C and Assembly Languages

Synopsis

A. A C Refresher
- Datatypes
- Simple and Compound Statements
- Systematic Translation Scheme
- Recursion

B. Assembly Languages
- Vanilla Assembly Language
- VAL programming example

Why C?

- Portable assembly language
- Low overheads compared to real assembly languages
- Most compilers and interpreters for other languages are written in C
- Good background in PL:
  - Relationship of other high level languages to C
  - Relationship of C to assembly languages

A C Refresher

```
int power(int a, int b) ;

int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

A C Refresher

A program is a collection of "functions"

```
int power(int a, int b) ;
```

```
int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

A C Refresher

Entry point into the program

```
int power(int a, int b) ;

int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

A C Refresher

Function body

```
int power(int a, int b) ;

int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

A C Refresher — Blocks

```
int power(int a, int b) ;

int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

Block

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

A C Refresher — Blocks

```
int power(int a, int b) ;

int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

Nested Block

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

A C Refresher — Declarations

Local variable declaration; at top of any block

```
int power(int a, int b) ;

int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

A C Refresher — Declarations

```
int power(int a, int b) ;

int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

Scope of local declaration

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

A C Refresher — Declarations

```
int power(int a, int b) ;

int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

Global declaration

```
int s = 1 ;
int power(int a, int b) {
  int cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

Scope of global declaration

A C Refresher

```
int power(int a, int b) ;

int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

Declarations

Each declaration has a type, a list of variables, possibly initialized

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

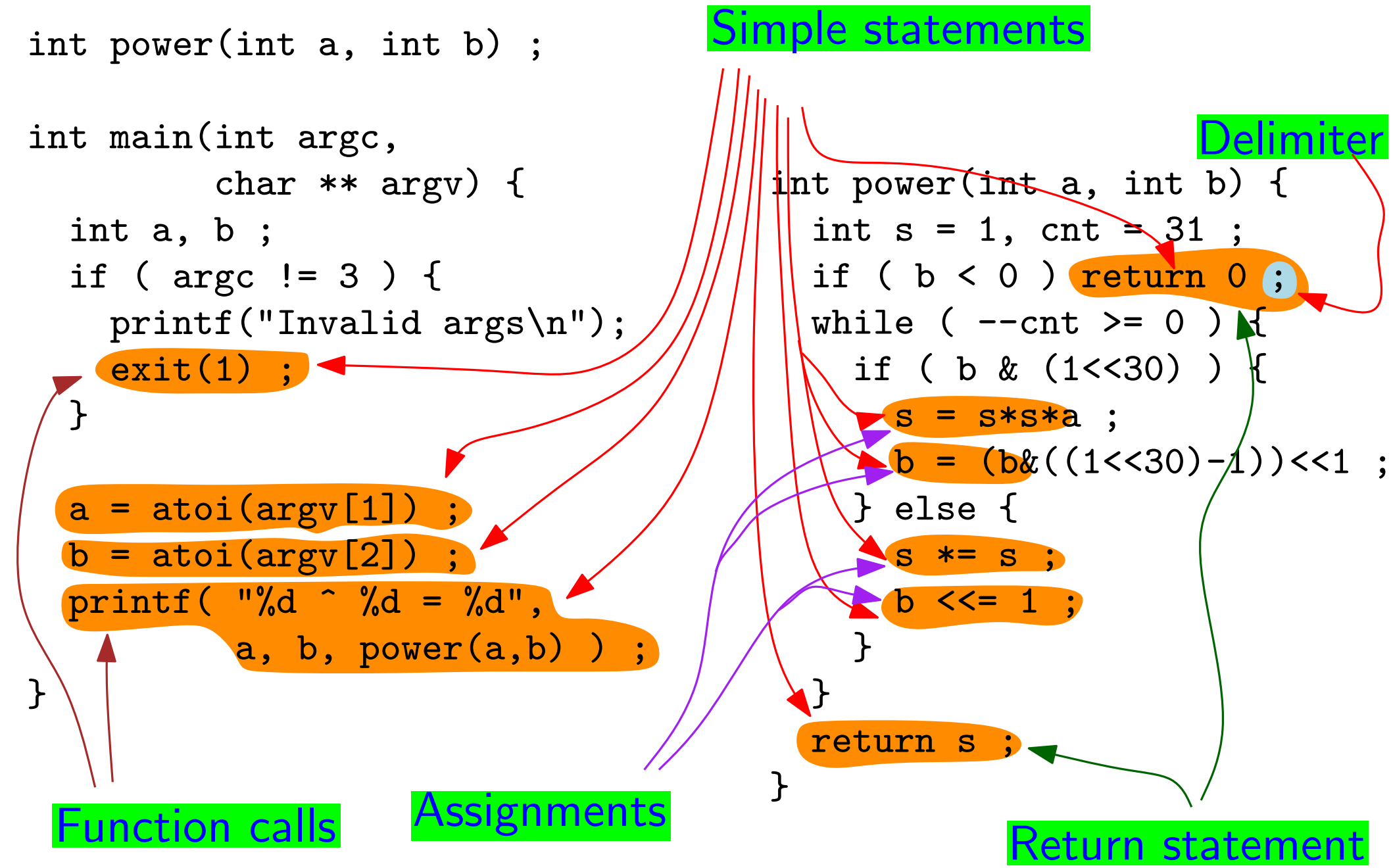
A C Refresher — Expressions have values

```
int power(int a, int b) ;

int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

Expressions

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b = s * s ;
    }
  }
  return s ;
}
```

A C Refresher — Statements



A C Refresher — Statements

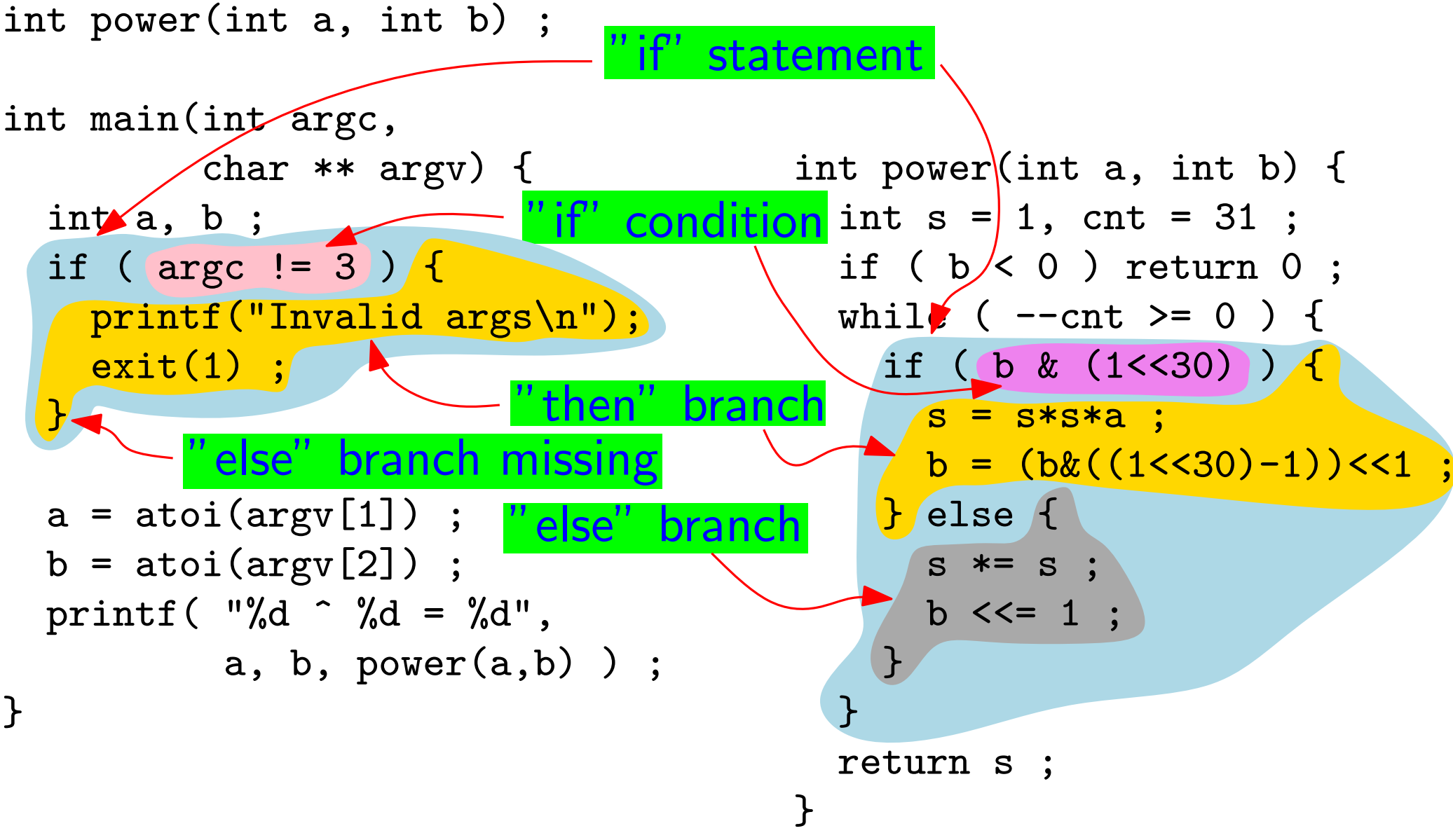
```
int power(int a, int b) ;

int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

Compound statements

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

Nested statement

A C Refresher — if statements



A C Refresher — while statements (loops)

```
int power(int a, int b) ;

int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

"while" statement

"while" condition

"while" body

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

A C Refresher — Function Terminology

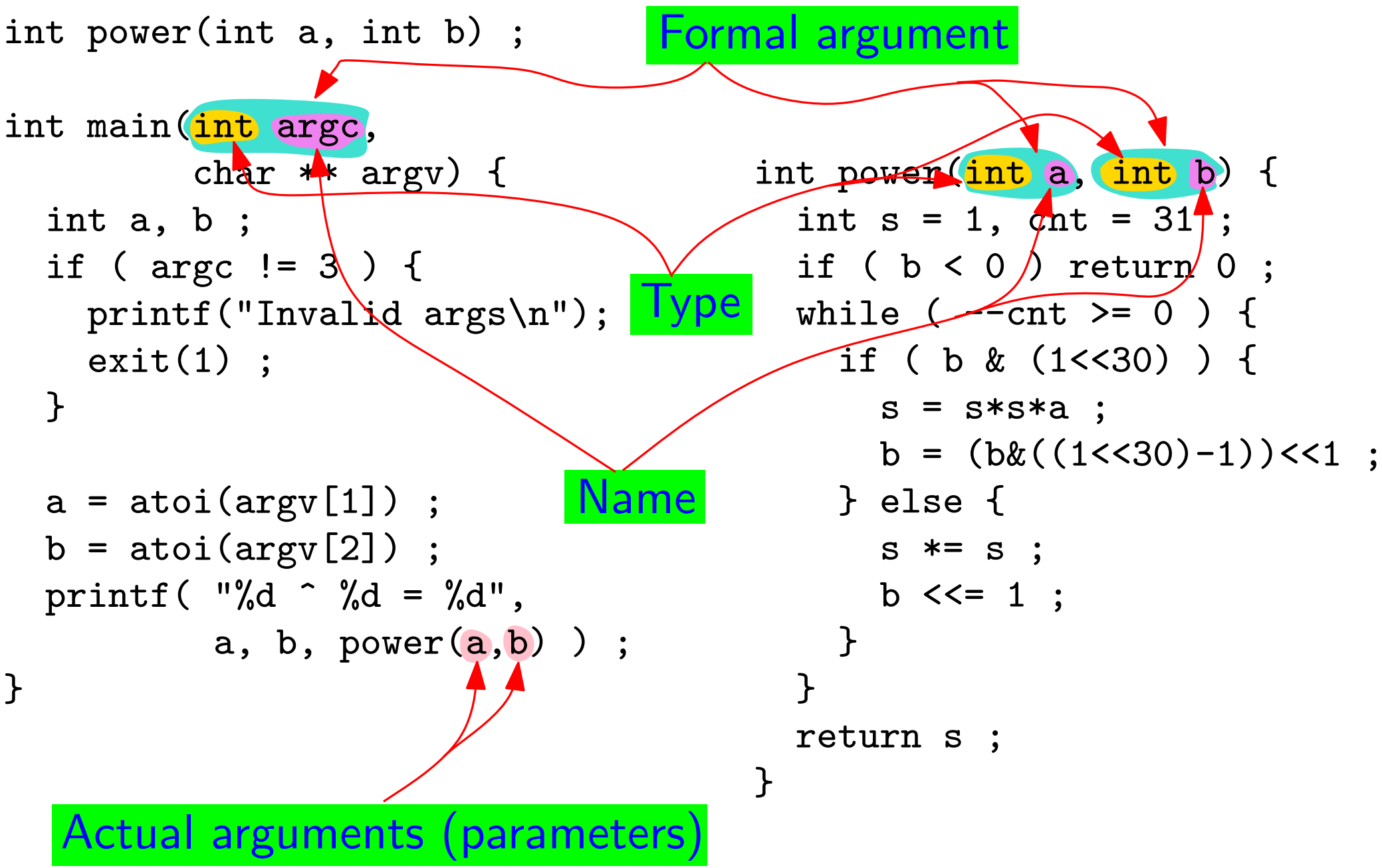
```
int power(int a, int b) ;

int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

Formal argument (parameter) list

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

Actual argument (parameter) list

A C Refresher — Function Terminology



A C Refresher — Function Terminology

```
int power(int a, int b) ;
```

Function prototype

```
int main(int argc,
         char ** argv) {
  int a, b ;
  if ( argc != 3 ) {
    printf("Invalid args\n");
    exit(1) ;
  }

  a = atoi(argv[1]) ;
  b = atoi(argv[2]) ;
  printf( "%d ^ %d = %d",
          a, b, power(a,b) ) ;
}
```

Function definition

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

Execution of C Programs

- Imperative paradigm
  - Sequential execution of statements
  - Based on the notion of **state**
    - Entire contents of memory accessible to the program
      - global and local variables/procedure arguments
      - dynamically allocated memory
  - Each statement takes the current state to a new state
- Demonstrated in **step-by-step execution** in IDEs/debuggers
  - Visual C++ Express Edition (Windows)
  - Code::Blocks (Windows/Linux)

More C-by-Example: Datatypes

```
int main() {
    int a = 1234567890 ;
    int asize = sizeof(a) ;
    long long b = 1234567890123456L ;
    int bsize = sizeof(b) ;
    int btrunc = b ;
    float c = b / a ;
    float d = b / (float)a ;
    float afloat = (float)a ;
    float bfloat = (float)b ;
    double e = b / (double)a ;
    char f = a ;
    unsigned char g = f ;
    float h = f ;
    *(char*)&h = g ;
    *((char*)&h+3) =  g ;
    return 0;
}
```

Demo in Debugger!

More C-by-Example: Pointers

```
int main() {
    int a[3][5] ;
    int asize = sizeof(a) ;
    int alinesize = sizeof(a[0]) ;
    int aelemsize = sizeof(a[0][0]) ;
    a[1][1] = 10 ;
    int * p = (int*) &a ;
    int * q = (int*) a ;
    int * r = a ;
    *r = 100 ;
    int (*s)[3][5] = &a ;
    int b = *(p+alinesize/aelemsize+1) ;
    *(p+2*alinesize/aelemsize+2) = 20 ;
    int c = (*s)[2][2] ;
    return 0;
}
```

Demo in Debugger!

More C-by-Example: `for` Loops

```
int main() {
  int sum, i ;
  for ( sum=0, i = 0 ; i <= 10 ; i ++ ) {
    sum += i ;
  }
  ...
}
```

Simple loop computing sum of integers from 0 to 10 in variable `sum`.

Alternative way of computing the same thing.

```
int main() {
  int sum, i ;
  for ( sum=0, i = 0 ; i <= 10 ; sum+=i, i ++ ) ;
  ...
}
```

```
int main() {
  int sum, i ;
  sum = i = 0 ;
  while ( i <= 10 ) {
    sum += i ;
    i ++ ;
  }
  ...
}
```

Equivalent `while` loop.

More C-by-Example: `for` Loops

```
int main() {
  int sum, i ;
  for ( sum=0, i = 0 ; i <= 10 ; i ++ ) {
    sum += i ;
  }
  ...
}
```

Simple loop computing sum of integers from 0 to 10 in variable `sum`.

Alternative way of computing the same thing.

```
int main() {
  int sum, i ;
  for ( sum=0, i = 0 ; i <= 10 ; sum+=i, i ++ ) ;
  ...
}
```

```
int main() {
  int sum, i ;
  sum = i = 0 ;
  while ( i <= 10 ) {
    sum += i ;
    i ++ ;
  }
  ...
}
```

Equivalent `while` loop.

More C-by-Example: `for` Loops

```
int main() {
  int sum, i ;
  for ( sum=0, i = 0 ; i <= 10 ; i ++ ) {
    sum += i ;
  }
  ...
}
```

Simple loop computing sum of integers from 0 to 10 in variable `sum`.

Alternative way of computing the same thing.

```
int main() {
  int sum, i ;
  for ( sum=0, i = 0 ; i <= 10 ; sum+=i, i ++ ) ;
  ...
}
```

```
int main() {
  int sum, i ;
  sum = i = 0 ;
  while ( i <= 10 ) {
    sum += i ;
    i ++ ;
  }
  ...
}
```

Equivalent `while` loop.

More C-by-Example: `for` Loops

```
int main() {
  int sum, i ;
  for ( sum=0, i = 0 ; i <= 10 ; i ++ ) {
    sum += i ;
  }
  ...
}
```

Simple loop computing sum of integers from 0 to 10 in variable `sum`.

Alternative way of computing the same thing.

```
int main() {
  int sum, i ;
  for ( sum=0, i = 0 ; i <= 10 ; sum+=i, i ++ ) ;
  ...
}
```

```
int main() {
  int sum, i ;
  sum = i = 0 ;
  while ( i <= 10 ) {
    sum += i ;
    i ++ ;
  }
  ...
}
```

Equivalent `while` loop.

More C-by-Example: `do-while` Loops

```
int main() {
  int sum, i ;
  sum = i = 0 ;
  do {
    i ++ ;
    sum += i ;
  } while ( i < 10 ) ;
  ...
}
```

Snippet that computes the sum of all integers from 1 to 10.

Equivalent `while` loop.

```
int main() {
  int sum, i ;
  sum = i = 0 ;
  i ++ ;
  sum += i ;
  while ( i < 10 ) {
    i ++ ;
    sum += i ;
  }
}
```

More C-by-Example: `do-while` Loops

```
int main() {
  int sum, i ;
  sum = i = 0 ;
  do {
    i ++ ;
    sum += i ;
  } while ( i < 10 ) ;
  ...
}
```

Snippet that computes the sum of all integers from 1 to 10.

Systematic translation.

```
int main() {
  int sum, i ;
  sum = i = 0 ;
  i ++ ;
  sum += i ;
  while ( i < 10 ) {
    i ++ ;
    sum += i ;
  }
}
```

Equivalent `while` loop.

More C-by-Example: `do-while` Loops

```
int main() {
  int sum, i ;
  sum = i = 0 ;
  do {
    i ++ ;
    sum += i ;
  } while ( i < 10 ) ;
  ...
}
```

Snippet that computes the sum of all integers from 1 to 10.

Equivalent `while` loop.

```
int main() {
  int sum, i ;
  sum = i = 0 ;
  i ++ ;
  sum += i ;
  while ( i < 10 ) {
    i ++ ;
    sum += i ;
  }
}
```

More C-by-Example: `do-while` Loops

```
int main() {
  int sum, i ;
  sum = i = 0 ;
  do {
    i ++ ;
    sum += i ;
  } while ( i < 10 ) ;
  ...
}
```

Snippet that computes the sum of all integers from 1 to 10.

Equivalent `while` loop.

```
int main() {
  int sum, i ;
  sum = i = 0 ;
  i ++ ;
  sum += i ;
  while ( i < 10 ) {
    i ++ ;
    sum += i ;
  }
}
```

More C-by-Example: `do-while` Loops

```
int main() {
  int sum, i ;
  sum = i = 0 ;
  do {
```

Snippet that computes the sum of all integers from 1 to 10.

`while`, `for`, end `do-while` are called *looping statements*, or *loops*.

Equivalent `while` loop.

```
  while ( i < 10 ) {
    i ++ ;
    sum += i ;
  }
}
```

More C-by-Example: `break` statements

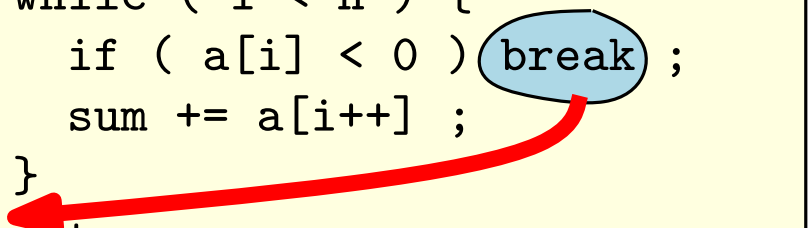
Add elements of an array up to first negative number (non-inclusive)

```
int sumpos ( int a[], int n ) {
  int i = 0, sum = 0 ;
  while ( i < n ) {
    if ( a[i] < 0 ) break ;
    sum += a[i++] ;
  }
  return sum ;
}
```

More C-by-Example: `break` statements

Add elements of an array up to first negative number (non-inclusive)

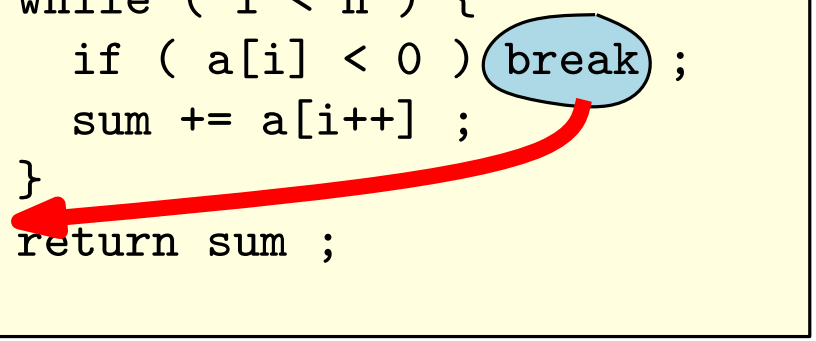
```
int sumpos ( int a[], int n ) {
  int i = 0, sum = 0 ;
  while ( i < n ) {
    if ( a[i] < 0 ) break ;
    sum += a[i++] ;
  }
  return sum ;
}
```



More C-by-Example: `break` statements

Add elements of an array up to first negative number (non-inclusive)

```
int sumpos ( int a[], int n ) {
  int i = 0, sum = 0 ;
  while ( i < n ) {
    if ( a[i] < 0 ) break ;
    sum += a[i++] ;
  }
  return sum ;
}
```

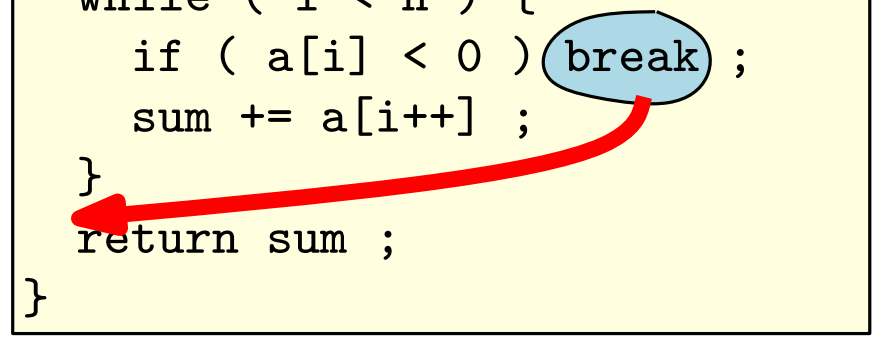


```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ;
        i < n ;
        sum += a[i++] )
    if ( a[i] < 0 ) break ;
  return sum ;
}
```

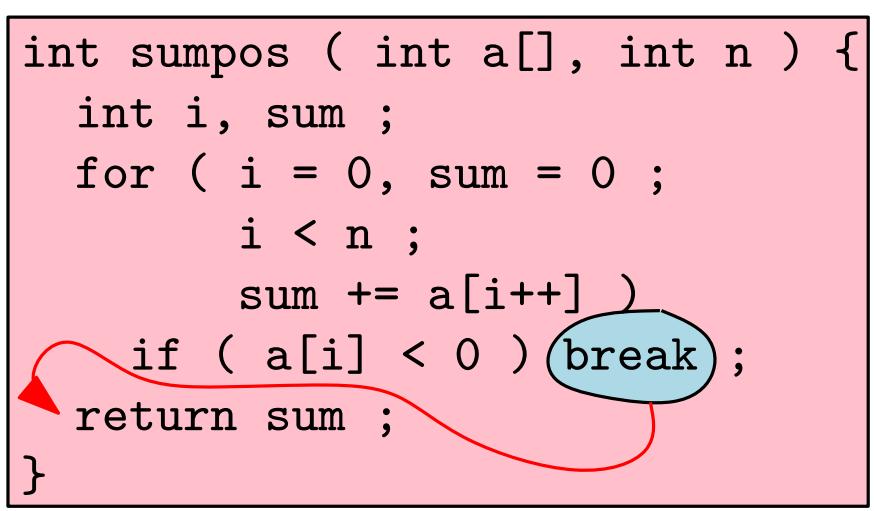
More C-by-Example: `break` statements

Add elements of an array up to first negative number (non-inclusive)

```
int sumpos ( int a[], int n ) {
  int i = 0, sum = 0 ;
  while ( i < n ) {
    if ( a[i] < 0 ) break ;
    sum += a[i++] ;
  }
  return sum ;
}
```



```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ;
        i < n ;
        sum += a[i++] )
    if ( a[i] < 0 ) break ;
  return sum ;
}
```



More C-by-Example: `break` statements

Add elements of an array up to first negative number (non-inclusive)

```
int sumpos ( int a[], int n ) {
  int i = 0, sum = 0 ;
  while ( i < n ) {
    if ( a[i] < 0 ) break ;
    sum += a[i++] ;
  }
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum, flag ;
  for ( i = 0, sum = 0, flag = 1 ;
        i < n && flag ;
        sum += a[i++] )
    if ( a[i] < 0 ) flag = 0 ;
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ;
        i < n ;
        sum += a[i++] )
    if ( a[i] < 0 ) break ;
  return sum ;
}
```

More C-by-Example: `break` statements

Add elements of an array up to first negative number (non-inclusive)

```
int sumpos ( int a[], int n ) {
  int i = 0, sum = 0 ;
  while ( i < n ) {
    if ( a[i] < 0 ) break ;
    sum += a[i++] ;
  }
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum, flag ;
  for ( i = 0, sum = 0, flag = 1 ;
        i < n && flag ;
        sum += a[i++] )
    if ( a[i] < 0 ) flag = 0 ;
  return sum ;
}
```

Is this equivalent?

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ;
        i < n ;
        sum += a[i++] )
    if ( a[i] < 0 ) break ;
  return sum ;
}
```

More C-by-Example: `break` statements

Add elements of an array up to first negative number (non-inclusive)

```
int sumpos ( int a[], int n ) {
  int i = 0, sum = 0 ;
  while ( i < n ) {
    if ( a[i] < 0 ) break ;
    sum += a[i++] ;
  }
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum, flag ;
  for ( i = 0, sum = 0, flag = 1 ;
        i < n && flag ;
        sum += a[i++] )
    if ( a[i] < 0 ) flag = 0 ;
  return sum ;
}
```

Is this equivalent?

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ;
        i < n ;
        sum += a[i++] )
    if ( a[i] < 0 ) break ;
  return sum ;
}
```

More C-by-Example: `break` statements

Add elements of an array up to first negative number (non-inclusive)

```
int sumpos ( int a[], int n ) {
  int i = 0, sum = 0 ;
  while ( i < n ) {
    if ( a[i] < 0 ) break ;
    sum += a[i++] ;
  }
  return sum ;
}
```

Nothing to execute after `flag` is set to 0

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ;
        i < n ;
        sum += a[i++] )
    if ( a[i] < 0 ) break ;
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum, flag ;
  for ( i = 0, sum = 0, flag = 1 ;
        i < n && flag ;  )
    if ( a[i] < 0 ) flag = 0 ;
    else sum += a[i++] ;
  return sum ;
}
```

More C-by-Example: `continue` statements

Add all the positive elements of an array

```
int sumpos ( int a[], int n ) {
  int i = -1, sum = 0 ;
  while ( ++i < n ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

More C-by-Example: `continue` statements

Add all the positive elements of an array

```
int sumpos ( int a[], int n ) {
  int i = -1, sum = 0 ;
  while ( ++i < n ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

More C-by-Example: `continue` statements

Add all the positive elements of an array

```
int sumpos ( int a[], int n ) {
  int i = -1, sum = 0 ;
  while ( ++i < n ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ; i < n ; i++ ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

More C-by-Example: `continue` statements

Add all the positive elements of an array

```
int sumpos ( int a[], int n ) {
  int i = -1, sum = 0 ;
  while ( ++i < n ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ; i < n ; i++ ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

More C-by-Example: `continue` statements

Add all the positive elements of an array

```
int sumpos ( int a[], int n ) {
  int i = -1, sum = 0 ;
  while ( ++i < n ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i = -1, sum = 0 ;
  while ( ++i < n ) {
    if ( a[i] >= 0 ) {
      sum += a[i] ;
    }
  }
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ; i < n ; i++ ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

More C-by-Example: `continue` statements

Add all the positive elements of an array

```
int sumpos ( int a[], int n ) {
  int i = -1, sum = 0 ;
  while ( ++i < n ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i = -1, sum = 0 ;
  while ( ++i < n ) {
    if ( a[i] >= 0 ) {
      sum += a[i] ;
    }
  }
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ; i < n ; i++ ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

More C-by-Example: `continue` statements

Add all the positive elements of an array

```
int sumpos ( int a[], int n ) {
  int i = -1, sum = 0 ;
  while ( ++i < n ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

negate

```
int sumpos ( int a[], int n ) {
  int i = -1, sum = 0 ;
  while ( ++i < n ) {
    if ( a[i] >= 0 ) {
      sum += a[i] ;
    }
  }
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ; i < n ; i++ ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

negate

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ;
        i < n ; i++ ) {
    if ( a[i] >= 0 ) {
      sum += a[i] ;
    }
  }
  return sum ;
}
```

C-by-Example: goto statements

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

```
int power(int a, int b) {
  int s = 1 ;
  if ( b < 0 ) return 0 ;
loop:
  if ( ! (--cnt >= 0)  )
          goto loop_exit ;
  if ( b & (1<<30) ) {
    s *= s*a ;
    b = (b&((1<<30)-1))<<1 ;
  } else {
    s *= s ;
    b <<= 1 ;
  }
  goto loop ;
loop_exit:
  return s ;
}
```

C-by-Example: `goto` statements

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

```
int power(int a, int b) {
  int s = 1 ;
  if ( b < 0 ) return 0 ;
loop:
  if ( ! (--cnt >= 0)  )
          goto loop_exit ;
  if ( b & (1<<30) ) {
    s *= s*a ;
    b = (b&((1<<30)-1))<<1 ;
  } else {
    s *= s ;
    b <<= 1 ;
  }
  goto loop ;
loop_exit:
  return s ;
}
```

C-by-Example: `goto` statements

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

```
int power(int a, int b) {
  int s = 1 ;
  if ( b < 0 ) return 0 ;
loop:
  if ( ! (--cnt >= 0)  )
          goto loop_exit ;
  if ( b & (1<<30) ) {
    s *= s*a ;
    b = (b&((1<<30)-1))<<1 ;
  } else {
    s *= s ;
    b <<= 1 ;
  }
  goto loop ;
loop_exit:
  return s ;
}
```

C-by-Example: `goto` statements

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

```
int power(int a, int b) {
  int s = 1 ;
  if ( b < 0 ) return 0 ;
loop:
  if ( ! (--cnt >= 0) )
          goto loop_exit ;
  if ( b & (1<<30) ) {
    s *= s*a ;
    b = (b&((1<<30)-1))<<1 ;
  } else {
    s *= s ;
    b <<= 1 ;
  }
  goto loop ;
loop_exit:
  return s ;
}
```

Systematic translation scheme

Systematic Translation Scheme

- Algorithmic procedure for translating a *program skeleton* into an equivalent one.
- Works for all possible programs.
- Can be implemented as a translator or compiler
- Must be specified in enough detail to make the implementation possible.

Simulation of `break`/`continue` with `goto`

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ;
        i < n ;
        sum += a[i++] )
    if ( a[i] < 0 ) break ;
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ; i < n ; i++ ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ;
        i < n ;
        sum += a[i++] ) {
    if ( a[i] < 0 ) goto brk;
  }
brk:
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ; i < n ; i++ ) {
    if ( a[i] < 0 ) goto cont ;
    sum += a[i] ;
  cont:{}
  }
  return sum ;
}
```

Simulation of break/continue with goto

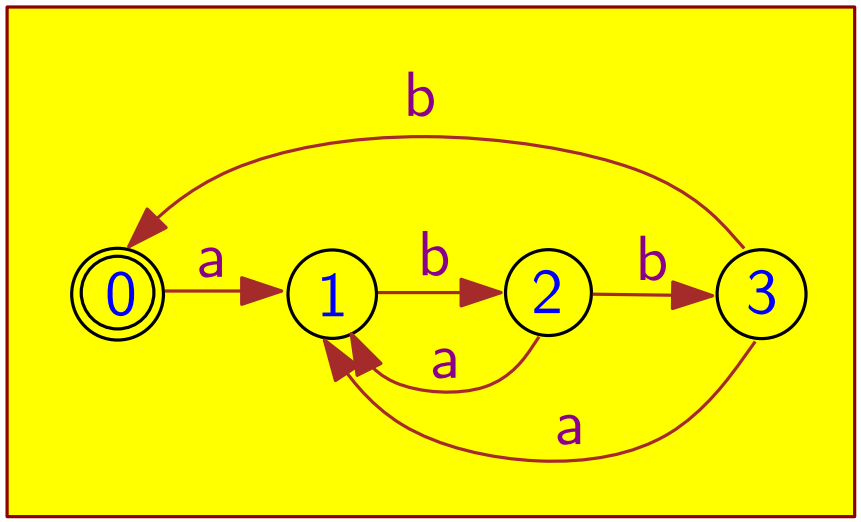
```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ;
        i < n ;
        sum += a[i++] )
    if ( a[i] < 0 ) break ;
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ; i < n ; i++ ) {
    if ( a[i] < 0 ) continue ;
    sum += a[i] ;
  }
  return sum ;
}
```

```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ;
        i < n ;
        sum += a[i++] ) {
    if ( a[i] < 0 ) goto brk;
  }
brk:
  return sum ;
}
```

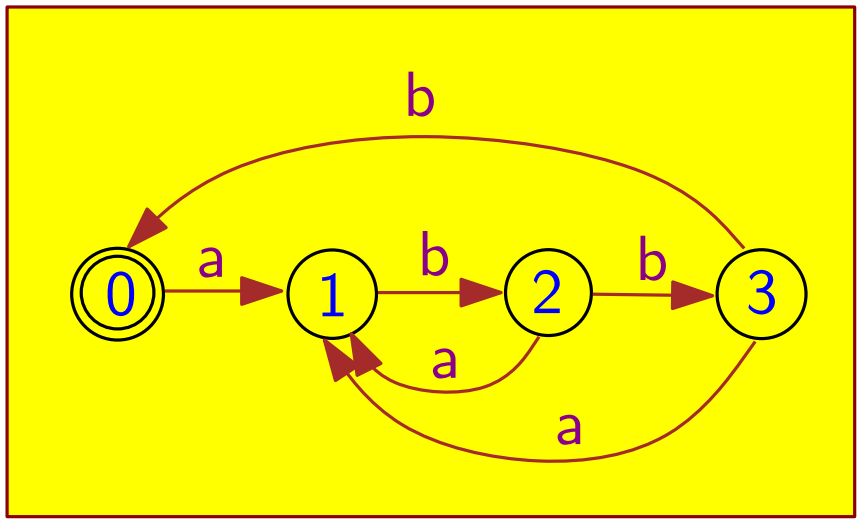
```
int sumpos ( int a[], int n ) {
  int i, sum ;
  for ( i = 0, sum = 0 ; i < n ; i++ ) {
    if ( a[i] < 0 ) goto cont ;
    sum += a[i] ;
  cont:{}
  }
  return sum ;
}
```

switch statement



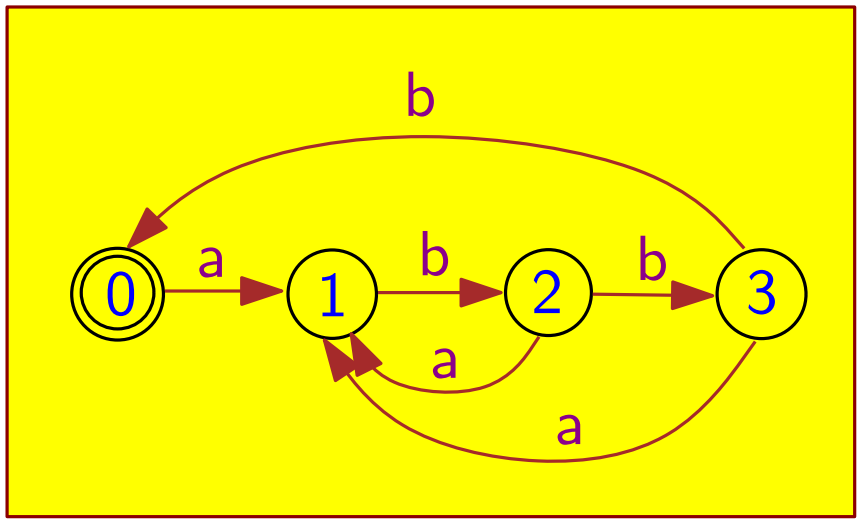

```
int accept(char a[], int n) {
  int i = 0, state = 0 ;
  while ( i < n ) {
    switch ( state ) {
      case 0 :
        if (a[i] == 'a')
         { state = 1 ; break ; }
        else return 0 ;
      case 1 :
        if (a[i] == 'b')
         { state = 2 ; break ; }
        else return 0 ;
      case 2 :
        switch(a[i]) {
           case 'a' : state = 1 ; break ;
           case 'b' : state = 3 ; break ;
           default  : return 0 ;
        }
        break ;
      case 3 :
        switch(a[i]) {
           case 'a' : state = 1 ; break ;
           case 'b' : state = 0 ; break ;
           default  : return 0 ;
        }
        break ;
      default : return 0 ;
    }
    i ++ ;
  }
  if ( state == 0 ) return 1 ;
  else return 0 ;
}
```

`switch` statement



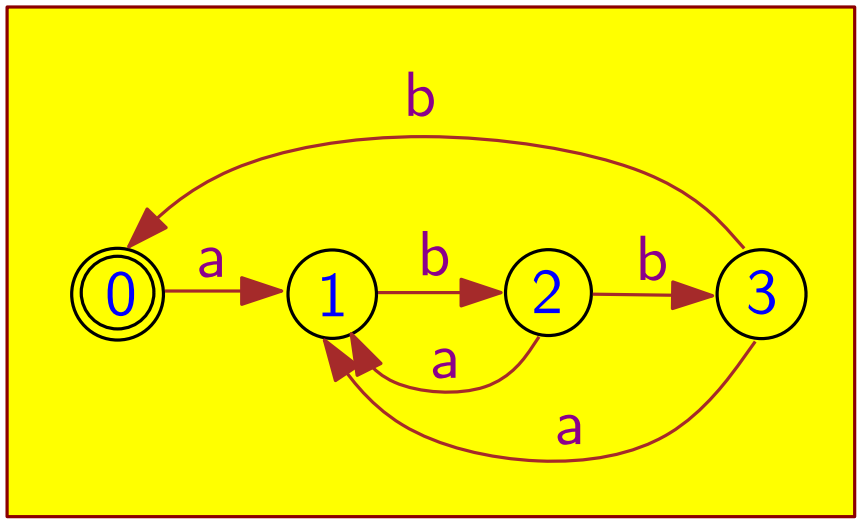

```
int accept(char a[], int n) {
  int i = 0, state = 0 ;
  while ( i < n ) {
    switch ( state ) {
      case 0 :
        if (a[i] == 'a')
         { state = 1 ; break ; }
        else return 0 ;
      case 1 :
        if (a[i] == 'b')
         { state = 2 ; break ; }
        else return 0 ;
      case 2 :
        switch(a[i]) {
           case 'a' : state = 1 ; break ;
           case 'b' : state = 3 ; break ;
           default  : return 0 ;
        }
        break ;
      case 3 :
        switch(a[i]) {
           case 'a' : state = 1 ; break ;
           case 'b' : state = 0 ; break ;
           default  : return 0 ;
        }
        break ;
      default : return 0 ;
    }
    i ++ ;
  }
  if ( state == 0 ) return 1 ;
  else return 0 ;
}
```

switch statement




```
int accept(char a[], int n) {
  int i = 0, state = 0 ;
  while ( i < n ) {
    switch ( state ) {
      case 0 :
        if (a[i] == 'a')
         { state = 1 ; break ; }
        else return 0 ;
      case 1 :
        if (a[i] == 'b')
         { state = 2 ; break ; }
        else return 0 ;
      case 2 :
        switch(a[i]) {
            case 'a' : state = 1 ; break ;
            case 'b' : state = 3 ; break ;
            default  : return 0 ;
        }
        break ;
      case 3 :
        switch(a[i]) {
            case 'a' : state = 1 ; break ;
            case 'b' : state = 0 ; break ;
            default  : return 0 ;
        }
        break ;
      default : return 0 ;
    }
    i ++ ;
  }
  if ( state == 0 ) return 1 ;
  else return 0 ;
}
```

`switch` statement




```
int accept(char a[], int n) {
  int i = 0, state = 0 ;
  while ( i < n ) {
    switch ( state ) {
      case 0 :
        if (a[i] == 'a')
         { state = 1 ; break ; }
        else return 0 ;
      case 1 :
        if (a[i] == 'b')
         { state = 2 ; break ; }
        else return 0 ;
      case 2 :
        switch(a[i]) {
          case 'a' : state = 1 ; break ;
          case 'b' : state = 3 ; break ;
          default  : return 0 ;
        }
        break ;
      case 3 :
        switch(a[i]) {
          case 'a' : state = 1 ; break ;
          case 'b' : state = 0 ; break ;
          default  : return 0 ;
        }
        break ;
      default : return 0 ;
    }
    i ++ ;
  }
  if ( state == 0 ) return 1 ;
  else return 0 ;
}
```

Computed `goto` statement

```
int accept(char a[], int n) {
  int i = 0; unsigned int state = 0 ;
  void *caseptr[4] =
      { &&L1, &&L2, &&L3, &&L4 } ;
  while ( i < n ) {
    if (state > 3) return 0 ;
    goto *caseptr[state] ;
    L1 : // old case 0
      if (a[i] == 'a')
        state = 1 ; goto L5 ;
      else return 0 ;
    L2 : // old case 1
      if (a[i] == 'b')
        state = 2 ; goto L5 ;
      else return 0 ;
```

```
    L3 : // old case 2
      switch(a[i]) {
        case 'a' : state = 1 ;
                   goto L5 ;
        case 'b' : state = 3 ;
                   goto L5 ;
        default  : return 0 ;
      }
    L4 : // old case 3
      switch(a[i]) {
        case 'a' : state = 1 ;
                   goto L5 ;
        case 'b' : state = 0 ;
                   goto L5 ;
        default  : return 0 ;
      }
    }
   L5: i ++ ;
  }
  if ( state == 0 ) return 1 ;
  else return 0 ;
}
```

Computed `goto` statement

Addresses of labels L1, L2, L3, L4

```
int accept(char a[], int n) {
  int i = 0; unsigned int state = 0 ;
  void *caseptr[4] =
      { &&L1, &&L2, &&L3, &&L4 } ;
  while ( i < n ) {
    if (state > 3) return 0 ;
    goto *caseptr[state] ;
    L1 : // old case 0
      if (a[i] == 'a')
        state = 1 ; goto L5 ;
      else return 0 ;
    L2 : // old case 1
      if (a[i] == 'b')
        state = 2 ; goto L5 ;
      else return 0 ;
    L3 : // old case 2
      switch(a[i]) {
        case 'a' : state = 1 ;
                   goto L5 ;
        case 'b' : state = 3 ;
                   goto L5 ;
        default  : return 0 ;
      }
    L4 : // old case 3
      switch(a[i]) {
        case 'a' : state = 1 ;
                   goto L5 ;
        case 'b' : state = 0 ;
                   goto L5 ;
        default  : return 0 ;
      }
    }
   L5: i ++ ;
  }
  if ( state == 0 ) return 1 ;
  else return 0 ;
}
```

Computed `goto` statement

Addresses of labels L1, L2, L3, L4

```
int accept(char a[], int n) {
  int i = 0; unsigned int state = 0 ;
  void *caseptr[4] =
      { &&L1, &&L2, &&L3, &&L4 } ;
  while ( i < n ) {
    if (state > 3) return 0 ;
    goto *caseptr[state] ;
    L1 : // old case 0
      if (a[i] == 'a')
        state = 1 ; goto L5 ;
      else return 0 ;
    L2 : // old case 1
      if (a[i] == 'b')
        state = 2 ; goto L5 ;
      else return 0 ;
    L3 : // old case 2
      switch(a[i]) {
        case 'a' : state = 1 ;
                   goto L5 ;
        case 'b' : state = 3 ;
                   goto L5 ;
        default  : return 0 ;
      }
    L4 : // old case 3
      switch(a[i]) {
        case 'a' : state = 1 ;
                   goto L5 ;
        case 'b' : state = 0 ;
                   goto L5 ;
        default  : return 0 ;
      }
    }
   L5: i ++ ;
  }
  if ( state == 0 ) return 1 ;
  else return 0 ;
}
```

Computed goto jumps to one of L1, L2, L3, L4 depending on state

Computed goto statement

Addresses of labels L1, L2, L3, L4

```
int accept(char a[], int n) {
  int i = 0; unsigned int state = 0 ;
  void *case_ptr[4] =
      { &&L1, &&L2, &&L3, &&L4 } ;
  while ( i < n ) {
    goto *case_ptr[state] ;
    L1 : // old case 0
      switch(a[i]) {
        case 'a' : state = 1 ;
                   goto L5 ;
        case 'b' : state = 3 ;
                   goto L5 ;
        default  : return 0 ;
      }
    L2 : // old case 1
      switch(a[i]) {
        case 'b' : state = 2 ;
                   goto L5 ;
        default  : return 0 ;
      }
    L3 : // old case 2
      switch(a[i]) {
        case 'a' : state = 1 ;
                   goto L5 ;
        default  : return 0 ;
      }
    L4 : // old case 3
      switch(a[i]) {
        case 'b' : state = 3 ;
                   goto L5 ;
        default  : return 0 ;
      }
    L5: i ++ ;
  }
  if ( state == 0 ) return 1 ;
  else return 0 ;
}
```

Computed gotos are a GCC extension only, and are not part of the standard language.

Computed goto jumps to one of L1, L2, L3, L4 depending on state

Recursion Refresher

```
int power( int a, int b ) {
  int x ;
  if ( b <= 0 ) return 1 ;
  if ( b & 1 )
    return a * power( a, b-1 ) ;
  x = power( a, b>>1 ) ;
  return x * x ;
}
```

$$a^b = \begin{cases} 1, & \text{if } b \leq 0 \\ a \cdot \left(a^{b-1}\right), & \text{if } b \text{ is odd} \\ \left(a^{b/2}\right)^2, & \text{if } b \text{ is even} \end{cases}$$

Recursion Refresher

```
int power( int a, int b ) {
  int x ;
  if ( b <= 0 ) return 1 ;
  if ( b & 1 )
    return a * power( a, b-1 ) ;
  x = power( a, b>>1 ) ;
  return x * x ;
}
```

$$a^b = \begin{cases} 1, & \text{if } b \leq 0 \\ a \cdot \left(a^{b-1}\right), & \text{if } b \text{ is odd} \\ \left(a^{b/2}\right)^2, & \text{if } b \text{ is even} \end{cases}$$

Recursion Refresher

```
int power( int a, int b ) {
  int x ;
  if ( b <= 0 ) return 1 ;
  if ( b & 1 )
    return a * power( a, b-1 ) ;
  x = power( a, b>>1 ) ;
  return x * x ;
}
```

$$a^b = \begin{cases} 1, & \text{if } b \leq 0 \\ a \cdot \left(a^{b-1}\right), & \text{if } b \text{ is odd} \\ \left(a^{b/2}\right)^2, & \text{if } b \text{ is even} \end{cases}$$

Recursion Refresher

```
int power( int a, int b ) {
  int x ;
  if ( b <= 0 ) return 1 ;
  if ( b & 1 )
    return a * power( a, b-1 ) ;
  x = power( a, b>>1 ) ;
  return x * x ;
}
```

$$a^b = \begin{cases} 1, & \text{if } b \leq 0 \\ a \cdot \left(a^{b-1}\right), & \text{if } b \text{ is odd} \\ \left(a^{b/2}\right)^2, & \text{if } b \text{ is even} \end{cases}$$

Recursion Refresher

```
int power( int a, int b ) {
  int x ;
  if ( b <= 0 ) return 1 ;
  if ( b & 1 )
    return a * power( a, b-1 ) ;
  x = power( a, b>>1 ) ;
  return x * x ;
}
```

$$
\begin{aligned}
2^7 &= \\
2 \cdot 2^6 &= \\
2 \cdot \text{square}(2^3) &= \\
2 \cdot \text{square}(2 \cdot 2^2) &= \\
2 \cdot \text{square}(2 \cdot \text{square}(2))
\end{aligned}
$$

Mathematical definitions are naturally recursive!

$$
a^b = \begin{cases} 1, & \text{if } b \leq 0 \\ a \cdot \left(a^{b-1}\right), & \text{if } b \text{ is odd} \\ \left(a^{b/2}\right)^2, & \text{if } b \text{ is even} \end{cases}
$$

Recursion in programming allows implementation of mathematical definitions!

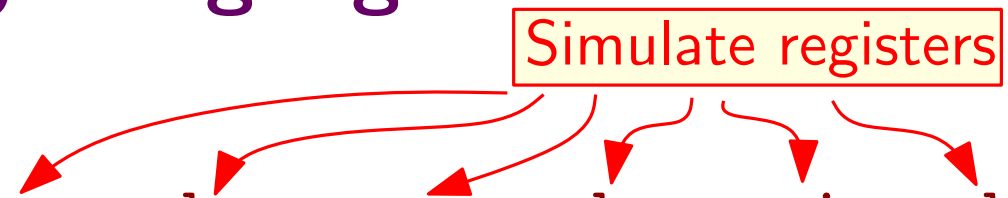
Assembly Languages

- Means of making machine languages more readable
- Execution unit: *instruction*
  - Very limited amout of computation
- No structured programming
- Programs: large in terms of lines of code
- Different for each architecture
  - Pentium AL $\neq$ MIPS AL
- We abstract AL as a subset of C
  - Interest in low-level programming skill
  - No interest in specific architecture

VAL: Vanilla Assembly Language

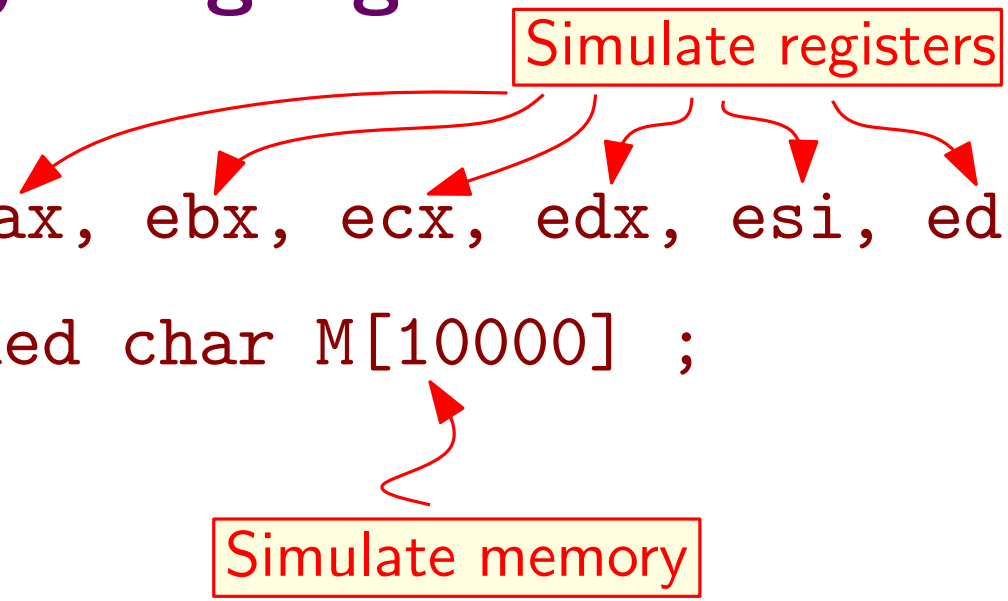
- Data:
  - 6 global variables: `int eax, ebx, ecx, edx, esi, edi ;`
  - One global array: `unsigned char M[10000] ;`
  - No local variables!
- Functions:
  - One single function contains all the code
  - Prototype: `void exec(void) ;`

VAL: Vanilla Assembly Language



- Data:
  - 6 global variables: `int eax, ebx, ecx, edx, esi, edi ;`
  - One global array: `unsigned char M[10000] ;`
  - No local variables!
- Functions:
  - One single function contains all the code
  - Prototype: `void exec(void) ;`

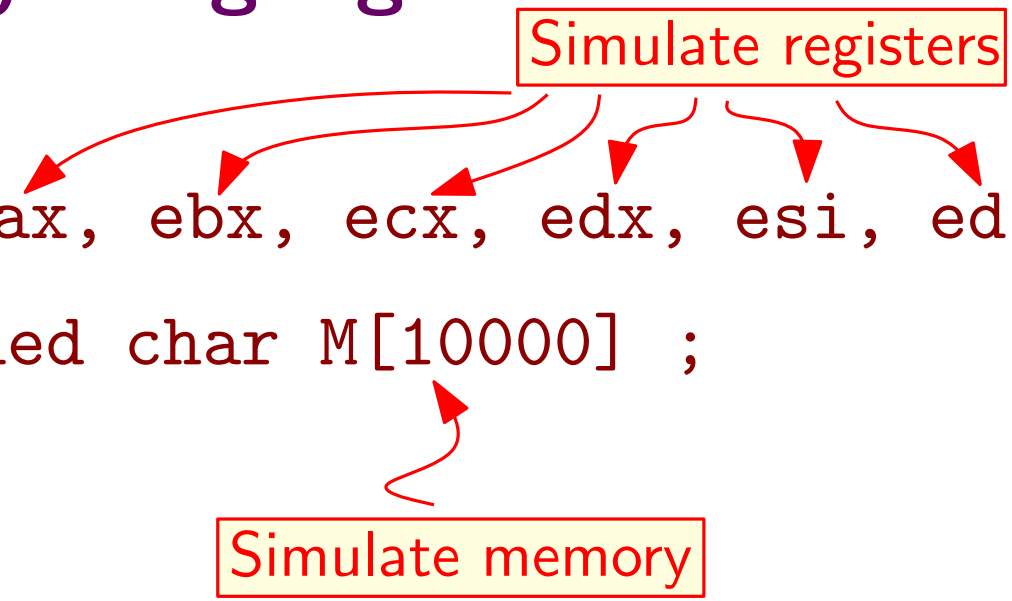
VAL: Vanilla Assembly Language



- Data:
  - 6 global variables: `int eax, ebx, ecx, edx, esi, edi ;`
  - One global array: `unsigned char M[10000] ;`
  - No local variables!
- Functions:
  - One single function contains all the code
  - Prototype: `void exec(void) ;`

VAL: Vanilla Assembly Language

- Data:
  - 6 global variables: `int eax, ebx, ecx, edx, esi, edi ;`
  - One global array: `unsigned char M[10000] ;`
  - No local variables!
- Functions:
  - One single function contains all the code
  - Prototype: `void exec(void) ;`



Vanilla Assembly Language (VAL): Code

- Operands:
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant,*
    glob_var+constant, glob_var-constant
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`],`
    `*(unsigned int *)&M[`*index_expr*`],`
    `*(short*)&M[`*index_expr*`],`
    `*(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: `M[ecx+edx], M[ecx*2],`
      `eax*ebx, *(int*)eax`

Vanilla Assembly Language (VAL): Code

- Operands: Simulates Pentium AL operands
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant,*
    glob_var+constant, glob_var-constant
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`],`
    `*(unsigned int *)&M[`*index_expr*`],`
    `*(short*)&M[`*index_expr*`],`
    `*(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: `M[ecx+edx], M[ecx*2],`
      `eax*ebx, *(int*)eax`

Vanilla Assembly Language (VAL): Code

- Operands:
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant,*
    glob_var+constant, glob_var-constant
    → 1, -10, 20, 'a', &&label, etc
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`],`
    `*(unsigned int *)&M[`*index_expr*`],`
    `*(short*)&M[`*index_expr*`],`
    `*(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: `M[ecx+edx], M[ecx*2], eax*ebx, *(int*)eax`

Vanilla Assembly Language (VAL): Code

- Operands:
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant, glob_var+constant, glob_var-constant*
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`], *(unsigned int *)&M[`*index_expr*`], *(short*)&M[`*index_expr*`], *(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: `M[ecx+edx], M[ecx*2], eax*ebx, *(int*)eax`

Memory is byte-accessible.

To form 4-byte entities (integers), we need to be explicit.

Vanilla Assembly Language (VAL): Code

- Operands:
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant,*
    glob_var+constant, glob_var-constant
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`],`
    `*(unsigned int *)&M[`*index_expr*`],`
    `*(short*)&M[`*index_expr*`],`
    `*(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: `M[ecx+edx], M[ecx*2],`
      `eax*ebx, *(int*)eax`

Vanilla Assembly Language (VAL): Code

- Operands:
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant,*
    glob_var+constant, glob_var-constant
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`],`
    `*(unsigned int *)&M[`*index_expr*`],`
    `*(short*)&M[`*index_expr*`],`
    `*(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: `M[ecx+edx], M[ecx*2],`
      `eax*ebx, *(int*)eax`

Vanilla Assembly Language (VAL): Code

- Operands:
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant,*
    glob_var+constant, glob_var-constant
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`],`
    `*(unsigned int *)&M[`*index_expr*`],`
    `*(short*)&M[`*index_expr*`],`
    `*(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: `M[ecx+edx], M[ecx*2],`
      `eax*ebx, *(int*)eax`

Vanilla Assembly Language (VAL): Code

- Operands:
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant,*
    glob_var+constant, glob_var-constant
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`],`
    `*(unsigned int *)&M[`*index_expr*`],`
    `*(short*)&M[`*index_expr*`],`
    `*(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: `M[ecx+edx], M[ecx*2],`
      `eax*ebx, *(int*)eax`

Vanilla Assembly Language (VAL): Code

- Operands:
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant,*
    glob_var+constant, glob_var-constant
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`],`
    `*(unsigned int *)&M[`*index_expr*`],`
    `*(short*)&M[`*index_expr*`],`
    `*(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: `M[ecx+edx], M[ecx*2],`
      `eax*ebx, *(int*)eax`

Vanilla Assembly Language (VAL): Code

- Operands:
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant,*
    glob_var+constant, glob_var-constant
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`],`
    `*(unsigned int *)&M[`*index_expr*`],`
    `*(short*)&M[`*index_expr*`],`
    `*(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: `M[ecx+edx], M[ecx*2],`
      `eax*ebx, *(int*)eax`

Vanilla Assembly Language (VAL): Code

- Operands:
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant,*
    glob_var+constant, glob_var-constant
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`],`
    `*(unsigned int *)&M[`*index_expr*`],`
    `*(short*)&M[`*index_expr*`],`
    `*(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: ~~`M[ecx+edx]`~~, `M[ecx*2]`, `eax*ebx`, `*(int*)eax`

Vanilla Assembly Language (VAL): Code

- Operands:
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant,*
    glob_var+constant, glob_var-constant
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`],`
    `*(unsigned int *)&M[`*index_expr*`],`
    `*(short*)&M[`*index_expr*`],`
    `*(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: ~~`M[ecx+edx]`~~, ~~`M[ecx*2]`~~, `eax*ebx`, `*(int*)eax`

Vanilla Assembly Language (VAL): Code

- Operands:
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant,*
    glob_var+constant, glob_var-constant
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`],`
    `*(unsigned int *)&M[`*index_expr*`],`
    `*(short*)&M[`*index_expr*`],`
    `*(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: ~~`M[ecx+edx]`~~, ~~`M[ecx*2]`~~, ~~`eax*ebx`~~, `*(int*)eax`

Vanilla Assembly Language (VAL): Code

- Operands:
  - Global vars: `eax, ebx, ecx, edx, esi, edi`
  - Index expression: *glob_var, constant,*
    glob_var+constant, glob_var-constant
  - Array ref: `M[`*index_expr*`], *(int*)&M[`*index_expr*`],`
    `*(unsigned int *)&M[`*index_expr*`],`
    `*(short*)&M[`*index_expr*`],`
    `*(unsigned short *)&M[`*index_expr*`]`
  - Examples:
    - register operand: `ebx`
    - integer in memory (4 bytes) : `*(int*)&M[esi]`
    - unsigned short in memory (2 bytes) : `*(unsigned short*)&M[edi+4]`
    - char in memory (1 byte) : `M[ecx-1]`
    - unsigned int in memory (4 bytes): `*(unsigned int*)&M[12]`
    - not allowed: ~~`M[ecx+edx]`~~, ~~`M[ecx*2]`~~, ~~`eax*ebx`~~, ~~`*(int*)eax`~~

VAL: Code

- Assignments
  - No two memory references (`M[...]`) in the same instruction!!!
  - *operand* = *operand*
  - *operand* += *operand*
  - *operand* -= *operand*
  - *operand* *= *operand*
  - etc... all shortcut assignment operators allowed
- Examples:
  - `eax = ebx`
  - `eax <<= *(int*)&M[ebx]`
  - `eax = M[ebx]`
  - `*(int*)&M[ebx+4] = esi`
  - `*(int*)&M[ebx-12] = 10`
  - `M[esi] += '0'`

Illegal:

- `eax = ebx+ecx`
- `eax = M[ebx+ecx]`
- `M[eax] += M[ebx]`

VAL: Code

`goto` Instructions

`goto` *label*

`goto * operand` – operand's value must be of the form && label

Boolean expressions:

operand < *operand*, *operand* <= *operand*, *operand* == *operand*

operand != *operand*, *operand* > *operand*, *operand* >= *operand*

No two memory operands in the same expression!!!

"If" statements:

`if ( boolean_expr ) goto label/operand ;`

VAL: Code

`goto` Instructions

`goto` *label*

`goto * operand` – operand's value must be of the form && label

Boolean expressions:

operand < *operand*, *operand* <= *operand*, *operand* == *operand*

operand != *operand*, *operand* > *operand*, *operand* >= *operand*

No two memory operands in the same expression!!!

"If" statements:

`if ( boolean_expr ) goto label/operand ;`

Example

Original code:

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

Equivalent VAL code

```
int eax, ebx, ecx, edx, esi, edi ;
unsigned char M[10000] ;

// arguments a,b in ecx, edx
// return value in eax, at end of func
void exec() {
  eax = 1; ebx = 31 ;
  if ( edx < 0 ) goto return_0 ;
loop:
  ebx -= 1 ;
  if ( ebx < 0 ) goto return_p ;
  edi = edx ;  edi &= 0x40000000 ;
  if ( edi ) goto then_branch ;
  eax *= eax ;
  edx <<= 1 ;
  goto end_if ;
then_branch:
  eax *= eax ; eax *= ecx ;
  edx &= 0x3fffffff ; edx <<= 1 ;
end_if:
  goto loop ;
return_0:
  eax = 0 ;
return_p:{}
}
```

Example

Original code:

convention

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

Equivalent VAL code

```
int eax, ebx, ecx, edx, esi, edi ;
unsigned char M[10000] ;

// arguments a,b in ecx, edx
// return value in eax, at end of func
void exec() {
  eax = 1; ebx = 31 ;
  if ( edx < 0 ) goto return_0 ;
loop:
  ebx -= 1 ;
  if ( ebx < 0 ) goto return_p ;
  edi = edx ;  edi &= 0x40000000 ;
  if ( edi ) goto then_branch ;
  eax *= eax ;
  edx <<= 1 ;
  goto end_if ;
then_branch:
  eax *= eax ; eax *= ecx ;
  edx &= 0x3fffffff ; edx <<= 1 ;
end_if:
  goto loop ;
return_0:
  eax = 0 ;
return_p:{}
}
```

Label must point to a statement

Example

Original code:

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

Equivalent VAL code

```
int eax, ebx, ecx, edx, esi, edi ;
unsigned char M[10000] ;

// arguments a,b in ecx, edx
// return value in eax, at end of func
void exec() {
  eax = 1; ebx = 31 ;
  if ( edx < 0 ) goto return_0 ;
loop:
  ebx -= 1 ;
  if ( ebx < 0 ) goto return_p ;
  edi = edx ;  edi &= 0x40000000 ;
  if ( edi ) goto then_branch ;
  eax *= eax ;
  edx <<= 1 ;
  goto end_if ;
then_branch:
  eax *= eax ; eax *= ecx ;
  edx &= 0x3fffffff ; edx <<= 1 ;
end_if:
  goto loop ;
return_0:
  eax = 0 ;
return_p:{}
}
```

Example

Original code:

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

Equivalent VAL code

```
int eax, ebx, ecx, edx, esi, edi ;
unsigned char M[10000] ;

// arguments a,b in ecx, edx
// return value in eax, at end of func
void exec() {
  eax = 1; ebx = 31 ;
  if ( edx < 0 ) goto return_0 ;
loop:
  ebx -= 1 ;
  if ( ebx < 0 ) goto return_p ;
  edi = edx ;  edi &= 0x40000000 ;
  if ( edi ) goto then_branch ;
  eax *= eax ;
  edx <<= 1 ;
  goto end_if ;
then_branch:
  eax *= eax ; eax *= ecx ;
  edx &= 0x3fffffff ; edx <<= 1 ;
end_if:
  goto loop ;
return_0:
  eax = 0 ;
return_p:{}
}
```

Example

Original code:

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

Equivalent VAL code

```
int eax, ebx, ecx, edx, esi, edi ;
unsigned char M[10000] ;

// arguments a,b in ecx, edx
// return value in eax, at end of func
void exec() {
  eax = 1; ebx = 31 ;
  if ( edx < 0 ) goto return_0 ;
loop:
  ebx -= 1 ;
  if ( ebx < 0 ) goto return_p ;
  edi = edx ;  edi &= 0x40000000 ;
  if ( edi ) goto then_branch ;
  eax *= eax ;
  edx <<= 1 ;
  goto end_if ;
then_branch:
  eax *= eax ; eax *= ecx ;
  edx &= 0x3fffffff ; edx <<= 1 ;
end_if:
  goto loop ;
return_0:
  eax = 0 ;
return_p:{}
}
```

Example

Original code:

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

Equivalent VAL code

```
int eax, ebx, ecx, edx, esi, edi ;
unsigned char M[10000] ;

// arguments a,b in ecx, edx
// return value in eax, at end of func
void exec() {
  eax = 1; ebx = 31 ;
  if ( edx < 0 ) goto return_0 ;
loop:
  ebx -= 1 ;
  if ( ebx < 0 ) goto return_p ;
  edi = edx ;  edi &= 0x40000000 ;
  if ( edi ) goto then_branch ;
  eax *= eax ;
  edx <<= 1 ;
  goto end_if ;
then_branch:
  eax *= eax ; eax *= ecx ;
  edx &= 0x3fffffff ; edx <<= 1 ;
end_if:
  goto loop ;
return_0:
  eax = 0 ;
return_p:{}
}
```

Example

Original code:

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

Equivalent VAL code

```
int eax, ebx, ecx, edx, esi, edi ;
unsigned char M[10000] ;

// arguments a,b in ecx, edx
// return value in eax, at end of func
void exec() {
  eax = 1; ebx = 31 ;
  if ( edx < 0 ) goto return_0 ;
loop:
  ebx -= 1 ;
  if ( ebx < 0 ) goto return_p ;
  edi = edx ;  edi &= 0x40000000 ;
  if ( edi ) goto then_branch ;
  eax *= eax ;
  edx <<= 1 ;
  goto end_if ;
then_branch:
  eax *= eax ; eax *= ecx ;
  edx &= 0x3fffffff ; edx <<= 1 ;
end_if:
  goto loop ;
return_0:
  eax = 0 ;
return_p:{}
}
```

Example

Original code:

```
int power(int a, int b) {
  int s = 1, cnt = 31 ;
  if ( b < 0 ) return 0 ;
  while ( --cnt >= 0 ) {
    if ( b & (1<<30) ) {
      s = s*s*a ;
      b = (b&((1<<30)-1))<<1 ;
    } else {
      s *= s ;
      b <<= 1 ;
    }
  }
  return s ;
}
```

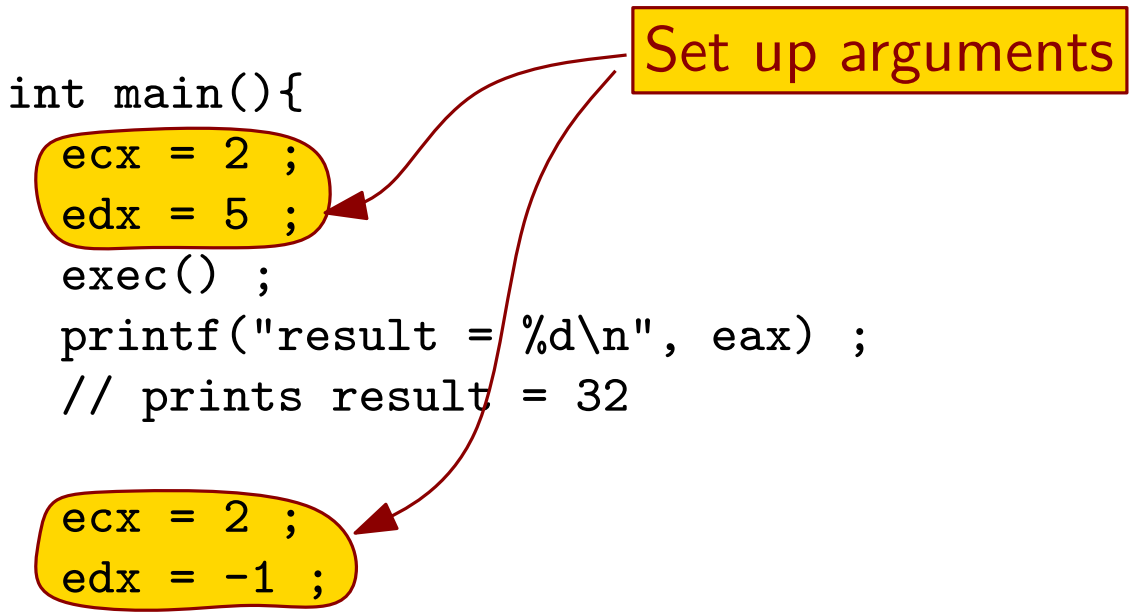
Equivalent VAL code

```
int eax, ebx, ecx, edx, esi, edi ;
unsigned char M[10000] ;

// arguments a,b in ecx, edx
// return value in eax, at end of func
void exec() {
  eax = 1; ebx = 31 ;
  if ( edx < 0 ) goto return_0 ;
loop:
  ebx -= 1 ;
  if ( ebx < 0 ) goto return_p ;
  edi = edx ;  edi &= 0x40000000 ;
  if ( edi ) goto then_branch ;
  eax *= eax ;
  edx <<= 1 ;
  goto end_if ;
then_branch:
  eax *= eax ; eax *= ecx ;
  edx &= 0x3fffffff ; edx <<= 1 ;
end_if:
  goto loop ;
return_0:
  eax = 0 ;
return_p:{}
}
```

Testing the Code

```
int main(){
  ecx = 2 ;
  edx = 5 ;
  exec() ;
  printf("result = %d\n", eax) ;
  // prints result = 32

  ecx = 2 ;
  edx = -1 ;
  exec() ;
  printf("result = %d\n", eax) ;
  // prints result = 0

  getchar() ; // prevent closing the window
}
```

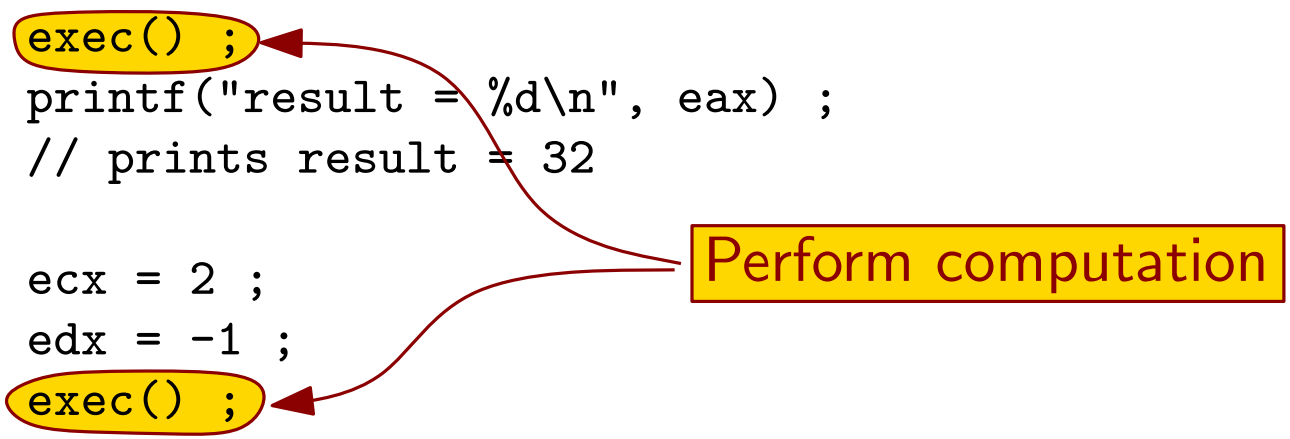
Testing the Code




```
int main(){
  ecx = 2 ;
  edx = 5 ;
  exec() ;
  printf("result = %d\n", eax) ;
  // prints result = 32

  ecx = 2 ;
  edx = -1 ;
  exec() ;
  printf("result = %d\n", eax) ;
  // prints result = 0

  getchar() ; // prevent closing the window
}
```

Testing the Code

```
int main(){
  ecx = 2 ;
  edx = 5 ;
  exec() ;
  printf("result = %d\n", eax) ;
  // prints result = 32

  ecx = 2 ;
  edx = -1 ;
  exec() ;
  printf("result = %d\n", eax) ;
  // prints result = 0

  getchar() ; // prevent closing the window
}
```



Testing the Code

```
int main(){
  ecx = 2 ;
  edx = 5 ;
  exec() ;
  printf("result = %d\n", eax) ;
  // prints result = 32

  ecx = 2 ;
  edx = -1 ;
  exec() ;
  printf("result = %d\n", eax) ;
  // prints result = 0

  getchar() ; // prevent closing the window
}
```

Print results, so as to verify correctness

Testing the Code

```
int main(){
  ecx = 2 ;
  edx = 5 ;
  exec() ;
  printf("result = %d\n", eax) ;
  // prints result = 32

  ecx = 2 ;
  edx = -1 ;
  exec() ;
  printf("result = %d\n", eax) ;
  // prints result = 0

  getchar() ; // prevent closing the window
}
```

No computation is allowed in the `main()` of a VAL program!

The `main()` is only for testing purposes.

- Set up arguments
- Call `exec()`
- Print result

The `main()` is not part of your answer to an exercise that requires you to write a VAL program. If you provide a `main()` as part of your VAL program in an exam answer, the `main()` function will be ignored!

Conclusion

- All software executes, in the end, assembly language code
- C was devised as a portable assembly language
- All other languages are implemented in C
- It is important to understand the relationship of C to AL, as well as C to other languages
- We explore this relationship with the aid of *translation schemes*