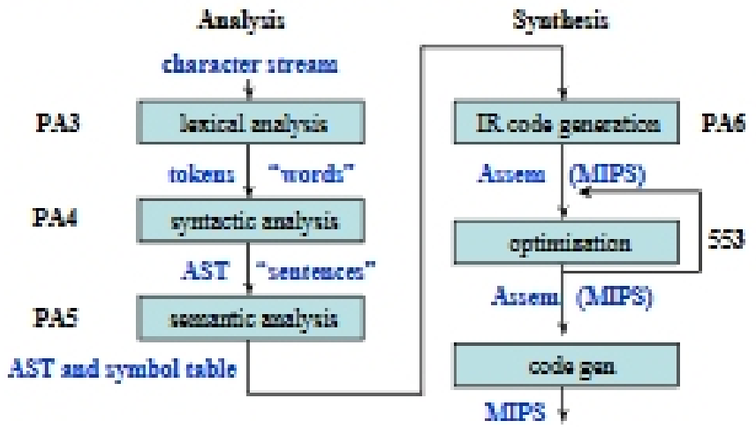
CS 453: Compiler Construction Review

Phases of the compiler

- lexicographical analysis, or scanning (regular expressions)
- syntactic analysis, or parsing (context free grammars)
  - building the abstract syntax tree (syntax-directed translation)
- building the symbol table (visitor design pattern)
- semantic analysis, or type checking (visitor design pattern)
- code generation (visitor design pattern)
  - 3-address code
  - Assem(MIPS)

How would adding floats to the MiniJava compiler affect each phase?

Structure of the MiniJava Compiler



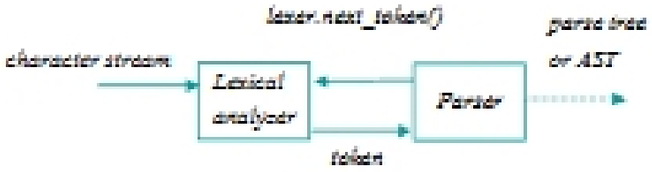
Specifying Tokens with JFlex

JFlex example input file:

```
package mjparser;
import java_cup.runtime.Symbol;


%%
%line
%char
%cup
%public

%eofval{
  return new Symbol(sym.EOF, new
Tokenvalue("EOF", yyline, yychar));
%eofval}
```

```
LETTER=[a-Za-z]
DIGIT=[0-9]
UNDERSCORE="_"
LETT_DIG_UND=(LETTER)|(DIGIT)|(UNDERSCORE)
ID=(LETTER)((LETT_DIG_UND))*

%%
"&&" { return new Symbol(sym.AND, new
Tokenvalue(yytext(), yyline, yychar)); }

"boolean" {return new
Symbol(sym.BOOLEAN,...

{ID} { return new Symbol(sym.ID, new ...
```


Interaction Between Scanning and Parsing



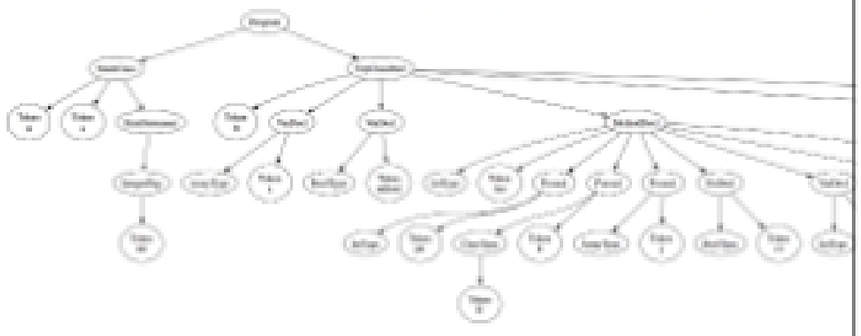
Specifying Grammar with JavaCUP

JavaCUP example input file:

```
package mjparser;
import java_cup.runtime.*;
import ast.node.*;
...
terminal AND, ASSIGN, INT;
terminal mjparser.TokenValue
    NUMBER;
...
non terminal Program  program;

non terminal List<IClassDecl>
    class_decl_list;

non terminal MainClass
    main_class;
```

```
start with program;

program ::=
    main_class:m class_decl_list:l
    {: RESULT = new Program(m,l); :}
    ;

exp ::=
  |   NUMBER:n
      {: Token token = new Token(n.text,
n.line, n.pos);
         RESULT = new IntegerExp( token );
      :}
  ...
```


Abstract Syntax Tree for Memory Layout Example



Semantic Analysis

Determine whether source is meaningful

- Check for semantic errors
- Check for type errors
- Gather type information for subsequent stages
  - Relate variable uses to their declarations

Example errors (from C)

```
functionl = 3.14159;
x = 570 + "hello, world!"
scalar[i]
```


Compiler Data Structures

Symbol Tables

- Compile-time data structure
- Holds names, type information, and *scope* information for variables

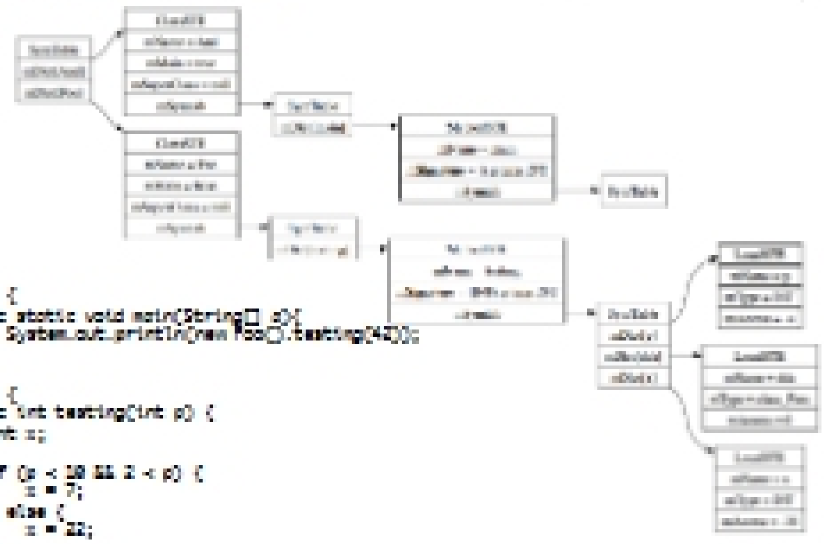
Scopes

- A name space
  e.g. In Pascal, each procedure creates a new scope
  e.g. In C, each set of curly braces defines a new scope
- Can create a separate symbol table for each scope
- What are the scopes in Minifloat?

Using Symbol Tables

- For each variable declaration:
  - Check for symbol table entry
  - Add new entry; add type info
- For each variable use:
  - Check symbol table entry

Example Symbol Table



```
class And {
    public static void main(String[] a){
          System.out.println(new Foo().testing(42));
    }}

class Foo {
    public int testing(int p) {
        int x;

        if (p < 10 && 2 < p) {
            x = 7;
        } else {
            x = 22;
        }
        return x;}}
```

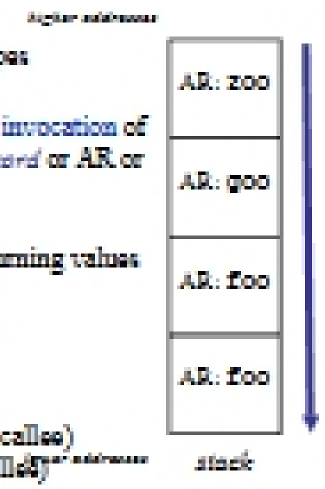

Compiling Procedures

Properties of procedures

- Procedures/methods/functions define scopes
- Procedure lifetimes are nested
- Can store information related to dynamic invocation of a procedure on a call stack (*activation record* or AR or stack frame):
  - Space for saving registers
  - Space for passing parameters and returning values
  - Space for local variables
  - Return address of calling instruction

Stack management

- Push an AR on procedure entry (caller or callee)
- Pop an AR on procedure exit (caller or callee)
- Why do we need a stack?

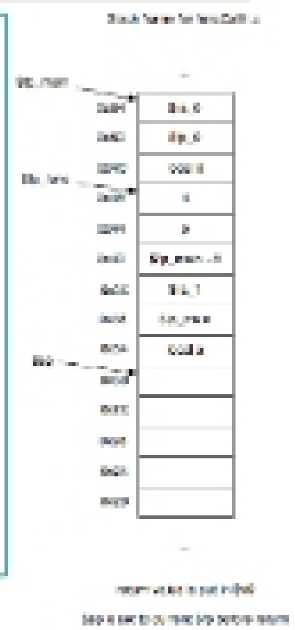


Stack Frame for MiniJava Compiler

```
int foo(int x,int y,int *z) {
    int a;
    a = x * y - *z;
    return a;
}
void main() {
    int x;
    x = 2;
    cout << foo(4,5,&x);
    cout << "\n";
}
```

```
    .text
_foo:
        sw      $ra, 0($sp)     #PUSH
        subu    $sp, $sp, 4
        sw      $fp, 0($sp)     #PUSH
        subu    $sp, $sp, 4
        addu    $fp, $sp, [illegible]
        subu    $sp, $fp, [illegible]
        ...
        lw      $t0, [illegible]($fp)
        move    $v0, $t0
        lw      $ra, [illegible]($fp)
        move    $t0, $fp
        lw      $fp, [illegible]($fp)
        move    $sp, $t0
        jr      $ra
```

```
    .text
        .globl  main
main:
        sw      $ra, 0($sp)     #PUSH
        subu    $sp, $sp, 4
        sw      $fp, 0($sp)     #PUSH
        subu    $sp, $sp, 4
        addu    $fp, $sp, [illegible]
        subu    $sp, $fp, [illegible]
        li      $t0, 2
        sw      $t0, [illegible]($fp)
        li      $t0, 4
        sw      $t0, 0($sp)     #PUSH
        subu    $sp, $sp, 4
        li      $t0, 5
        sw      $t0, 0($sp)     #PUSH
        subu    $sp, $sp, 4
        subu    $t0, $fp, [illegible]
        sw      $t0, 0($sp)     #PUSH
        subu    $sp, $sp, 4
        jal     _foo
        move    $a0, $v0
        ...
        lw      $ra, 0($fp)
        move    $t0, $fp
        lw      $fp, -4($fp)
        move    $sp, $t0
        jr      $ra
```



Assem intermediate representation

Assem.Instr

- "assembly language instruction without register assignments"

OPER(String assem, List<Temp> dst, List<Temp> src, List<Label> jumps)

- contains a string with holes for registers indicated by `d# and `s# and holes for labels indicated by `j#
- dst and src are lists of Temps whose register assignment should fill holes
  - first entry in src is associated with `s0, second with `s1, etc.
  - first entry in dst is associated with `d0, etc.
- jumps is a list of labels for filling in label holes