CS 536 Announcements for Tuesday, April 12, 2022

Last Time
- type checking
- type-system concepts
- type-system vocabulary
- minim
  - type rules
  - how to apply type rules

Today
- runtime environments
- runtime storage layout
- activation records
- static allocation
- stack allocation
- activation records revisited
- what happens on function call, entry, return

Next Time
- parameter passing

Type checking in minim

minim's type system
- primitive types: int bool void
- type constructors: struct
- coercion: a bool cannot be used as an int is expected and vice versa

Type errors in minim
- operators applied to operands of wrong type
- expressions that, because of context, must be a particular type but are not
- related to function calls

Type checking
- Recursively walks the AST to
  - determine the type of each expression and sub-expression using the type rules of the language
  - find type errors
- Add a typeCheck method to AST nodes
Type checking: errors

Goals:

- report as many \textit{distinct} errors as possible
- don't report \textit{same} error multiple times – avoid error cascading

Introduce internal \texttt{error} type

- when type incompatibility is discovered
  - report the error
  - pass \texttt{error} up the tree
- when a type check gets \texttt{error} as an operand
  - don't (re)report an error
  - pass \texttt{error} up the tree

Example:

```c
int a;
bool b;
a = true + 1 + 2 + b;
b = 2;
```
**Back to the big picture**
Before code generation, we need to consider the runtime environment:

Program piggybacks on the operating system (OS)

Compiler must use runtime environment as best it can

We need to create/impose conventions on the way our program accesses memory

**Issues to consider**

**Variables**
- How are they stored?
- What happens when a variable's value is needed?

**How do functions work?**
- What information should be stored for each function?
- What should happen when client code calls a function?
- What should happen when a function is entered?
- What should happened when a function returns?
Storage layout

Memory layout: static allocation

Region for global memory

One "frame" for each procedure
- memory "slot" for each local, parameter
- memory "slot" for caller
Memory layout: stack allocation

Allocate one activation record (AR) per invocation
- use the stack
- push a new AR on function entry
- pop AR on function exit
- to reduce the size, put static data in the global area

Stack size not known at compile time
- don't know (at compile-time) how many ARs there will be
- size of local variables may not be known
- each AR keeps track of:
  - its boundaries
  - the previous AR's boundaries

Activation record keeps track of
- local variables
- info about the call made by the caller
  - data context

  - control context

Non-local dynamic memory

Don't always want all data allocated in a function call to disappear on return

Don't know how much space we'll need

The Heap
- region of memory independent of the stack
- allocated according to calls in the program
- how is memory "given back"?
Function calls

Instruction pointer ($ip$) tracks the line (address) of code that it is executing
- if $ip$ points to code generated for some function, we'll say we are *in* that function

caller = function doing the invocation
callee = function being invoked

Activation records revisited
Function entry: caller responsibilities
Store the *caller-saved* registers in its own AR
Set up the actual parameters
  - set aside slot for the return value
  - push parameters onto the stack
Copy return address out of $ip
Jump to first instruction of the callee

Function entry: callee responsibilities
Save $fp (it will need to be restored when the callee returns)
Update the base of the new AR to be the end of the old AR
Save *callee-saved* registers (if necessary)
Make space for locals

Function exit: callee responsibilities
Set the return value
Restore callee-saved registers
Grab stored return address
Restore *old* $sp
Restore *old* $fp
Jump to the stored return address

Function exit: caller responsibilities
Pop the return value (or copy from register)
Restore caller-saved registers
Example

```c
int summation(int max) {
    int sum = 1;
    for (int k = 1; k <= max; k++) {
        sum += k;
    }
    return sum;
}

void main() {
    int x = summation(4);
    print(x);
}
```