Programming Language Implementation X

In this lecture we will look at **code generation** and in particular at

- **runtime environment**
- **intermediate code generation**
- **register allocation**

The material is (loosely) based on Aho et al Chapters 7, 8 and 9.

Run-time environment

To understand code generation one needs to understand what should happen at run-time. In particular we need to understand **allocation** and **deallocation** of data objects. This is managed by the **run-time support** package.

The design of the **run-time support** package is influenced by the HL language it supports.

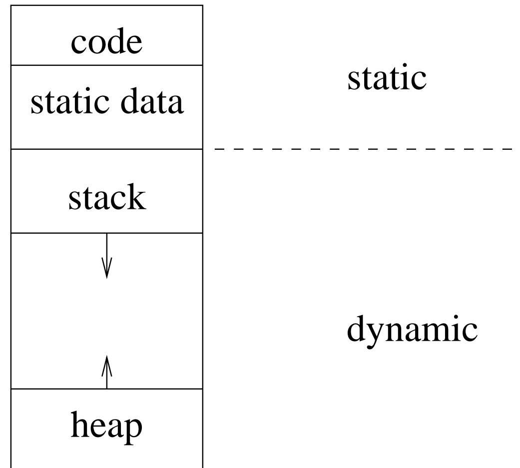
FORTRAN: The size of data structures is determined at compile time and sub-programs cannot be recursive.

Pascal family: Storage may be allocated dynamically and procedures can be passed as parameters and called recursively.

Functional and Logic languages: Allocation and deallocation of storage is invisible to the programmer.

We shall focus on the Pascal family.

Activation Records



Memory is split into
– **program code**
– **static data**
– **stack**
– **heap**.

An **activation record** is pushed on to the stack when a procedure is called and popped off when control returns from the procedure.

The size of the activation record depends on the procedure's parameters and local variables. It may not be known at compile-time. **Why?**

Activation Records (Cont)

result if function
dynamic link
static link
return address
saved environment
parameters
local variables
temporary variables

The activation record might contain:

- A location for the result in the case it is a function.
- A pointer to the stack frame of the calling procedure (ie the **dynamic predecessor**).
- A pointer to the stack frame of the textually surrounding procedure (ie the **static predecessor**).
- The return address for the calling procedure.
- Environment information such as register values which need to be restored on return.
- Memory locations for the parameters.
- Memory locations for the local variables.
- Memory locations for the temporary variables generated in expression evaluation.

Normally we keep the **stack top (ST)** and the **local base (LB)** in registers.

Procedure Invocation

Calling a procedure/function

- Evaluates arguments and assigns values to the parameters.
- Sets the **link** data (ie dynamic and static predecessor and the return address.
- Jumps to procedure entry (found in some static table).

Returning from a procedure/function

- Restore the calling environment.
- Jump to the return address.
- If there is a return value, copy to temporary variable.

Static Links

The obvious question is why do we need a **static link** as well as a **dynamic link**?

Give the stack for the Cascal program:

```
program h {
  int i, j;

  int function p {
     int x;

     int function r {
       int y;
       j := j+1;
     }

     if i>0 then {
       i:= i-1;
       p();
       pr:  }
      else {
       r();
       rr: }
  }

  i := 2;
  j := 1;
  p();
  p2r: }
```

Static Links (Cont.)

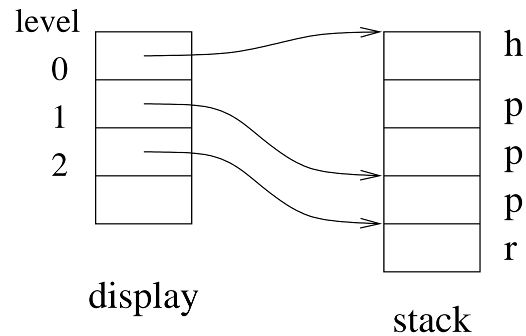
The **textual level** of a procedure (block) is the number of procedures surrounding it.

In a Pascal like language a procedure at level K can only call procedure at level $\leq K + 1$.

To access a variable at offset x level j from a procedure at level i we follow $i - j$ static links up the stack to find the right activation record and then go to off set x.

Exercise: How do you compute the value of the static link in the procedure being called?

Displays



We can avoid the unravelling of static links by maintaining an array of registers indexed by textual level which point to the apropriate activation record.

This is called a **display**.

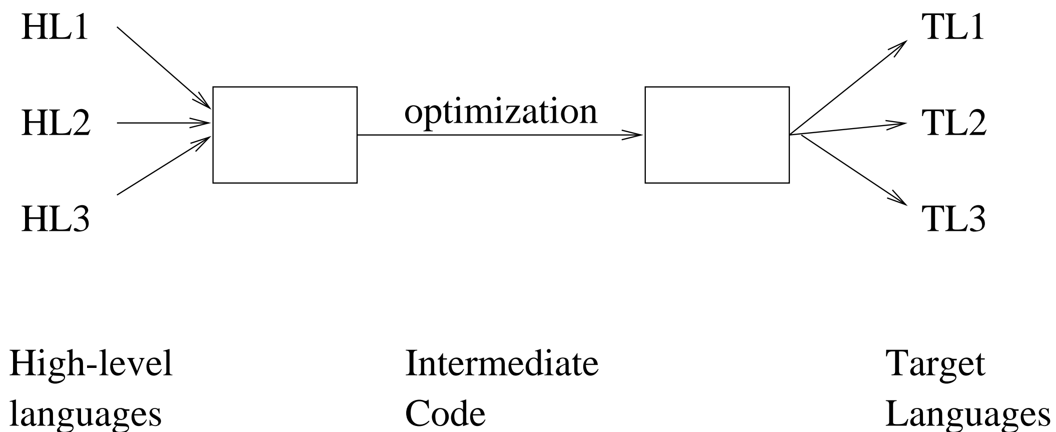
If i is the level of the called procedure we need to set $display[i]$ on call and restore its value on return from the call.

Code Generation

The aim is to produce code which is:
– **efficient**
– **compact**
– uses **registers** wisely
– is **correct**.

Typical steps in code generation are:

- **Intermediate code generation**
- **Optimization** (this is optional)
- **Target code generation**



Using a machine-independent intermediate language is good because:
– **retargeting** is facilitated
– facilitates machine-independent **code optimization**

Abstract Machines / Virtual Machines

Instead of compiling directly into machine code, the target is often a virtual machine.

A virtual machine is essentially an interpreter for a virtual language that runs on the target machine.

The virtual language provides an intermediate level between high-level language and native machine code that is specifically designed for a particular class of languages. (e.g. procedural, object-oriented, logic, functional) and provides the basic data structures and basic control mechanisms in these languages.

As a consequence, translation into virtual machine code is easier. Native machine code is more complex, requires explicit data structure and address management, provides only very simple forms of control structures and requires more optimization.

The main arguments for using a virtual machine (such as the JVM) are

- higher protability
- security (easier encapsulation)

However, native code is typically much faster.

Intermediate Code

One common form of intermediate code is **three address** code. This is a sequence of statements of form

```
x := y <op> z
```

Thus a complex source language expression like

```
x := y + u * z
```

will give rise to the statements

```
t1 := u * z
x := y + t1
```

Three address statements are similar to assembly code:

- **Assignment statements** of the above form.
- **Assignment statements** of the form

  ```
  x := <op> z
  ```

  where `<op>` is a unary operation.
- **Copy statements** of the form

  ```
  x := z
  ```

- **Unconditional jumps** of form

```
goto L
```

- **Conditional jumps** of form

```
if x <relop> y goto L
```

- **Parameter setting** statements and **procedure call** and **return y**.

```
param x1
param x2
  ...
param xn
call p, n
```

- **Indexed assignments** of form

```
x := y[i]
y[i] := x
```

- **Address** and **pointer assignments** of form

```
x := &y
x := *y
*x :=y
```

This is only one possible intermediate code. See Chapter 8 of Aho et al for more details.

Intermediate Code Generation

It is straightforward to use attribute grammars to construct intermediate code.

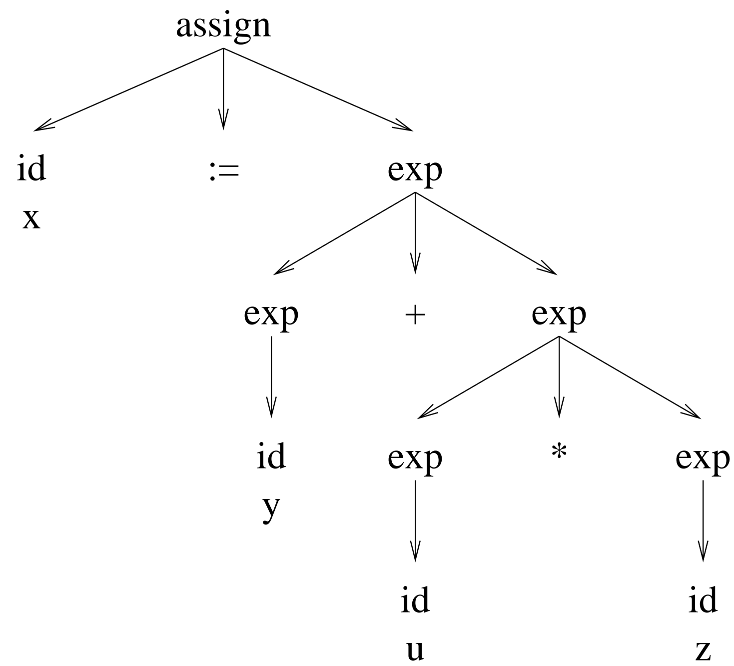
Production	Semantic Rules
$assgn \rightarrow \mathbf{id} := exp$	$assgn.code :=$ $exp.code @ gen(\mathbf{id}.place\ :=\ exp.place)$
$exp_0 \rightarrow exp_1 + exp_2$	$exp_0.place := newtemp$ $exp_0.code :=$ $exp_1.code @ exp_2.code @$ $gen(exp_0.place\ :=\ exp_1.place\ +\ exp_2.place)$
$exp_0 \rightarrow exp_1 * exp_2$	$exp_0.place := newtemp$ $exp_0.code :=$ $exp_1.code @ exp_2.code @$ $gen(exp_0.place\ :=\ exp_1.place\ *\ exp_2.place)$
$exp_0 \rightarrow (exp_1)$	$exp_0.place := exp_1.place$ $exp_0.code := exp_1.code$
$exp_0 \rightarrow \mathbf{id}$	$exp_0.place := \mathbf{id}.place$ $exp_0.code :=$ nil

Intermediate Code Generation (Cont)

For example

```
x := y + u * z
```

will give rise to:



Target Code Generation

Instruction selection

If we do not care about efficiency it is usually easy to generate target code from each three address statement.

Unfortunately we usually do care about efficiency!

Register allocation

It is better to use register operands. Use of registers involves two steps

- **Register allocation** in which we select the variables to be stored in registers.
- **register assignment** in which we select the specific registers.

We note that optimal assignment is NP-hard.

Register Allocation

In fact register allocation can be viewed as k-**graph colouring**.

In the graph, variables are nodes and nodes are connected if they are alive at the same time. If we have k registers we try to color the graph nodes so that no two connected nodes have the same color using only k colours.

Consider the intermediate code

```
 <x,y alive>
t1 := x + 1
t2 := y + t1
t3 := y * t2
z := x + t3
 <x,z alive>
```

What is the register allocation graph?

What is the minimal number of registers needed?

Peephole Optimization

Peephole optimization shifts an inspection window over the generated code to discover code segements that can be optimized locally. This is normally used for machine-dependant optimization.

Assuming that the target machine has an increment instruction INC working on a memory location, 9-11 can be optimized to the single instruction INC 19.

label	address	instruction	operand		
...					
	3	STORE	19	count	
	4	LOADC	1		
	5	STORE	20	result	
Label1	6	LOAD	19	count	
	7	SUB	21	value	
	8	JUMPGE	16	Label2	
	9	LOAD	19	count	
	10	ADDC	1		$\Rightarrow$
	11	STORE	19	count	
	12	LOAD	20	result	
	13	MUL	19	count	

Such code modifications have to be done prior to assembly.

Summary

We have looked at

- runtime environment
- intermediate code generation
- register allocation
- peephole optimization

Homework

- Read Chapters 7, 8 and 9 of Aho et al.
- Give attribute rules to generate code for a **while** loop and a procedure call (first give the grammar!).