Lecture 4

Note: Some stuff done in lecture 4 uses notes for lecture 3, specfically:

1. Remind of LL grammar example
2. frl3 5.2 left recursion inkl "slide 2"
3. FIRST(P) concept, computation w and w/o nullables
   FIRST(P) relation to LL grammars
4. frl3 5.3 Eliminating left recursion
5. frl3 6 LR sketch

1 Administrativia

Project launch Tue 10-12 in L1 DKM 30. 2 bonus points for attendence.

Project pages and Tigris pages live.

Schedule now finalised until mid-April.

As you have noticed, lectures are somewhat terse. 8 lectures not enough for a detailed exposition of course material. Don't forget to read theory!

2 Today

More on syntax tree abstraction levels: Removing grammatical elements using some algorithms.

Symbol tables.

Type checking.

Perhaps some words on frame-layout.

3 More on compiler tools

(Move this to proj launch)

Java-world tools: JavaCC, SableCC, JavaCup
C/C++ world tools: flex, yacc, bison
Haskell: alex, happy

These tools allow a grammar–of some class–to be specified. Syntax differs slightly. Some are LL(k), other are LR(1) (actually LALR).

Allows grammar as productions, meta information like precedence, action-for-matched-production.

The tools either have a hardwired action when a production is chosen (SableCC) or allow us to specify an action.

In SableCC, a concrete parse tree generator is hardwired. In JavaCC, one may specify own actions, but with JJTree one may get concrete parse tree.

4 Semantical analysis

We have lexical analysis, with a perspective of a single token. By repeatedly calling the lexer until EOF, we will get any lexical error.

Then we have parsing, checking the program's conformance to a (context-free) grammar representation of our language. If the parser terminates without errors, the program is *syntactically* correct.

Which errors remain? Types have not yet been checked! Lots of buggy programs have been accepted.

Now, use semantical analysis, to handle the final phase in checking formal program correctness. (Now just algorithm bugs, implementation bugs, type overflows, out-of-bounds bugs, ... remain).

Note: We are still in the *compiler front end*, i.e., the source language parts.

Focus is on languages with block structure which allows declarations at (each) block start. We need to create something for vardecls visible for scope. Lots of freedom how to do this!

Consider some simple example.

Get formal:

Goals:
(1) Report used-but-undeclared objects
(2) Report type errors
(3) Set things up for later passes: bind use to decl to

Typed things:
(1) Explicitly declared variables
(2) Formal parameters
(4) Methods
(5) Types ("typedef") (not really an object...)
(6) Constants
All things but last are identifiers.

Kinds of types:
(1) Basic types: int, long, float, double, boolean, ...
(2) Other types: class, struct, method

Type of struct determined by subtypes (and their subtypes...):
(1) Field names+types relevant for checking Id.Id expressions.
(2) Exact (recursive) type matching might imply type compatibility (in some languages).

Type of classes determined similar to structs, except no compatibility between identical classes.

Scoping and visibility:
(1) Nesting
(2) Subtypes: objects within a class

How do we maintain *symbol tables*?
(1) Scoping { int i; {int i; { ... } } } makes requirements.
(2) Compound types (structs, classes) require symbol table (hierarchy)

Show example.

Some symbol table implementation alternatives:
(1) Create a new symbol table for each {. remove for }. Maintain ST:s as linked list.
(2) Have just one symbol table active, insert after {. undo insertions at }

Tradeoff: Do we have many symbol references, or deep nesting?

For efficiency: map each Id to a unique number, e.g., a ref to a string hash table.

Back to errors.

Source program long gone, how do we report errors to user? A: We need to store line numbers in syntax tree.

5 Activation records

It is the job of the compiler-generated assembly program to maintain the memory state, i.e., stack and heap.

So: The compiler needs to model the runtime environment, and generate code for it.

For JVM, most of this job done for us, though.

Heap usually maintained with library assistance ("malloc").

The stack is main abstraction for method calls

- Typically grows downwards
- Operations: PUSH and POP
- Stack pointer points at last item PUSHed
- Stack pointer is special CPU register
- Operating systems supports the stack

5.1 Activation records

Activation record = data a method needs during execution and for returning to its caller

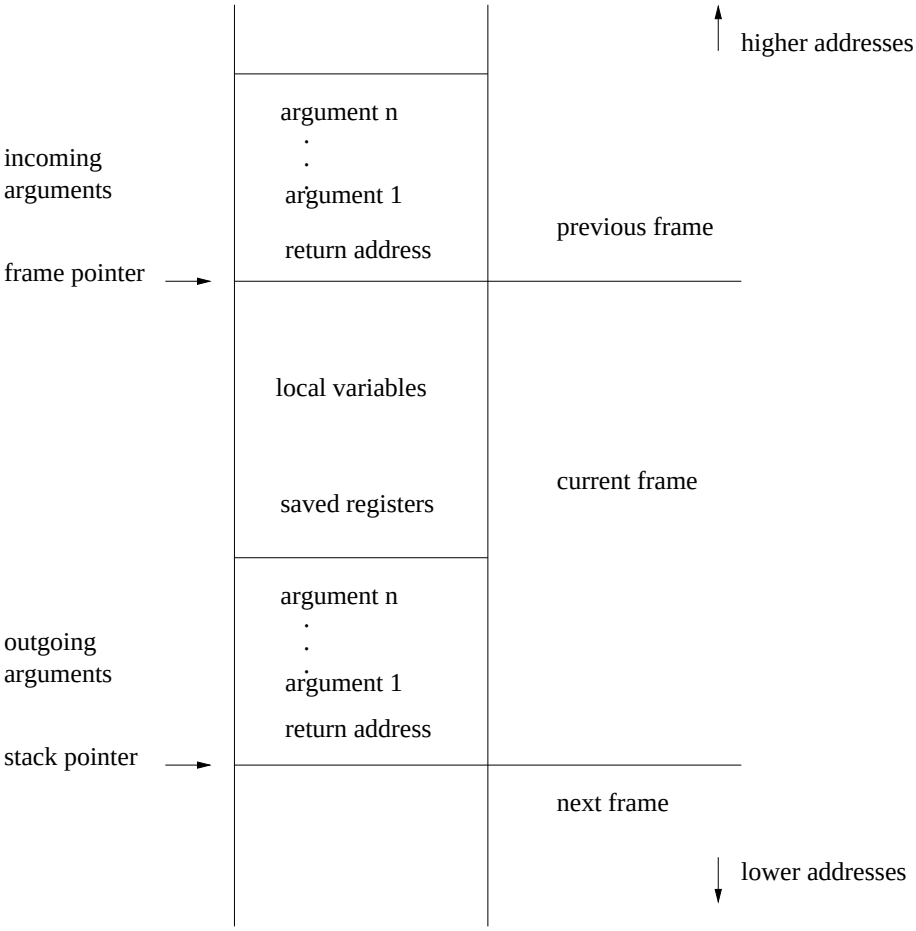
Activation records live on the stack

In fact, activation records are the only thing there

Contents:

- Incoming parameters
- Return address
- Register save area
- Space for local variables
- Space for compiler-generated temporaries
- (Static link)

.


higher addresses
argument n
incoming
arguments
argument 1
return address
previous frame
frame pointer
local variables
current frame
saved registers
argument n
outgoing
arguments
argument 1
return address
stack pointer
next frame
lower addresses

5.2 Managing activation records

The generated code must manage activation records

For JVM: Higher abstraction level—details in its documentation

For ASM: Lower abstraction level—follow "ABI"

- Compiler computes exact layout
- Compiler generates instructions for adjusting stack pointer, saving registers, updating static link, etc.

6 Final words

Note: next week L1 Tue, D3 Thu
After that, 4 weeks without komp lectures!
Hack away, come to progress seminars (with progress for bonus pts)