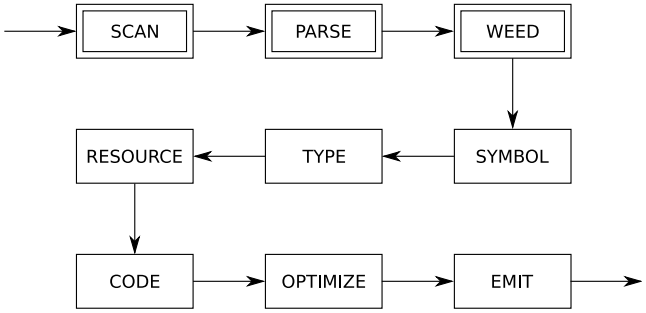
Abstract Syntax Trees (part 2)

COMP 520: Compiler Design (4 credits)

Professor Laurie Hendren

hendren@cs.mcgill.ca



Today ....

- Introduction to the JOOS language
- More on building ASTs and weeding.
- Interesting examples from JOOS and OncoTime

The Java language:

- was originally called Oak;
- was developed as a small, clean, OO language for programming consumer devices;
- was built into the Webrunner browser;
- matured into Java and HotJava;
- is now supported by many browsers, allowing Java programs to be embedded in WWW pages;
- is also used by web servers, even if the client user is not running Java; and
- is the implementation language for several large applications.

Basic compilation (`.java` $\longrightarrow$ `.class`):

- Java programs are developed as source code for a collection of Java classes;
- each class is compiled into Java Virtual Machine (JVM) bytecode;
- bytecode is interpreted or JIT-compiled using some implementation of the JVM;
- Java supports a GUI; and
- many browsers have Java plugins for executing JVM bytecode.

Major benefits of Java:

- it's object-oriented;
- it's a "cleaner" OO language than C++;
- it's portable (except for native code);
- it's distributed and multithreaded;
- it's secure;
- it supports windowing and applets;
- it's semantics is completely standardized;
- it has a huge class library; and
- it's finally finally finally officially open source.

Java security has many sources:

- programs are strongly type-checked at compile-time;
- array bounds are checked at run-time;
- `null` pointers are checked at run-time;
- there are no explicit pointers;
- dynamic linking is checked at run-time; and
- class files are verified at load-time.

Major drawbacks of Java:

- it misses some language features, e.g. genericity (until 1.5), multiple inheritance, operator overloading;
- it does not have one *single* standard (`JDK 1.0.2` vs. `JDK 1.1.*` vs. ...) and probably never will;
- it can be slower than C++ for expensive numeric computations due to dynamic array-bounds checks; $Z^{Z^{Z^{ZZ}}}$ and
- it's not JOOS.

Goals in the design of JOOS:

- extract the object-oriented essence of Java;
- make the language small enough for course work, yet large enough to be interesting;
- provide a mechanism to link to existing Java code; and
- ensure that every JOOS program is a valid Java program, such that JOOS is a strict subset of Java.

Programming in JOOS:

- each JOOS program is a collection of classes;
- there are ordinary classes which are used to develop JOOS code; and
- there are external classes which are used to interface to Java libraries.

An ordinary class consists of:

- protected fields;
- constructors; and
- public methods.

```
$ cat Cons.java
public class Cons {
  protected Object first;
  protected Cons rest;

  public Cons(Object f, Cons r)
  { super(); first = f; rest = r; }

  public void setFirst(Object newfirst)
  { first = newfirst; }

  public Object getFirst()
  { return first; }

  public Cons getRest()
  { return rest; }

  public boolean member(Object item)
  { if (first.equals(item))
      return true;
    else if (rest==null)
      return false;
    else
      return rest.member(item);
```

```
  }

  public String toString()
  { if (rest==null)
      return first.toString();
    else
      return first + " " + rest;
  }
}
```

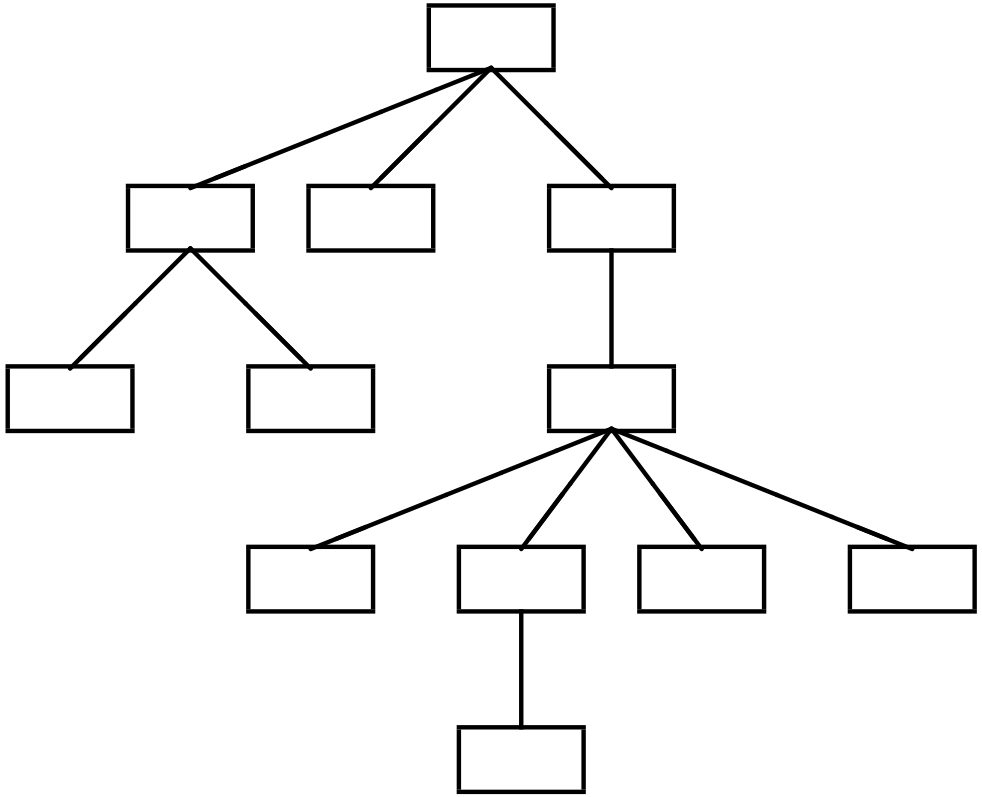
Notes on the `Cons` example:

- fields in JOOS must be *protected*: they can only be accessed via objects of the class or its subclasses;
- constructors in JOOS must start by invoking a constructor of the superclass, i.e. by calling `super(...)` where the argument types determine the constructor called;
- methods in JOOS must be *public*: they can be invoked by any object; and
- only constructors in JOOS can be overloaded, other methods cannot.

Other important things to note about JOOS:

- subclassing must not change the signature of a method;
- local declarations must come at the beginning of the statement sequence in a block; and
- every path through a non-void method must return a value. (In Java such methods can also throw exceptions.)

The class hierarchies in JOOS and Java are both single inheritance, i.e. each class has exactly one superclass, except for the root class:

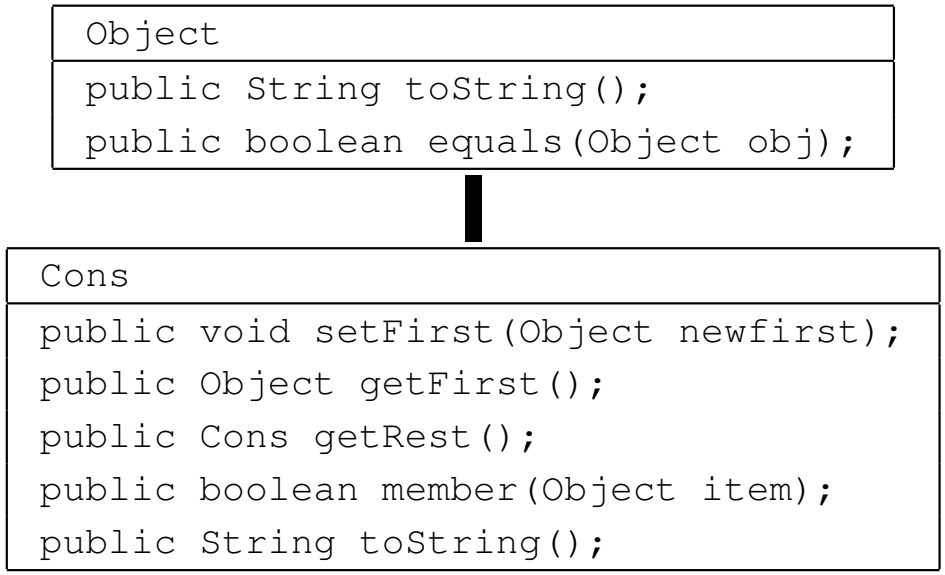


The root class is called `Object`, and any class without an explicit `extends` clause is a subclass of `Object`.

The definition of `Cons` is equivalent to:

```
public class Cons extends Object
{ ... }
```

which gives the tiny hierarchy:



The class `Object` has two methods:

- `toString()` returns a string encoding the type and object id; and
- `equals()` returns true if the object reference denotes the current object.

These methods are often overridden in subclasses:

- `toString()` encodes the value as a string; and
- `equals()` decides a more abstract equality.

When overriding a method, the argument types and return types must remain the same.

When overriding `equals()`, `hashcode()` must also be overridden: equal objects *must* produce the same hashcode.

Extending the `Cons` class:

```
$ cat ExtCons.java
```

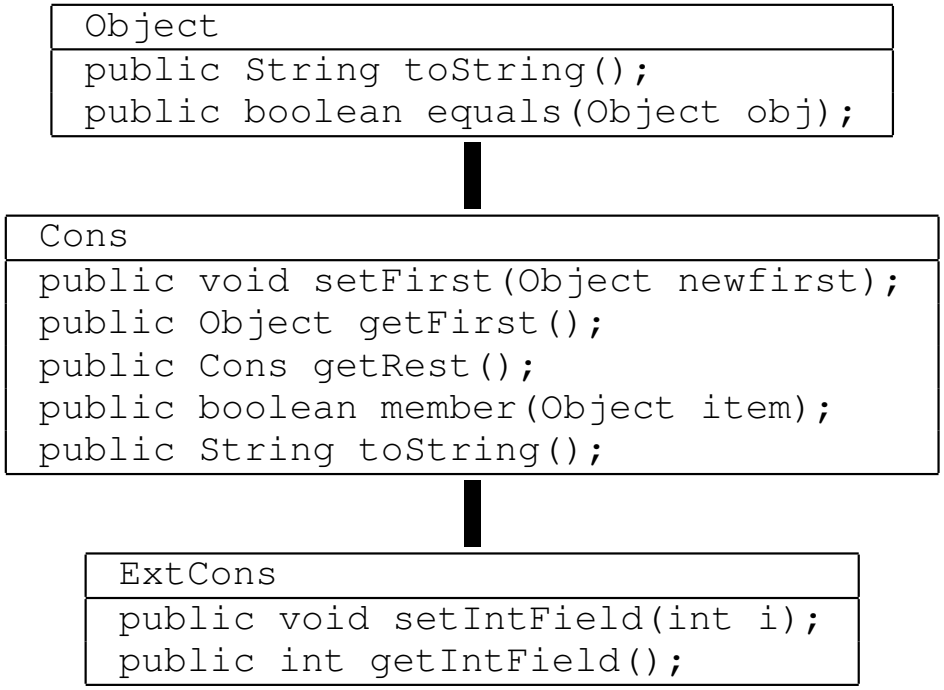
```
public class ExtCons extends Cons {
  protected int intField;

  public ExtCons(Object f, Cons r, int i)
  { super(f,r);
    intField = i;
  }

  public void setIntField(int i)
  { intField = i; }

  public int getIntField()
  { return(intField); }
}
```

The extended hierarchy:



Using the `Cons` class:

```
$ cat UseCons.java
```

```
import joos.lib.*;

public class UseCons {

  public UseCons() { super(); }

  public static void main(String argv[])
    { Cons l;
      JoosIO f;

      l = new Cons("a",new Cons("b",new Cons("c",null)));
      f = new JoosIO();
      f.println(l.toString());
      f.println("first is " + l.getFirst());
      f.println("second is " + l.getRest().getFirst());
      f.println("a member? " + l.member("a"));
      f.println("z member? " + l.member("z"));
    }
}
```

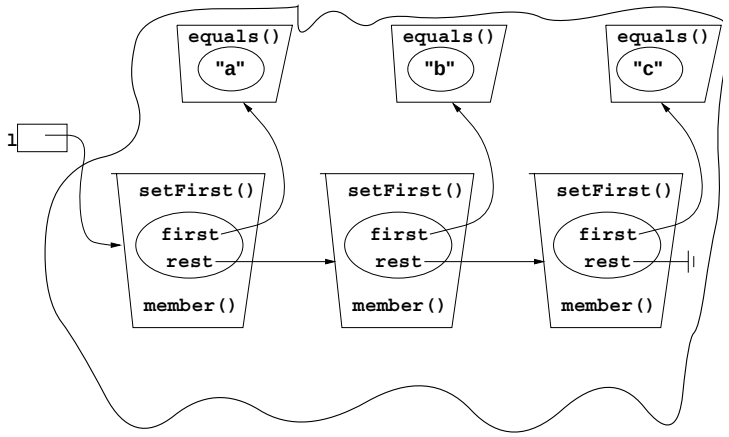
A Java program (not an applet) requires a `main()` method.

It is necessary to `import` library functions such as `println()`.

Compile and run the `UseCons` program:

```
$ javac joos/lib/*.java
$ joosc UseCons.java Cons.java
$ java UseCons
```

The `UseCons` program builds these objects:



The output of the `UseCons` program is:

```
a b c
first is a
second is b
a member? true
z member? false
```

Types in JOOS are either primitive types:

- `boolean`: `true` and `false`;
- `int`: $-2^{31} \ldots 2^{31} - 1$;
- `char`: the ASCII characters;

or user-defined class types;

or externally defined class types:

- `Object`;
- `Boolean`;
- `Integer`;
- `Character`;
- `String`;
- `BitSet`;
- `Vector`;
- `Date`.

Note that `boolean` and `Boolean` are different.

Types in Java and JOOS:

- Java is strongly-typed;
- Java uses the name of a class as its type;
- given a type of class `C`, any instance of class `C` or a subclass of `C` is a permitted value;
- there is "down-casting" which is automatically checked at run-time:

```
SubObject subobj = (SubObject) obj;
```

- there is an explicit `instanceof` check:

```
if (subobj instanceof Object)
  return true;
else
  return false;
```

- and finally some type-checking must be done at run-time.

Statements in JOOS:

- expression statements:

```
x = y + z;
x = y = z;
a.toString(l);
new Cons("abc",null);
```

- block statements:

```
{ int x;
  x = 3;
}
```

- control structures:

```
if (l.member("z")) {
   // do something
}

while (l != null) {
   l = l.getRest(); // do something
}
```

- return statements:

```
return;

return true;
```

Expressions in JOOS:

- constant expressions:

  ```
  true, 13, '\n', "abc", null
  ```

- variable expressions:

  ```
  i, first, rest
  ```

- binary operators:

  ```
  ||
  &&
  != ==
  < > <= >= instanceof
  + -
  * / %
  ```

- unary operators:

  ```
  -
  !
  ```

Expressions in JOOS:

- class instance creation:

```
new Cons("abc",null)
```

- cast expressions:

```
(String) getFirst(list)
(char) 119
```

- method invocation:

```
l.getFirst()
super.getFirst();
l.getFirst().getFirst();
this.getFirst();
```

Abstract methods and classes:

- a method may be `abstract`, where no implementation is given;
- if a class contains one or more `abstract` methods, it must be defined as an `abstract` class;
- the constructor of an `abstract` class cannot be invoked;
- `abstract` classes are used to define “frameworks”.

```
import joos.lib.*;

public abstract class Benchmark {
  protected JoosSystem s; // JOOS interface to the Java System Class

  public Benchmark()
  { super(); s = new JoosSystem(); }

  public abstract void benchmark();// Hook for actual benchmark

  public int myrepeat(int count) // driver to time repeated executions
  { int start;
    int i;
    start = s.currentTimeMillis();
    i = 0;
    while (i < count) {
      this.benchmark();
      i = i+1;
    }
    return s.currentTimeMillis()-start;
  }
}
```

```
public class ExtBenchmark extends Benchmark {
  public ExtBenchmark() {
    super();
  }
  public void benchmark() {} // timing an empty method
}

import joos.lib.*;
public class UseBenchmark {
  public UseBenchmark() { super(); }
  public static void main(String argv[])
  { ExtBenchmark b;
    JoosIO f;
    int reps;
    int time;
    b = new ExtBenchmark();
    f = new JoosIO();
    f.print("Enter number of repetitions: ");
    reps = f.readInt();
    time = b.myrepeat(reps);
    f.println("time is " + time + " millisecs");
  }
}
```

Final methods and classes:

- the `final` keyword is used when no modifications to functionality are allowed;
- a `final` method cannot be overridden by subclasses;
- a `final` class cannot be extended;
- `final` classes typically belong to libraries: `Boolean`, `Integer`, and `String` (for security purposes).

Note that JOOS does not provide `final` *fields* like Java does.

Synchronized methods:

- Java and JOOS programs can start multiple threads;
- sometimes access to a shared resource must be protected, such that only one thread is in a *critical section* at a time;
- each object has an associated lock; and
- JOOS provides `synchronized` methods, such that when a thread invokes a `synchronized` method on an object, the thread does not enter the method until it has successfully acquired the target object's lock and it holds on to the lock until the method execution completes.

Note that JOOS does not provide `synchronized` blocks like Java does.

```
public class SyncBox {
  protected Object boxContents;

  public SyncBox() { super(); }

  // return contents of the box, set contents to null
  public synchronized Object get()
  { Object contents;
    contents = boxContents;
    boxContents = null;
    return contents;
  }

  // put something in the box,
  // if the box already has something in it, return false
  // else fill the box, return true
  public synchronized boolean put (Object contents)
  { if (boxContents != null) return false;
    boxContents = contents;
    return true;
  }
}
```

External classes in Java:

- Java compiles programs with respect to a set of libraries of precompiled class files; and
- when a Java compiler encounters an unknown method, it searches the precompiled bytecode for an implementation.

External classes in JOOS:

- JOOS compiles programs with respect to a set of libraries of precompiled class files; but
- external classes must be explicitly presented to the JOOS compiler.

```
$ cat joos/extern/javalib.joos
[...]
// java.lang.String
extern public final class String in "java.lang" {
  public String();
  public String(String value);
  public String(StringBuffer buffer);
  public String vlaueOf(boolean b);
  public char charAt(int index);
  public int compareTo(String anotherString);
  public boolean endsWith(String suffix);
  public boolean equals(Object obj);
  public boolean equalsIgnoreCase(String anotherString);
  public int indexOf(String str, int fromIndex);
  public int lastIndexOf(String str, int fromIndex);
  public int length();
  public boolean regionMatches(boolean ignoreCase,
     int toffset, String other, int ooffset, int len);
  public boolean startsWith(String prefix, int toffset);
  public String substring(int beginIndex, int endIndex);
  public String concat(String str);
  public String toLowerCase();
  public String toUpperCase();
  public String toString();
  public String trim();
}
[...]
```

External declarations for Java libraries:

- `javalib.joos`
- `appletlib.joos`
- `awtlib.joos`
- `netlib.joos`
- `BigDecimal.joos`

External declarations for JOOS libraries:

- `jooslib.joos`

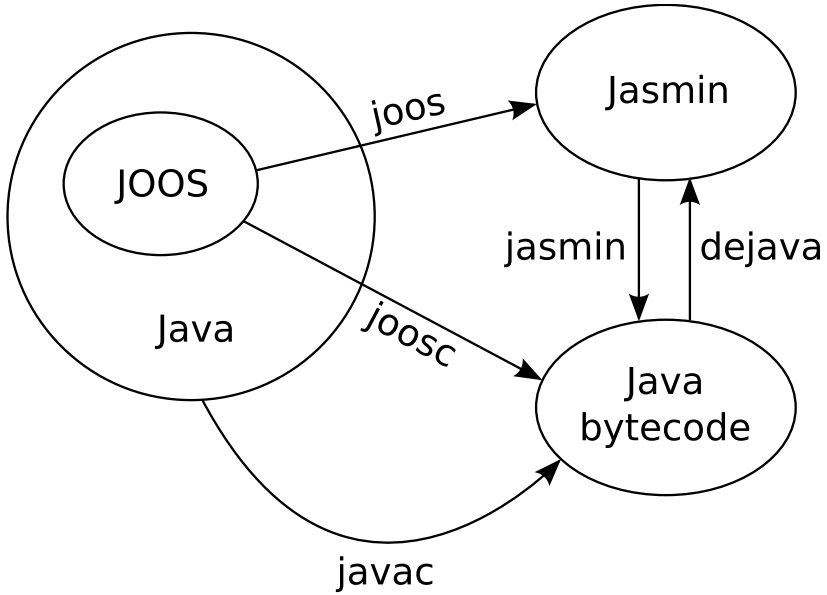
Example JOOS programs:

- `AppletGraphics`: simple graphics programs to be displayed via a browser;
- `AwtDemos`: examples of using the Abstract Windows Toolkit;
- `ImageDemos`: two techniques for displaying an animation;
- `Network`: simple examples of interacting over the network;
- `Simple`: a relatively large collection of simple programs;
- `Threads`: simple multithreaded programs; and
- `WIGapplets`: examples of WIG applets.

When compared to Java, JOOS:

- does not support packages, interfaces, exceptions, some control structures, mixed statements and declarations;
- has only `protected` fields and `public` methods;
- does not allow overloading of methods;
- does not support arrays;
- does not allow static methods;
- supports only `int`, `boolean`, and `char` as primitive types; and
- uses external class declarations.

Converting between JOOS & Java source code (`*.java`, `*.joos`), Jasmin assembler (`*.j`) and Java bytecode (`*.class`):



`joosc` simply calls `joos` and then `jasmin`.

Now lets look at some highlights of the JOOS scanner/parser/AST:

- Remember that there are three implementations of JOOS (A- version), available on the web site: flex/bison, SableCC2 and SableCC3.
- We will do one assignment on the JOOS compiler (peephole optimization of bytecode).

The JOOS compiler has the AST node types:

```
PROGRAM      CLASSFILE    CLASS
FIELD        TYPE         LOCAL
CONSTRUCTOR  METHOD       FORMAL
STATEMENT    EXP          RECEIVER
ARGUMENT     LABEL        CODE
```

with many extra fields:

```
typedef struct METHOD {
  int lineno;
  char *name;
  ModifierKind modifier;
  int localslimit; /* resource */
  int labelcount; /* resource */
  struct TYPE *returntype;
  struct FORMAL *formals;
  struct STATEMENT *statements;
  char *signature; /* code */
  struct LABEL *labels; /* code */
  struct CODE *opcodes; /* code */
  struct METHOD *next;
} METHOD;
```

The JOOS constructors are as we expect:

```
METHOD *makeMETHOD(char *name, ModifierKind modifier, TYPE *returntype,
                   FORMAL *formals, STATEMENT *statements, METHOD *next)
{ METHOD *m;
  m = NEW(METHOD);
  m->lineno = lineno;
  m->name = name;
  m->modifier = modifier;
  m->returntype = returntype;
  m->formals = formals;
  m->statements = statements;
  m->next = next;
  return m;
}

STATEMENT *makeSTATEMENTwhile(EXP *condition, STATEMENT *body)
{ STATEMENT *s;
  s = NEW(STATEMENT);
  s->lineno = lineno;
  s->kind = whileK;
  s->val.whileS.condition = condition;
  s->val.whileS.body = body;
  return s;
}
```

Highlights from the JOOS scanner:

```
[ \t]+          /* ignore */;
\n              lineno++;
\/\/[^\n]*      /* ignore */;
abstract        return tABSTRACT;
boolean         return tBOOLEAN;
break           return tBREAK;
byte            return tBYTE;
...
"!="            return tNEQ;
"&&"            return tAND;
"||"            return tOR;
"+"             return '+';
"-"             return '-';
...
```

Setting values

```
0|([1-9][0-9]*) {yylval.intconst = atoi(yytext);
                 return tINTCONST;}
true            {yylval.boolconst = 1;
                 return tBOOLCONST;}
false           {yylval.boolconst = 0;
                 return tBOOLCONST;}
\"([^\"])*\" {yylval.stringconst =
                 (char*)malloc(strlen(yytext)-1);
              yytext[strlen(yytext)-1] = '\0';
              sprintf(yylval.stringconst,"%s",yytext+1);
              return tSTRINGCONST;}
```

Highlights from the JOOS parser:

```
method : tPUBLIC methodmods returntype
       tIDENTIFIER '(' formals ')' '{' statements '}'
       {$$ = makeMETHOD($4,$2,$3,$6,$9,NULL);}
     | tPUBLIC returntype
       tIDENTIFIER '(' formals ')' '{' statements '}'
       {$$ = makeMETHOD($3,modNONE,$3,$5,$8,NULL);}
     | tPUBLIC tABSTRACT returntype
       tIDENTIFIER '(' formals ')' ';'
       {$$ = makeMETHOD($4,modABSTRACT,$3,$6,NULL,NULL);}
     | tPUBLIC tSTATIC tVOID
       tMAIN '(' mainargv ')' '{' statements '}'
       {$$ = makeMETHOD("main",modSTATIC,
                        makeTYPEvoid(),NULL,$9,NULL);}
;

whilestatement : tWHILE '(' expression ')' statement
               {$$ = makeSTATEMENTwhile($3,$5);}
;
```

Notice the conversion from concrete syntax to abstract syntax that involves dropping unnecessary tokens.

Building LALR(1) lists:

```
formals : /* empty */
          {$$ = NULL;}
        | neformals
          {$$ = $1;}
;

neformals : formal
            {$$ = $1;}
          | neformals ',' formal
            {$$ = $3; $$->next = $1;}
;

formal : type tIDENTIFIER
         {$$ = makeFORMAL($2,$1,NULL);}
;
```

The lists are naturally backwards.

Using backwards lists:

```
typedef struct FORMAL {
  int lineno;
  char *name;
  int offset; /* resource */
  struct TYPE *type;
  struct FORMAL *next;
} FORMAL;


void prettyFORMAL(FORMAL *f)
{ if (f!=NULL) {
    prettyFORMAL(f->next);
    if (f->next!=NULL) printf(", ");
    prettyTYPE(f->type);
    printf(" %s",f->name);
  }
}
```

What effect would a call stack size limit have?

The JOOS grammar calls for:

```
castexpression :
   '(' identifier ')' unaryexpressionnotminus
```

but that is not LALR(1).

However, the more general rule:

```
castexpression :
   '(' expression ')' unaryexpressionnotminus
```

is LALR(1), so we can use a clever action:

```
castexpression :
   '(' expression ')' unaryexpressionnotminus
   {if ($2->kind!=idK) yyerror("identifier expected");
    $$ = makeEXPcast($2->val.idE.name,$4);}
;
```

Hacks like this only work sometimes.

LALR(1) and Bison are not enough when:

- our language is not context-free;
- our language is not LALR(1) (for now let's ignore the fact that Bison now also supports GLR); or
- an LALR(1) grammar is too big and complicated.

In these cases we can try using a more liberal grammar which accepts a slightly larger language.

A separate phase can then weed out the bad parse trees.

***Example*: disallowing division by constant 0:**

```
exp : tIDENTIFIER
    | tINTCONST
    | exp '*' exp
    | exp '/' pos
    | exp '+' exp
    | exp '-' exp
    | '(' exp ')'
;

pos : tIDENTIFIER
    | tINTCONSTPOSITIVE
    | exp '*' exp
    | exp '/' pos
    | exp '+' exp
    | exp '-' exp
    | '(' pos ')'
;
```

We have doubled the size of our grammar.

This is not a very modular technique.

Instead, weed out division by constant 0:

```
int zerodivEXP(EXP *e)
{ switch (e->kind) {
    case idK:
    case intconstK:
        return 0;
    case timesK:
        return zerodivEXP(e->val.timesE.left) ||
               zerodivEXP(e->val.timesE.right);
    case divK:
        if (e->val.divE.right->kind==intconstK &&
           e->val.divE.right->val.intconstE==0) return 1;
        return zerodivEXP(e->val.divE.left) ||
               zerodivEXP(e->val.divE.right);
    case plusK:
        return zerodivEXP(e->val.plusE.left) ||
               zerodivEXP(e->val.plusE.right);
    case minusK:
        return zerodivEXP(e->val.minusE.left) ||
               zerodivEXP(e->val.minusE.right);
  }
}
```

A simple, modular traversal.

Requirements of JOOS programs:

- all local variable declarations must appear at the beginning of a statement sequence:

```
int i;
int j;
i=17;
int b;  /* illegal */
b=i;
```

- every branch through the body of a non-`void` method must terminate with a return statement:

```
boolean foo (Object x, Object y) {
   if (x.equals(y))
      return true;
}                    /* illegal */
```

Also may not return from within a while-loop etc.

These are hard or impossible to express through an LALR(1) grammar.

Weeding bad local declarations:

```
int weedSTATEMENTlocals(STATEMENT *s,int localsallowed)
{ int onlylocalsfirst, onlylocalssecond;
  if (s!=NULL) {
    switch (s->kind) {
      case skipK:
          return 0;
      case localK:
          if (!localsallowed) {
            reportError("illegally placed local declaration",s->lineno);
          }
          return 1;
      case expK:
          return 0;
      case returnK:
          return 0;
      case sequenceK:
          onlylocalsfirst =
             weedSTATEMENTlocals(s->val.sequenceS.first,localsallowed);
          onlylocalssecond =
             weedSTATEMENTlocals(s->val.sequenceS.second,onlylocalsfirst);
          return onlylocalsfirst && onlylocalssecond;
      case ifK:
          (void)weedSTATEMENTlocals(s->val.ifS.body,0);
          return 0;
```

```
    case ifelseK:
        (void)weedSTATEMENTlocals(s->val.ifelseS.thenpart,0);
        (void)weedSTATEMENTlocals(s->val.ifelseS.elsepart,0);
        return 0;
    case whileK:
        (void)weedSTATEMENTlocals(s->val.whileS.body,0);
        return 0;
    case blockK:
        (void)weedSTATEMENTlocals(s->val.blockS.body,1);
        return 0;
    case superconsK:
        return 1; } } }
```

Weeding missing returns:

```
int weedSTATEMENTreturns(STATEMENT *s)
{ if (s!=NULL) {
    switch (s->kind) {
      case skipK:
          return 0;
      case localK:
          return 0;
      case expK:
          return 0;
      case returnK:
          return 1;
      case sequenceK:
          return weedSTATEMENTreturns(s->val.sequenceS.second);
      case ifK:
          return 0;
      case ifelseK:
          return weedSTATEMENTreturns(s->val.ifelseS.thenpart) &&
                 weedSTATEMENTreturns(s->val.ifelseS.elsepart);
      case whileK:
          return 0;
      case blockK:
          return weedSTATEMENTreturns(s->val.blockS.body);
      case superconsK:
          return 0; } } }
```

The testing strategy for a parser that constructs an abstract syntax tree T from a program P usually involves a pretty printer.

If *parse*(P) constructs T and *pretty*(T) reconstructs the text of P, then:

$$\textit{pretty}(\textit{parse}(P)) \approx P$$

Even better, we have that:

$$\textit{pretty}(\textit{parse}(\textit{pretty}(\textit{parse}(P)))) \equiv \textit{pretty}(\textit{parse}(P))$$

Of course, this is a necessary but not sufficient condition for parser correctness.