Downarrow V \mathbin{\widehat{\odot}} W}$$

"Evaluating" a variable means looking up its value in the state

The rules for constants and binary operators are similar to before, but also include the program state

L_{While} Semantics (Statements, 1)

$\sigma, s \Downarrow \sigma'$

Skip returns the state unchanged

$$\frac{}{\sigma,\ \texttt{skip} \Downarrow \sigma}$$

Evaluate the first statement to get updated state σ'

Evaluate the second statement with σ' to get second state σ''

$$\frac{\sigma,\ s \Downarrow \sigma' \qquad \sigma', t \Downarrow \sigma''}{\sigma,\ s; t \Downarrow \sigma''}$$

Final state is σ''

Evaluate the expression to a value V

$$\frac{\sigma, M \Downarrow V}{\sigma,\ x := M \Downarrow \sigma[x \mapsto V]}$$

Update state to map variable x to V

$$\frac{\sigma, M \Downarrow \texttt{true} \quad \sigma, s \Downarrow \sigma'}{\sigma,\ \texttt{if } M \texttt{ then } s \texttt{ else } t \Downarrow \sigma'} \qquad \frac{\sigma,\ M \Downarrow \texttt{false} \quad \sigma, t \Downarrow \sigma'}{\sigma,\ \texttt{if } M \texttt{ then } s \texttt{ else } t \Downarrow \sigma'}$$

if-statements are similar to the those for if-expressions: evaluate the test, then either evaluate the then- or else- branch accordingly

L_{While} Semantics (Statements, 2)

$\sigma, s \Downarrow \sigma'$

If the test expression evaluates to false...

$$\frac{\sigma,\ M \Downarrow \texttt{false}}{\sigma,\ \texttt{while } M \{ s \} \Downarrow \sigma}$$

Return the state unchanged

If the test expression evaluates to true...

Evaluate the body, producing a new state σ'

Evaluate the while loop again with the updated state

$$\frac{\sigma,\ M \Downarrow \texttt{true} \qquad \sigma,\ s \Downarrow \sigma' \qquad \sigma',\ \texttt{while } M \{ s \} \Downarrow \sigma''}{\sigma,\ \texttt{while } M \{ s \} \Downarrow \sigma''}$$

Return the final state

Example L_{While} Evaluation

$$\frac{\dfrac{}{\cdot,\ 1 \Downarrow 1}}{\cdot,\ x := 1 \Downarrow x \mapsto 1} \qquad \frac{\dfrac{}{x \mapsto 1,\ x \Downarrow 1} \quad \dfrac{}{x \mapsto 1,\ 1 \Downarrow 1}}{x \mapsto 1,\ x + 1 \Downarrow ?}}{x \mapsto 1,\ y := x + 1 \Downarrow ?}$$

$$\cdot,\ x := 1; y := x + 1 \Downarrow ?$$

$$\frac{\dfrac{}{\cdot,\ 1 \Downarrow 1}}{\cdot,\ x := 1 \Downarrow x \mapsto 1} \qquad \frac{\dfrac{}{x \mapsto 1,\ x \Downarrow 1} \quad \dfrac{}{x \mapsto 1,\ 1 \Downarrow 1}}{x \mapsto 1,\ x + 1 \Downarrow 2}}{x \mapsto 1,\ y := x + 1 \Downarrow x \mapsto 1, y \mapsto 2}$$

$$\cdot,\ x := 1; y := x + 1 \Downarrow x \mapsto 1, y \mapsto 2$$

Major missing features in L_{While}

Scoped Variables

{ }

An L_{While} variable is assumed to be in scope from the point of declaration onwards, rather than block scope

Functions / Procedures

L_{While} programs are monolithic blocks of code – but functions (which return a result) and procedures (which don't) can be added reasonably easily

Syntactic Sugar

```
if M then s
⇔
if M then s else skip
```

```
do { s } while M
⇔
s; while M { s }
```

```
for (x in 1..n) { s }
⇔
x := 1;
while (x <= n) {
  s; x := x + 1
}
```

sli.do 1544288

Calc: A simple imperative arithmetic language

```
set x = 13
set y = x * (x + 1)
put x
put y / 2
```

- The first small case study language used in Michele's part of the course is called **Calc**
- It models a simple calculator with a memory, where memory locations are given variable names
  - Expressions are just arithmetic expressions
  - The **set** command sets a memory location to the result of an expression
  - The **put** statement prints the result of an expression to the terminal
- Apart from the **put** statement (which can be added easily), Calc is expressible by L_{While}

The Fun Programming Language

```
func int fact (int n):
  int f = 1
  while n > 1:
    f = f * n
    n = n - 1 .
  return f .

proc main():
  int num = read()
  write(num)
  write(fact(num)) .
```

- Fun is a simple imperative language
- Language features:
  - Global variables
  - Procedures and functions
  - Two data types: bool and int
  - Variable assignment
  - Procedure/Function calls
  - If statements and while loops
  - Console IO
- A Fun program consists of a series of global variables, procedures/functions, and a procedure called `main`

Fun Abstract Syntax (Types, Expressions, and Statements)

sli.do 1544288

```
Variables f, p, x, y, z
Types
  A, B ::= int | bool
Operators
  ⊙ ::= < | > | ==
      | + | - | * | /
Expressions
  L, M, N ::= x | c | f(M?)
            | not M
            | M ⊙ N
Statements
  s, t ::= x = M
         | p(M?)
         | if M: s⃗.
         | if M: s⃗ else s⃗.
         | while M: s⃗.
```

- The syntax of types, expressions, and statements should be familiar from the languages we've seen so far
- We write $\vec{s}$ for a sequence of statements, and $M^?$ for an optional expression
- Minor differences:
  - Unary negation: **not** `M`
  - Function calls $f(M^?)$ and procedure calls $p(M^?)$

Fun Language Abstract Syntax (Definitions and Programs)

sli.do 1

```
Variable Declaration
  v ::= A x = M
Definitions D ::= P | F
Procedure Definitions
  P ::=
    proc p(x?):
      v⃗ s⃗
    .
Function Definitions
  F ::=
    func A f(x?):
      v⃗ s⃗
      return M
    .
```

e.g. `int foo = 5`

A procedure takes an optional parameter, and contains a list of variable declarations followed by a list of statements

A function takes an optional parameter, and contains a list of variable declarations, followed by a list of statements, followed by an expression to return

Fun Programs

Programs are then represented as a series of global variables followed by a sequence of procedure / function definition

$$\texttt{Prog} ::= \vec{v}\ \vec{D}$$

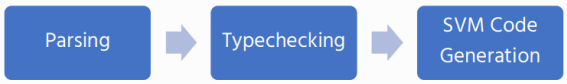
Fun expects a distinguished procedure `main`

It provides a predefined procedure `write`, to write an expression to the terminal

It also provides a predefined function `read`, to read an integer from the terminal

Fun Materials (On Moodle)

- There's an (informal) specification for Fun
  - Full (concrete) syntax, and informal specification of typing and semantics
- There's also a full compiler from Fun to SVM

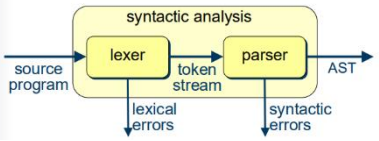


- Main difference to our implementations is that the Fun compiler **does not have explicit classes for ASTs**, instead using ANTLR visitors
- Also, read "syntactic analysis" as "parsing", and "contextual analysis" as "typechecking" in the Fun source code / specification

Week 6: Syntactic analysis

Syntactic analysis checks that the source program is well-formed and determines its phrase structure.
▪ Syntactic analysis can be decomposed into:
– a lexer (which breaks the source program down into tokens)
– a parser (which determines the phrase structure of the source program)

▪ *Recall:* The syntactic analyser inputs a source program and outputs an AST.
▪ Inside the syntactic analyser, the lexer channels a stream of tokens to the parser:

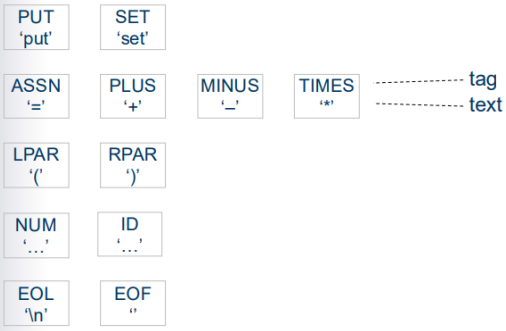


Tokens are textual symbols that influence the source program's phrase structure, e.g.: – literals – identifiers – operators – keywords – punctuation (parentheses, commas, colons, etc.)
▪ Each token has a tag and a text. E.g.: – the addition operator might have tag PLUS and text '+' – a numeral might have tag NUM, and text such as '1' or '37' – an identifier might have tag ID, and text such as 'x' or 'a1'

▪ Separators are pieces of text that do not influence the phrase structure, e.g.: – spaces – comments.
▪ An end-of-line is: – a separator in most PLs – a token in Python (since it delimits a command).

The lexer converts source code to a token stream.
▪ At each step, the lexer inspects the next character of the source code and acts accordingly (see next slide).
▪ When no source code remains, the lexer outputs an EOF token.
▪ E.g., if the next character of the source code is: – a space: discard it.
– the start of a comment: scan the rest of the comment, and discard it.
– a punctuation mark: output the corresponding token.
– a digit: scan the remaining digits, and output the corresponding token (a numeral).
– a letter: scan the remaining letters, and output the corresponding token (which could be an identifier or a keyword).

▪ Complete list of Calc tokens:

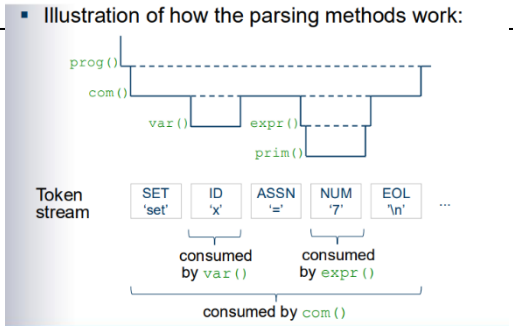


The parser converts a token stream to an AST.
▪ There are many possible parsing algorithms.
▪ Recursive-descent parsing is particularly simple and attractive.
▪ Given a suitable grammar for the source language, we can quickly and systematically write a recursive-descent parser for that language
▪ A recursive-descent parser consists of: – a family of parsing methods N(), one for each nonterminal symbol N of the source language's grammar – an auxiliary method match() .
▪ These methods "consume" the token stream from left to right.

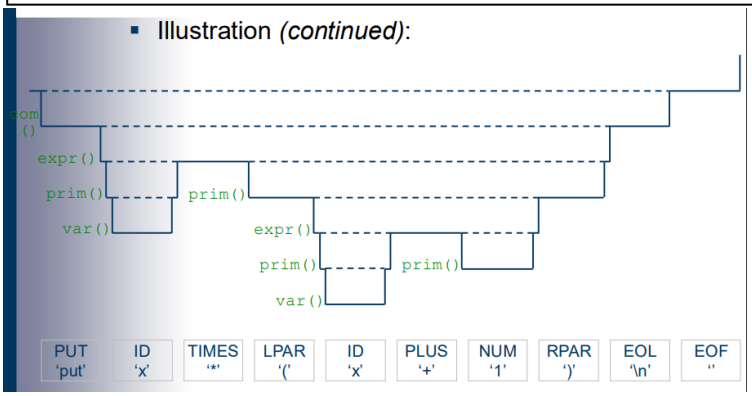
▪ Method match(t) checks whether the next token has tag t. – If yes, it consumes that token. – If no, it reports a syntactic error.
▪ For each nonterminal symbol N, method N() checks whether the next few tokens constitute a phrase of class N.
– If yes, it consumes those tokens (and returns an AST representing the parsed phrase).
– If no, it reports a syntactic error.

▪ Parsing methods: prog(), com(), expr(), prim(), var() parses a program, parses a command, parses an expression, parses a primary expression, parses a variable

▪ Illustration of how the parsing methods work:



▪ Illustration *(continued)*:



- Recall the EBNF production rule for commands:

  com = 'put' *expr eol*
  | 'set' *var* '=' *expr eol*

- Parsing method for commands *(outline)*:

```
void com () {
  if (…) {                    ---- if the next token is 'put'
    match(PUT);
    expr();
    match(EOL);
  }
```

- Parsing method for commands *(continued)*:

```
  else if (…) {               ---- if the next token is 'set'
    match(SET);
    var();
    match(ASSN);
    expr();
    match(EOL);
  }
  else
    …                         ---- report a syntactic error
}
```

- Recall the EBNF production rule for programs:

  prog = *com* eof*

- Parsing method for programs *(outline)*:

```
void prog () {
  while (…) {                 ---- while the next token is 'set' or 'put'
    com();
  }
  match(EOF);
}
```

- Consider the EBNF production rule:

  N = *RE*

- The corresponding parsing method is:

```
void N () {
  match the pattern RE
}
```

- To match the pattern *t* (where *t* is a terminal symbol):

  match(*t*);

- To match the pattern *N* (where *N* is a nonterminal symbol):

  N();

- To match the pattern RE_1 RE_2:

  match the pattern RE_1
  match the pattern RE_2

- To match the pattern RE_1 | RE_2:

```
if (the next token can start RE1)
  match the pattern RE1
else if (the next token can start RE2)
  match the pattern RE2
else
  report a syntactic error
```

- *Note:* This works only if no token can start both RE_1 and RE_2.
  - In particular, this does not work if a production rule is left-recursive, e.g., *N* = *X* | *N Y*.

- To match the pattern *RE* *:

```
while (the next token can start RE)
  match the pattern RE
```

- *Note:* This works only if no token can both start and follow *RE*.

▪ Syntactic analysis has a variety of applications: – in compilers – in XML applications (parsing XML documents and converting them to tree form) – in web browsers (parsing and rendering HTML documents) – in natural language applications (parsing and translating NL documents).

▪ A compiler generation tool automates the process of building compiler components.
▪ The input to a compiler generation tool is a specification of what the compiler component is to do. E.g.: – The input to a parser generator is a grammar .
▪ Examples of compiler generation tools: – lex and yacc (see Advanced Programming) – JavaCC – SableCC – ANTLR.

▪ ANTLR (ANother Tool for Language Recognition) is the tool we shall use here. See www.antlr.org.
▪ ANTLR can automatically generate a lexer and recursive-descent parser, given a grammar as input: – The developer starts by expressing the source language's grammar in ANTLR notation (which resembles EBNF). – Then the developer enhances the grammar with actions and/or tree-building operations.
▪ ANTLR can also generate contextual analysers and code generators (see §8).

- Calc grammar expressed in ANTLR notation:

```
grammar Calc;

prog
     : com* EOF
     ;

com
     : PUT expr EOL
     | SET var ASSN expr EOL
     ;

var
     : ID
     ;
```

- Calc grammar *(continued)*:

```
expr
     : prim
       ( PLUS prim
       | MINUS prim
       | TIMES prim
       )*
     ;

prim
     : NUM
     | var
     | LPAR expr RPAR
     ;
```

- Calc grammar *(continued – lexicon)*:

```
PUT   : 'put' ;
SET   : 'set' ;

ASSN  : '=' ;
PLUS  : '+' ;
MINUS : '-' ;
TIMES : '*' ;
LPAR  : '(' ;
RPAR  : ')' ;

ID    : 'a'..'z' ;
NUM   : '0'..'9'+ ;

EOL   : '\r'? '\n' ;
SPACE : (' ' | '\t')+    {skip();} ;
```

Tokens and separators have upper-case names.

This says that a SPACE is a separator.

- Write a driver program that calls `CalcParser`'s method `prog()`:

```
public class CalcRun {
  public static void main (String[] args) {
    InputStream source =
      new InputStream(args[0]);
    CalcLexer lexer = new CalcLexer(
      new ANTLRInputStream(source));
    CommonTokenStream tokens =
      new CommonTokenStream(lexer);
    CalcParser parser =
      new CalcParser(tokens);
    parser.prog();
  }
```

creates an input stream
creates a lexer
runs the lexer, creating a token stream
creates a parser
runs the parser

- Put the above grammar in a file named `Calc.g`.
- Feed this as input to ANTLR:

  …$ `java org.antlr.Tool Calc.g`

- ANTLR automatically generates the following classes:
  - Class `CalcLexer` contains methods that convert an input stream (source code) to a token stream.
  - Class `CalcParser` contains parsing methods `prog()`,
- When compiled and run, `CalcRun` performs syntactic analysis on the source program, reporting any syntactic errors.
- However, `CalcRun` does nothing else!

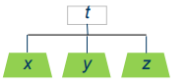
- Normally we want to make the parser do something useful.
- To do this, we enhance the ANTLR grammar with either actions or tree-building operations.
- An ANTLR **action** is a segment of Java code:

```
{ code }
```

- An ANTLR **tree-building operation** has the form:

```
-> ^(t x y z)
```

where *t* is a token and *x*, *y*, *z* are subtrees.



- Augmented Calc grammar:

```
grammar Calc;

@members {
    private int[] store
      = new int[26];
}

prog
    : com* EOF
    ;
```

storage for variables 'a', …, 'z'

- Suppose that we want `CalcRun` to perform actual calculations:
  - The command "`put` *expr*" should evaluate the expression *expr* and then print the result.
  - The command "`set` *var* = *expr*" should evaluate the expression *expr* and then store the result in the variable *var*.

- Augmented Calc grammar *(continued)*:

```
com
    : PUT v=expr EOL   { println(v); }

    | SET ID ASSN
      v=expr EOL       { int a =
                           $ID.text.charAt(0)
                           - 'a';
                         store[a] = v; }
    ;
```

`$ID.text` is the text of `ID` (a string of letters)
`$ID.text.charAt(0)` is the 1st letter.
`$ID.text.charAt(0)-'a'` is in the range 0..25.

- We can augment the Calc grammar with actions to do this:
  - Create storage for variables 'a', …, 'z'.
  - Declare that `expr` will return a value of type `int`. Add actions to compute its value. And similarly for `prim`.
  - Add an action to the `put` command to print the value returned by `expr`.
  - Add an action to the `set` command to store the value returned by `expr` at the variable's address in the store.

- Augmented Calc grammar *(continued)*:

```
expr                    returns [int val]
    : v1=prim           { $val = v1; }
      ( PLUS v2=prim    { $val += v2; }
      | MINUS v2=prim   { $val -= v2; }
      | TIMES v2=prim   { $val *= v2; }
      )*
    ;
```

- Run ANTLR as before: …$ java org.antlr.Tool Calc.g
- ANTLR inserts the @members{…} code into the CalcParser class.
- ANTLR inserts the above actions into the com(), expr(), and prim() methods of CalcParser.

- At the top of the expr production rule, " expr returns [int val]" declares that parsing an expr will return an integer result named val.
- Within actions in the expr production rule, "$val = …" sets the result.
- In any production rule, "v=expr" sets a local variable v to the result of parsing the expr.

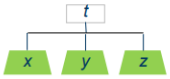
- When compiled and run, `CalcRun` again performs syntactic analysis on the source program, but now it also performs the actions:

```
…$ javac CalcLexer.java CalcParser.java \
      CalcRun.java
…$ java CalcRun test.calc
16
56
72
```

```
set c = 8
set e = 7
put c*2
put e*8
set m = (c*2) + (e*8)
put m
```

- What if the parser is required to build an AST?
- Start with an EBNF grammar of the source language, together with a summary of the ASTs to be generated.
- Express the grammar in ANTLR's notation. Then add tree-building operations to specify the translation from source language to ASTs.
- *Recall:* An ANTLR tree-building operation has the form:

```
-> ^(t x y z)
```



- Fun grammar *(outline)*:

```
grammar Fun;

prog
    : var_decl* proc_decl+ EOF
    ;

var_decl
    : type ID ASSN expr
    ;

type
    : BOOL
    | INT
    ;
```

- Fun grammar *(continued)*:

```
com
    : ID ASSN expr
    | IF expr COLON seq_com DOT
    | …
    ;

seq_com
    : com*
    ;
```

- Fun grammar *(continued)*:

```
expr : sec_expr …

sec_expr
    : pri_expr
        ( (PLUS | MINUS | TIMES | DIV)
          pri_expr
        )*
    ;

pri_expr
    : NUM
    | ID
    | LPAR expr RPAR
    | …
    ;
```

- Augmented Fun grammar *(outline)*:

```
grammar Fun;

options {
    output = AST;
    …;
}

tokens {
    PROG;
    SEQ;
    …;
```

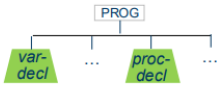
states that this grammar will generate an AST

lists special tokens to be used in the AST (in addition to lexical tokens)

- Augmented Fun grammar *(outline)*:

```
prog
    : var_decl* proc_decl+ EOF
                    -> ^(PROG
                         var_decl*
                         proc_decl+)
    ;
```

builds an AST like this:



- Augmented Fun grammar *(continued)*:

```
com
    : ID ASSN expr      -> ^(ASSN ID expr)
    | IF expr COLON
      seq_com DOT       -> ^(IF expr seq_com)
    | …
    ;

seq_com
    : com*              -> ^(SEQ com*)
```

- Augmented Fun grammar *(continued)*:

```
expr : sec-expr …

sec_expr
    : prim_expr
        ( (PLUS^ | MINUS^ | TIMES^ | DIV^)
          prim_expr
        )*
    ;

prim_expr
    : NUM               -> NUM
    | ID                -> ID
    | LPAR expr RPAR    -> expr
    | …
    ;
```

builds an AST like this:

- Put the above grammar in a file named `Fun.g`.
- Run ANTLR to generate a lexer and a parser:

```
…$ java org.antlr.Tool Fun.g
```

- ANTLR creates the following classes:
  - Class `FunLexer` contains methods that convert an input stream (source code) to a token stream.
  - Class `FunParser` contains parsing methods `prog()`, `var_decl()`, `com()`, …, that consume the token stream.
- The `prog()` method now returns an AST.

Week 7: VM code generation, svm specs

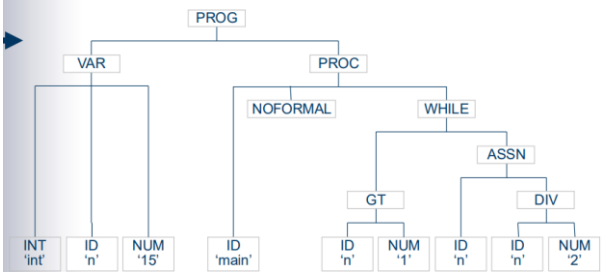
Code generation translates the source program (represented by an AST / syntax tree) into equivalent object code.

- In general, code generation can be broken down into:
  - address allocation (deciding the representation and address of each variable in the source program)
  - code selection (selecting and generating object code)
  - register allocation (where applicable) (assigning registers to local and temporary variables).

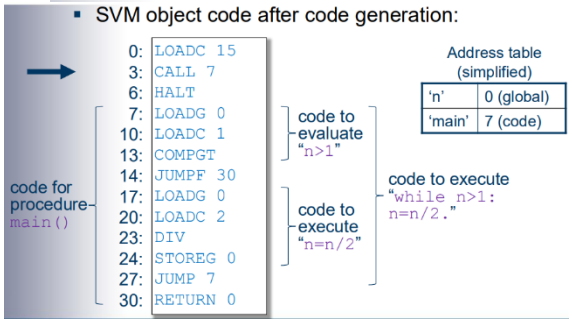
Source program:

```
int n = 15
# pointless program
proc main ():
  while n > 1:
    n = n/2 .
.
```

AST after syntactic analysis (slightly simplified):



SVM object code after code generation:



- Address allocation requires collection and dissemination of information about declared variables, procedures, etc.
- The code generator employs an **address table**. This contains the address of each declared variable, procedure, etc. E.g.:

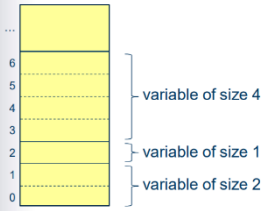
'x'	0 (global)
'y'	2 (global)
'fac'	7 (code)
'main'	30 (code)

variables / procedures

At each variable declaration, allocate a suitable address, and put the identifier and address into the address table.

- Wherever a variable is used (e.g., in a command or expression), retrieve its address.
- At each procedure declaration, note the address of its entry point, and put the identifier and address into the address table.
- Wherever a procedure is called, retrieve its address.

- Allocate consecutive addresses to variables, taking account of their sizes. E.g.:



- *Note:* Fun is simpler: all variables are of size 1.

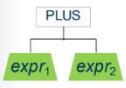
- The code generator must distinguish between three kinds of addresses:
  - A code address refers to an instruction within the space allocated to the object code.
  - A global address refers to a location within the space allocated to global variables.
  - A local address refers to a location within a space allocated to a group of local variables.

Implementation in Java:

```
public class Address {

   public static final int
      CODE = 0, GLOBAL = 1, LOCAL = 2;

   public int offset;
   public int locale; // CODE, GLOBAL, or LOCAL

   public Address (int off, int loc) {
     offset = off;  locale = loc;
   }

}
```

- The code generator will walk the AST.
- For each construct (expression, command, etc.) in the AST, the code generator must emit suitable object code.
- The developer must plan what object code will be selected by the code generator.

Code template for binary operator:



code to evaluate *expr*$_1$
code to evaluate *expr*$_2$
`ADD`

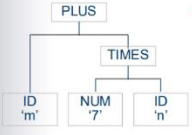
- For each construct in the source language, the developer should devise a code template. This specifies what object code will be selected.
- The code template to evaluate an expression should include code to evaluate any subexpressions, together with any other necessary instructions.
- The code template to execute a command should include code to evaluate any subexpressions and code to execute any subcommands, together with any other necessary instructions.

Code generator action for binary operator:

walk *expr*$_1$ generating code;
walk *expr*$_2$ generating code;
emit instruction "`ADD`"

- Compare:
  - The *code template* specifies what code should be selected.
  - The *action* specifies what the code generator will actually do to generate the selected code.

- E.g., code to evaluate "`m+(7*n)`":

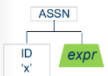



```
LOADG 3
LOADC 7
LOADG 4
MULT
ADD
```

code to evaluate "`m`"
code to evaluate "`7*n`"

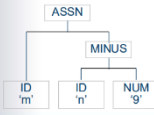
- We are assuming that `m` and `n` are global variables at addresses 3 and 4, respectively.

Code template for assignment-command:



code to evaluate *expr*
`STOREG` *d* or `STOREL` *d*
where *d* is the address offset of 'x'

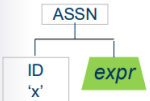
- E.g., code to execute "`m = n-9`":




```
LOADG 4
LOADC 9
SUB
STOREG 3
```

code to evaluate "`n-9`"

Code generator action for assignment-command:



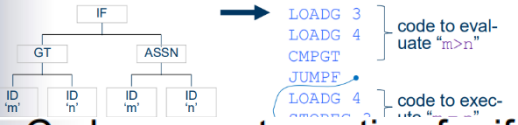
walk *expr* generating code;
lookup 'x' and retrieve its address *d*;
emit instruction "`STOREG` *d*" (if x is global)
or "`STOREL` *d*" (if x is local)

- The code generator emits instructions one by one. When an instruction is emitted, it is added to the end of the object code.
- At the destination of a jump instruction, the code generator must note the destination address and incorporate it into the jump instruction.

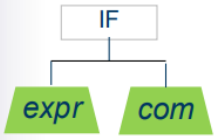
- For a backward jump, the destination address is already known when the jump instruction is emitted.
- For a forward jump, the destination address is unknown when the jump instruction is emitted. Solution:
  – Emit an incomplete jump instruction (with 0 in its address field), and note its address.
  – When the destination address becomes known later, patch that address into the jump instruction.

- Code template for if-command:

- E.g., code to execute "`if m>n: m = n.`":



Code generator action for if-command:



walk *expr*, generating code;
emit instruction "`JUMPF 0`";
walk *com*, generating code;
patch the correct address into the above `JUMPF` instruction

- Code generator action:

note the current instruction address c_1
walk *expr*, generating code
note the current instruction address c_2
emit "JUMPF 0"
walk *com*, generating code
emit "JUMP c_1"
note the current instruction address c_3
patch c_3 into the jump at c_2

c_1 7 c_2 14 c_3 30

```
 0: ...
    ...
 7: LOADG 0
10: LOADC 1
13: COMPGT
14: JUMPF 30
17: LOADG 0
20: LOADC 2
23: DIV
24: STOREG 0
27: JUMP 7
30:
```

- The code generator is a visitor, with a similar structure to the contextual analysis visitor.
- For each type of syntax tree node, the visit method implements the code generation action.

```
class FunEncoderVisitor extends AbstractParseTreeVisitor<Void>
                        implements FunVisitor<Void> {
   SVM obj = new SVM();
   int globalvaraddr = 0;
   int localvaraddr = 0;
   int currentLocale = Address.GLOBAL;

   SymbolTable<Address> addrTable = new SymbolTable<Address>();
...
}
```

Creates an instance of the SVM. The code generator will emit instructions directly into its code store.

```
Void visitNum(FunParser.NumContext ctx) {
   int value = Integer.parseInt(ctx.NUM().getText());
   obj.emit12(SVM.LOADC, value); // emit12 means 1 opcode +
   return null;                  // 2 byte operand
}

Void visitId(FunParser.IdContext ctx) {
   String id = ctx.ID().getText();
   Address varaddr = addrTable.get(id);
   switch (varaddr.locale) {
      case Address.GLOBAL:
         obj.emit12(SVM.LOADG,varaddr.offset);
         break;
      case Address.LOCAL:
         obj.emit12(SVM.LOADL,varaddr.offset);
   }
   return null;
}
```

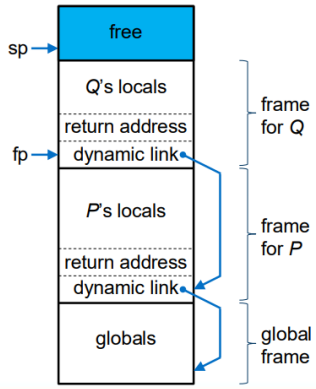
```
// com : ID ASSN expr # assn

Void visitAssn(FunParser.AssnContext ctx) {
   visit(ctx.expr()); // Generate code to evaluate expr
   String id = ctx.ID().getText();
   // Find the address of the variable.
   // This always succeeds, because we assume that the
   // program has been through the contextual analyser.
   Address varaddr = addrTable.get(id);
   switch (varaddr.locale) {
      case Address.GLOBAL:
         obj.emit12(SVM.STOREG,varaddr.offset);
         break;
      case Address.LOCAL:
         obj.emit12(SVM.STOREL,varaddr.offset);
      }
   return null;
}
```

```
// expr : e1=sec_expr (op=(EQ | LT | GT) e2=sec_expr)?

Void visitExpr(FunParser.ExprContext ctx) {
   visit(ctx.e1); // Generate code to evaluate e1
   if (ctx.e2 != null) {
      visit(ctx.e2); // Generate code to evaluate e2
      switch (ctx.op.getType()) { // Generate an
         case FunParser.EQ:       // instruction for the
            obj.emit1(SVM.CMPEQ); // operator.
            break;
         case FunParser.LT:
            obj.emit1(SVM.CMPLT); // emit1 means 1 opcode
            break;
         case FunParser.GT:
            obj.emit1(SVM.CMPGT);
            break;
      }
   }
   return null;
}
```

```
// IF expr COLON c1=seq_com (DOT | ELSE COLON c2=seq_com DOT) # if

Void visitIf(FunParser.IfContext ctx) {
   visit(ctx.expr());
   int condaddr = obj.currentOffset();
   obj.emit12(SVM.JUMPF, 0); // This has to be patched later.
   if (ctx.c2 == null) { // IF without ELSE
      visit(ctx.c1);
      int exitaddr = obj.currentOffset();
      obj.patch12(condaddr, exitaddr);
   }
   else {                // IF ... ELSE
      visit(ctx.c1);
      int jumpaddr = obj.currentOffset();
      obj.emit12(SVM.JUMP, 0); // This also has to be patched.
      int elseaddr = obj.currentOffset();
      obj.patch12(condaddr, elseaddr);
      visit(ctx.c2);
      int exitaddr = obj.currentOffset();
      obj.patch12(jumpaddr, exitaddr);
   }
   return null;
}
```

```
// var_decl : type ID ASSN expr  # var

Void visitVar(FunParser.VarContext ctx) {
   visit(ctx.expr());
   String id = ctx.ID().getText();
   switch (currentLocale) {
      // Adding the variable to the address table always succeeds,
      // because we assume we have done contextual analysis, so it
      // is guaranteed to be a new variable name.
      case Address.LOCAL:
         addrTable.put(id, new Address(localvaraddr++,
                                       Address.LOCAL));
         break;
      case Address.GLOBAL:
         addrTable.put(id, new Address(globalvaraddr++,
                                       Address.GLOBAL));
   }
   return null;
}
```

- SVM data store when the main program has called *P*, and *P* has called *Q*:
- **sp** (stack pointer) points to the first free cell above the top of the stack.
- **fp** (frame pointer) points to the first cell of the topmost frame.



Storage Organisation

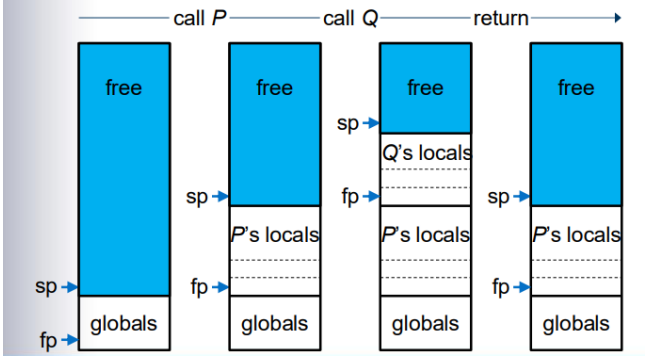
Each variable occupies storage space throughout its lifetime. That storage space must be:
– allocated at the start of the variable's lifetime
– deallocated at the end of the variable's lifetime.
▪ Assumptions:
– The PL is statically typed, so every variable's type is known to the compiler.
– All variables of the same type occupy the same amount of storage space.

▪ A global variable's lifetime is the program's entire run-time.
▪ For global variables, the compiler allocates fixed storage space.
▪ A local variable's lifetime is an activation of the block in which the variable is declared.
The lifetimes of local variables are nested.
▪ For local variables, the compiler allocates storage space on a stack

▪ At any given time, the stack contains one or more activation frames:
– The frame at the base of the stack contains the global variables.
– For each active procedure P, there is a frame containing P's local variables.
An active procedure is one that has been called but not yet returned.
▪ A frame for procedure P is:
– pushed on to the stack when P is called – popped off the stack when P returns.

▪ The compiler fixes the size and layout of each frame.
▪ The offset of each global/local variable (relative to the base of the frame) is known to the compiler.

- Effect of calls and returns:



Such a table can be implemented by a pair of hash-tables, one for globals and one for locals:

```
public class SymbolTable<A> {

   // A SymbolTable<A> object represents a scoped
   // table in which each entry consists of an identifier
   // and an attribute of type A.

   private HashMap<String,A>
      globals, locals;

   public SymbolTable () {
      globals = new HashMap<String,A>();
      locals = null;  // Initially there are no locals.
   }
```

Implementation in Java *(continued)*:

```
   public void enterLocalScope () {
      locals = new HashMap<String,A>();
   }

   public void exitLocalScope () {
      locals = null;
   }
```

Implementation in Java *(continued)*:

```
   public void put (String id, A attr) { … }
   // Add an entry (id, attr) to the locals (if not null),
   // otherwise add the entry to the globals.

   public A get (String id) { … }
   // Retrieve the attribute corresponding to id in
   // the locals (if any), otherwise retrieve it from
   // the globals.

}
```

Now the type table can be declared thus:

```
SymbolTable<Type> typeTable;
```

Week 8: Variables and Scope

In functional PLs (and in mathematics), a variable stands for a fixed (but possibly unknown) value.
▪ In imperative and OO PLs, a variable contains a value. The variable may be inspected and updated as often as desired.
– Such a variable can be used to model a real-world object whose state changes over time

▪ A simple variable is one that contains a primitive value or pointer. – A simple variable occupies a single allocated cell.
▪ A composite variable is one that contains a composite value. – A composite variable occupies a group of adjacent allocated cells.

▪ A variable of a composite type has the same structure as a value of the same type. For instance: – A tuple variable is a tuple of component variables. – An array variable is a mapping from an index range to a group of component variables.
▪ Depending on the PL, a composite variable can be: – totally updated (all at once), and/or – selectively updated (one component at a time).

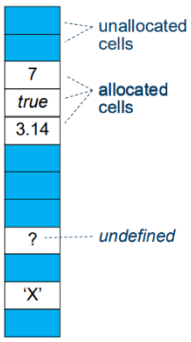
▪ Every variable is created at some definite time, and destroyed at some later time when it is no longer needed.
▪ A variable's lifetime is the interval between its creation and destruction.
▪ A variable occupies cells only during its lifetime. When the variable is destroyed, these cells may be deallocated. – And these cells may subsequently be re-allocated to other variable(s).

▪ A global variable's lifetime is the program's entire run-time. It is created by a global declaration.
▪ A local variable's lifetime is an activation of a block. It is created by a declaration within that block, and destroyed on exit from that block.
▪ A heap variable's lifetime is arbitrary, but bounded by the program's run-time. It can be created at any time (by an allocator), and may be destroyed at any later time. It is accessed through a pointer.

Outline of C program:

```
void main () {
    float m;
    ... r(3); ...
}

void r (int n) {
    int i;
    if (n > 1) {
        ... r(n-1); ...
    }
}
```

- To understand such variables, assume an abstract storage model:
  - A **store** is a collection of **cells**, each of which has a unique **address**.
  - Each cell is either *allocated* or *unallocated*.
  - Each allocated cell contains either a simple value or *undefined*.
  - An allocated cell can be **inspected**, unless it contains *undefined*.
  - An allocated cell can be **updated** at any time.



- Declaration and updating of struct variables:

```
struct Date {int y, m, d;}
struct Date xmas, today;

xmas.d = 25;
xmas.m = 12;
xmas.y = 2008;
```
selective updating

```
today = xmas;
```
total updating

- Declaration and updating of array variable:

```
float a[10];

a[i] = 3.1417;
```
selective updating

Outline of C program:

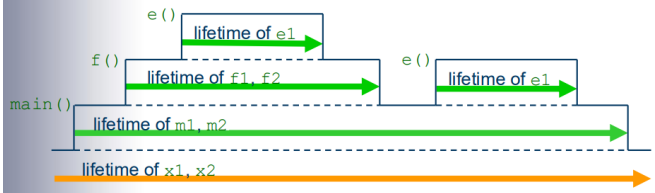
```
extern int x1, x2;
```
global variables

```
void main () {
    int m1; float m2;
    ... f(); ... e(); ...
}
```
local variables

```
void f () {
    float f1; int f2;
    ... e(); ...
}
```
local variables

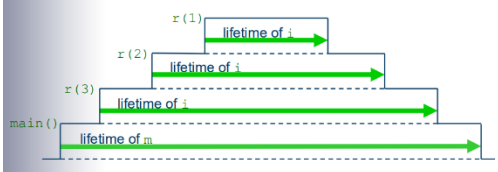
```
void e () {
    int e1;
```
local variable

- Lifetimes of global and local variables:



- Global and local variables' lifetimes are nested. They can never be overlapped.

- Lifetimes of global and local variables:



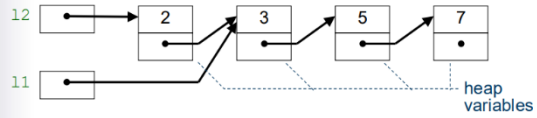
- *Note:* A local variable of a recursive procedure/function has several nested lifetimes.

Example: C heap variables *(1)*

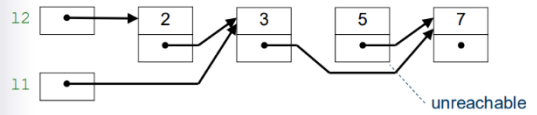
Outline of C program:

```
struct IntNode {int elem; IntList succ;}
typedef struct IntNode * IntList;

IntList c (int h, IntList t) {
    // Return an IntList with head h and tail t.
    IntList ns =
        (IntList) malloc (sizeof IntNode);
    ns->elem = h; ns->succ = t;
    return ns;
}

void d (IntList ns) {
    ns->succ = ns->succ->succ;
}
void main {
    IntList l1, l2;
    l1 = c(3, c(5, c(7, NULL)));
    l2 = c(2, l1);
    d(l1);
}
```

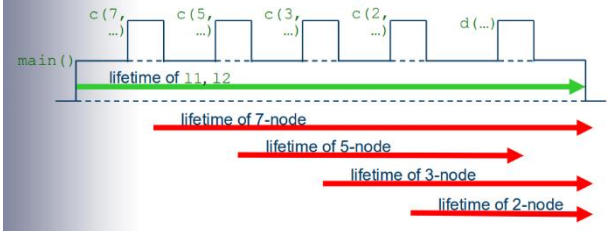
- After initializing `l1` and `l2`:



- After calling `d(l1)`:



- Lifetimes of local and heap variables:



- Heap variables' lifetimes *can* overlap one another and local/global variables' lifetimes.

- An allocator is an operation that creates a heap variable, yielding a pointer to that heap variable.
E.g.: – C's allocator is a library function, malloc(). – Java's allocator is an expression of the form " new C(...)".
- A deallocator is an operation that explicitly destroys a designated heap variable.
E.g.: – C's deallocator is a library function, free(). – Java has no deallocator at all.

- A heap variable remains reachable as long as it can be accessed by following pointers from a global or local variable.
- A heap variable's lifetime extends from its creation until: – it is destroyed by a deallocator, or – it becomes unreachable, or – the program terminates.

Consider this C code:

```
struct Date {int y, m, d;}
typedef Date * DatePtr;

DatePtr date1 = (DatePtr)
   malloc(sizeof Date);
date1->y = 2008;
date1->m = 1;
date1->d = 1;

DatePtr date2 = date1;

free(date2);

printf("%d4", date1->y);
date2->y = 2009;
```

makes `date1` point to a newly-allocated heap variable

makes `date2` point to that same heap variable

deallocates that heap variable – `date1` and `date2` now contain dangling pointers

behaves unpredictably

- A pointer is a reference to a particular variable. (In fact, pointers are sometimes called references.)
- A pointer's referent is the variable to which it refers.
- A null pointer is a special pointer value that has no referent.
- A pointer is essentially the address of its referent in the store. – However, each pointer also has a type, and the type of a pointer allows us to infer the type of its referent.

- Pointers and heap variables can be used to represent recursive values such as lists and trees.
- But the pointer itself is a low-level concept. Manipulation of pointers is notoriously errorprone and hard to understand.
- For example, the C assignment "p->succ = q;" appears to manipulate a list, but which list? Also: – Does it delete nodes from the list? – Does it stitch together parts of two different lists? – Does it introduce a cycle?

- A dangling pointer is a pointer to a variable that has been destroyed.
- Dangling pointers arise when: – a pointer to a heap variable still exists after the heap variable is destroyed by a deallocator – a pointer to a local variable still exists at exit from the block in which the local variable was declared.
- A deallocator immediately destroys a heap variable; all existing pointers to that heap variable then become dangling pointers. – Thus deallocators are inherently unsafe.

- C is highly unsafe: – After a heap variable is destroyed, pointers to it might still exist. – At exit from a block, pointers to its local variables might still exist (e.g., if stored in global variables).
- Java is very safe: – It has no deallocator. – Pointers to local variables cannot be obtained.

- A command (often called a statement) is a program construct that will be executed to update variables.
- Commands are characteristic of imperative and OO PLs (but not functional PLs).
- Simple commands: – A skip command is a command that does nothing. – An assignment command is a command that uses a value to update a variable. – A procedure call is a command that calls a proper procedure with argument(s). Its net effect is to update some variables.
- Compound commands: – A sequential command is a command that executes its sub-commands in sequence. – A conditional command is a command that chooses one of its sub-commands to execute. – An iterative command is a command that executes its sub-command repeatedly. This may be:
  - definite iteration (where the number of repetitions is known in advance)
  - indefinite iteration (where the number of repetitions is not known in advance).
- A block command is a command that contains declarations of local variables, etc.

Java single assignment:

```
m = n + 1;
```

Java multiple assignment:

```
m = n = 0;
```

Java assignment combined with binary operator:

```
m += 7;      ---------- equivalent to "m = m+7;"
n /= b;      ---------- equivalent to "n = n/b;"
```

Java if-command:

```
if (x > y)
   out.print(x);
else
   out.print(y);
```

Java switch-command:

```
Date today = …;
switch (today.m) {
   case  1:  out.print("JAN"); break;
   case  2:  out.print("FEB"); break;
   …
   case 11:  out.print("NOV"); break;
   case 12:  out.print("DEC");
}
```

Breaks are essential here!

Java while-command:

```
Date[] dates;
…
int i = 0;
while (i < dates.length) {
   out.print(dates[i]);
   i++;
}
```

Java for-commands (both forms):

```
for (int i = 0; i < dates.length; i++)
   out.print(dates[i]);

for (Date d : dates)
   out.print(d);
```

Java do-while-command:

```
static String render (int n) {
   String s = "";
   int m = n;
   do {
      char d = (char)(m % 10 +
(int)'0');
      s = d + s;
      m /= 10;
   } while (m > 0);
   return s;
}
```

Here the loop condition is evaluated *after* each repetition of the loop body.

Java block command:

```
if (x > y) {
   int z = x;
   x = y;
   y = z;
}
```

The primary purpose of evaluating an expression is to yield a value.

In most imperative and OO PLs, evaluating an expression can also update variables – these are **side effects**.

In C and Java, the body of a function is a *command*. If that command updates global or heap variables, calling the function has side effects.

In C and Java, assignments are in fact expressions with side effects: "$V = E$" stores the value of E in V as well as yielding that value.

The C function `getchar(fp)` reads a character and updates the file variable that `fp` points to.

The following C code is correct:

```
char ch;
while ((ch = getchar(fp)) != NULL)
   putchar(ch);
```

The following C code is incorrect (why?):

```
enum Gender {FEMALE, MALE};
Gender g;
if (getchar(fp) == 'F')  g = FEMALE;
else if (getchar(fp) == 'M')  g = MALE;
else …
```

The meaning of an expression/command depends on the declarations of any identifiers used by the expression/command.

- A binding is a fixed association between an identifier and an entity (such as a value, variable, or procedure).
- An environment (or name-space) is a set of bindings.
- Each declaration produces some bindings, which are added to the surrounding environment.
- Each expression/command is interpreted in a particular environment. Every identifier used in the expression/command must have a binding in that environment.

- The scope of a declaration (or of a binding) is the portion of the program text over which it has effect.
- In some early PLs (such as Cobol), the scope of every declaration was the whole program.
- In modern PLs, the scope of each declaration is controlled by the program's block structure

- C program outline, showing environments:

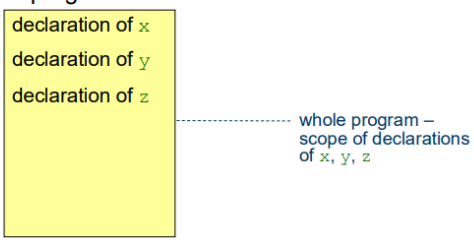
```
extern int z;
extern const float c = 3.0e6;

void f () {
   …
}

void g (float x) {
   char c;
   int i;
   …
}
```

{ c → the FLOAT value 3.0×10^6,
f → a VOID→VOID function,
g → a FLOAT→VOID function,
z → an INT global variable }

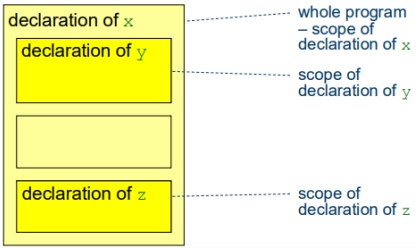
{ c → a CHAR local variable,
f → a VOID→VOID function,
g → a FLOAT→VOID function,
i → an INT local variable,
x → a FLOAT local variable,
z → an INT global variable }

▪ A block is a program construct that delimits the scope of any declarations within it.
▪ Each PL has its own forms of blocks: – C: block commands ("{ ... }"), function bodies, compilation-units. – Java: block commands ("{ ... }"), method bodies, class declarations. – Haskell: block expressions ("let ... in ..."), function bodies, modules.
▪ A PL's block structure is the way in which blocks are arranged in the program text.

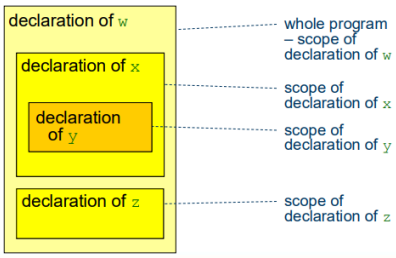
Some PLs (such as Cobol) have **monolithic block structure**: the whole program is a single block. The scope of every declaration is the whole program.



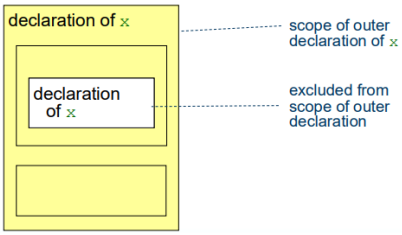
Some PLs (such as Fortran) have **flat block structure**: the program is partitioned into blocks, but these blocks may not contain inner blocks.



▪ Modern PLs have **nested block structure**: blocks may be nested freely within other blocks.



With nested block structure, the scope of a declaration excludes any inner block where the same identifier is declared:



▪ C has flat block structure for functions, but nested block structure for variables:

```
extern int x1, x2;
void main () {
    int m1; float m2;
    … f(); …
}
void f () {
    float f1;
    while (…) {
        int f2;
        …
    }
    …
}
```

▪ A binding occurrence of identifier I is an occurrence of I where I is bound to some entity e.
▪ A bound occurrence of identifier I is an occurrence of I where use is made of the entity e to which I is bound.
▪ If the PL is statically scoped (see later), every bound occurrence of I should correspond to exactly one binding occurrence of I.

▪ C program outline, showing binding occurrences and bound occurrences:

```
extern int z;
extern const float c = 3.0e6;
void f () {
    … c … z …
}
void g (float x) {
    int i;
    char c;
    … c … i … x … z …
}
```

▪ A PL is statically scoped if the body of a procedure is executed in the environment of the procedure definition. – Then we can decide at compile-time which binding occurrence of an identifier corresponds to a given bound occurrence.
▪ A PL is dynamically scoped if the body of a procedure is executed in the environment of the procedure call site. – Then we cannot decide until run-time which binding occurrence of an identifier corresponds to a given bound occurrence, since the environment may vary from one call site to another.

▪ Dynamic scoping fits badly with static typing. – In the previous slide, what if the two declarations of s had different types?
▪ Nearly all PLs (including Pascal, Ada, C, Java, Haskell) are statically scoped.
▪ Only a few PLs (such as Smalltalk and Lisp) are dynamically scoped.

Program in a *statically* scoped PL (C):

```
const int s = 2;
int f (int x) {
    return x * s;
}
void g (int y) {
    print (f (y));
}
void h (int z) {
    const int s = 3;
    print (f (z));
}
```

The value of s here is always 2.
prints 2 × y
prints 2 × z

Similar program in a hypothetical *dynamically* scoped PL:

```
const int s = 2;
int f (int x) {
    return x * s;
}
void g (int y) {
    print (f (y));
}
void h (int z) {
    const int s = 3;
    print (f (z));
}
```

The value of s here depends on the call site.
prints 2 × y
prints 3 × z

▪ A declaration is a program construct that will be elaborated to produce binding(s). – A declaration may also have side effects (such as creating a variable).
▪ A definition is a declaration whose only effect is to produce binding(s). – A definition has no side effects.
▪ Simple declarations:
– A type declaration binds an identifier to an existing or new type.
– A constant definition binds an identifier to a value (possibly computed).
– A variable declaration binds an identifier to a newlycreated variable.
– A procedure definition binds an identifier to a procedure. – And similarly for other entities (depending on the PL).

▪ Compound declarations:
– A sequential declaration combines several subdeclarations, such that the later sub-declarations can use bindings produced by the earlier sub-declarations.
– A recursive declaration is one that uses the bindings it produces itself.
▪ A recursive declaration is one that uses the bindings it produces itself.
▪ In almost all PLs, recursion is restricted to: – type (or class) declarations – procedure (or method) definitions.

▪ Java classes may be recursive.
▪ Java method definitions may be recursive.

```
class IntList {
    int head;
    IntList tail;
    static int length (IntList list) {
        if (list == null)
            return 0;
        else
            return 1 + length (list.tail);
    }
}
```

▪ C struct type declarations may be recursive (but only via pointers).
▪ C function definitions may be recursive.

```
struct IntList {
    int head;
    struct IntList * tail;
}
int length (IntList * list) {
    if (list == NULL)
        return 0;
    else
        return 1 + length(list->tail);
}
```

Week 9: Abstraction

In programming, abstraction means the distinction between what a program-unit does and how it does it.
▪ This supports a separation of concerns between the implementor (who codes the program-unit) and the application programmer (who uses it).
▪ Program-units include: – procedures (here) – packages, abstract data types, classes (see §12) – generic packages and classes (see §13).

▪ A proper procedure (or just procedure) embodies a command to be executed. – A procedure call is a command. – It causes the procedure's body to be executed. – Its net effect is to update some variables. ▪ A function procedure (or just function) embodies an expression to be evaluated. – A function call is an expression. – It causes the function's body to be evaluated. – Its net effect is to yield a value (the function's result)

▪ Imperative PLs usually support both proper procedures and function procedures. – In Pascal and Ada, proper procedures and function procedures are syntactically distinct. – In C and Java, the only distinction is that a proper procedure's result type is VOID.
▪ Functional PLs support function procedures only.
▪ OO PLs also support procedures, in the guise of methods: – Static methods are procedures exported by classes. – Instance methods are procedures attached to objects.

▪ In most imperative and OO PLs, the function's body is syntactically a block-command. This is executed until a return determines the function's result.
▪ Pros and cons: + The full expressive power of commands is available to define the function. – This is a roundabout way to compute a result. – A return might never be executed. – Side effects are possible.

▪ In functional PLs, the function's body is syntactically an expression. This is evaluated to yield the function's result.
▪ Pros and cons of this design: + This design is simple and natural. – Expressive power is limited, unless the PL has conditional expressions, iterative expressions, etc.
▪ An argument is a value (or other entity) that is passed to a procedure.
▪ An actual parameter is an expression that yields an argument.
▪ A formal parameter is an identifier through which a procedure can access an argument.
▪ What may be passed as arguments? – values (in all PLs) – variables, or pointers to variables (in many PLs) – procedures, or pointers to procedures (in some PLs).

▪ A parameter mechanism is a means by which a formal parameter provides access to the corresponding argument.
▪ Different PLs support a bewildering variety of parameter mechanisms: value, result, valueresult, constant, variable, procedural, and functional parameters.
▪ These can all be understood in terms of two underlying concepts: – copy parameter mechanisms – reference parameter mechanisms
▪ With a copy parameter mechanism, a value is copied into and/or out of a procedure: – The formal parameter FP is bound to a local variable of the procedure. – A value is copied into that local variable on calling the procedure; or copied out of that local variable (to an argument variable) on return.
▪ Principal copy parameter mechanisms: – copy-in parameter – copy-out parameter.

▪ Copy-in parameter (or value parameter): – The argument is a value. – On call, a local variable is created and initialized with the argument value. – On return, that local variable is destroyed.
▪ Copy-out parameter (or result parameter): – The argument is a variable. – On call, a local variable is created but not initialized. – On return, that local variable's final value is assigned to the argument variable, then the local variable is destroyed.
Example of Copy in
▪ C function:

```
void print (Date date) {
    printf ("%d-%d-%d",
        date.y, date.m,
        date.d);
}
```

Local variable `date` is initialized to the argument value.

▪ Call:

```
Date today = {2008, 11, 5};
print (today);
```

The argument value is the triple (2008, 11, 5).

▪ With a reference parameter mechanism, the formal parameter is a reference to the argument. – The formal parameter FP is bound to a reference to the argument. – Every access to FP is an indirect access to the argument.
▪ Principal reference parameter mechanisms: – constant parameters – variable parameters – procedural parameters.
Constant parameter: the argument is a value.
▪ Variable parameter: the argument is a variable.
▪ Procedural parameter: the argument is a procedure.
Java method:

```
void print (Date date) {
    out.print (date.y
        & "-" & date.m
        & "-" & date.d);
    date.y++;
}
```

`date` is a reference to the argument object.

Call:

```
Date today = new Date (2008, 11, 5);
print (today);
```

The argument is the object to which `today` refers.

C supports only the copy-in parameter mechanism.
▪ However, we can achieve the effect of a variable parameter by passing a pointer: – If a C function has a parameter of type T*, the corresponding argument must be a pointer to a variable of type T. The function can then indirectly inspect or update that variable.

▪ Java supports the copy-in parameter mechanism for primitive types (such as int, float, ...). ▪ In effect, Java supports the reference parameter mechanism for object types (such as T[], String, List, ...).

▪ A package (or module) is a named group of components declared for a common purpose.
▪ These components may be types, constants, variables, procedures, inner packages, etc. (depending on the PL).
▪ The meaning of a package is the set of bindings exported by the package – often called the package's application program interface (API).

Outline of a module (dictionary.py)

```
words = [...]

def contains (word):
    global words
    return (word in words)

def add (word):
    global words
    if word not in words:
        words += [word]
```

This module's API:
`words` → a list of words,
`contains` → a function that tests whether a word is in the list,
`add` → a procedure that adds a word to the list }

Some of the components of a program-unit (package/class) may be private. This is called encapsulation.
▪ Levels of privacy: – A component is private if it is visible only inside the program-unit. – A component is protected if it is visible only inside the program-unit and certain "friendly" program-units. – A component is public if it is visible to application code outside the program-unit.
▪ A program-unit's API consists of its public bindings only.

Most PLs (such as Ada, Java, Haskell) allow individual components of a program-unit to be specified as private/protected/public.
▪ Python has a convention that components whose names start with "_" are private, whilst those whose names start with letters are public. – This convention is not enforced by the Python compiler

Outline of a module (dictionary.py)

```
_words = [...]

def contains (word):
    global _words
    return (word in _words)

def add (word):
    global _words
    if word not in _words:
        _words += [word]
```

This module's API:
`contains` → a function that tests whether a word is in the list,
`add` → a procedure that adds a word to the list }

In Java, the components of a package are classes and inner packages.

Package components are added incrementally.

Outline of a class declaration within a package:

```
package sprockets;
import widgets.*;
public class C {
    ...
}
```

`package sprockets;` declares that class `C` is a component of package `sprockets`
`import widgets.*;` declares that class `C` uses public components of package `widgets`

An object is a tagged tuple of variable components (instance variables), equipped with operations that access these instance variables.
- A constructor is an operation that initializes a newly created object.
- An instance method is an operation that inspects and/or updates an existing object of class C. That object (known as the receiver object) is determined by the method call.
- A class is a set of similar objects. All objects of a given class C have similar instance variables, and are equipped with the same operations.

- If C' is a subclass of C (or C is a superclass of C'), then C' is a set of objects that are similar to one another but richer than the objects of class C: – An object of class C' has all the instance variables of an object of class C, but may have extra instance variables. – An object of class C' is equipped with all the instance methods of class C, but may override some of them, and may be equipped with extra instance methods.

By default, a subclass inherits (shares) its superclass's instance methods.
- Alternatively, a subclass may override some of its superclass's instance methods, by providing more specialized versions of these methods.

Suppose: – the Animal class defines a method named move – the Mammal and Flier classes both override that method.
- Which method does the Bat class inherit? Bat b = ...; b.move(...);
- Possible answers: – Make it call the Mammal method. – Force the programmer to choose. – Make this method call illegal.

Class declaration:

```
class Dict {
    private int size;
    private String[] words;
    public Dict (int capacity)
    { ... }
    public void add (String w)
    {   if (! this.contains(w))
            this.words[this.size++] = w; }
    public boolean contains (String w)
    { ... }
}
Shape s = new Shape();
Circle c = new Circle(10.0);
s.move(12.0, 5.0);
c.move(3.0, 4.0);
... c.diameter() ...
s.draw();   ---- draws a point at (12, 5)
c.draw();   ---- draws a circle centered at (3, 4)
s = c;
s.draw();   ---- ditto! (dynamic dispatch)
```

Possible application code:

```
Dict mainDict = new Dict (10000);
Dict userDict = new Dict (1000);
...
if (! mainDict.contains (currentWord)
  && ! userDict.contains (currentWord))
   userDict.add (currentWord);
```

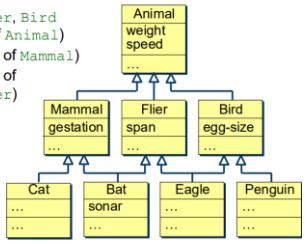
Illegal application code:

```
userDict.size = 0;          // illegal
out.print (userDict.words[0]);
```

- An OO PL supports single inheritance if each class has at most one superclass.
- Single inheritance gives rise to a hierarchy of classes.
- Single inheritance is supported by most OO PLs, including Java
- Multiple inheritance allows each class to have any number of superclasses.
- Multiple inheritance is supported by C++.
- Nevertheless, multiple inheritance gives rise to both conceptual and implementation problems.

Declared classes:
- Animal
- Mammal, Flier, Bird (subclasses of Animal)
- Cat (subclass of Mammal)
- Bat (subclass of Mammal, Flier)
- etc.

Class relationships:



Each instance method of a class C is inherited by the subclass C', unless it is overridden by C'.
- The overriding method in class C' has the same name and type as the original method in class C.
- Most OO PLs allow the programmer to specify whether an instance method is virtual (may be overridden) or not: – In C++, an instance method specified as virtual may be overridden. – In Java, an instance method specified as final may not be overridden.

- In every OO PL, a variable of type C may refer to an object of any subclass of C.
- If method M is virtual, then the method call "O.M(...)" entails dynamic dispatch: – The compiler infers the type of O, say class C. It then checks that class C is equipped with an instance method named M, of the appropriate type. – At run-time, however, it might turn out that the receiver object is of class C', a subclass of C. The receiver object's tag is used to determine its actual class, and hence determine which of the methods named M is to be called.

A program-unit is generic if it is parameterized with respect to a type on which it depends.
- Many reusable program-units (e.g., stack, queue, list, and set ADTs/classes) are naturally generic.
- Generic program-units include: – generic packages (not covered here) – generic classes – generic procedures.

- In Java, a **generic class** *GC* is parameterized with respect to a type *T* on which it depends:

```
class GC <T> {
    ... T ... T ...
}
```

- The generic class must be **instantiated**, by substituting a type argument *A* for the type parameter *T*:

```
GC<A>
```

- This instantiation generates an ordinary class.

The following instantiation generates a class that encapsulates lists with `Date` elements:

```
List<Date> holidays;
holidays = new List<Date>();
holidays.add (new Date(2009, 1, 1));
```

In an instantiation, the type argument must be a class, *not a primitive type*:

```
List<int> primes;   // illegal
```

Consider a class that encapsulates lists with elements of type `T`. This class can be made generic with respect to type `T`.

Generic class declaration:

```
class List <T> {                    // type parameter
    // A List<T> object is a list of elements of type
    // T, where the number of elements ≤ cap.
    private static final cap = 100;
    private int size;
    private T[] elems;
```

Java also supports generic interfaces.

From `java.lang`:

```
interface Comparable <T> {
    public int compareTo (T that);
}
```

If class `C` is declared as implementing `Comparable<C>`, `C` *must* be equipped with a `compareTo` method that compares objects of type `C`.

Generic class declaration *(continued)*:

```
    public List () {
    // Construct an empty list.
        size = 0;
        elems = (T[]) new Object[cap];
    }
    public void add (T elem) {
    // Add elem to the end of this list.
        elems[size++] = elem;
    }
}
```

Consider a generic class *GC‹T›* that requires *T* to be equipped with some specific methods.

T may be specified as **bounded** by a class *C*:

```
class GC <T extends C> { ... }
```

- *T* is known to be equipped with all the methods of *C*.
- The type argument must be a subclass of *C*.

Alternatively, *T* may be specified as **bounded** by an interface *I*:

```
class GC <T extends I> { ... }
```

- *T* is known to be equipped with all the methods of *I*.
- The type argument must be a class that implements *I*.

Recall a generic class *GC‹T›* that does *not* require *T* to be equipped with any specific methods. As we have seen, it is enough just to name *T*:

```
class GC <T> { ... }
```

This is actually an abbreviation for:

```
class GC <T extends Object> { ... }
```

- *T* is known to be equipped with all the `Object` methods, such as `equals`.
- The type argument must be a subclass of `Object`, i.e., any class.

Class `String` implements `Comparable<String>`. So the following generates a class that encapsulates priority queues with `String` elements:

```
PQueue<String> pq;
pq = new PQueue<String>();
pq.add("beta");
pq.add("alpha");
out.print(pq.first());
```

Consider a class `Pqueue<T>` that encapsulates priority queues with elements of type `T`. It is required that `T` is equipped with a `compareTo` method.

Generic class declaration:

```
class PQueue <T extends Comparable<T>> {
    private static final cap = 20;
    private int size;
    private T[] elems;
    public PQueue () {
        size = 0;
        elems = (T[]) new Object[cap];
    }
```

Generic class declaration *(continued)*:

```
    public void add (T elem) {
    // Add elem to this priority queue.
        if (elem.compareTo(elems[0])<0) ...
        ...
    }
    public T first () {
    // Return the first element of this priority queue.
        return elems[0];
    }
}
```

The key concept being illustrated is that a priority queue automatically orders its elements based on their natural ordering (defined by the Comparable interface). When strings are added to the priority queue, they are automatically sorted lexicographically (alphabetically).

So even though "beta" was added first and "alpha" was added second, when `pq.first()` is called, it will return "alpha" because "alpha" comes before "beta" alphabetically. This demonstrates how the priority queue maintains elements in their priority order rather than insertion order.

This example highlights how generics and the Comparable interface work together to create type-safe collections with automatic ordering capabilities.

Suppose that class `Date` implements `Comparable<Date>`. Then the following generates a class that encapsulates priority queues with `Date` elements:

```
PQueue<Date> holidays;
```

But `PQueue` *cannot* be instantiated with a class `C` that does not implement `Comparable<C>`:

```
PQueue<Button> buttons;
```

illegal

A method that chooses between two arguments of type `T` can be made generic w.r.t. type `T`:

```
public static <T>
       T either (boolean b, T y, T z) {
   return (b ? y : z);
}
```

Calls:

```
... either (isletter(c), c, '*')
... either (m > n, m, n)
```

implicitly substituting type `Character` for `T`

implicitly substituting type `Integer` for `T`

Week 10: Runtime organisation and native code generation

Assumptions: – The PL is statically-typed. – The compiler decides the size and layout of each type. – All variables of the same type have the same size.

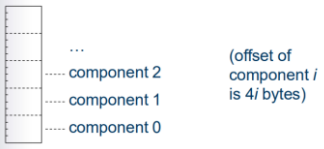
Representation of each primitive type may be language-defined or implementation-defined.
- Typically 8, 16, 32, or 64 bits.
- BOOL: 00000000 or 00000001.
- CHAR: 00000000, ..., 11111111 (if 8-bit)
- INT: 16-bit or 32-bit or 64-bit two's complement.
- FLOAT: 32-bit or 64-bit IEEE floating-point.

Example: representation of C arrays

C type definition:

```
typedef int[] Arr;
```
Assume size of INT is 4 bytes

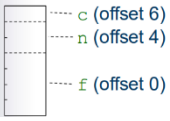
Possible representation of values of type `Arr`:



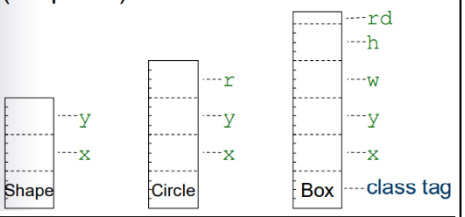
C struct type definition:

```
struct Str {
   float f;
   int n;
   char c;
};
```
Assume sizes: FLOAT 4 bytes INT 2 bytes CHAR 1 byte

Possible representation of structs of type `Str`:



Representing objects of above classes (simplified):



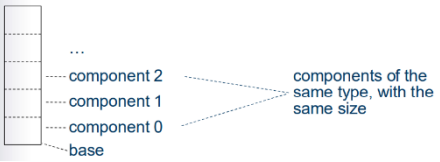
If the PL has no deallocator, the heap manager must be able to find and destroy any unreachable heap variables automatically. This is done by a garbage collector.
- A garbage collector must visit all reachable heap variables. This is inevitably time-consuming.
- A mark-sweep garbage collector is the simplest. It first marks all reachable heap variables, then deallocates all unmarked heap variables.

Mark-sweep algorithm: To mark all heap variables reachable from pointer p:
1 Let heap variable v be the referent of p.
2 If v is unmarked: 2.1 Mark v. 2.2 For each pointer q in v: 2.2.1 Mark all heap variables reachable from q.

To collect garbage: 1 For each pointer p in a global/local variable: 1.1 Mark all heap variables reachable from p.
2 For each heap variable v: 2.1 If v is unmarked, destroy v. 2.2 Else, if v is marked, unmark v.

Represent an array by juxtaposing its components.

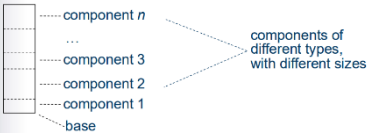
Representation of arrays of type {0, 1, 2, …} → *T*:



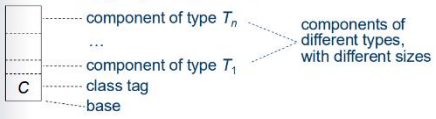
Representation of cartesian products (1)

Represent a tuple (or record or struct) by juxtaposing its components.

Representation of tuples of type $T_1 \times T_2 \times \ldots \times T_n$:



- Consider class *C* with components (instance variables) of types $T_1, \ldots, T_n$.
- Representing objects of class *C*:



- The compiler knows the offset of each component (relative to the object's base address).

The offset of array component *i* (relative to the array's base address) is linearly related to *i*:

offset of component *i* = (size of type *T*) × *i*

known to the compiler — unknown

Since *i* is unknown, the offset of component *i* must be calculated at run-time.

The compiler knows the offset of each tuple component (relative to the tuple's base address):

offset of component 1 = 0

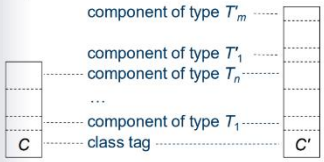
offset of component 2 = size of type T_1

offset of component 3 = size of type T_1 + size of type T_2

...

offset of component *n* = size of type T_1 + size of type T_2 + ... + size of type T_{n-1}

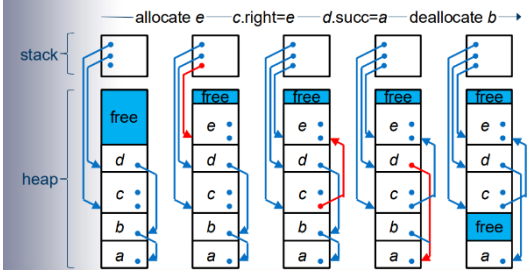
- Now consider class *C'* (a subclass of *C*) with *additional* instance variables of types $T'_1, \ldots, T'_m$.
- Representation of objects of classes *C* and *C'*:



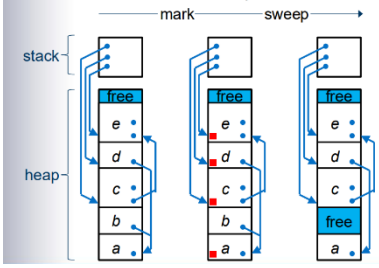
- Each component has a known offset in objects of a given class *C and all subclasses of C*.

Heap variables occupy a storage region called the heap. At any given time, the heap contains all currently-live heap variables, interspersed with free space. – When a new heap variable is to be created, some free space is allocated to it. – When a heap variable is to be destroyed, its allocated space reverts to being free.
The heap manager (part of the PL's run-time system) keeps track of free space within the heap – usually by means of a free-list: a linked list of free areas of various sizes.
- The heap manager provides: – a routine to create a heap variable (called by the PL's allocator) – a routine to destroy a heap variable (called by the PL's deallocator, if any).

- Effect of allocations and deallocations:



- Effect of mark and sweep:



Time complexity of mark-sweep garbage collection is $O(n_r + n_h)$
– n_r = number of reachable heap variables
– n_h = total number of heap variables.
- The heap tends to become fragmented: – There might be many small free areas, but none big enough to allocate a new large heap variable. – Partial solution: coalesce adjacent free areas in the heap. – Better solution: use a copying or generational garbage collector.

A copying garbage collector maintains two separate heap spaces: – Initially, space 1 contains all heap variables; space 2 is spare. – Whenever the garbage collector reaches a marked heap variable v, it copies v from space 1 to space 2. – At the end of garbage collection, spaces 1 and 2 are swapped.
- Pros and cons: + Heap variables can be consolidated when copied into space 2.

– All pointers to a copied heap variable must be found and redirected from space 1 to space 2.

A generational garbage collector maintains two (or more) separate heap spaces: – One space (the old generation) contains only long-lived heap variables; the other space (the young generation) contains shorter-lived heap variables. – The old generation is garbage-collected infrequently (since long-lived heap variables are rarely deallocated). – The young generation is garbage-collected frequently (since short-lived heap variables are often deallocated). – Heap variables that live long enough may be promoted from the young generation to the old generation.
- Pro: + Garbage collection is more focussed.

▪ Code selection is difficult because: – CISC machines have very complicated instructions, with multiple addressing modes. – even RISC machines have some fairly complicated instructions.
▪ Register allocation is an issue: – Registers should be used as much as possible. – RISC machines typically have only general-purpose registers. – CISC machines typically have a variety of special-purpose registers (e.g., int registers, float registers, address registers).
Aim to use registers as much as possible for local variables and intermediate results of expressions.
▪ Problem: The number of registers is limited – especially when some are dedicated (e.g., fp, sp).
▪ Opportunity: Different variables can be allocated to the same register if they are live at different times.
▪ Here, a variable is deemed to be live only if it might be inspected later.

A control-flow graph is a directed graph in which: – each vertex is a BB – each edge is a jump from the end of one BB to the start of another BB – one vertex is designated as the entry point – one vertex is designated as the exit point.

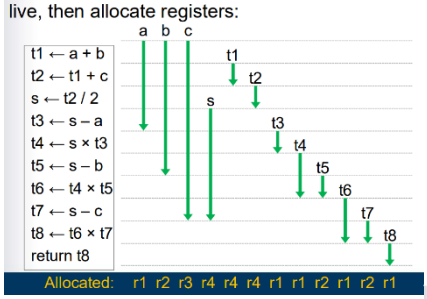
A **basic-block** (BB) is a straight-line sequence of instructions using **three address code**:
- no jumps except at the end of a BB
- no jumps to anywhere except the start of a BB.

Within a BB, break up complicated expressions using temporary variables, such that each assignment instruction contains at most one operator. E.g.:

```
a = (b+c)*(d-e);
```
→
```
t1 ← b + c
t2 ← d – e
a ← t1 × t2
```

Note: Basic-blocks are unrelated to block structure.

Within the BB, determine where each variable is live, then allocate registers:



Define the following sets for each BB *b* in a control-flow graph:
- *in*[*b*] is the set of variables live at the start of *b*
- *out*[*b*] is the set of variables live at the end of *b*
- *use*[*b*] is the set of variables *v* such that *b* inspects *v* (before any update to *v*)
- *def*[*b*] is the set of variables *v* such that *b* updates *v* (before any inspection of *v*)

Data flow equations for liveness analysis:

in[*b*] = *use*[*b*] U (*out*[*b*] – *def*[*b*])

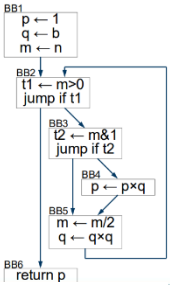
out[*b*] = *in*[*b'*] U *in*[*b''*] U …
(where *b'*, *b''*, … are the successors of *b* in the flow graph)

- The liveness analysis algorithm follows directly from the data flow equations:

To compute in[*b*] and out[*b*] for all BBs in a control-flow graph:
1. For each *b*:
   1.1. Set *in*[*b*] = *out*[*b*] = { }.
2. Repeat until the sets *in*[*b*] and *out*[*b*] stop changing:
   2.1. For each *b*:
      2.1.1. Set *in*[*b*] = *use*[*b*] U (*out*[*b*] – *def*[*b*]).
      2.1.2. Set *out*[*b*] = *in*[*b'*] U *in*[*b''*] U …
      (where *b'*, *b''*, … are the successors of *b*).

- The live variables in block b are in[b] U def[b]

The *use* and *def* sets for each basic block in the control flow graph:



use[BB1] = { b, n }	def[BB1] = { p, q, m }
use[BB2] = { m }	def[BB2] = { t1 }
use[BB3] = { m }	def[BB3] = { t2 }
use[BB4] = { p, q }	def[BB4] = { }
use[BB5] = { m, q }	def[BB5] = { }
use[BB6] = { p }	def[BB6] = { }

The equations for the *in* and *out* sets:

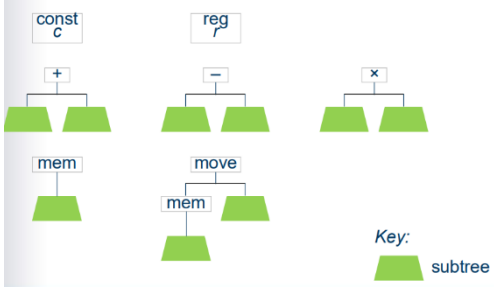
in[BB1] = { b, n } U (out[BB1] – { p, q, m })	out[BB1] = in[BB2]
in[BB2] = { m } U (out[BB2] – { t1 })	out[BB2] = in[BB6] U in[BB3]
in[BB3] = { m } U (out[BB3] – { t2 })	out[BB3] = in[BB5] U in[BB4]
in[BB4] = { p, q } U out[BB4]	out[BB4] = in[BB5]
in[BB5] = { m, q } U out[BB5]	out[BB5] = in[BB2]
in[BB6] = { p } U out[BB6]	out[BB6] = { }

Iterate the equations, starting with in[b] = { } for each b:

in[BB1]	in[BB2]	in[BB3]	in[BB4]	in[BB5]	in[BB6]
{ }	{ }	{ }	{ }	{ }	{ }
{ b, n }	{ m }	{ m }	{ p, q }	{ m, q }	{ p }
{ b, n }	{ m, p }	{ m, p, q }	{ m, p, q }	{ m, p, q }	{ p }
{ b, n }	{ m, p, q }	{ m, p, q }	{ m, p, q }	{ m, p, q }	{ p }
{ b, n }	{ m, p, q }	{ m, p, q }	{ m, p, q }	{ m, p, q }	{ p }

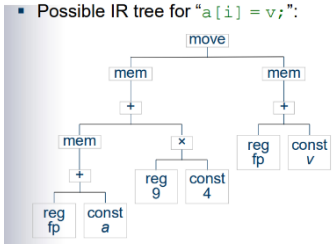
The process terminates because each step can only increase the sets, and they can't increase beyond the set of all variables.

IR trees:



Native code generation is simplified by using a low-level intermediate representation (IR) of the source program.
▪ The IR should be capable of: – representing the semantics of the source code – representing the semantics of target-machine instructions.
▪ The IR should ideally be independent of the target machine.

- Possible IR tree for "`a[i] = v;`":



Consider the C function:

```
int tri (int a, int b, int c) {
  int s = (a+b+c)/2;
  return s*(s-a)*(s-b)*(s-c);
}
```

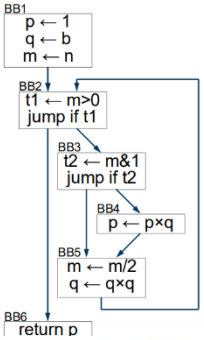
This function's body is a single BB:

```
t1 ← a + b
t2 ← t1 + c
s ← t2 / 2
t3 ← s – a
t4 ← s × t3
t5 ← s – b
t6 ← t4 × t5
t7 ← s – c
t8 ← t6 × t7
return t8
```

Consider the C function:

```
int pow (int b, int n) {
  int p = 1, q = b, m = n;
  while (m > 0) {
    if (m & 1) p = p*q;
    m = m/2;
    q = q*q;
  }
  return p;
```

This function's body is a control-flow graph:



Where is each variable live?
- b and n are live only in BB1
- p is live everywhere
- m and q are live everywhere except in BB6
- t1 is live only in BB2
- t2 is live only in BB3.

Substituting for the *out* sets gives equations for the *in* sets alone:

in[BB1] = { b, n } U (in[BB2] – { p, q, m })

in[BB2] = { m } U ((in[BB6] U in[BB3]) – { t1 })

in[BB3] = { m } U ((in[BB5] U in[BB4]) – { t2 })

in[BB4] = { p, q } U in[BB5]

in[BB5] = { m, q } U in[BB2]

in[BB6] = { p }

- Consider the C assignment "`a[i] = v;`".
- Assume that:
  - `a` has type `int*` and `v` has type `int`
  - each `int` occupies 4 bytes
  - variable `a` is located at offset *a* within the topmost activation frame (that location contains the base address of `a`)
  - variable `i` is located in register r9
  - variable `v` is located at offset *v* within the topmost activation frame.
- Address of `a[i]` is:
  (base address of `a`) + 4×(content of `i`).

Jouette is a hypothetical RISC machine – invented by Andrew Appel for his Modern Compiler Implementation books.

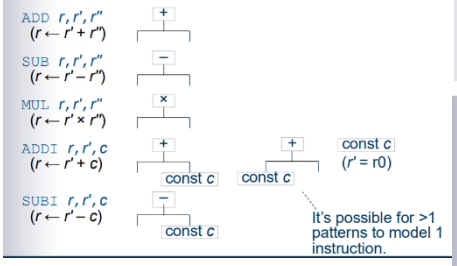
- Jouette architecture: – general-purpose registers r0, r1, r2, ..., r31
  – r0 always contains zero.

Jouette instruction set:

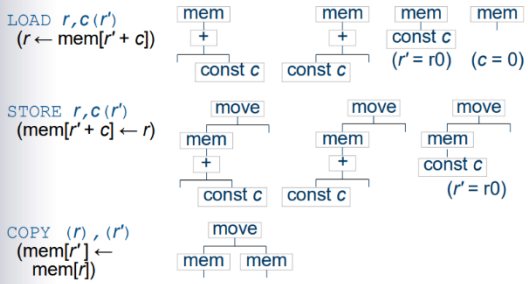
Mnemonic	*Behaviour*
`ADD r,r',r''`	$r \leftarrow r' + r''$
`SUB r,r',r''`	$r \leftarrow r' - r''$
`MUL r,r',r''`	$r \leftarrow r' \times r''$
`ADDI r,r',c`	$r \leftarrow r' + c$
`SUBI r,r',c`	$r \leftarrow r' - c$
`LOAD r,c(r')`	$r \leftarrow \text{mem}[r' + c]$
`STORE r,c(r')`	$\text{mem}[r' + c] \leftarrow r$
`COPY (r),(r')`	$\text{mem}[r'] \leftarrow \text{mem}[r]$

r, *r'*, *r''* are registers; *c* is a constant

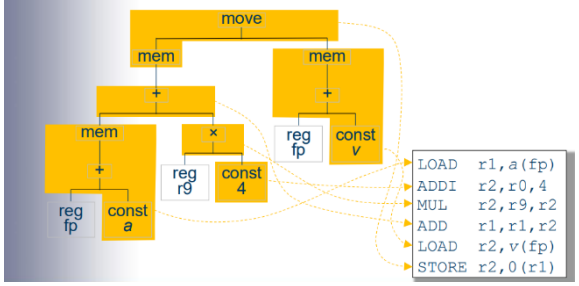
- Jouette arithmetic instructions:



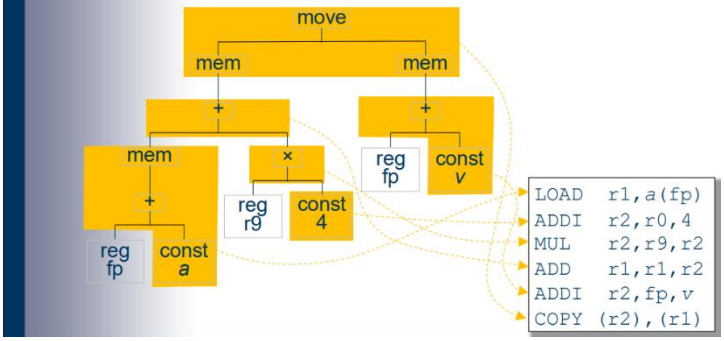
- Jouette load/store instructions:



- One way to cover the IR for "`a[i] = v;`":



- Different way to cover the IR for "`a[i] = v;`":



- Maximal-munch code selection algorithm:

  To cover the IR tree *t* using instruction patterns *ps*:

  1. Find the largest pattern *p* in *ps* that covers the top of *t*.
  2. For each uncovered subtree *s* of *t* (from left to right):
     2.1. Cover *s* using *ps*.
     2.2. Emit the instruction corresponding to *p*.

- The time complexity is O(size of *t*).
- The emitted code is optimal in the sense that:
  - no two adjacent patterns could be replaced by a single pattern
  - the number of instructions is minimal.

versity of Glasgow

Code selection: method

- Translate the source code or AST into an IR tree.
- "Cover" the tree with IR instruction patterns.
- Emit code corresponding to these instruction patterns
  - performing register allocation as you go.