

Compilation

0368-3133 (Semester A, 2013/14)

Lecture 8: Activation Records + Register Allocation

Noam Rinetzky

Slides credit: Roman Manevich, Mooly Sagiv and Eran Yahav

1

What is a Compiler?

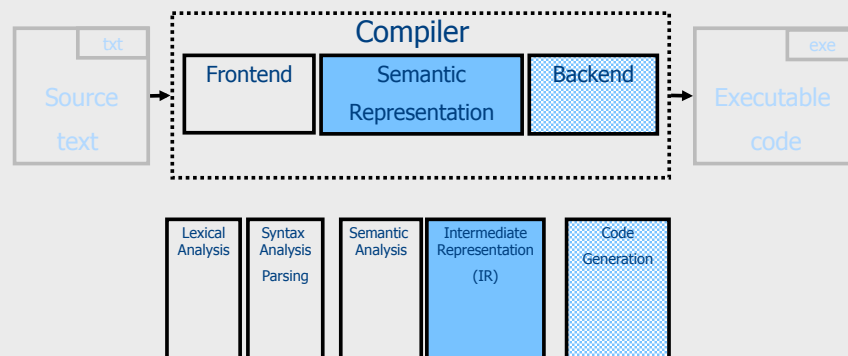
“A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).”

The most common reason for wanting to transform source code is to create an executable program.”

--Wikipedia

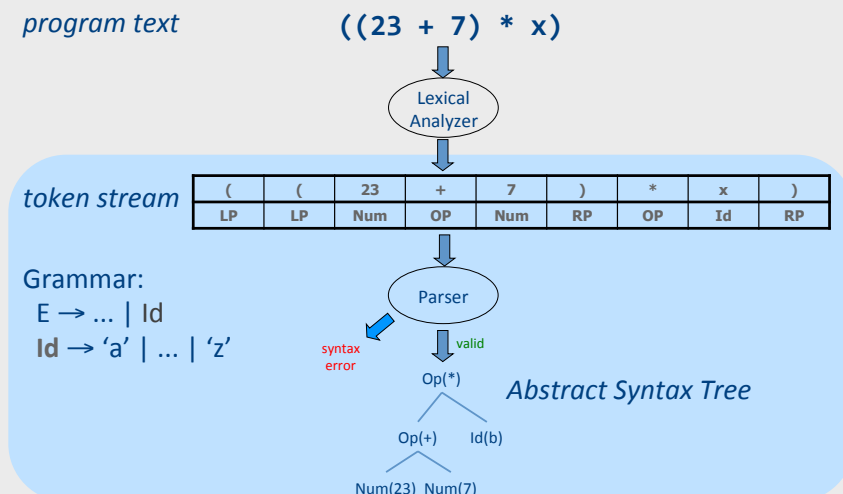
2

Conceptual Structure of a Compiler



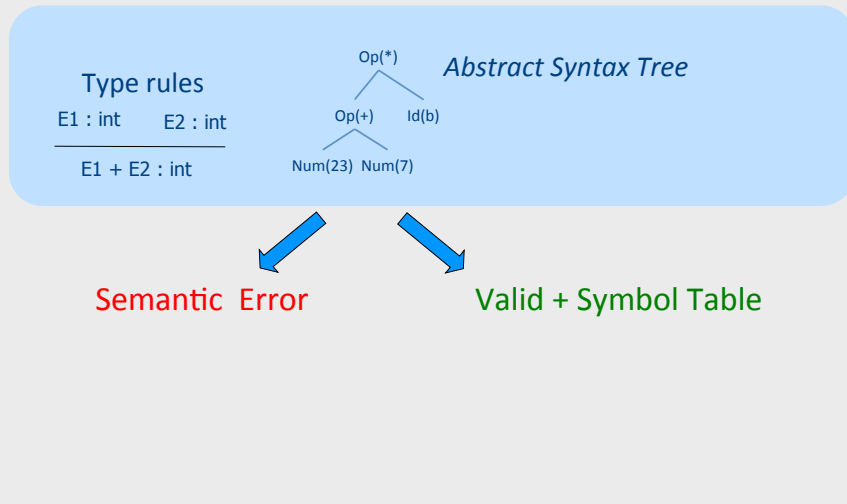
3

From scanning to parsing



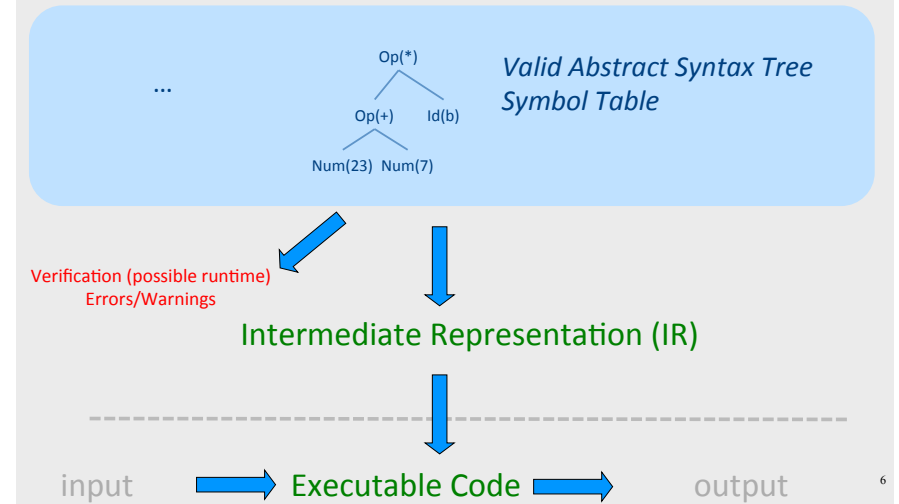
4

Context Analysis



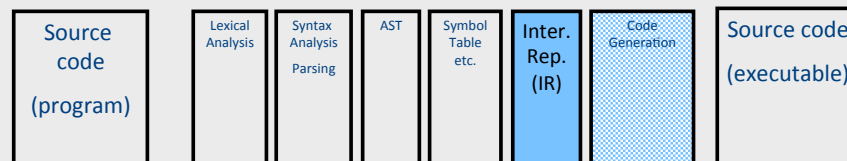
5

Code Generation



6

Code Generation: IR

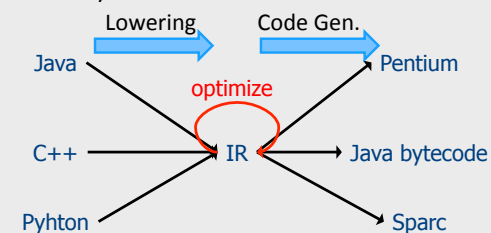


- Translating from abstract syntax (AST) to intermediate representation (IR)
 - **Three-Address Code**
 - Primitive statements, control flow, procedure calls

7

Intermediate representation

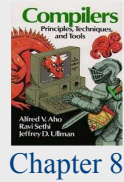
- A language that is between the source language and the target language – not specific to any machine
- Goal 1: retargeting compiler components for different source languages/target machines
- Goal 2: machine-independent optimizer
 - Narrow interface: small number of node types (instructions)



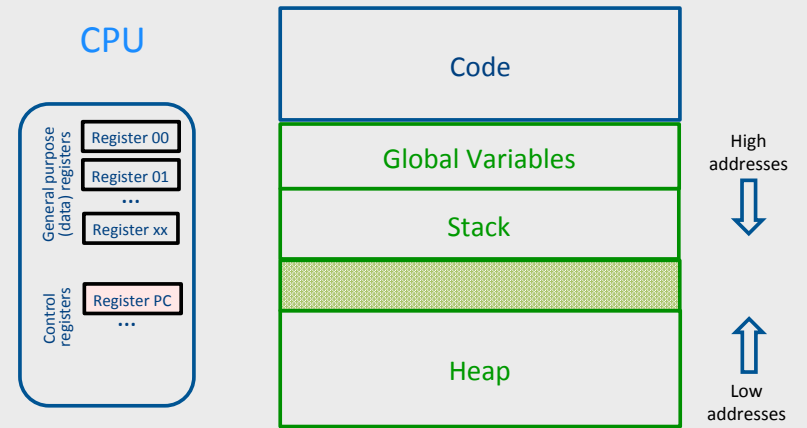
8

Three-Address Code IR

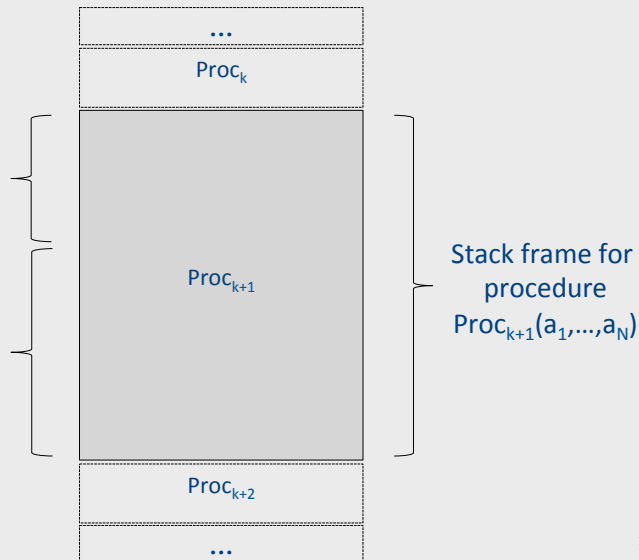
- A popular form of IR
- High-level assembly where instructions have at most three operands



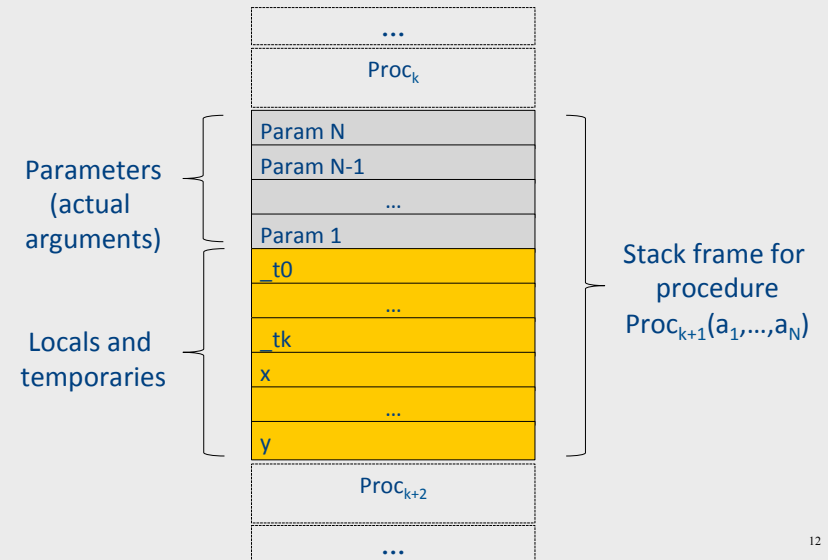
Abstract Register Machine



Abstract Activation Record Stack



Abstract Stack Frame



TAC generation for expressions

- **cgen**(*atomic_expr*) directly generates TAC for atomic expressions
 - Constants, identifiers,...
- **Cgen**(*compound_expr*) recursively generates TAC for compound expressions
 - binary operators, procedure calls, ...
 - use temporary variables (**registers**) to store values of intermediate expressions

13

cgen example

```
cgen(5 + x) = {  
  Choose a new temporary t  
  Let  $t_1 = \mathbf{cgen}(5)$   
  Let  $t_2 = \mathbf{cgen}(x)$   
  Emit(  $t = t_1 + t_2$  )  
  Return t  
}
```

14

Naïve cgen for expressions

- Maintain a counter for temporaries in **c**
- Initially: **c = 0**
- **cgen**($e_1 \text{ op } e_2$) = {
 - Let A = **cgen**(e_1)
 - c = c + 1**
 - Let B = **cgen**(e_2)
 - c = c + 1**
 - Emit(**_tc = A op B;**)
 - Return **_tc**}

15

TAC Generation for Control Flow Statements

- Label introduction
_label_name:
Indicates a point in the code that can be jumped to
- Unconditional jump: go to instruction following label L
Goto L;
- Conditional jump: test condition variable t;
if 0, jump to label L
IfZ t Goto L;
- Similarly : test condition variable t;
if 1, jump to label L
IfNZ t Goto L;

16

Control-flow example – conditions

```
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;
z = z * z;
```

```
    _t0 = x < y;
    IfZ _t0 Goto _L0;
    z = x;
    Goto _L1;
_L0:
    z = y;
_L1:
    z = z * z;
```

17

cgen for if-then-else

```
cgen(if (e) s1 else s2)  Let _t = cgen(e)
                             Let Lfalse be a new label
                             Let Lafter be a new label
                             Emit( IfZ _t Goto Lfalse; )
                             cgen(s1)
                             Emit( Goto Lafter; )
                             Emit( Lfalse: )
                             cgen(s2)
                             Emit( Goto Lafter; )
                             Emit( Lafter: )
```

18

Naive cgen for expressions

- Maintain a counter for temporaries in `c`
- Initially: `c = 0`
- $\text{cgen}(e_1 \text{ op } e_2) = \{$
 - Let `A = cgen(e1)`
 - `c = c + 1`
 - Let `B = cgen(e2)`
 - `c = c + 1`
 - Emit(`_tc = A op B;`)
 - Return `_tc`
- **Observation:** temporaries in `cgen(e1)` can be reused in `cgen(e2)`

19

Improving cgen for expressions

- Observation – naïve translation needlessly generates temporaries for leaf expressions
- Observation – temporaries used exactly once
 - Once a temporary has been read it can be reused for another sub-expression
- $\text{cgen}(e_1 \text{ op } e_2) = \{$
 - Let `_t1 = cgen(e1)`
 - Let `_t2 = cgen(e2)`
 - Emit(`_t = _t1 op _t2;`)
 - Return `t`
- Temporaries `cgen(e1)` can be reused in `cgen(e2)`

20

Sethi-Ullman translation

- Algorithm by Ravi Sethi and Jeffrey D. Ullman to emit optimal TAC
 - Minimizes number of temporaries
- Main data structure in algorithm is a stack of temporaries
 - Stack corresponds to recursive invocations of $_t = \text{cgen}(e)$
 - All the temporaries on the stack are live
 - Live = contain a value that is needed later on

21

Weighted register allocation

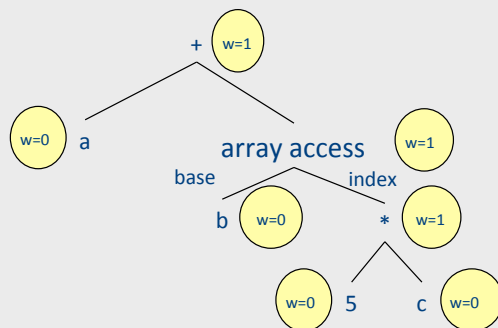
- Can save registers by re-ordering subtree computations
- Label each node with its **weight**
 - Weight = number of registers needed
 - Leaf weight known
 - Internal node weight
 - $w(\text{left}) > w(\text{right})$ then $w = \text{left}$
 - $w(\text{right}) > w(\text{left})$ then $w = \text{right}$
 - $w(\text{right}) = w(\text{left})$ then $w = \text{left} + 1$
- Choose **heavier** child as first to be translated
- **WARNING:** have to check that no side-effects exist before attempting to apply this optimization (pre-pass on the tree)

22

Weighted reg. alloc. example

$_t0 = \text{cgen}(a+b[5*c])$

Phase 1: - check absence of side-effects in expression tree
- assign weight to each AST node

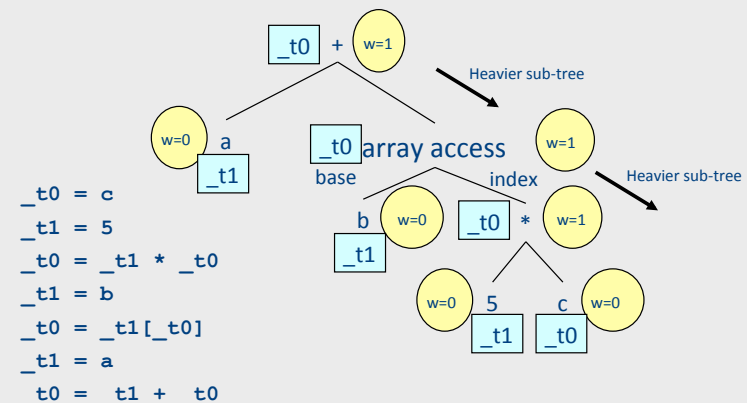


23

Weighted reg. alloc. example

$_t0 = \text{cgen}(a+b[5*c])$

Phase 2: - use weights to decide on order of translation



TAC Generation for Memory Access Instructions

- **Copy** instruction: $a = b$
- **Load/store** instructions:
 $a = *b$ $*a = b$
- **Address of** instruction $a = \&b$
- **Array accesses:**
 $a = b[i]$ $a[i] = b$
- **Field accesses:**
 $a = b[f]$ $a[f] = b$
- **Memory allocation** instruction:
 $a = \text{alloc}(\text{size})$
 - Sometimes left out (e.g., malloc is a procedure in C)

25

Array operations

$x := y[i]$

```
t1 := &y ; t1 = address-of y
t2 := t1 + i ; t2 = address of y[i]
x := *t2 ; value stored at y[i]
```

$x[i] := y$

```
t1 := &x ; t1 = address-of x
t2 := t1 + i ; t2 = address of x[i]
*t2 := y ; store through pointer
```

26

TAC Generation for Control Flow Statements

- Label introduction
_label_name:
Indicates a point in the code that can be jumped to
- Unconditional jump: go to instruction following label L
Goto L;
- Conditional jump: test condition variable t;
if 0, jump to label L
IfZ t Goto L;
- Similarly : test condition variable t;
if 1, jump to label L
IfNZ t Goto L;

27

Control-flow example – conditions

```
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;
z = z * z;
```

```
_t0 = x < y;
IfZ _t0 Goto _L0;
z = x;
Goto _L1;
_L0:
z = y;
_L1:
z = z * z;
```

28

cgen for if-then-else

```
cgen(if (e) s1 else s2)  Let _t = cgen(e)
                          Let Lfalse be a new label
                          Let Lafter be a new label
                          Emit( IfZ _t Goto Lfalse; )
                          cgen(s1)
                          Emit( Goto Lafter; )
                          Emit( Lfalse: )
                          cgen(s2)
                          Emit( Goto Lafter; )
                          Emit( Lafter: )
```

29

Control-flow example – loops

```
int x;
int y;

while (x < y) {
    x = x * 2;
}

y = x;

_L0:  _t0 = x < y;
      IfZ _t0 Goto _L1;
      x = x * 2;
      Goto _L0;
_L1:  y = x;
```

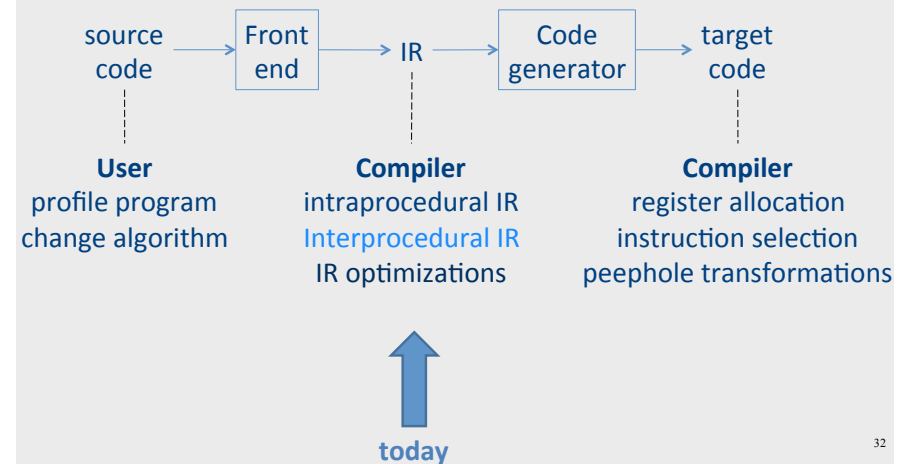
30

cgen for while loops

```
cgen(while (expr) stmt)  Let Lbefore be a new label.
                        Let Lafter be a new label.
                        Emit( Lbefore: )
                        Let t = cgen(expr)
                        Emit( IfZ t Goto Lafter; )
                        cgen(stmt)
                        Emit( Goto Lbefore; )
                        Emit( Lafter: )
```

31

Optimization points



32

Interprocedural IR

- Compile time generation of code for procedure invocations
- Activation Records (aka Stack Frames)

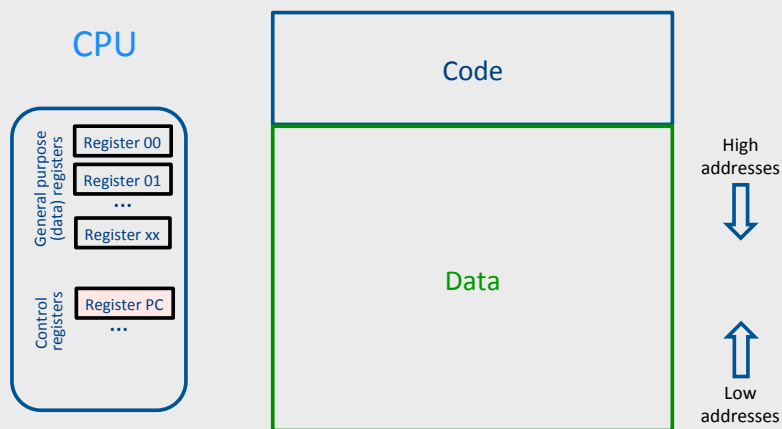
33

Supporting Procedures

- New computing environment
 - at least temporary memory for local variables
- Passing information into the new environment
 - Parameters
- Transfer of control to/from procedure
- Handling return values

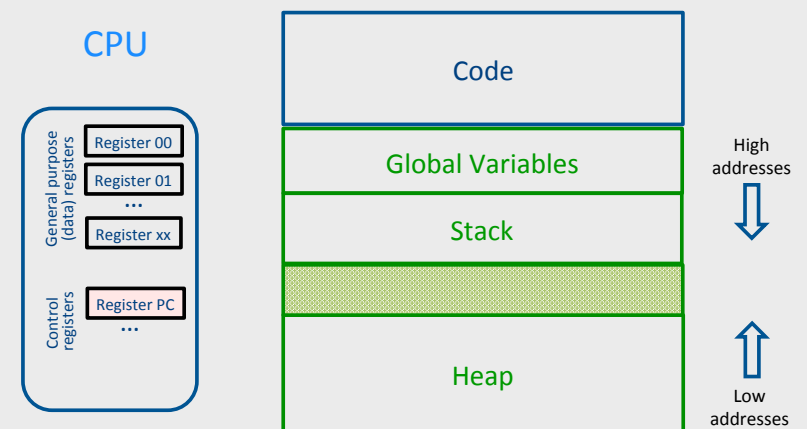
34

Abstract Register Machine (High Level View)



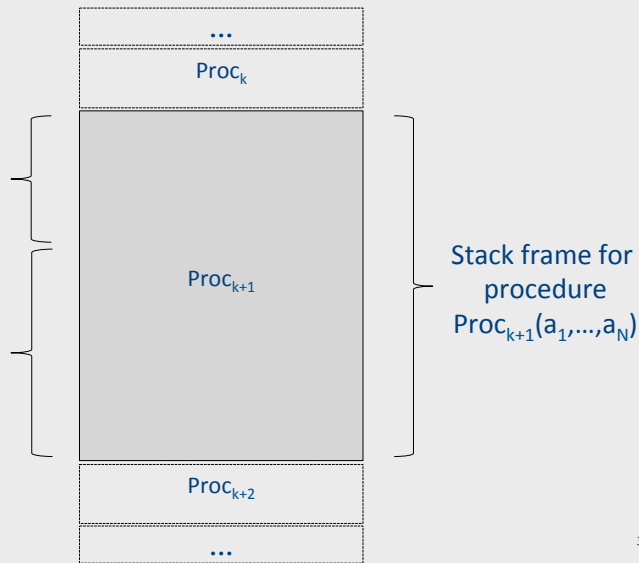
35

Abstract Register Machine (High Level View)

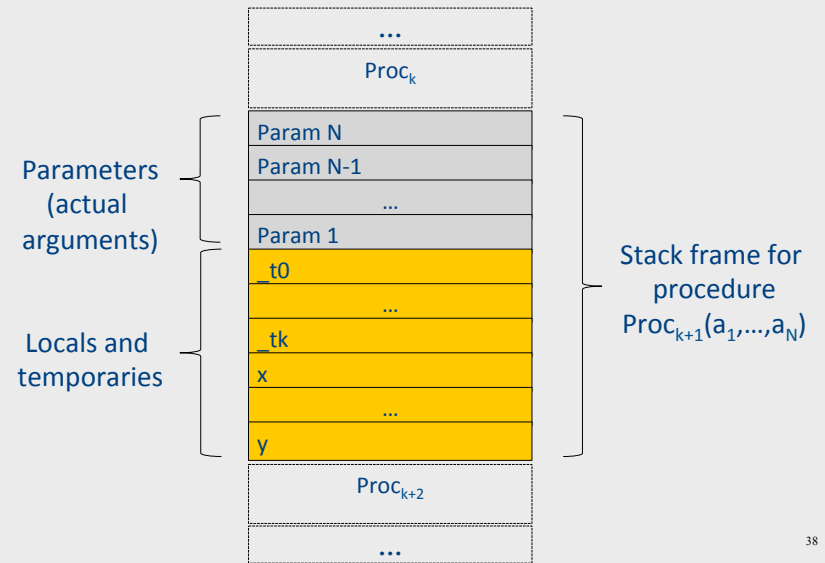


36

Abstract Activation Record Stack



Abstract Stack Frame



Handling Procedures

- Store local variables/temporaries in a **stack**
- A function call instruction pushes arguments to stack and jumps to the function label
A statement $x=f(a_1, \dots, a_n)$; looks like

```

Push a1; ... Push an;
Call f;
Pop x; // copy returned value
    
```
- Returning a value is done by pushing it to the stack (**return x;**)

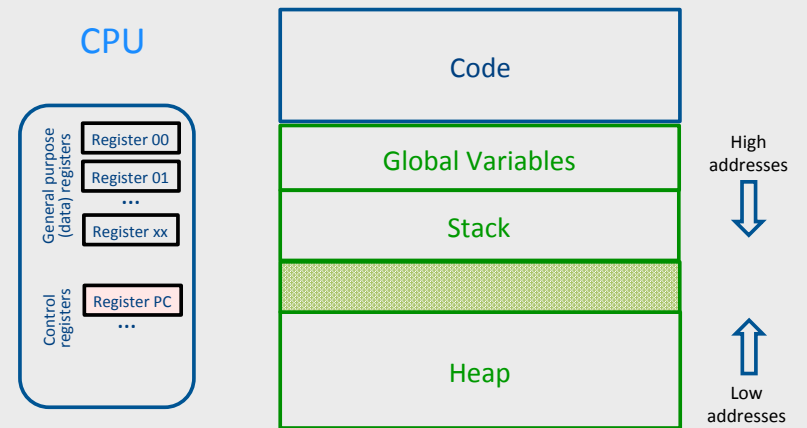
```

Push x;
    
```
- Return control to caller (and roll up stack)

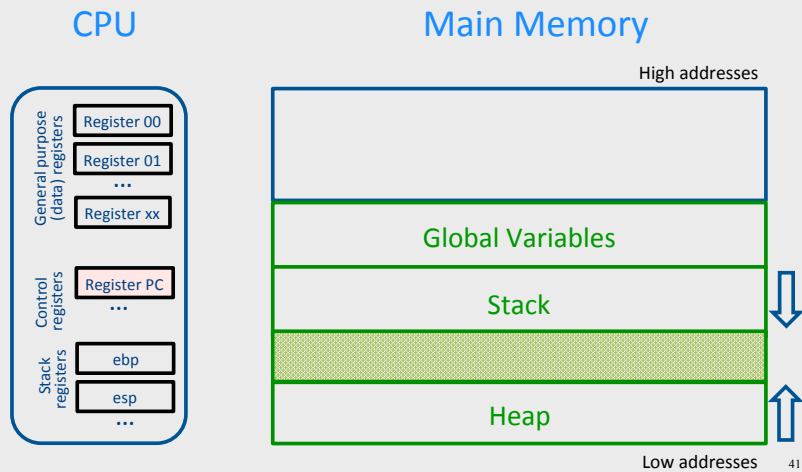
```

Return;
    
```

Abstract Register Machine



Semi-Abstract Register Machine



41

Intro: Functions Example

```
int SimpleFn(int z) {  
    int x, y;  
    x = x * y * z;  
    return x;  
}  
  
void main() {  
    int w;  
    w = SimpleFunction(137);  
}
```

```
_SimpleFn:  
_t0 = x * y;  
_t1 = _t0 * z;  
x = _t1;  
Push x;  
Return;  
  
main:  
_t0 = 137;  
Push _t0;  
Call _SimpleFn;  
Pop w;
```

42

What Can We Do with Procedures?

- Declarations & Definitions
- Call & Return
- Jumping out of procedures
- Passing & Returning procedures as parameters

43

Design Decisions

- Scoping rules
 - Static scoping vs. dynamic scoping
- Caller/callee conventions
 - Parameters
 - Who saves register values?
- Allocating space for local variables

44

Static (lexical) Scoping

```
main ()
{
  int a = 0;
  int b = 0;
  {
    int b = 1;
    {
      B2 int a = 2;
      printf ("%d %d\n", a, b);
    }
    B1 {
      int b = 3;
      B3 printf ("%d %d\n", a, b);
    }
    printf ("%d %d\n", a, b);
  }
  printf ("%d %d\n", a, b);
}
```

a name refers to
its (closest)
enclosing **scope**

**known at
compile time**

Declaration	Scopes
a=0	B0,B1,B3
b=0	B0
b=1	B1,B2
a=2	B2
b=3	B3

45

Dynamic Scoping

- Each identifier is associated with a global stack of bindings
- When entering scope where identifier is declared
 - push declaration on identifier stack
- When exiting scope where identifier is declared
 - pop identifier stack
- **Evaluating the identifier in any context binds to the current top of stack**
- **Determined at runtime**

46

Example

```
int x = 42;
```

```
int f() { return x; }
```

```
int g() { int x = 1; return f(); }
```

```
int main() { return g(); }
```

- What value is returned from main?
- Static scoping?
- Dynamic scoping?

47

Why do we care?

- We need to generate code to access variables
- Static scoping
 - Identifier binding is known at compile time
 - Address of the variable is known at compile time
 - Assigning addresses to variables is part of code generation
 - No runtime errors of “access to undefined variable”
 - Can check types of variables

48

Variable addresses for static scoping: first attempt

```
int x = 42;
```

```
int f() { return x; }  
int g() { int x = 1; return f(); }  
int main() { return g(); }
```

identifier	address
x (global)	0x42
x (inside g)	0x73

49

Variable addresses for static scoping: first attempt

```
int a [11];  
  
void quicksort(int m, int n) {  
    int i;  
    if (n > m) {  
        i = partition(m, n);  
        quicksort (m, i-1);  
        quicksort (i+1, n);  
    }  
  
    main() {  
        ...  
        quicksort (1, 9);  
    }  
}
```

**what is the address
of the variable "i" in
the procedure
quicksort?**

50

Compile-Time Information on Variables

- Name
- Type
- Scope
 - when is it recognized
- Duration
 - Until when does its value exist
- Size
 - How many bytes are required at runtime
- Address
 - Fixed
 - Relative
 - Dynamic

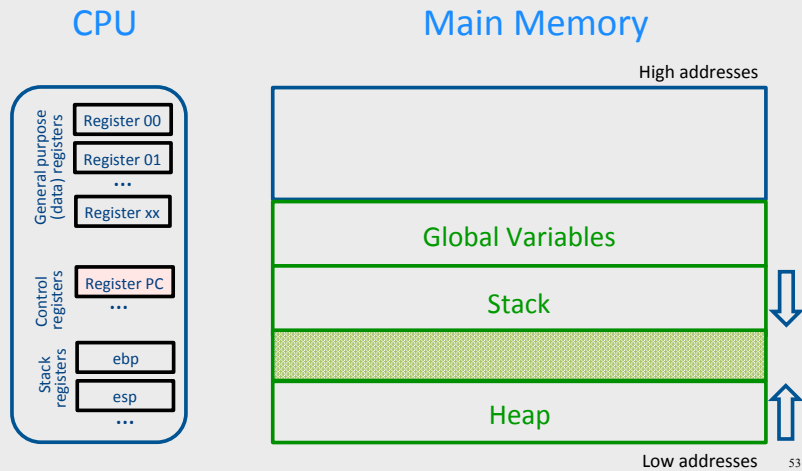
51

Activation Record (Stack Frames)

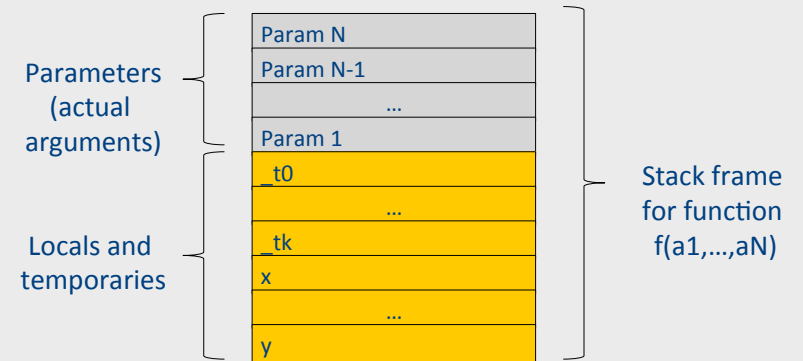
- separate space for each procedure invocation
- **managed at runtime**
 - code for managing it generated by the compiler
- desired properties
 - efficient allocation and deallocation
 - procedures are called frequently
 - variable size
 - different procedures may require different memory sizes

52

Semi-Abstract Register Machine



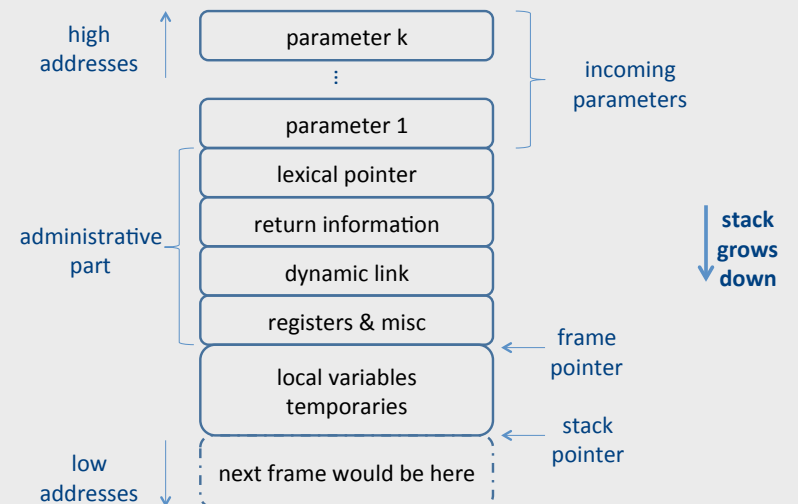
A Logical Stack Frame (Simplified)



Runtime Stack

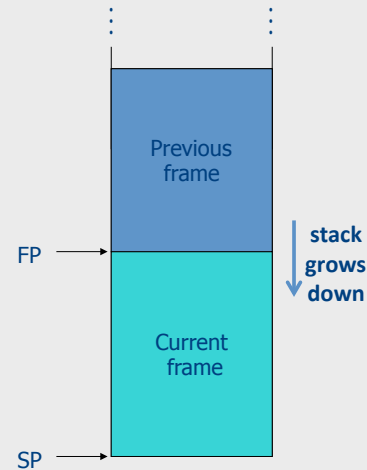
- Stack of activation records
- Call = push new activation record
- Return = pop activation record
- Only one "active" activation record – top of stack
- How do we handle recursion?

Activation Record (frame)



Runtime Stack

- SP – stack pointer
 - top of current frame
- FP – frame pointer
 - base of current frame
 - Sometimes called BP (base pointer)



57

Code Blocks

- Programming language provide code blocks

```
void foo()
{
    int x = 8 ; y=9;//1
    { int x = y * y ;//2 }
    { int x = y * 7 ;//3 }
    x = y + 1;
}
```

administrative
x1
y1
x2
x3
...

58

L-Values of Local Variables

- The offset in the stack is known at compile time
- $L\text{-val}(x) = FP + \text{offset}(x)$
- $x = 5 \Rightarrow \text{Load_Constant } 5, R3$
Store $R3, \text{offset}(x)(FP)$

59

Pentium Runtime Stack

Register	Usage
ESP	Stack pointer
EBP	Base pointer

Pentium stack registers

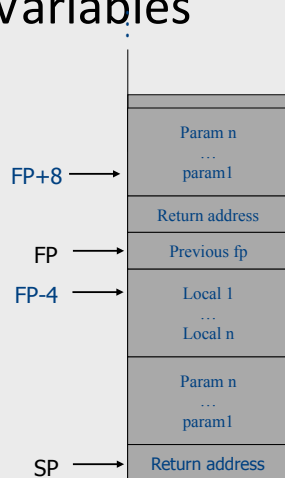
Instruction	Usage
push, pusha,...	push on runtime stack
pop, popa,...	Base pointer
call	transfer control to called routine
return	transfer control back to caller

Pentium stack and call/ret instructions

60

Accessing Stack Variables

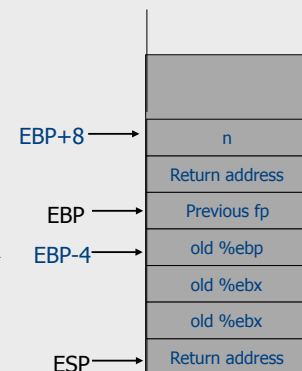
- Use offset from FP (%ebp)
- Remember – stack grows downwards
- Above FP = parameters
- Below FP = locals
- Examples
 - %ebp + 4 = return address
 - %ebp + 8 = first parameter
 - %ebp – 4 = first local



61

Factorial – fact(int n)

```
fact:
pushl %ebp           # save ebp
movl %esp,%ebp      # ebp=esp
pushl %ebx          # save ebx
movl 8(%ebp),%ebx   # ebx = n
cmpl $1,%ebx        # n = 1 ?
jle .lresult        # then done
leal -1(%ebx),%eax  # eax = n-1
pushl %eax          #
call fact           # fact(n-1)
imull %ebx,%eax     # eax=retv*n
jmp .lreturn        #
.lresult:
movl $1,%eax        # retv
.lreturn:
movl -4(%ebp),%ebx  # restore ebx
movl %ebp,%esp      # restore esp
popl %ebp           # restore ebp
```



(stack in intermediate point)

(disclaimer: real compiler can do better than that)

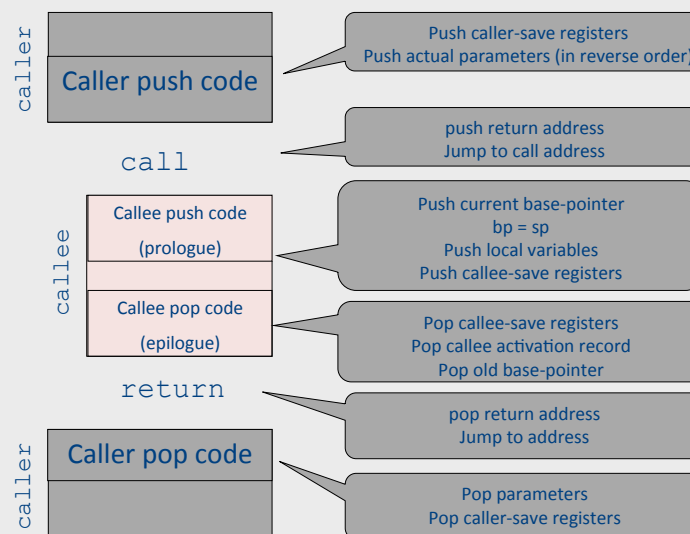
62

Call Sequences

- The processor does not save the content of registers on procedure calls
- So who will?
 - Caller saves and restores registers
 - Callee saves and restores registers
 - But can also have both save/restore some registers

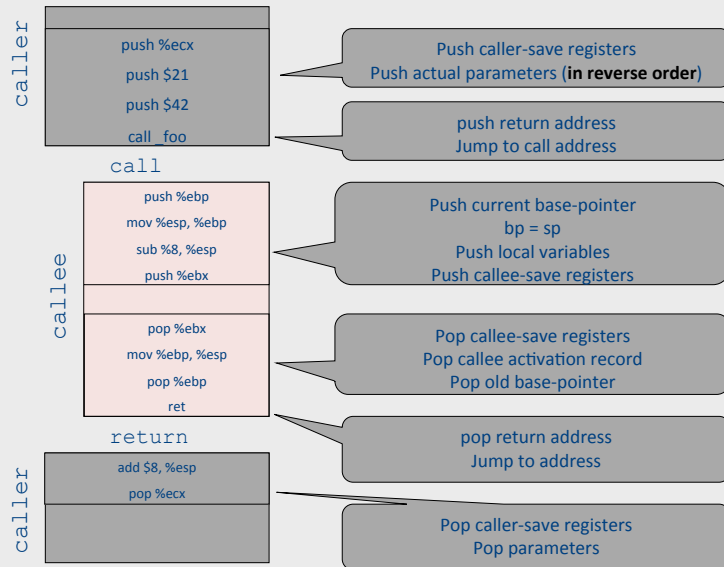
63

Call Sequences



64

Call Sequences – Foo (42, 21)



65

“To Callee-save or to Caller-save?”

- Callee-saved registers need only be saved when callee modifies their value
- some heuristics and conventions are followed

66

Caller-Save and Callee-Save Registers

- Callee-Save Registers
 - Saved by the callee before modification
 - Values are automatically preserved across calls
- Caller-Save Registers
 - Saved (if needed) by the caller before calls
 - Values are not automatically preserved across calls
- Usually the architecture defines caller-save and callee-save registers
- Separate compilation
- Interoperability between code produced by different compilers/languages
- But compiler writers decide when to use caller/callee registers

67

Callee-Save Registers

- Saved by the callee before modification
- Usually at procedure prolog
- Restored at procedure epilog
- Hardware support may be available
- Values are automatically preserved across calls

```
int foo(int a) {
    int b=a+1;
    f1();
    g1(b);
    return(b+2);
}

.global _foo
Add_Constant -K, SP //allocate space for foo
Store_Local R5, -14(FP) // save R5
Load_Reg R5, R0; Add_Constant R5, 1
JSR f1 ; JSR g1;
Add_Constant R5, 2; Load_Reg R5, R0
Load_Local -14(FP), R5 // restore R5
Add_Constant K, SP; RTS // deallocate
```

68

Caller-Save Registers

- Saved by the caller before calls when needed
- Values are not automatically preserved across calls

```

void bar (int y) {
    int x=y+1;
    f2(x);
    g2(2);
    g2(8);
}

.global _bar
    Add_Constant -K, SP //allocate space for bar
    Add_Constant R0, 1
    JSR f2
    Load_Constant 2, R0 ;   JSR g2;
    Load_Constant 8, R0 ;   JSR g2
    Add_Constant K, SP // deallocate space for bar
    RTS
    
```

69

Parameter Passing

- 1960s
 - In memory
 - No recursion is allowed
- 1970s
 - In stack
- 1980s
 - In registers
 - First k parameters are passed in registers (k=4 or k=6)
 - Where is time saved?
- Most procedures are leaf procedures
- Interprocedural register allocation
- Many of the registers may be dead before another invocation
- Register windows are allocated in some architectures per call (e.g., sun Sparc)

70

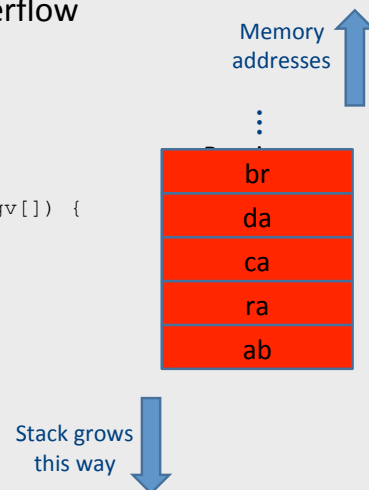
Windows Exploit(s) Buffer Overflow

```

void foo (char *x) {
    char buf[2];
    strcpy(buf, x);
}

int main (int argc, char *argv[]) {
    foo(argv[1]);
}

./a.out abracadabra
Segmentation fault
    
```



71

Buffer overflow

```

int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);
    if(strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) { if(argc < 2) {
    printf("Usage: %s <password>\n", argv[0]); exit(0); }
    if(check_authentication(argv[1])) {
        printf("\n===== \n");
        printf(" Access Granted.\n");
        printf("===== \n"); }
    else {
        printf("\nAccess Denied.\n");
    }
}
    
```

(source: "hacking – the art of exploitation, 2nd Ed")

72

Buffer overflow

```
int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);
    if(strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) { if(argc < 2) {
    printf("Usage: %s <password>\n", argv[0]); exit(0); }
    if(check_authentication(argv[1])) {
        printf("\n-----\n");
        printf("    Access Granted.\n");
        printf("-----\n"); }
    else {
        printf("\nAccess Denied.\n");
    }
}
```

(source: "hacking – the art of exploitation, 2nd Ed")

73

Buffer overflow

```
int check_authentication(char *password) {
    char password_buffer[16];
    int auth_flag = 0;

    strcpy(password_buffer, password);
    if(strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) { if(argc < 2) {
    printf("Usage: %s <password>\n", argv[0]); exit(0); }
    if(check_authentication(argv[1])) {
        printf("\n-----\n");
        printf("    Access Granted.\n");
        printf("-----\n"); }
    else {
        printf("\nAccess Denied.\n");
    }
}
```

(source: "hacking – the art of exploitation, 2nd Ed")

74

Buffer overflow

```
0x08048529 <+69>: movl    $0x8048647, (%esp)
0x08048530 <+76>: call   0x8048394 <puts@plt>
0x08048535 <+81>: movl    $0x8048664, (%esp)
0x0804853c <+88>: call   0x8048394 <puts@plt>
0x08048541 <+93>: movl    $0x804867a, (%esp)
0x08048548 <+100>: call  0x8048394 <puts@plt>
0x0804854d <+105>: jmp    0x804855b <main+119>
0x0804854f <+107>: movl    $0x8048696, (%esp)
0x08048556 <+114>: call  0x8048394 <puts@plt>
```

75

Nested Procedures

- For example – Pascal
- Any routine can have sub-routines
- Any sub-routine can access anything that is defined in its containing scope or inside the sub-routine itself
 - “non-local” variables

76

Example: Nested Procedures

```

program p() {
  int x;
  procedure a() {
    int y;
    [ procedure b() { ... c() ... };
    procedure c() {
      int z;
      [ procedure d() {
        y := x + z
      };
      ... b() ... d() ...
    }
    ... a() ... c() ...
  }
  ... a() ... c() ...
}
a()

```

Possible call sequence:
 $p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d$

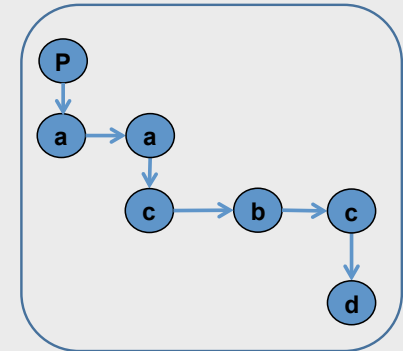
what are the addresses of variables "x," "y" and "z" in procedure d?

77

Nested Procedures

- can call a sibling, ancestor
- when "c" uses (non-local) variables from "a", which instance of "a" is it?
- how do you find the right activation record at runtime?

Possible call sequence:
 $p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d$

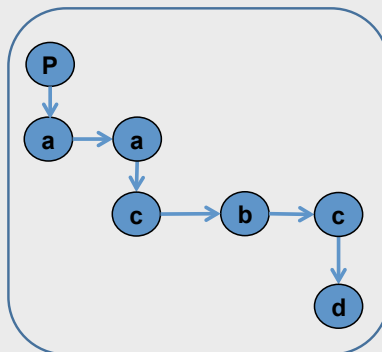


78

Nested Procedures

- goal: find the closest routine in the stack from a given nesting level
- if we reached the same routine in a sequence of calls
 - routine of level k uses variables of the same nesting level, it uses its own variables
 - if it uses variables of nesting level $j < k$ then it must be the last routine called at level j
- If a procedure is last at level j on the stack, then it must be ancestor of the current routine

Possible call sequence:
 $p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d$



79

Nested Procedures

- problem: a routine may need to access variables of another routine that contains it statically
- solution: lexical pointer (a.k.a. access link) in the activation record
- lexical pointer points to the last activation record of the nesting level above it
 - in our example, lexical pointer of d points to activation records of c
- lexical pointers created at runtime
- number of links to be traversed is known at compile time

80

Lexical Pointers

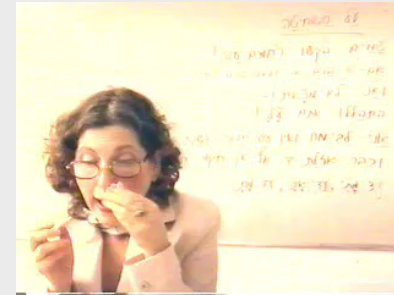
```

program p(){
  int x;
  procedure a(){
    int y;
    [procedure b(){ c() };
    procedure c(){
      int z;
      [procedure d(){
        y := x + z;
      };
      ... b() ... d() ...
    };
    ... a() ... c() ...
  };
  a();
}

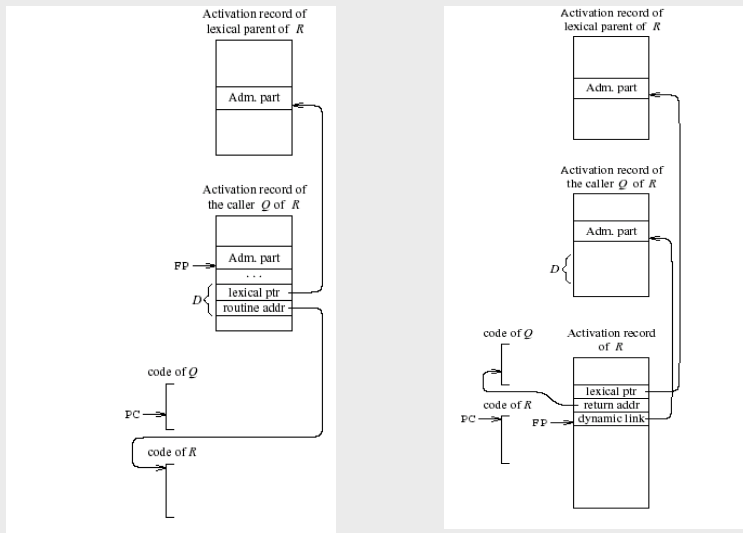
```

Possible call sequence:
 $p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d$

What is a Compiler?



Procedure Q Calls R



Activation Records: Remarks

Non-Local goto in C syntax

```
void level_0(void) {
    void level_1(void) {
        void level_2(void) {
            ...
            goto L_1;
            ...
        }
        ...
    L_1:...
    ...
}
...
}
```

85

Non-local gotos in C

- setjmp remembers the current location and the stack frame
- longjmp jumps to the current location (popping many activation records)

86

Non-Local Transfer of Control in C

```
#include <setjmp.h>
void find_div_7(int n, jmp_buf *jmpbuf_ptr) {
    if (n % 7 == 0) longjmp(*jmpbuf_ptr, n);
    find_div_7(n + 1, jmpbuf_ptr);
}
int main(void) {
    jmp_buf jmpbuf;          /* type defined in setjmp.h */
    int return_value;

    if ((return_value = setjmp(jmpbuf)) == 0) {
        /* setting up the label for longjmp() lands here */
        find_div_7(1, &jmpbuf);
    }
    else {
        /* returning from a call of longjmp() lands here */
        printf("Answer = %d\n", return_value);
    }
    return 0;
}
```

87

Stack Frames

- Allocate a separate space for every procedure incarnation
- Relative addresses
- Provide a simple mean to achieve modularity
- Supports separate code generation of procedures
- Naturally supports recursion
- Efficient memory allocation policy
 - Low overhead
 - Hardware support may be available
- LIFO policy
- Not a pure stack
 - Non local references
 - Updated using arithmetic

88

The Frame Pointer

- The caller
 - the calling routine
- The callee
 - the called routine
- caller responsibilities:
 - Calculate arguments and save in the stack
 - Store lexical pointer
- call instruction:
 - M[--SP] := RA
 - PC := callee
- callee responsibilities:
 - FP := SP
 - SP := SP - frame-size
- Why use both SP and FP?

89

Variable Length Frame Size

- C allows allocating objects of unbounded size in the stack

```
void p() {  
    int i;  
    char *p;  
    scanf("%d", &i);  
    p = (char *) alloca(i*sizeof(int));  
}
```

- Some versions of Pascal allows conformant array value parameters

90

Limitations

- The compiler may be forced to store a value on a stack instead of registers
- The stack may not suffice to handle some language features

91

Frame-Resident Variables

- A variable x cannot be stored in register when:
 - x is passed by reference
 - Address of x is taken (&x)
 - is addressed via pointer arithmetic on the stack-frame (C varargs)
 - x is accessed from a nested procedure
 - The value is too big to fit into a single register
 - The variable is an array
 - The register of x is needed for other purposes
 - Too many local variables
- An escape variable:
 - Passed by reference
 - Address is taken
 - Addressed via pointer arithmetic on the stack-frame
 - Accessed from a nested procedure

92

The Frames in Different Architectures

$g(x, y, z)$ where x escapes

	Pentium	MIPS	Sparc
x	InFrame(8)	InFrame(0)	InFrame(68)
y	InFrame(12)	InReg(X_{157})	InReg(X_{157})
z	InFrame(16)	InReg(X_{158})	InReg(X_{158})
View Change	$M[sp+0] \leftarrow fp$ $fp \leftarrow sp$ $sp \leftarrow sp-K$	$sp \leftarrow sp-K$ $M[sp+K+0] \leftarrow r_2$ $X_{157} \leftarrow r_4$ $X_{158} \leftarrow r_5$	save %sp, -K, %sp $M[fp+68] \leftarrow i_0$ $X_{157} \leftarrow i_1$ $X_{158} \leftarrow i_2$

93

Limitations of Stack Frames

- A local variable of P cannot be stored in the activation record of P if its duration exceeds the duration of P

- Example 1: Static variables in C (own variables in Algol)

```
void p(int x)
{
    static int y = 6 ;
    y += x;
}
```

- Example 2: Features of the C language

```
int * f()
{ int x ;
  return &x ;
}
```

- Example 3: Dynamic allocation

```
int * f() { return (int *)
malloc(sizeof(int)); }
```

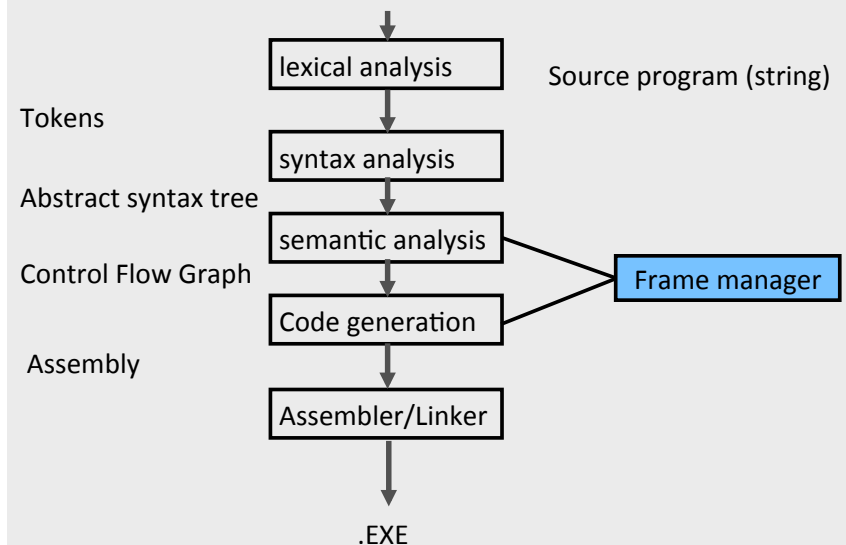
94

Compiler Implementation

- Hide machine dependent parts
- Hide language dependent part
- Use special modules

95

Basic Compiler Phases



96

Hidden in the frame ADT

- Word size
- The location of the formals
- Frame resident variables
- Machine instructions to implement “shift-of-view” (prologue/epilogue)
- The number of locals “allocated” so far
- The label in which the machine code starts

97

Invocations to Frame

- “Allocate” a new frame
- “Allocate” new local variable
- Return the L-value of local variable
- Generate code for procedure invocation
- Generate prologue/epilogue
- Generate code for procedure return

98

Activation Records: Summary

- compile time memory management for procedure data
- works well for data with well-scoped lifetime
 - deallocation when procedure returns

99

The End

100