Runtime Support

Runtime Environments

- **Objective**:
  - Managing the relationship between names in the program and data objects at runtime.
- Runtime memory contains:
  - generated target code
  - data objects: constants, static/local data, dynamic data
  - procedure activations: activation records (execution of a procedure is referred to as an activation of the procedure)

Runtime Environments

140

Design of runtime environment influenced by questions like the following:

- May storage be allocated dynamically under program control.
- Must storage be deallocated explicitly.
- May procedures be recursive.
- How are parameters passed when a procedure is called.
- What happens to the values of local names when control returns from an activation of a procedure.
- etc.

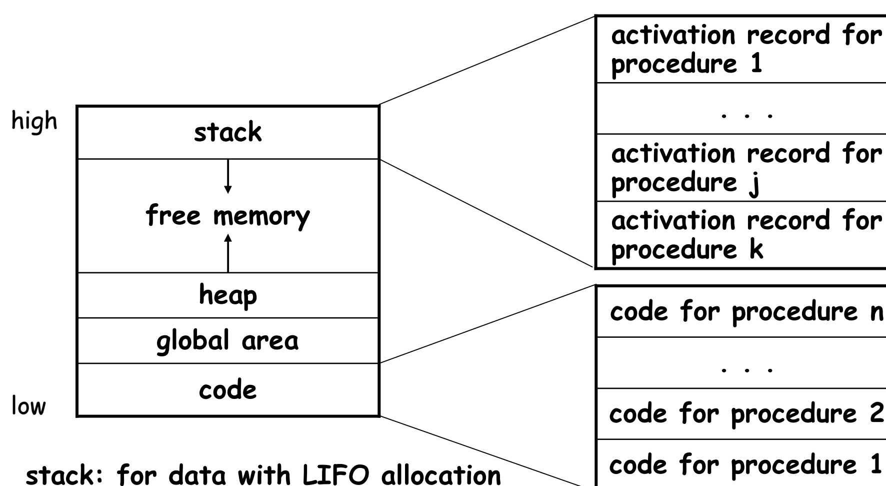
Important issues:

- Interoperability: procedure call standards, data layout
- System standards (application binary interface)

Runtime Environments

141

Typical runtime memory subdivision for stack-based approach:



stack: for data with LIFO allocation

heap: dynamically allocated data without LIFO strategy

Allocating Activation Records

1. Stack allocation
   - AR corresponds to lifetime of procedure
   - calls/returns balanced – LIFO strategy
   - e.g. Ada, C, Java
2. Static allocation
   - languages without recursion
   - for each procedure statically single AR – can be memory expensive!
   - e.g. Fortran 77
3. etc.

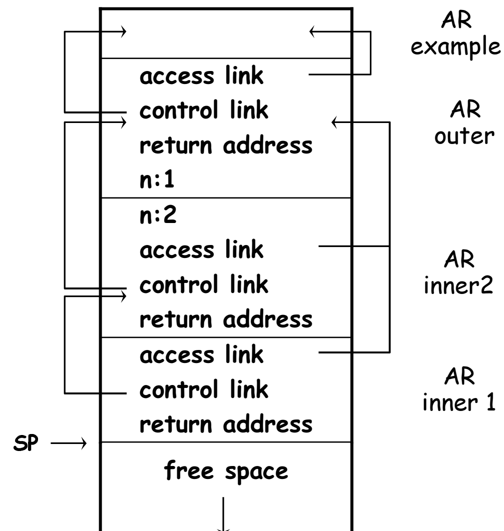
Activation Record

- Space for local data.
- Space for local temporaries needed for evaluating expressions.
- Space for actual parameters.
- Space for the returned value used by the callee to return a value to the caller.
- Space for bookkeeping information:
  - machine status information just before the procedure is called (PC, registers)
  - control link: points to the activation record of the caller
  - access link: access to non-local names in case of block structured declarations with the most closely nested rule.

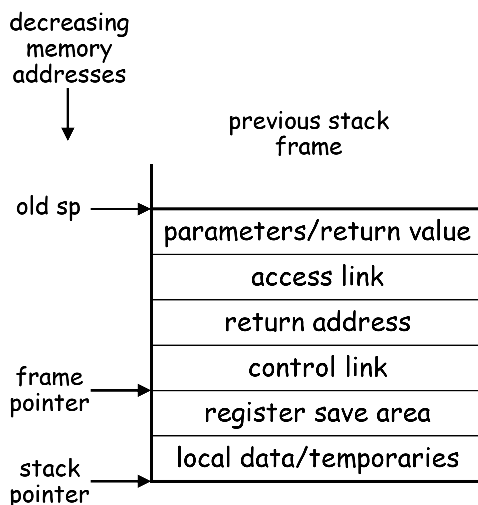
AR: Access Link and Control Link

Pascal program:

```
program example;

procedure outer;
var n:integer;

  procedure inner1;
  begin writeln(n); end;

  procedure inner2(n:integer);
  begin inner1; end;

begin n:=1; inner2(2); end;

begin (* main *)
  outer;
end.
```



Activation Record



- parameters next to AR of caller (access without knowing locals possible)
- fp points to control link
- access link (static link): optional, points to control link, not required for C
- control link (dynamic link): points to control link of previous AR
- register save area: by caller or callee
- fp alone or sp alone or both (e.g. alloca-fcn: not ISO-C)

Parameter Passing

Call site: actual arguments; Procedure: formal parameters

1. call by value
   - copying argument's value to parameter
   - large arrays?
2. call by reference
   - address of argument is passed
   - no copying
   - what about arguments in registers or constants?
3. call by result: out parameters, value copied to argument
4. call by value-result: in-out parameters, value copied in to parameter and copied back out to argument
5. call by name (Algol60, unpopular)

Saving Registers

Issue: Any register value that caller expects to survive must be saved to memory.

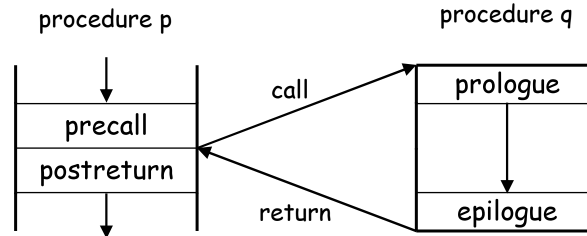
1. **Caller saves**:
   - caller knows live registers
   - caller performs the task
2. **Callee saves**:
   - callee knows registers which will be used
   - callee performs the task

Which approach is better?

Strategy in between:

- one half of register set: caller saves
- other half: callee saves

Procedure Calls: Call Sequence, Return Sequence



1. **Precall**: starts constructing callee's environment at call site.
2. **Prologue**: completes constructing its environment on entry to procedure.
3. **Epilogue**: starts reconstructing caller's environment on exit from procedure.
4. **Postreturn**: completes reconstructing its environment at call site.

Precall, Prologue, Epilogue, Postreturn

1. Precall:
   a) each argument evaluated and put in register or stack location
   b) address of the code of callee is determined (if not at compile time or link time)
   c) registers in use and saved by caller stored to memory
   d) if needed: access link of callee is computed
   e) return address is saved and branch to callee executed
2. Prologue:
   a) the old frame pointer is saved in the control link
   b) registers used by callee and saved by callee are stored to memory
   c) display is constructed (if used)

Precall, Prologue, Epilogue, Postreturn (cont'd)

3. Epilogue:
   a) registers saved by callee are restored from memory
   b) old frame pointer is recovered
   c) return value (if any) put at appropriate place
   d) branch to return address is executed
4. Postreturn:
   a) registers saved by caller are restored from memory
   b) returned value is used

Passing Parameters

1. Parameters in registers
   - arguments assigned to registers
   - usually for architectures with large register files
2. Parameters on runtime stack
   - arguments pushed onto runtime stack
   - architectures with few registers like Intel 386

Saving and restoring registers:
- Look for architecture support

Code Generation

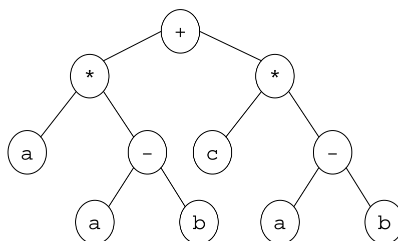
Intermediate CG: 3-Address Code from AST

Three-adress code:

- linearized representation of AST
- unique temporary names for interior nodes of AST (simple approach: sometimes even for leaves)

Example:

AST:



Code:

```
t1 := a
t2 := a - b
t3 := t1 * t2
t4 := c
t5 := a - b
t6 := t4 * t5
t7 := t3 + t6
```

Comments on Temporary Names

154

- Previous approach: Whenever a temorary is needed, a unique name is newly created.
- However: Temporary names can be shared, if not used simultaneously.

```
t1 := a
t2 := a - b
t3 := t1 * t2
t4 := c
t5 := a - b
t6 := t4 * t5
t7 := t3 + t6
```

```
t1 := a
t2 := a - b
t1 := t1 * t2
t2 := c
t3 := a - b
t2 := t2 * t3
t1 := t1 + t2
```

- Efficient utilization of registers important for good code.
- It is tried to place temporaries in registers.

Register Allocation

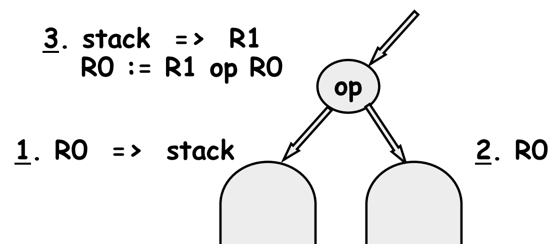
155

- Values in registers are accessed faster than in memory.
- **Goal of register allocation**:
  - Try to keep frequently used values in registers and reduce in this way memory accesses.
- Intermediate code can assume "infinite" number of "virtual registers".
- **Register allocation**:
  - Determine which virtual registers will be mapped to physical registers at any given program point.
- When a register is needed but all registers are in use, one register is freed by storing its contents in memory - this is called **spilling**.
- Note that register allocation is an NP-complete problem.

Calculating Expressions with 2 Registers

Simple approach – usage of temporary stack ST:

- **evaluate left branch to R0**
- **save left value: ST[top] := R0; top++;**
- **evaluate right branch to R0**
- **restore left value: R1 := ST[top]; top--;**
- **handle current node:**
  R0 := R1 op R0



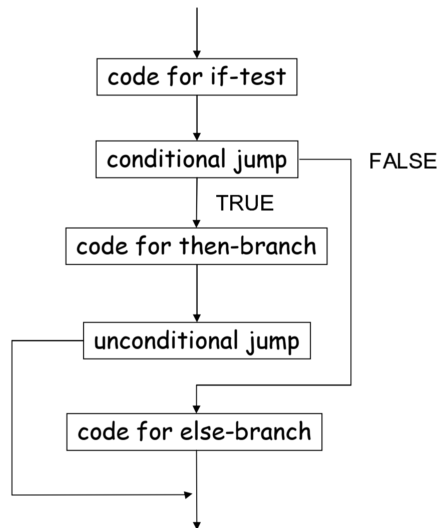
Code Generation

- Different approaches exist for code generation.
- Approach taken here: postorder traversal of syntax tree.
- Basic algorithm for syntax tree node with two children:

```
procedure genCode (T:treenode)
{
  if (T ≠ NULL)
    {
      generate code prior to left child;
      genCode(T->left);
      generate code prior to right child;
      genCode(T->right);
      generate code for T;
    }
}
```

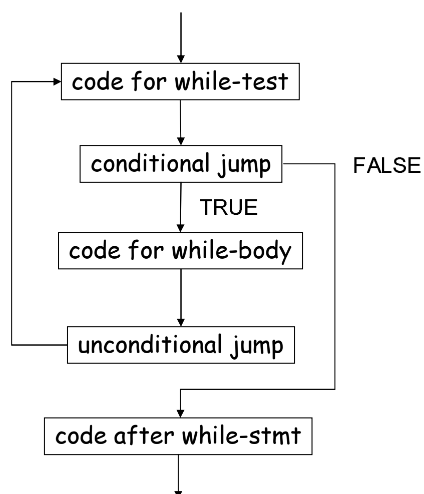
Code Generation

- Typical code arrangement for if-statement:



Code Generation

- Typical code arrangement for while-statement:



Assembly Code

For information about

- NASM (80x86 assembler) and
- virtual machine

see lecture notes of lab course.