Imperative Programming

• Most widely used paradigm
• Oldest paradigm
• Largest number of languages
  – E.g. C, FORTRAN, Java/C++ (without the objects)
• Features closely related to machine architecture
  – Based on the von Neumann stored program model
• We will skip a number of topics in this chapter as you should already be familiar with them from previous courses
Imperative Programming

- Language Structure
  - Declarations
    - Associate variables with memory locations
  - Expressions
    - Evaluated in the current environment
  - Commands
    - Execution and flow of control
    - Similar to the machine instructions
      - Assignment
        - Dynamically updates data store
      - Conditionals
        - Statements executed or skipped
      - Branching
        - Alters the flow of control
        - Supports looping
  - Basis for defining an effective programming language

Turing Completeness

- A programming language is said to be Turing Complete if it contains
  - Integer variables, values and operations
  - Assignments, branching, sequencing, and conditionals
- Other features make languages easier to program for complex applications
  - Loops
  - Procedures
  - Object Orientation
- But they have the same power as any other programming language
  - i.e. what is possible to compute in one is possible in the other
  - Jay is Turing Complete
Other Features

• Typical Turing complete language today supports:
  – Data types for reals, chars, strings, booleans
  – For, while, switch
  – Arrays
  – Records
  – I/O commands
  – Pointers
  – Procedures and functions

Imperative Programming Design Principles

• Structured Programming
  – The structure of the program text should be the guide to understanding what it does.
    • Can be more efficient
    • Improved readability
    • Easier to tune and modify
  – Efficiency
    • A language must allow an underlying assignment-oriented machine to be used directly and efficiently.
    • Driving factor in the implementation of many imperative languages
## Types in Typical Imperative Languages

- Emphasis on data structures with assignable components
  - Based on values that can be manipulated directly by underlying machine
- Size and layout fixed at compile time
- Storage allocated and deallocated explicitly
- Strongly typed
- Storage efficiency is the key

## Naming and Variables

- Skipping in Text
  - Reserved words
  - Variables unique
  - Concept of scope, scoping rules
Types, Values and Expressions in Jay

• Let’s look at defining types, values, and expressions in Jay

• Jay supports only two types
  – Boolean and Integer
    • Boolean
      – Values: {true, false}
      – Operations: &&, ||, !
    • Integer
      – Values: {..., -2, -1, 0, 1, 2, .....}
      – Operations
        » Arithmetic: +, -, *, /
        » Relational: ==, !=, <=, <, >=, >

Expressions in Jay

• Expression has the forms
  – Value
  – Variable
  – Result of a binary operation
  – Result of a unary operation

• Type of an expression is defined by the types of its constituents

<table>
<thead>
<tr>
<th>Constituent</th>
<th>Expression Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Type of the value</td>
</tr>
<tr>
<td>Variable</td>
<td>Type of the variable</td>
</tr>
<tr>
<td>Arithmetic Binary operator</td>
<td>int</td>
</tr>
<tr>
<td>Relational Binary operator</td>
<td>boolean</td>
</tr>
<tr>
<td>Unary operator</td>
<td>boolean</td>
</tr>
</tbody>
</table>
Formally Defining Types

• Back to the typemap format from chapter 3:

\[ tm = \{ <v_1, t_1>, <v_2, t_2>, \ldots, <v_n, t_n> \} \]

- \( v_i \) denotes a Variable
- \( t_i \) denotes its declared type
- \( t_i \in T \) (set of all types supported in the language)

Types of Expressions

• Formally defining expression types in Jay:

\[
\text{typeOf (Expression } e, \text{TypeMap } tm) = \begin{cases} \\
\text{e.type} & \text{if } e \text{ is a Value} \\
\text{e} \in tm \subseteq tm(e) & \text{if } e \text{ is a Variable} \\
\text{e.op} \in \{\text{ArithmeticOp} \} \supseteq \text{int} & \text{if } e \text{ is a Binary} \\
\text{e.op} \in \{\text{BooleanOp} \} \supseteq \text{Boolean} & \text{if } e \text{ is a Binary} \\
\text{e.op} \in \{\text{RelationalOp} \} \supseteq \text{Boolean} & \text{if } e \text{ is a Binary} \\
\text{e.op} \in \{\text{UnaryOp} \} \supseteq \text{Boolean} & \text{if } e \text{ is a Unary} \\
\end{cases}
\]
Jay Expression Example

- Declaration: int x, y;
- Expressions:
  - x + 2*y
  - x < 2*y
  - x < 2*y && x > 0
- Typemap \( \text{tm} = \{<\text{x}, \text{int}>, <\text{y}, \text{int}>\} \)

\[
\begin{align*}
\text{typeof}(x + 2*y, \text{tm}) &= \text{int} & + & \in \text{ArithmeticOp} \\
\text{typeof}(x < 2*y, \text{tm}) &= \text{boolean} & \text{<} & \in \text{RelationalOp} \\
\text{typeof}(x < 2*y \&\& x > 0, \text{tm}) &= \text{boolean} & \& \& & \in \text{BooleanOp}
\end{align*}
\]

Validity of Expressions

- typeof function determines the type of expressions
- typeof function does NOT determine the validity of the expression
- Need to check separately
Validity of Expressions

- An expression is valid if it is:
  - int or boolean Value
  - Variable in the type map
  - Unary with a UnaryOp operator and a valid boolean operand
  - Binary with ArithmeticOp with two valid int operands
  - Binary with RelationalOp with two valid int operands
  - Binary with BooleanOp with two valid boolean operands

Formal Validity of Expressions

- More formally:

\[
V(\text{Expression } e, \text{TypeMap } tm) = \begin{cases} 
true & \text{if } e \text{ is a Value} \\
= e \in tm & \text{if } e \text{ is a Variable} \\
= V(e.\text{term1},tm) \land V(e.\text{term2},tm) \land \\
\text{typeOf}(e.\text{term1},tm) = \text{int} & \text{if } e \text{ is a Binary} \\
\text{typeOf}(e.\text{term2},tm) = \text{int} & e.\text{op} \in \{\text{ArithmeticOp}\} 
\end{cases}
\]
Validity of Expressions

\[ V(\text{Expression } e, \text{TypeMap } tm) \]
\[ = V(e.\text{term1}, tm) \land V(e.\text{term2}, tm) \land \]
\[ \text{typeOf}(e.\text{term1}, tm) = \text{int} \land \text{typeOf}(e.\text{term2}, tm) = \text{int} \land \]
\[ \text{if } e \text{ is a Binary \lor } e.\text{op} \in \{\text{RelationalOp}\} \]
\[ = V(e.\text{term1}, tm) \land V(e.\text{term2}, tm) \land \]
\[ \text{typeOf}(e.\text{term1}, tm) = \text{boolean} \land \text{typeOf}(e.\text{term2}, tm) = \text{boolean} \land \]
\[ \text{if } e \text{ is a Binary \lor } e.\text{op} \in \{\text{BooleanOp}\} \]
\[ = V(e.\text{term}, tm) \land \]
\[ \text{typeOf}(e.\text{term}, tm) = \text{boolean} \land e.\text{op} = \! \]
\[ \text{if } e \text{ is a Unary} \]

Expression Example

- Validity of \( x + 2*y \)
  - Note recursive nature of \( V \)

\[ V(x + 2*y, tm) = V(x, tm) \land V(2*y, tm) \land \]
\[ \text{typeOf}(x, tm) = \text{int} \land \text{typeOf}(2*y, tm) = \text{int} \land \]
\[ \text{since } + \in \text{ ArithmeticOp} \]
Semantics of Jay

- Semantic domains
  - Integers (I)
  - Boolean (B)
- Meaning of a Jay Expression can be defined functionally:

\[ M(\text{Expression } e, \text{ Stat } \sigma) = \begin{cases} 
  e.\text{val} & \text{if } e \text{ is a value} \\
  \sigma(e) & \text{if } e \text{ is a variable} \\
  \text{ApplyBinary}(e.\text{op}, M(e.\text{term1}, \sigma), M(e.\text{term2}, \sigma)) & \text{if } e \text{ is binary} \\
  \text{ApplyUnary}(e.\text{op}, M(e.\text{term}, \sigma)) & \text{if } e \text{ is unary} 
\end{cases} \]

- ApplyBinary: Computes a value given a binary operator and two operands
- ApplyUnary: Computes a value given a unary operator and one operand
- Uses the properties of the underlying semantic domains
ApplyBinary Function

ApplyBinary : Operator × Value × Value → Value

ApplyBinary(Operator op, Value v₁, Value v₂)

= v₁ + v₂ if op = +
= v₁ − v₂ if op = −
= v₁ × v₂ if op = *
= floor \left( \frac{v₁}{v₂} \right) \times \text{sign}(v₁ \times v₂) if op = /

ApplyBinary, Cont.

ApplyBinary(Operator op, Value v₁, Value v₂)

= v₁ < v₂ if op = <
= v₁ ≤ v₂ if op = <=
= v₁ = v₂ if op = ===
= v₁ ≠ v₂ if op = !==
= v₁ ≥ v₂ if op = >=
= v₁ > v₂ if op = >
= v₁ ∧ v₂ if op = &&
= v₁ ∨ v₂ if op = ||
Semantics Example

• Meaning of expression $x + 2y$
  – Note recursive definitions

$$\sigma = \{<x, 4>, <y, -7>\}$$

$$M(x + 2y, \sigma) = \text{ApplyBinary}(+, M(x, \sigma), M(2y, \sigma))$$

$$= \text{ApplyBinary}(+, 4, \text{ApplyBinary}(\ast, M(2, \sigma), M(y, \sigma)))$$

$$= \text{ApplyBinary}(+, 4, \text{ApplyBinary}(\ast, 2, -7))$$

$$= \text{ApplyBinary}(+, 4, -14)$$

$$= -10$$

Semantics

• ApplyBinary depends on the meaning of the semantic domain of I and B
• Sometimes it is possible to define the meanings directly without using domains, e.g. for the definitions of \&\& and || without using \& and \lor

$$\text{ApplyBinary}(\text{Operator } op, \text{Value } v_1, \text{Value } v_2)$$

$$= \text{if } v_1 \text{ then } v_2 \text{ else false } \text{ if } op = \&\&$$

$$= \text{if } v_1 \text{ then true else } v_2 \text{ if } op = ||$$

Using the Short Circuit evaluation method
Elementary Data Types

- Skipping from book
  - Nibble, byte, word, quadword, etc.
  - IEEE 754 number format
  - Unicode
  - Operator overloading, type conversion (e.g. int/float)
- See textbook for similarity in C++ and Java operators
  - Book uses ?: for conditional
    - E.g. \( x = (y>0) ? 1 : 2; \)
      - If \( y>0 \) \( x=1 \); else \( y=2; \)

Syntax and Semantics of Jay

- Recall the following concrete and abstract syntax for Jay statements

  \[
  \text{Statement} \rightarrow ; | \text{Block} | \text{Assignment} | \text{IfStatement} | \text{WhileStatement} \\
  \text{Statement} = \text{Skip} | \text{Block} | \text{Assignment} | \text{Conditional} | \text{Loop}
  \]

- We can define validity for statements:

\[
\begin{align*}
V(\text{Statement} s, \text{TypeMap} tm) &= \text{true} \\
&= s.\text{target} \in tm \land V(s.\text{source}, tm) \land tm(s.\text{target}) = \text{typeOf}(s.\text{source}, tm) \\
&\quad \text{if } s \text{ is a Skip} \\
&\quad \text{if } s \text{ is an Assignment} \\
&
\end{align*}
\]

\[
\begin{align*}
&= V(s.\text{test}, tm) \land \text{typeOf}(s.\text{test}, tm) = \text{boolean} \land V(s.\text{thenbranch}, tm) \land V(s.\text{elsebranch}, tm) \\
&\quad \text{if } s \text{ is a Conditional} \\
&
\end{align*}
\]

\[
\begin{align*}
&= V(s.\text{test}, tm) \land \text{typeOf}(s.\text{test}, tm) = \text{boolean} \land V(s.\text{body}, tm) \\
&\quad \text{if } s \text{ is a Loop} \\
&
\end{align*}
\]

\[
\begin{align*}
&= V(b_1, tm) \land V(b_2, tm) \land \ldots \land V(b_n, tm) \\
&\quad \text{if } s \text{ is a Block} = b_1, b_2, \ldots, b_n \land n \geq 0
\end{align*}
\]
Semantics of Skip

- Skip

\[ M(\text{Skip } s, \text{State } \sigma) = \sigma \]

- Skip statement can’t change the state

Semantics of Assignment

- Discussed in chapter 3

\[ M : \text{Assignment} \times \Sigma \rightarrow \Sigma \]

\[ M(\text{Assignment } a, \text{State } \sigma) = \sigma \overrightarrow{U} \{ \langle a, \text{target} , M(\text{a.source} , \sigma) \rangle \} \]

- For \( x=a+b \)

\[ \sigma = \{<a,3>,<b,1>,<x,\overline{88}>\} \]

\[ M(x=a+b, \sigma) = \sigma \overrightarrow{U} \{<x, M(a+b, \sigma)>\} \]

\[ \sigma = \{<a,3>,<b,1>,<x,4>\} \]
Semantics of Conditional

\[ M(\text{Conditional } c, \text{State } \sigma) \]
\[ = M(c.\text{thenbranch}, \sigma) \quad \text{if } M(c.\text{test}, \sigma) \text{ is true} \]
\[ = M(c.\text{elsebranch}, \sigma) \quad \text{otherwise} \]

If \((a>b)\) max\(=a;\) else max\(=b\)

\[ \sigma = \{<a,3>,<b,1>\} \]

\[ M(\text{if } (a > b) \text{max} = a; \text{else max} = b;, \sigma) \]
\[ = M(\text{max} = a;, \sigma) \quad \text{if } M(a > b, \sigma) \text{ is true} \]
\[ = M(\text{max} = b;, \sigma) \quad \text{otherwise;} \]

Conditional, continued

\[ \sigma = \{<a,3>,<b,1>\} \]

\[ M(\text{if } (a > b) \text{max} = a; \text{else max} = b;, \sigma) \]
\[ = M(\text{max} = a;, \sigma) \quad \text{since } M(a > b, \sigma) \text{ is true} \]
\[ = \sigma \bar{U} \{<\text{max},3>\} \]
\[ = \sigma\{<a,3>,<b,1>,<\text{max},3>\} \]
Semantics of Block

• Block is just a sequence of statements

\[
M(\text{Block } b, \text{State } \sigma) \\
= \sigma \quad \text{if } b = \emptyset \\
= M((\text{Block})b_{2..n}, M((\text{Statement})b_1, \sigma)) \quad \text{if } b = b_1 b_2 \ldots b_n
\]

• Example for Block b:
  
  fact = fact * i;
  i = i – 1;

Block example

• \( b_1 = \{ \text{fact} = \text{fact} \ast i; \} \)
• \( b_2 = \{ i = i – 1; \} \)
• \( M(b, \sigma) = M(b_2, M(b_1, \sigma)) \)
  
  = M(i = i - 1, M(\text{fact} = \text{fact} \ast i, \sigma))
  
  = M(i = i - 1, M(\text{fact} = \text{fact} \ast i, \{ <i,3>, <\text{fact},1> \}))
  
  = M(i = i - 1, \{ <i,3>, <\text{fact},3> \})
  
  = \{ <i,2>, <\text{fact},3> \}
Semantics of Loop

- Loop = Expression test; Statement body

\[ M(\text{Loop } l, \text{State } \sigma) \]
\[ = M(l, M(l\.body, \sigma)) \quad \text{if } M(l\.test, \sigma) \text{ is true} \]
\[ = \sigma \quad \text{otherwise} \]

- Recursive definition

Loop Example

- Initial state \( \sigma=\{<N,3>\} \)

```plaintext
fact=1;
i=N;
while (i>1) {
    fact = fact * i;
i = i -1;
}
```

After first two statements, \( \sigma = \{<\text{fact},1>,<\text{N},3>,<\text{i},3>\} \)
Loop Example

\[ \sigma = \{<\text{fact}, 1>, <\text{N}, 3>, <i, 3>\} \]
\[ M(\text{while}(i > 1) \{\ldots\}, \sigma) \]
\[ = M(\text{while}(i > 1) \{\ldots\}, M(\text{fact}=\text{fact}*i; \ i=i-1;; \ \sigma) \]
\[ = M(\text{while}(i > 1) \{\ldots\}, \{<\text{fact}, 3>, <\text{N}, 3>, <i, 2>\}) \]
\[ = M(\text{while}(i > 1) \{\ldots\}, \{<\text{fact}, 6>, <\text{N}, 3>, <i, 1>\}) \]
\[ = M(\sigma) \]
\[ = \{<\text{fact}, 6>, <\text{N}, 3>, <i, 1>\} \]

Syntax and Semantics for Real Languages

- **For Loop**
  - Concrete Syntax
    - ForStatement \( \rightarrow \) for (Assign\(_{opt}\) Expr\(_{opt}\) Assign\(_{opt}\)) Statement
  - Abstract Syntax
    - Identical to Abstract Syntax for While
  - See text for details
- **Do statements**
  - Concrete Syntax
    - DoStatement \( \rightarrow \) do Statement while (Expression)
  - Abstract Syntax
    - Identical to Abstract Syntax for While
    - Different semantics, must make sure body is execute once
Syntax and Semantics of Real Languages

• Switch statement
  – Multi-way IF statement
  – See book for details, straightforward extension
• Break/Continue statement
  – Break terminates switch or while
  – Continue terminates current iteration of while and jumps back to the loop test
  – Semantics must be specified in definitions for switch and while
    • E.g. break statement in a switch:
      – Meaning if no break is Meaning of first matching true statement and all subsequent statements
      – Meaning if break is Meaning of first matching true statement, all subsequent statements until one contains a break

Scoping

• Skipping scope, visibility, and lifetime
  – Should be familiar with this from Java
  – Static scope
    • Scope computed at compile time
  – Dynamic scope
    • Scope compute at run time
Extension to Methods

- Suppose we would like to extend Jay to include methods (functions)
- Must augment the concrete syntax:

\[
\text{Program} \rightarrow \text{package Identifier \{ Declaratiens Methods MainMethod\}} \\
\text{Methods} \rightarrow \epsilon | \text{Methods Method} \\
\text{Method} \rightarrow \text{Type Identifier} (\text{Parameters}_{\text{opt}})\{\text{Declarations Statements}\} \\
\text{Type} \rightarrow \text{int} | \text{boolean} | \text{void} \\
\text{Parameters} \rightarrow \text{Parameter} | \text{Parameters},\text{Parameter} \\
\text{Parameter} \rightarrow \text{Type Identifier} \\
\text{MainMethod} \rightarrow \text{void main(\{Declarations Statements\}}
\]

Sample Program

```java
package K {
    int h, i;
    void A(int x, int y) {
        boolean i, j;
        B(h);
        ...
    }
    void B(int w) {
        int j, k;
        i = 2*W;
        w = w+1;
        ...
    }
    void main() {
        int a, b;
        h = 5; a = 3; b = 2;
        A(a, b);
        ...
    }
}
```
Syntax for Methods

• Must modify BNF rules for statements to allow method calls as a statement or a factor within an expression:

\[
\text{Statement} \rightarrow ; | \text{Block} | \text{Assignment} | \text{IfStatement} | \text{WhileStatement} | \text{CallStatement} | \text{ReturnStatement}
\]

\[
\text{CallStatement} \rightarrow \text{Identifier} (\text{Arguments}_{\text{opt}});
\]

\[
\text{ReturnStatement} \rightarrow \text{return Expression};
\]

\[
\text{Arguments} \rightarrow \text{Expression} | \text{Arguments, Expression}
\]

\[
\text{Factor} \rightarrow \text{Identifier} | \text{Literal} | \text{Call} | (\text{Expression})
\]

\[
\text{Call} \rightarrow \text{Identifier} (\text{Arguments}_{\text{opt}})
\]

Abstract Syntax for Methods

• Underlined rules indicate changes

\[
\begin{align*}
\text{Program} &= \text{Declarations} \text{ globals; Methods body} \\
\text{Methods} &= \text{Method}^* \\
\text{Method} &= \text{Type} \ t; \ \text{String id; Declarations params; Declarations locals; Block body} \\
\text{Type} &= \text{int} | \text{boolean} | \text{void} \\
\text{Block} &= \text{Statement}^* \\
\text{Statement} &= \text{Skip} | \text{Block} | \text{Assignment} | \text{Conditional} | \text{Loop} | \text{Call} | \text{Return} \\
\text{Call} &= \text{String name; Expressions args} \\
\text{Expressions} &= \text{Expression}^* \\
\text{Return} &= \text{Expression result} \\
\text{Expression} &= \text{Variable} | \text{Value} | \text{Binary} | \text{Unary} | \text{Call}
\end{align*}
\]
Static Type Checking for Methods

• New validity rules now need to be added:
  – Every method and global variable must have a unique ID
  – All local variables and parameters within a method must be unique with valid types
  – All statements and expressions within the body of each method must be valid with respect to the variables and parameters accessible to them
  – A return appears in every method with a nonvoid Type, and the type of the Expression in the return must be identical with the method’s Type
  – Every call has a name identical with the name of a method in the program and has the same number of arguments as the method
  – Every argument in a call has the same type as the type given in the corresponding parameter in the method of that name
• See book for denotational semantics to implement these rules