CSE 4251: Compiler Construction

Ryan Stansifer
Department of Computer Sciences
Florida Institute of Technology
Melbourne, Florida USA 32901

http://www.cs.fit.edu/~ryan/

computer-presentation edition

March 26, 2020
Review

• Regular expressions to NFA
• NFA to DFA
• Nullable, first, and follow
• LL(1) parsing
• LR(0), SLR, LR(1), LALR(1) parsing
More Generally

- Definition of formal language, regular expression
- Recursive descent parsers
- Scanners versus recognizers
- Definition of grammars, parse trees, ambiguity
- Hierarchy of formal languages
Overview of Assignment 5

• Study AST for MiniJava (in support.jar)

• Add semantic actions to JavaCC parser (new, textttnew, …)

• Understand how to make an AST visitor (e.g. syntax.PrettyPrintVisitor)

• Design a symbol table (see Appel 2nd, Figure 5.7)

• Code visitor to create global symbol table

• Code visitor to perform semantic checking.
Chapter 6: Activation Records
Activation Records

Because of recursion, several invocations of a subroutine may exist simultaneously.

Each invocation of a subroutine has its own instances of the local variables lasting from subroutine entry to return.

A subroutine invocation requires the runtime system maintain information about the invocation including the values of local variables (and much more: return address, saved state of the machine, temporaries values, access to the non-local environment, ...).

This collection of information is called an activation record.
A subroutine returns only after all the function it has called have returned. Thus subroutine invocations behave in a LIFO manner. So keep activation records in a stack.
Activation Records

Appel, 2nd edition, page 118. “For historical reasons, the run-time stack usually starts at a high memory address . . .”

Runtime Organization of a Program in Memory

- The stack (local variables)
- The heap (dynamically allocated variables)
- Globals variables, program code (read only)
Static link
Dynamic link
Return address
Parameters
Local variables
Register save area
Usual Layout of Activation Record for SPARC architecture
On an architecture with register Windows, such as SPARC . . . if we choose the stack pointer to be one of the \textit{out} registers and the frame point to be the corresponding \textit{in} register, as the SPARC UNIX System V API specifies, then the \textit{save} and the \textit{restore} instructions can be used to perform the entry and exit operations, with saving registers to memory and restoring left to the register-window spill and fill trap handlers.

Muchnick, page 113.
SPARC calling sequence

1. Caller puts arguments in %o0, %o1, ..., %o5. For example:
   
   ```
   mov 1,%o0       ! pass argument 0 in the register %o0
   set LABEL,%o1   ! pass argument 1 in the register %o1
   mov %l4,%o2    ! pass argument 2 in the register %o2
   ```

2. If more than six arguments, then the caller puts them in the (caller’s) argument build area. For example:
   
   ```
   mov 6,[%sp+4*(16+6)]       ! pass argument 6 on the stack
   set LABEL,[%sp+4*(16+7)]   ! pass argument 7 on the stack
   ```

3. Caller then executes `call` instruction, which saves the `pc` in %o7 and jumps to a label. For example:
   
   ```
   call <address> ; jmpl <address>, %o7
   ```
   
   `nop`
4. Callee executes **save** instruction, which shifts the register window set and reserves space for new stack frame. For example:

```
save %sp, -(16+1+t+x)&-8, %sp
```

where \( t \) is the number of locals and temporaries, and \( x \) is the maximum arguments needed by any subroutines of the callee. Stack pointers must be double word aligned.

5. If needed, the callee saves the incoming arguments (0 through 5) into the argument build area of the caller’s frame. For example:

```
st %i0, [%fp+4*(16+0)] ! store arg 0 into stack
st %i1, [%fp+4*(16+1)] ! store arg 1 into stack
```

Local variables and temporaries of the current frame are accessed with negative offsets from the \( %fp \). For example:

```
ld [%fp-4*(1+0)], %l0 ! load local/temp 0
ld [%fp-4*(1+1)], %l1 ! load local/temp 1
```
SPARC return sequence

1. Callee puts (integer) return value in %i0. For example:
   mov 1,%i0 ! put return value in %i0

2. Callee executes restore instruction, which resets the register window set and pops frame.

3. Callee then executes ret instruction, which jumps to %i7+8 (just past the call and its delay slot in the caller). For example:
   ret ! "jmpl %i7+8, %g0" after next instruction
   restore ! restore register window set
function:

; open next register window;
; and allocate N bytes for frame
save %sp -N, %sp

; perform function, result in %i0
ret       ; jumpl %i7+8, %g0
restore   ; restore %g0, %g0, %g0

SPARC Callee
Nonlocal variable access

1. Whenever a function $f$ is called, it can be passed a pointer to the frame of the function statically enclosing $f$; this pointer is the *static link*.

2. A global array can be maintained, containing – in position $i$ – a pointer to the frame of the most recently entered procedure whose *static nesting depth* is $i$. This array is called a *display*.

3. When $g$ calls $f$, each variable of $g$ that is actually accessed by $f$ (or by any function nested inside $f$) is passed to $f$ as an extra argument. This is called *lambda lifting*. 
Nonlocal Variable Access

A local variable is accessed as follows:

\[
\begin{align*}
    r & := \text{fp} \quad \text{! relative to frame pointer} \\
    r & := \text{M}[r+\text{offset}] \quad \text{! access local variable}
\end{align*}
\]

A non-local variable at static distance two is accessed as follows:

\[
\begin{align*}
    r & := \text{fp} \quad \text{! relative to frame pointer} \\
    r & := \text{M}[r+s1] \quad \text{! follow first static link} \\
    r & := \text{M}[r+s1] \quad \text{! follow second static link} \\
    r & := \text{M}[r+\text{offset}] \quad \text{! access the variable at sd=2}
\end{align*}
\]
Nonlocal Variable Access

\[ M[fp+\( k \)] \]

\[
\text{MEM (BINOP (PLUS, new TEMP (frame.FP())), CONST (\( k \)))}
\]
line 21. prettyprint calls show
    (show’s SL is set to prettyprint’s own frame pointer)

line 15. show calls indent
    (indent’s SL is set to show’s own frame pointer)

line 13. indent calls write

line 17. show calls itself recursively

line 15. show calls indent

line 13. indent calls write
line 14. indent accesses the variable output
An abstract structure to represent frame information

```java
package frame;

/* Appel, 2nd edition, Chapter 6, page 127 */
public abstract class Frame {
    public final Label instructionPointer;
    public final List<Access> formals; // including implicit static
    public Frame (Label i, List<? extends Access> f) {
        instructionPointer = i; formals = f;
    }
    public int getNumberOfFormals () { return formals.size(); }
    abstract public Frame newFrame (Label name, List formals);
    abstract public Access allocateLocal (boolean escape);
    public Access allocateLocal () { return allocateLocal(true); }

    public abstract temp.Temp FP(); /* Chapter 7, page 143 */
}
```
package frame;
import tree.Exp;

/*
*/

public abstract class Access {
    public abstract Exp access (Exp framePtr);
}

The class Frame holds information about formal parameters and local variables allocated in this frame. To make a new frame for a function \( f \) with \( k \) formal parameters, call \texttt{newFrame}(\( f, l \)), where \( l \) is a list of \( k \) booleans: \texttt{true} for each parameter that escapes and \texttt{false} for each parameter that does not. (In MiniJava, no parameters ever escape.) The result will be a Frame object.


Calling \texttt{newFrame()} and getting a new frame might change the caller’s frame. For example, the maximum size of the argument build area may increase.
The package `temp` is obsolete and its functionality is given in its entirety as part of the `tree` package in support.jar.

```java
package temp;
public class Temp {}
public class Label {}
```
Overview of Assignment 6

- Review the visitor pattern
- Review AST for MiniJava
- Study the given IR code in chapter 7
- Fix the design of the symbol table
- Detect semantics errors missed previously
- Study chapter 6
- Advise: use a \texttt{sparc} package, but ignore the book's abstract classes
- Study chapter 7
- Code a visitor to translate to IR code
- Use or don’t use "lazy IR trees"
Chapter 7: Translation to Intermediate Code
Intermediate Representation (IR)

- IR should be easy to convert from the abstract syntax; easy to convert to assembly. It should support machine-independent optimizations.

- Often compilers use several IRs.

\[
\text{syntax} \Rightarrow \text{IR}_1 \Rightarrow \text{IR}_2 \ldots \Rightarrow \text{IR}_k \Rightarrow \text{assembly code}
\]

![Diagram showing intermediate representations](image)

Figure 6.2: A compiler might use a sequence of intermediate representations

- MiniJava compiler uses only one IR: \texttt{tree.Exp}
An intermediate language helps modularize the back end of the compiler.

\textbf{7.1. INTERMEDIATE REPRESENTATION TREES}

\textbf{FIGURE 7.1.} Compilers for five languages and four target machines: (a) without an IR, (b) with an IR.
Figure 10.3: A middle-end and its ILs simplify construction of a compiler suite that must support multiple source languages and multiple target architectures.
GCC includes two ILs, one the represents source programs at a relatively high level and another that represents machine instructions abstractly. The Microsoft compiler suite uses Common Intermediate Language (CIL) as an ILs and the Common Language Runtime (CLR) as a generic interpreter of CIL.
abstract class Exp
  class CONST (int value)
  class NAME (tree.Label label)
  class TEMP (tree.NameOfTemp temp)
  class BINOP (int binop, Exp left, Exp right)
  class MEM (Exp exp)
  class CALL (Exp fun, ExpList args)
  class RET/ESEQ (Stm stm, Exp exp)

abstract class Stm
  class MOVE (Exp dst, Exp src)
  class EVAL (Exp exp)
  class JUMP (Exp exp, List<tree.Label> targets)
  class CJUMP (int rel, Exp l, Exp r, Label t, Label f)
  class SEQ (Stm left, Stm right)
  class LABEL (tree.NameOfLabel label)

[RET is more pronouncible than ESEQ.]
**NameOfTemp** The string name of temp

**NameOfLabel** The string name of a label

This are in support.jar file in the package `tree` and have factor methods of generating unique names.

(This differs significantly from the book.)
It is almost possible to give a formal semantics to the Tree language. However, there is no provision in this language for procedure and function definitions – we can specify only the body of each function. The procedure entry and exit sequences will be added after as special “glue” that is different for each target machine.

Appel, 2nd, Section 7.1, page 139.
Intermediate Representation

Expressions which stand for the computation of some value (possibly with side effects):

**CONST(i)** The integer constant i.

**NAME(n)** The symbolic constant n corresponding to an assembly language label.

**TEMP(t)** Temporary t. A temporary in the abstract machine is similar to a register in a real machine. However, the abstract machine has an infinite number of temporaries.

**BINOP()** The application of binary operator o to operands e1 and e2. Subexpression e1 is evaluated before e2.
Intermediate Representation

The remaining expressions which stand for the computation of some value (possibly with side effects):

**MEM(e)** The contents of a word of memory at address e.

**CALL()** Procedure call.

**RET(s,e)** The statement s is evaluated for side effects, then e is evaluated for a result. Called **ESEQ(s,e)** by Appel.
Intermediate Representation

The statements of the intermediate representation which perform side effects and control flow:

MOVE(TEMP \ t, e) Store the results of evaluating e into the temporary t.

MOVE(MEM e1, e2) Store the results of evaluating e2 at the address e1.

EVAL(e) Evaluate e for it side effects and discard the result.
(Called EXP by Appel.)

JUMP(e, l) Transfer control to address e
Intermediate Representation

The remaining statements which perform side effects and control flow:

CJUMP Evaluate e1 and e2 in that order; compare using relational operator o. If the result is true, jump to t, otherwise jump to f.

SEQ(s1,s2) The statement s1 followed by s2.

LABEL(n) Define name to be the current machine code address.
Example

new SEQ (new MOVE (temp,1), // temp := 1;
new SEQ (new CJUMP(<,x,5,T,F), // if x<5 goto T else F
new SEQ (new LABEL(F), // F:
new SEQ (new MOVE (temp,0), // temp := 0;
        new LABEL(T)  // T:
))))
Example 1

Abstract Syntax

\[ \text{OpExp(PLUS, IntExp(3), IntExp(4))} \]

Intermediate Trees

\[ \text{BINOP(PLUS, CONST 3, CONST 4)} \]
Example 2

Appel’s factorial program

```java
public int ComputeFac(int num) {
    int num_aux;
    if (num < 1)
        num_aux = 1;
    else
        num_aux = num * (this.ComputeFac(num - 1));
    return num_aux;
}
```

Factorial-07.txt
Lazy IR Trees

How do you generate good IR code bottom-up in a tree traversal?

The problem is the code you want to produce depends on the context in which it is used. The boolean conjunctions are the most notable example. In some contexts the code must produce a 0 (false) or a 1 (true). In other contexts the code is used to control a test.
Lazy IR Trees

A solution

Do not produce the IR tree directly. Instead use a class that will make the tree later, when it is known which one of three contexts is desired.
Lazy IR Trees

package translate;
class LazyIRTREE {
    abstract Exp asExp();
    abstract Stm asStm();
    Stm asCond (Label t, Label f) { throw new UnsupportedOperation(); }
    public String toString () {
        return String.format ("IR: %s", asStm().toString());
    }
}

Three views of the code:

1. as an expression
2. as a statement
3. as a conditional
“The whole point of the Cx representation is that conditional expressions can be combined easily with the MiniJava operator &&.” Appel, 2nd, Section 7.2, page 149.
Lazy IR Trees

class ExpIRTree extends LazyIRTree

class StmIRTree extends LazyIRTree

abstract class Cx extends LazyIRTree
    class RelCx extends Cx /* Page 149. */
Lazy IR Trees

class ExpIRTree extends LazyIRTree {
    private final tree.Exp exp;
    ExpIRTree (tree.Exp e) { exp = e; }
    tree.Exp asExp() { return exp; }
    tree.Stm asStm() { return new tree.EVAL(exp); }
    // asCond not implemented
}

class StmIRTree extends LazyIRTree {
    private final tree.Stm stm;
    StmIRTree (tree.Stm s) { stm = s; }
    tree.Stm asStm() { return stm; }
    // asExp, asCond not implemented
}
abstract class Cx extends LazyIRTree {
    tree.Exp unEx() { /* Program 7.2, page 161.*/ }  
    abstract tree.Stm unCx (Label t, Label f);
    // unNx "left as exercise"
}

Lazy IR Trees
“Making ‘simple’ Cx expressions from Absyn comparison operators is easy with the CJUMP operator.” Appel, 2nd, Section 7.2, page 149.

class RelCx extends Cx { /* Page 149. */
    final private int relop;
    final private tree.Exp left, right;

    RelCx (int op, tree.Exp l, tree.Exp r) {
        relop = op; left = l; right = r;
    }

    public tree.Stm asCond (Label t, Label f) {
        // new tree.CJUMP
    }
}
If \( i = \text{translate}("x<5") \), then \( i.\text{asCond}(t,f) \) should be \( \text{CJUMP}(\text{LT},x,5,t,f) \).
class Rel_LIRT extends Cond_LIRT {
    final private int relop;
    final private Exp left, right;
    Rel_LIRT (final int op, final Exp l, final Exp r) {
        relop = op; left = l; right = r;
    }
    @Override
    public Stm asCond (NameOfLabel t, NameOfLabel f) {
        return new CJUMP (relop, left, right, t, f);
    }
    // This conditional test is used only for its side effects!
    public Stm asStm() {
        return new SEQ(new EVAL(left),new EVAL(right));
    }
}
This sketch of the class appears in Appel, 2nd, Section 7.2, page 150.

class IfThenElseExp extends LazyIRTree {
    private final LazyIRTree cond, e2, e3;
    Label t,f,join;
    IfThenElseExp (LazyIRTree c, LazyIRTree thenClause, LazyIRTree elseClause) {
        assert c!=null; assert thenClause!=null;
        cond = c; e2 = thenClause; e3 = elseClause;
    }
    public Exp asExp() { /* ... */ }
    public Stm asStm() { /* ... */ }
    public Stm asCond (Label tt, Label ff) { /* ... */ }
}
A Simpler Translate

“To simplify the implementation of Translate, you may do without the Ex, Nx, Cx constructors. The entire Translate module can be done with ordinary value-expressions.” Appel, page 178.
Fragments

syntax. Program

symbol table

Chapter 5

fragments

Chapter 7

Chapter 4
Exercises

7.1 Draw a picture of the IR tree that results from each of the following expressions. Assume all variables escape.

b.

let
    var i := 8
    type intArray = array of int
    var b := intArray[10] of 0
in
    b[i+1]
end