CS202 Compiler Construction

February 18, 2003

Today:

Assignment 5 suggestions

Stack frames overview

Assignment 5:

Need to build parse trees for programs

Need to build parse trees for types
C/Lake/Pond type syntax:

Primitive, ad hoc (modern PLs better)

Type annotations “broken up”, e.g.

int *a[5]

Variable a is a 5-element array of pointers to integers ([] precedence over *)

Need to combine information from both annotation positions for complete type

Left-hand annotation type basis

Need notion of “partial” type when parsing type information surrounding variables

int *a[5]

Trick: use null to be a placeholder in partially constructed types

null will occupy the place of the basis of the type term

Use insertType to elaborate type as information is encountered

/* Insert a type into the base of the type chain
   * This is invalid if there is not a dangling recursive type
   * @param newType new type to insert at the end of the chain
   *
   * public void insertType(Type newType)
   */
“Left-hand” type annotations always represent the basis type:

```plaintext
int *a[5]
```

To complete type, use insertType to provide basis for partial type

When parsing variable:

```java
base_var_decl ::= ID:x
 |
| : LakeSym token = (LakeSym)
| CUP$parser$stack.elementAt(CUP$parser$top-0);
| RESULT = new Declarator(x, token.position());
| ;

Declarator constructor sets variable type to null
```

When parsing pointer type annotation:

```java
var_decl ::= STAR var_decl:d
 |
| : RESULT = d;
| RESULT.insertType(new PrType(null));
| ;

Still using null to occupy position of type term basis
When constructing type of declaration at the top level:

\[
\text{decl ::= base\_type: \text{var\_decl}} \ SEMI
\]
\[
\begin{align*}
&\text{d.insertType(t);} \\
&\text{RESULT = new DeclaratorList(1);} \\
&\text{RESULT.add(d);} \\
&; \\
\end{align*}
\]

Combine parsed basis and partial type, to generate complete type

One last hint…

To construct a void type:

\[
\text{primitive\_type ::= VOID}
\]
\[
\begin{align*}
&\text{RESULT = VoidType.getVoidType(); }; \\
\end{align*}
\]

Also…

This assignment works; if you’re getting uncaught exceptions, revise your work

Eventually want to generate code

Actually runs on hardware

Start considering \textit{runtime} (dynamic properties)

Today: activation frames and call stacks

Review of 101/222
High-level view of low-level details

For now, assume working with IDs
   Can use symbol table to store/look up values

Don’t worry about actual memory layout
details (yet)

During runtime, need to keep track of
values associated with function calls

- Function arguments
- Local variables
- Results
- Where to return to

_Activation records maintain this
information

One record per dynamic activation

One recursive function can have multiple
activations

Need to organize activation records…
Observation: In Algol-like languages, function calls form a tree

```
int g(int x) { return x; }
int h(int x) { return x; }
in f(int x) = { g(x); h(x) }
```

Upon completion of one each branch, control is returned to calling function

*Stack* can be used to *push* on new activations, *popped* upon completion

Popping call stack restores state of calling function

```
\[
\begin{align*}
f(a) \\
g(a) \\
h(a)
\end{align*}
\]
```

Modern langs: stack based local vars

```
frame pointer
```
```
```
```
```
```
```
```
```
Before calling function, evaluate args,

push args on stack

\[ \begin{array}{c}
\text{fp} \\
\text{sp} \\
\text{args} \\
\text{locals} \\
\text{arg2} \\
\text{arg1} \\
\end{array} \]

Calling function

push args on stack
jump to function
save return addr

\[ \begin{array}{c}
\text{fp} \\
\text{sp} \\
\text{args} \\
\text{locals} \\
\text{arg2} \\
\text{arg1} \\
\text{return pc} \\
\end{array} \]

Function establishes local frame

push args on stack
jump to function
save return addr
move fp to sp

\[ \begin{array}{c}
\text{fp} \\
\text{sp} \\
\text{args} \\
\text{locals} \\
\text{arg2} \\
\text{arg1} \\
\text{return pc} \\
\end{array} \]
Function establishes local frame
- push args on stack
- jump to function
- save return addr
- move fp to sp
- allocate locals

When returning, function
- restores fp, sp
- remember return pc

When returning, function
- restores fp, sp
- remember return pc
When returning, function
restores fp, sp
remember return pc
jump to return

Finally
restores fp, sp
remember return pc
jump to return
pop args

Simple nested blocks:
sp unchanged
Function entry allocates for all blocks

Sibling blocks share memory
Nested blocks simple
  Pre-allocate enough for all
  Overlap siblings storage

Nested functions hard
  Each has own fp
  Need to access variables in outer
  Only known as +/- (fp)
  How are they addressed?

Not legal in C/Lake/Pond, but might look like:

```c
int f(int a)
{
    int b;
    int g(int c)
    { int d = c+b-a; return d; }
    b = 7;
    return g(3);
}
```
Nested functions have extra param
fp of enclosing function

\[ g(3, fp); \]

Extra param to g
Call it \( f_{fp} \)
Gives pointer to parent fp
Gives access to parent vars/args

\[ g(int c, f_{fp}); \]

Assume in f:
- \( a \) was \( 4(fp) \)
- \( b \) was \( -32(fp) \)

Then in g:
- \( a \) is \( 4(f_{fp}) \)
- \( b \) is \( -32(f_{fp}) \)
Assume \( f_{fp} \) is \( 8(fp) \): 
- \( a \) is \( 4(8(fp)) \)
- \( b \) is \( -32(8(fp)) \)
Fact: everything on stack is expensive  
(must access memory)

Registers are much cheaper  
modern processors provide many

Assume unlimited supply of registers  
For each function  
Will restrict this later

Most args, return address in regs

Registers shared across functions  
What happens during call? Either:

Callee-save registers  
Saved by called function  
Caller-save registers  
Caller saves

No clear division of responsibility; choice  
defined by architecture and convention

General convention:

If values are to be maintained for use  
after function call, use callee-save  
Callee saves all

If some values can be discarded upon  
function call, use caller-save  
Caller can select which to save

Use together to minimize saving
Need to save registers in call, \textit{on stack}:

- Push args on stack
- Jump to function
- Save return addr
- Move fp to sp
- Save registers
- Allocate locals

Some args, locals still on stack:

- Too many to fit in registers
- Passed by reference
- Too big to fit in a register
- An unknown size
- An array and accessed by addr arithmetic
- Operand of an address of operator
- Visible to other threads/processes (volatile)

Will eventually be generating code for Sparc:

Sparc presents 32 registers:
- 8 global registers g0-g7
- 8 input i0-i7
- 8 local l0-l7
- 8 output o0-o7

Hardware has more:
- On call, slide which regs program sees
called sliding register window
Consider big array of registers
(fewer registers shown for simplicity)

After save, window slides down

Eventually we run out of registers
Save saves top of window and wraps registers

writes old values of A-D registers to stack save area

------------------

Saves only when call stack exceeds number of registers
Saves up higher in stack
Restores only on underflow

If n sets of registers on stack,
bottom n levels of call stack never write registers to memory

------------------