Symbol tables are used to describe and analyse definitions and uses of identifiers.

Grammars are too weak; the language:

$$\{w \alpha w \mid w \in \Sigma^*\}$$

is not context-free.

A symbol table is a map from identifiers to meanings:

- **i**: local, int
- **done**: local, boolean
- **insert**: method, ...
- **List**: class, ...
- **x**: formal, list
- ...

We must construct a symbol table for every program point.

Using symbol tables to analyse JOOS:

- which classes are defined;
- what is the inheritance hierarchy;
- is the hierarchy well-formed;
- which fields are defined;
- which methods are defined;
- what are the signatures of methods;
- are identifiers defined twice;
- are identifiers defined when used; and
- are identifiers used properly?

The standard of modern languages.
Old-style one-pass technology:

```
void weedPROGRAM(PROGRAM *p);
void weedCLASSFILE(CLASSFILE *c);
void weedCLASS(CLASS *c);
```

Forward declarations enable recursion.

Still haunts some languages:

```
The symbol table behaves like a stack:
```

The symbol table can be implemented as a simple stack:

```
• pushSymbol(SymbolTable *t, char *name, ...)
• popSymbol(SymbolTable *t)
• getSymbol(SymbolTable *t, char *name)
```

Still just linear search, though.
Implement symbol tables as a cactus stack of hash tables:

- each hash table contains the identifiers in a level;
- push a new hash table when a level is entered;
- each identifier is entered in the top hash table;
- it is an error if it is already there;
- a use of an identifier is looked up in the hash tables from top to bottom;
- it is an error if it is not found;
- pop a hash table when a level is left.

What is a good hash function on identifiers?

Use the initial letter:
- codePROGRAM, codeMETHOD, codeEXP, ...

Use the sum of the letters:
- doesn’t distinguish letter order

Use the shifted sum of the letters:

```
"j" = 106 = 0000000001101010
shift 0000000011010100
+ "o" = 111 = 0000000001101111
= 0000001011100111
shift 0000101110011110
+ "s" = 115 = 0000000001110011
= 0000011001011101
= 1629
```
How to handle mutual recursion:

A

...B...

B

...A...

A single traversal of the abstract syntax tree is not enough.

Make two traversals:

• collect definitions of identifiers; and
• analyse uses of identifiers.

For cases like recursive types, the definition is not completed before the second traversal.

Symbol information in JOOS:

$ cat tree.h

[...]

typedef enum(classSym,fieldSym,methodSym,
formalSym,localSym) SymbolKind;

typedef struct SYMBOL {
  char *name;
  SymbolKind kind;
  union {
    struct CLASS *classS;
    struct FIELD *fieldS;
    struct METHOD *methodS;
    struct FORMAL *formalS;
    struct LOCAL *localS;
  } val;
} SYMBOL;

[...]
Symbol tables are weaved together with abstract syntax trees:

```java
public class B extends A {
    protected A a;
    protected B b;

    public void m(A x, B y) {
        this.m(a,b);
    }
}
```

Complicated recursion in JOOS is resolved through multiple passes:

```bash
cat symbol.c
```

Each pass goes into further detail:

- **symInterfacePROGRAM**: define classes and their interfaces;
- **symInterfaceTypesPROGRAM**: build hierarchy and analyse interface types; and
- **symImplementationPROGRAM**: define locals and analyse method bodies.

Defining a JOOS class:

```c
void symInterfaceCLASS(CLASS *c, SymbolTable *sym) {
    SYMBOL *s;
    if (defSymbol(sym,c->name)) {
        reportStrError("class name %s already defined", c->name,c->lineno);
    } else {
        s = putSymbol(sym,c->name,classSym);
        s->val.classS = c;
        c->localsym = initSymbolTable();
        symInterfaceFIELD(c->fields,c->localsym);
        symInterfaceCONSTRUCTOR(c->constructors, c->name,c->localsym);
        symInterfaceMETHOD(c->methods,c->localsym);
    }
}
```

Defining a JOOS method:

```c
void symInterfaceMETHOD(METHOD *m, SymbolTable *sym) {
    SYMBOL *s;
    if (m!=NULL) {
        symInterfaceMETHOD(m->next,sym);
        if (defSymbol(sym,m->name)) {
            reportStrError("method name %s already defined", m->name,m->lineno);
        } else {
            s = putSymbol(sym,m->name,methodSym);
            s->val.methodS = m;
        }
    }
}
```

and its signature:

```c
void symInterfaceTypesMETHOD(METHOD *m, SymbolTable *sym) {
    if (m!=NULL) {
        symInterfaceTypesMETHOD(m->next,sym);
        symTYPE(m->returntype,sym);;
        symInterfaceTypesFORMAL(m->formals,sym);
    }
}
```
Analysing a JOOS class implementation:

```c
void symImplementationCLASS(CLASS *c)
{
    SymbolTable *sym;
    sym = scopeSymbolTable(classlib);
    symImplementationFIELD(c->fields,sym);
    symImplementationCONSTRUCTOR(c->constructors,c,sym);
    symImplementationMETHOD(c->methods,c,sym);
}
```

Analysing a JOOS method implementation:

```c
void symImplementationMETHOD(METHOD *m, CLASS *this, SymbolTable *sym)
{
    SymbolTable *msym;
    if (m!=NULL) {
        symImplementationMETHOD(m->next,this,sym);
        msym = scopeSymbolTable(sym);
        symImplementationFORMAL(m->formals,msym);
        symImplementationSTATEMENT(m->statements,this,msym,
                                  m->modifier==staticMod);
    }
}
```

Analysing JOOS local declarations:

```c
void symImplementationLOCAL(LOCAL *l, SymbolTable *sym)
{
    SYMBOL *s;
    if (l!=NULL) {
        symImplementationLOCAL(l->next,sym);
        symTYPE(l->type,sym);
        if (defSymbol(sym,l->name)) {
            reportStrError("local %s already declared",
                           l->name,l->lineno);
        } else {
            s = putSymbol(sym,l->name,localSym);
            s->val.localS = l;
        }
    }
}
```

Identifier lookup in the JOOS class hierarchy:

```c
SYMBOL *lookupHierarchy(char *name, CLASS *start)
{
    SYMBOL *s;
    if (start==NULL) return NULL;
    s = getSymbol(start->localsym,name);
    if (s!=NULL) return s;
    if (start->parent==NULL) return NULL;
    return lookupHierarchy(name,start->parent);
}
```

For which class do we return NULL on line 5 of each function?
Analysing expressions:

void symImplementationEXP(EXP *e, CLASS *this,
   SymbolTable *sym, int stat)
{
    switch (e->kind) {
      case idK:
        e->val.idE.idsym = symVar(e->val.idE.name, sym,
          this, e->lineno, stat);
        break;
      case assignK:
        e->val.assignE.leftsym =
          symVar(e->val.assignE.left, sym,
          this, e->lineno, stat);
        symImplementationEXP(e->val.assignE.right,
           this, sym, stat);
        break;
        ...
    }
}

Analysing an identifier:

SYMBOL *symVar(char *name, SymbolTable *sym,
   CLASS *this, int lineno, int stat)
{
    SYMBOL *s;
    s = getSymbol(sym, name);
    if (s==NULL) {
      s = lookupHierarchy(name, this);
      if (s==NULL) {
        reportStrError("identifier %s not declared",
          name, lineno);
      } else {
        if (s->kind!=fieldSym)
          reportStrError("%s is not a variable as expected",
            name, lineno);
      }
    } else {
      if ((s->kind!=fieldSym) && (s->kind!=formalSym) &&
        (s->kind!=localSym))
        reportStrError("%s is not a variable as expected",
          name, lineno);
      }
    } else {
      if ((s->kind=fieldSym) && (s->kind=formalSym) &&
        (s->kind=localSym))
        reportStrError("illegal static reference to %s",
          name, lineno);
      return s;
    }
}

The testing strategy for the symbol tables involves an extension of the pretty printer.

A textual representation of the symbol table is printed once for every scope area.

- In Java, use to\texttt{String}().

These tables are then compared to a corresponding manual construction for a sufficient collection of programs.

Furthermore, every error message should be provoked by some test program.