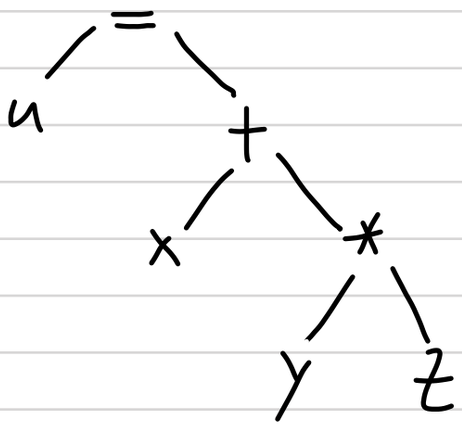
Straight Code generation

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



memory access is relative to register 28 which contains the base address of global variables

```
SP=0:  LDW 1, 28, adr(x)   reg[1] = x
SP=1:  LDW 2, 28, adr(y)   reg[2] = y
SP=2:  LDW 3, 28, adr(z)   reg[3] = z
SP=3:  MUL 2, 2, 3         reg[2] = reg[2] * reg[3]
SP=2:  ADD 1, 1, 2         reg[1] = reg[1] + reg[2]
SP=1:  STW 1, 28, adr(u)   u = reg[1]
SP=0:
```

stack-based register allocation (compiler-firm concept)

↝ we will later use a different register allocation strategy

Code Generation Rules

assignment = identifier "=" expression .

- Parser<expression>
- STW sp, 28, adr(variable)
- sp = sp - 1

expression = term . → Parser<term>

expression = expression "+" term .

- Parser<expression>
- Parser<term>
- sp = sp - 1
- ADD sp, sp, sp+1

not left-factored hier

expression = expression "-" term .

- Parser<expression>
- Parser<term>
- sp = sp - 1
- SUB sp, sp, sp+1

term = factor . → Parser<factor>

term = term "*" factor .

- Parser<term>
- Parser<factor>
- sp = sp - 1
- MUL sp, sp, sp+1

term = term "/" factor .

- Parser<term>
- Parser<factor>
- sp = sp - 1
- DIV sp, sp, sp+1

factor = identifier(variable) .

- sp = sp + 1
- LDW sp, 28, adr(variable)

factor = integer(value) .

- sp = sp + 1
- ADDI sp, 0, value

factor = "(" expression ")" .

- Parser<expression>

assumes reg[0] == 0! alternatively → F1: MOVI a,c : reg[a]=c

Encode instructions

```
int op;
int a;
int b;
int c;

int instruction;
```

see decoding (target machine)

```
encode() {
 // in compiler and linker!
 // assuming: 0 <= op <= 2^6-1 = 63
 // assuming: 0 <= a <= 2^5-1 = 31
 // assuming: 0 <= b <= 2^5-1 = 31
 // assuming: -32768 = -2^15 <= c <= 2^26-1 = 67108863
 // assuming: if c > 2^15-1 = 32767 then a == 0 and b == 0
 if (c < 0)
   c = c + 65536; // 0x10000: 2^16
 // if << is not available
 // replace (x << 5) by (x * 32) and (x << 16) by (x * 65536)
 instruction = (((((op << 5) + a) << 5) + b) << 16) + c;
}
```

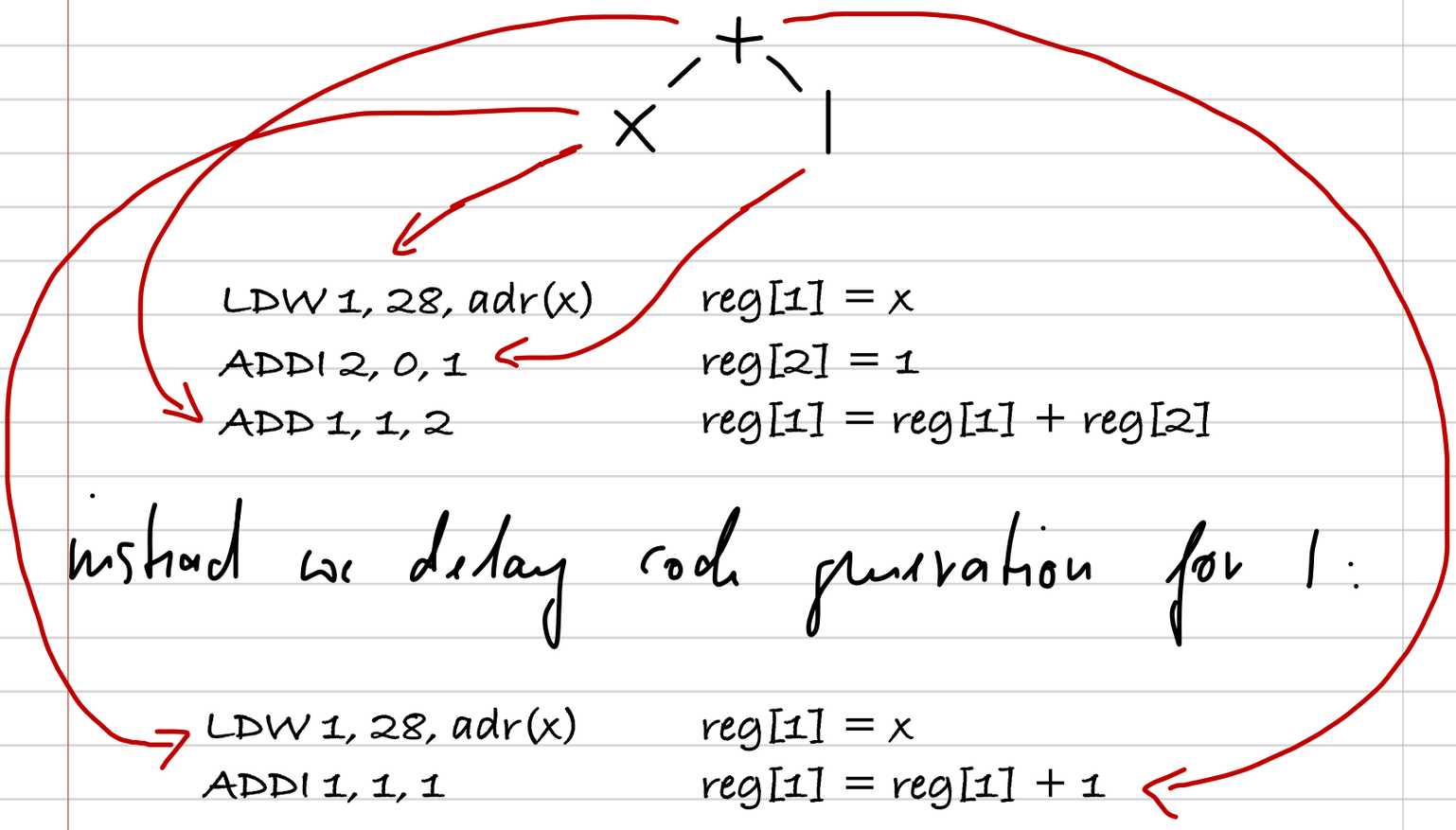
sign-preserving down-scaling to 16 bits (from 32 bits)

2^5

2^{16}

Delayed Code Generation

x + 1



instead we delay code generation for 1:

→ parser procedures get a (inout) parameter that represents an attribute for delayed code generation:

- CONST
- VAR
- REG
(- REF)

```
struct item_t {
  int mode;
  struct type_t *type;
  int reg;
  int offset;
  int value;
  ...
}
```

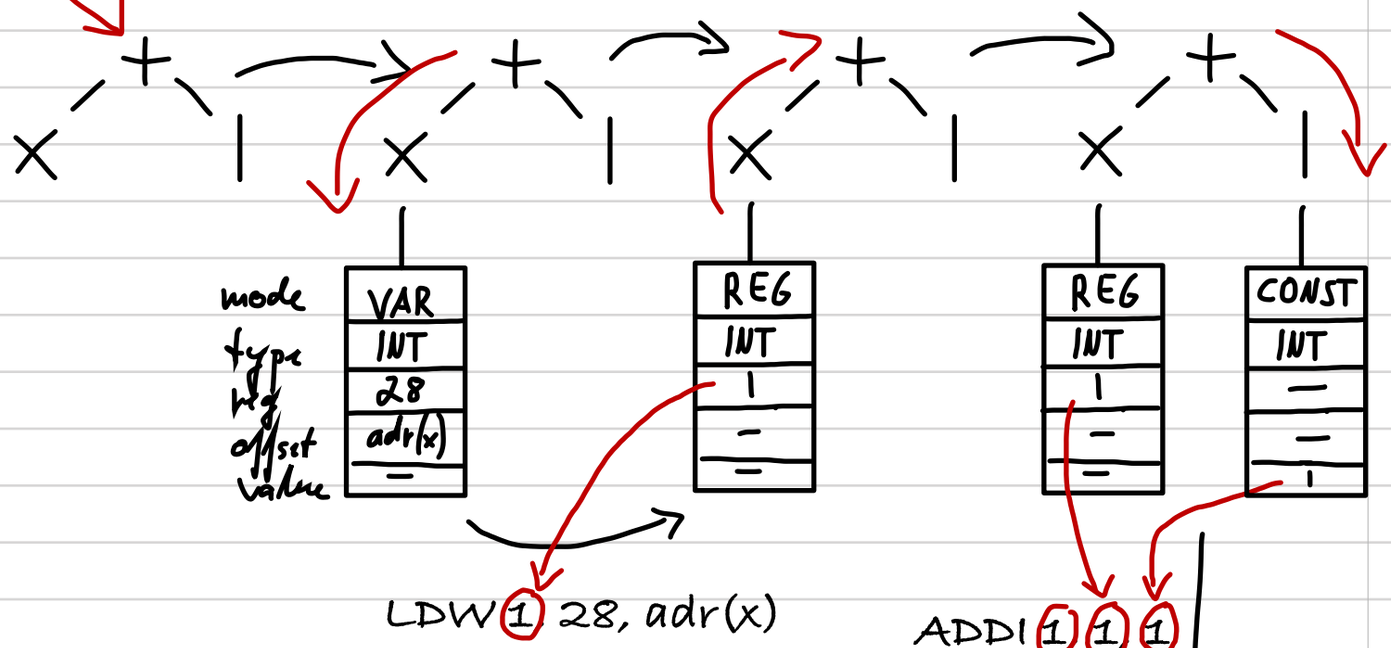
"reg[reg]" + offset → address

→ constant value

more later

Example

current parsing position



register allocation:

int requestRegister();

- returns unused register (first-fit)
- maintains a 32-element array of bits
- a set bit @ index i indicates that reg[i] is used

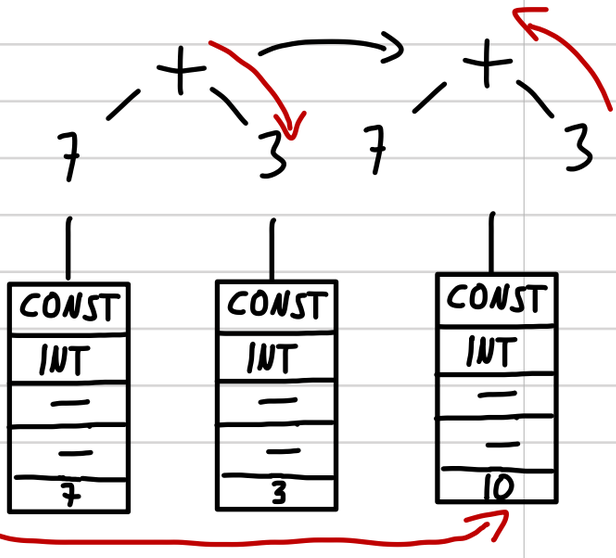
releaseRegister(int i);

- resets bit@index i to release reg[i]

7+3

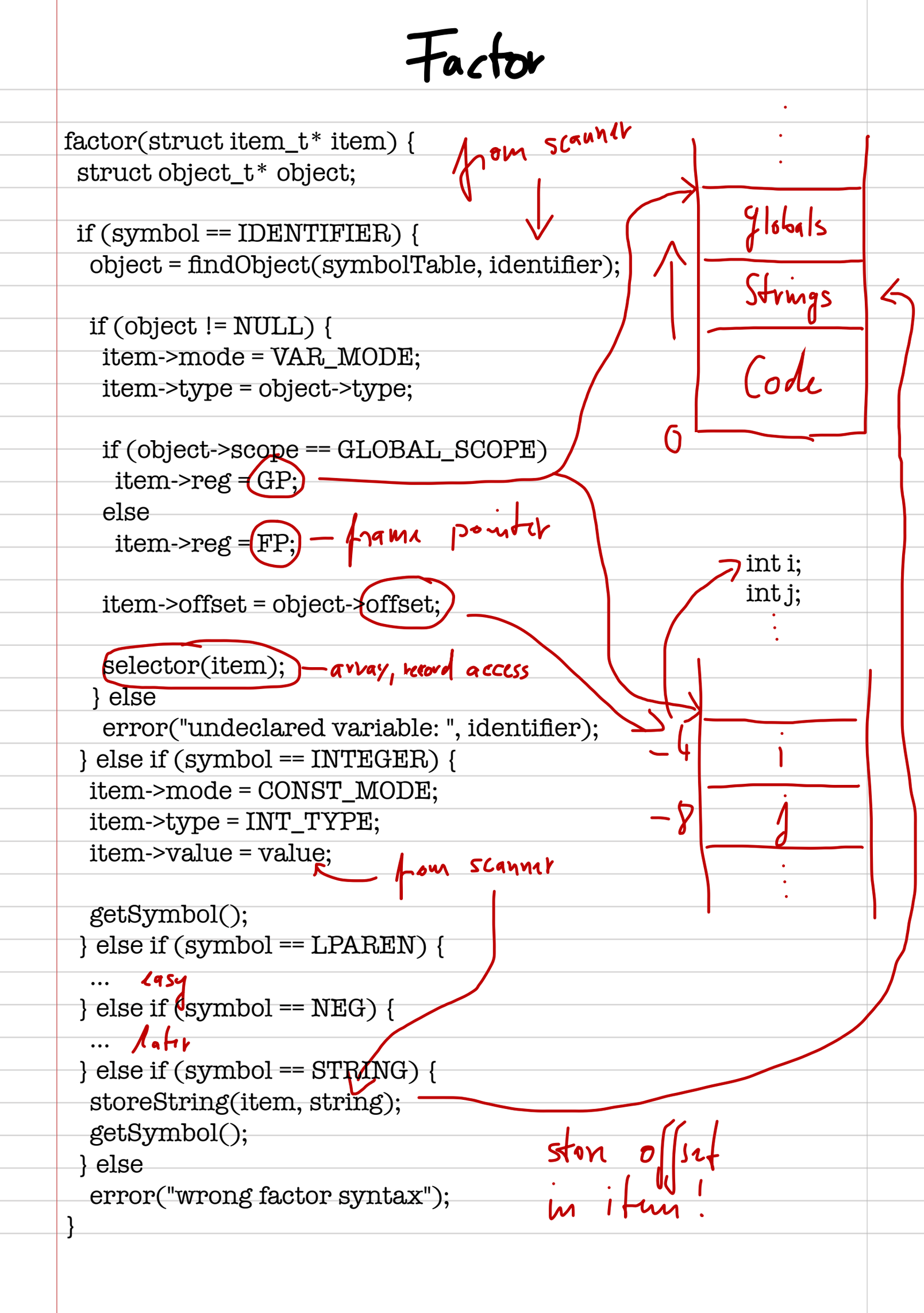
constant folding:

- evaluate constant subexpressions at compile time
- handle overflow!
- delay mode switch from CONST



Factor

```
factor(struct item_t* item) {
 struct object_t* object;

 if (symbol == IDENTIFIER) {
  object = findObject(symbolTable, identifier);

  if (object != NULL) {
   item->mode = VAR_MODE;
   item->type = object->type;

   if (object->scope == GLOBAL_SCOPE)
    item->reg = GP;
   else
    item->reg = FP;

   item->offset = object->offset;

   selector(item);
  } else
   error("undeclared variable: ", identifier);
 } else if (symbol == INTEGER) {
  item->mode = CONST_MODE;
  item->type = INT_TYPE;
  item->value = value;

  getSymbol();
 } else if (symbol == LPAREN) {
  ...
 } else if (symbol == NEG) {
  ...
 } else if (symbol == STRING) {
  storeString(item, string);
  getSymbol();
 } else
  error("wrong factor syntax");
}
```



Simple Expression

→ arithmetic and boolean operators, no comparison operators

```
simpleExpressionBinaryOperator(struct item_t* leftItem,
                               struct item_t* rightItem,
                               int operatorSymbol) {
 if (operatorSymbol == OR) {
  ... later
 } else if ((leftItem->type == INT_TYPE) &&
            (rightItem->type == INT_TYPE)) {
  if (rightItem->mode == CONST_MODE) {
   if (leftItem->mode == CONST_MODE) {
    if (operatorSymbol == ADD)
     leftItem->value = leftItem->value + rightItem->value;
    else if (operatorSymbol == SUB)
     leftItem->value = leftItem->value - rightItem->value;
   } else {
    load(leftItem);
    if (operatorSymbol == ADD)
     put(ADDI, leftItem->reg, leftItem->reg, rightItem->value);
    else if (operatorSymbol == SUB)
     put(SUBI, leftItem->reg, leftItem->reg, rightItem->value);
   }
  } else {
   load(leftItem);
   load(rightItem);
   if (operatorSymbol == ADD)
    put(ADD, leftItem->reg, leftItem->reg, rightItem->reg);
   else if (operatorSymbol == SUB)
    put(SUB, leftItem->reg, leftItem->reg, rightItem->reg);
   releaseRegister(rightItem->reg);
  }
 } else
  error("integer expressions expected");
}
```

← handle overflow!

→ switch to REG mode

← leftItem may be in CONST_MODE! (optimization possible, not done here)

← important to avoid register leak!

↝ similarly for terms

Mode Switching

```
load(struct item_t* item) {
 if (item->mode == CONST_MODE)
   const2Reg(item);
 else if (item->mode == VAR_MODE)
   var2Reg(item);
 else if (item->mode == REF_MODE)
   ref2Reg(item);
}
```

← later

↳ switches to REG mode by emitting code

```
const2Reg(struct item_t* item) {
 item->mode = REG_MODE;
 item->reg = requestRegister();

 // assumes: reg[0]==0 for MOVI semantics
 put(ADDI, item->reg, 0, item->value);

 item->value = 0;
 item->offset = 0;
}
```

loads item->value into reg[item->reg]

```
var2Reg(struct item_t* item) {
 int newReg;

 item->mode = REG_MODE;
 newReg = requestRegister();

 put(LDW, newReg, item->reg, item->offset);

 item->reg = newReg;
 item->offset = 0;
}
```

loads value of variable @ mem[reg[item->reg] + item->offset] into reg[newReg]

Assignment

```
assignment() {
  struct item_t* leftItem;
  struct item_t* rightItem;
  struct object_t* object;

  leftItem = malloc(sizeof(struct item_t));

  if (symbol == IDENTIFIER) {
    ...
  } else
    error("identifier expected");

  if (symbol == ASSIGN)
    getSymbol();
  else
    error("assignment expected");

  rightItem = malloc(sizeof(struct item_t));
  expression(rightItem);

  assignmentOperator(leftItem, rightItem);
}


assignmentOperator(struct item_t* leftItem,
                   struct item_t* rightItem) {
  if (leftItem->type != rightItem->type)
    warning("type mismatch in assignment");

  load(rightItem);

  // leftItem must be in VAR_MODE, rightItem must be in REG_MODE
  put(STW, rightItem->reg, leftItem->reg, leftItem->offset);

  if (leftItem->mode == REF_MODE)
    releaseRegister(leftItem->reg);
  releaseRegister(rightItem->reg);
}
```