Compiler construction: Set 3

method resolution is dynamic in resolving which class's methods are needed, searching up the class hierarchy as required.

method resolution is static in resolving overload of methods based on signature.

Symbol Table Questions
1) How store the members of a class?
   - variables
   - methods
   are a hash table specific to each class

2) How to store overloaded methods within a class?
   \[ a \cdot m(3, 4, 7, 12, 2) \] ; hashing ID + signature fails: too many possibilities
Algorithmic ASTs: goals for design

1) Ability to unparsable to a program equivalent to original program.
2) Hide implementation details: use getters to access data within classes.
3) API changes for different compilation phases:
   new methods: `uparse()`, `SemanticCheck()`, `generateCodec()`

ST modifications: `type`, `kind`

```
int, bool, array, type, kind
```
Java Virtual Machine

bytecode

Jasmin assembler

CSX-Go

Compiler

 assembler code

Jasmin

class file

bytecode:
compact: 0-address,
implicit stack
safe: stack size errors, type errors

disassembler
java: runs program (class file)
javac: compiles source code -> class
javap -c: disassembles a class file
The class file has several sections:

- **Code**
- **Constant pool**
- Values and ptrs to fields in classes associated type
- 16-bit indexed: each value/ptr uses 1 index
- Types are represented by strings:
  - `byte B`
  - `short S`
  - `char C`
  - `int I`
  - `long J`
  - `float F`
  - `double D`
  - `void V`
  - `array of int []I`
  - `array of type []E [E]`
  - `reference to type @L@`

**Arithmetic**: all stack entries are 32-bit quantities.

- **fadd**: pops 2 ints, adds, pushes result.
- **ldadd**: adds 2 longs, each is stored in 2 consecutive stack cells.

**Mmul**

The arguments and the result.

**Registers**: 32 bits
- As many as you need
- First ones: parameters 0, 1, 2, ...
- Static method: no instance
- Dynamic method: "this" is in register 0
- Next ones: general purpose for local variables
- 64-bit quantity: use 2 adjacent registers even, odd numbered registers.
load: register → stack
store: stack → register

load: o\text{load} \ (\text{reg})
i\text{load} 3 \ (3\text{ bytes})
i\text{load}_3 \ (1\text{ byte})
i\text{store}_7 \ (3\text{ bytes})
i\text{store}_7 \ (1\text{ byte})

fields (global variables in c,s,r,g, fields in Java)

static \[\rightarrow\] get static \[\text{name} \ 	ext{type} \rightarrow \text{java/io/PrintStream}]

dynamic

java/lang/System/out

from field to stack
put static \[\text{name} \ 	ext{type} \]
from stack to field

generic field (instance→stack) \[\text{name} \ 	ext{type} \]
put field (stack→instance)

to place value of an instance field into a local variable: get field, then store.
type conversion: pop old value, push converted value

\( \text{i2f} : \text{integer to float} \)

Calling a method:

\( \text{static} : \text{invokeStatic} \quad \text{Name} \)

\( \text{java/lang/Math/pow} \quad (D D) D \)

\( \text{dynamic} : \text{invokeVirtual} \quad \text{Name} \)

\( \text{return to object popped from stack} \)

\( \text{java/lang/System/Out/println} \quad (I) \)

call \text{foo} (1, 2, 3, \langle c \rangle)

\[
\begin{array}{c}
\text{4} \\
\text{3} \\
\text{2} \\
\text{1} \\
\text{0} \\
\end{array}
\quad
\text{return value}
\]

type of string:

\( \text{L java/lang/String;} \)

Java arrays are instances of an Array class.

Special instructions:

\( \text{iload} : \text{load an integer from an int array} \)

\[
\begin{array}{c}
\text{index} \\
\text{value} \\
\end{array}
\]

\( \text{istore} : \text{place an integer into an int array} \)

\[
\begin{array}{c}
\text{value} \\
\text{index} \\
\end{array}
\]

related: \( f, d, l, b, s, \) for basic types.
array of class, such as array of String:

- a load
- a astore
- ref to array

ptr to the array

value: ref to the item in the array

new array type

- length

Branching: 3 byte: opcode distance

5 byte:

- 1B
- 2B
- 4B

goto (loc)

conditional pop an integer

- ifgt (loc)
- ifeq
- iflt
- ifge
- ifle
- ifne

conditional based on 2 values, both popped.

- if cmp gt (loc)
Symbol tables ( CST )

Static scoping: scope of an identifier is known to compiler, nested scopes with inheritance, with hiding.

To organize ST
1) list of STs for currently active scopes
2) unified ST with a field for level.

bookstore: to close a scope.

Specific concerns of the source language:
- records (structs, classes)
  - can be nested
  - the type has components
    1) record
    2) ST of its fields
      - each entry: name, type, offset
    3) length (in bytes)

- explicit named types
  - entered into ST
    - name, type, kind = type
    - variables of the named type:
      - ST entry just points to the type entry in ST.

Pascal:

```
type
  r = record
    a: int; b: char end;
```

```
var
  v: r;
```

ST:

```
<table>
<thead>
<tr>
<th>v: type, record</th>
</tr>
</thead>
<tbody>
<tr>
<td>v: type</td>
</tr>
</tbody>
</table>
```

hash table

```
<table>
<thead>
<tr>
<th>a: int, offset 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>b: char, offset 4</td>
</tr>
</tbody>
</table>
```
overloading: multiple competing declarations of the same identifier within a scope.
- handle by building a list of all overloaded meanings.

- altered search rules
  Pascal with statement
  
  \[
  \text{with as do} \quad \begin{cases} \text{v.a := 7} \\ a := 7 \end{cases} \\
  \text{end;}
  \]
  push a new scope onto the ST list
  \( \rightarrow \) scope of the record's type.
  place a temporary value for "this" in ST.

- forward references
  2-pass compilation (2 passes through AST in checking semantics)
  check as a scope is closed that all promises are fulfilled.

- type descriptors
  easy: int, long
  others: arrays (element type, range, index type)
  records (see above)
  pointer (base type)
  difficult: dynamic-sized arrays.

  type descriptor must be available at run time.
Semantics checking during checking, ST can be updated, AST can be modified.

If:
  I/E
    expr
  Ss
  Ss

While
  label
  expr
  Ss
  bool
  in GT

Switch
  condition
  cases
  default
  label
  expr
  context
  reducible to a literal of right type

Other semantic checks
1) type consistency
2) reachability
3) exception handing
type consistency:
when is assignment valid?
LHS = RHS
Similarly, actual vs formal binding
if types are same, OK
if types are compatible, OK
name equivalence
names derived from constructors
structural equivalence
records
 laxities: field names ignored
flattening
arrays
base types: int, float, ...
record introduces new type
array
compatible: LHS member of subtype OK
coercible (int -> float)
Reachability

is Reachable (i)

terminates Normally (t)

if either branch has t.

null statement

return break continue

if true/false

while

while true

while false

error!

int v;

v += 10;  

return;

v >= 10;

v >= 20;
Exceptions

try {
  throws e1, e2
  catch (ExceptionType e1)
    throws e3
  catch (e2...)
}

1. must declare all exceptions a method can throw.
2. all exceptions declared as thrown must be possible.
3. a catch may only mention a possible exception.

implementation:
keep track of all exceptions S can throw, at end of method, verify that the exceptions thrown inside the method are covered by the declaration.
a catch block removes exceptions from the set, can add to the set

Try Node

S

C C C C

S1: throws e1
  try
    catch e1