Head and Body

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

 item = malloc(sizeof(struct item_t));
 returnType(item);

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

  if (object != NULL) {
   if (object->type != item->type)
    warning("return type mismatch in procedure: ", identifier);
   fixLink(object->offset);
  } else {
   object = createObject(symbolTable, identifier);
   object->class = PROC;
  }

  getSymbol();
  object->type = item->type;
  object->offset = PC;
  formalParameters(object);
 } else error("identifier expected");

 returnFJumpAddress = 0;

 if (symbol == BEGIN) getSymbol(); else error("missing '{'");

 prologue(variableDeclarationSequence(object) * 4);
 procedureContext = object;
 statementSequence();
 fixLink(returnFJumpAddress);
 epilogue(object->value * 4);

 if (symbol == END) getSymbol(); else error("missing '}'");
}
```

head

body

← procedure already declared or called

← check if procedure has already been implemented! (this would be an error)

← next instruction will be first instruction of the procedure

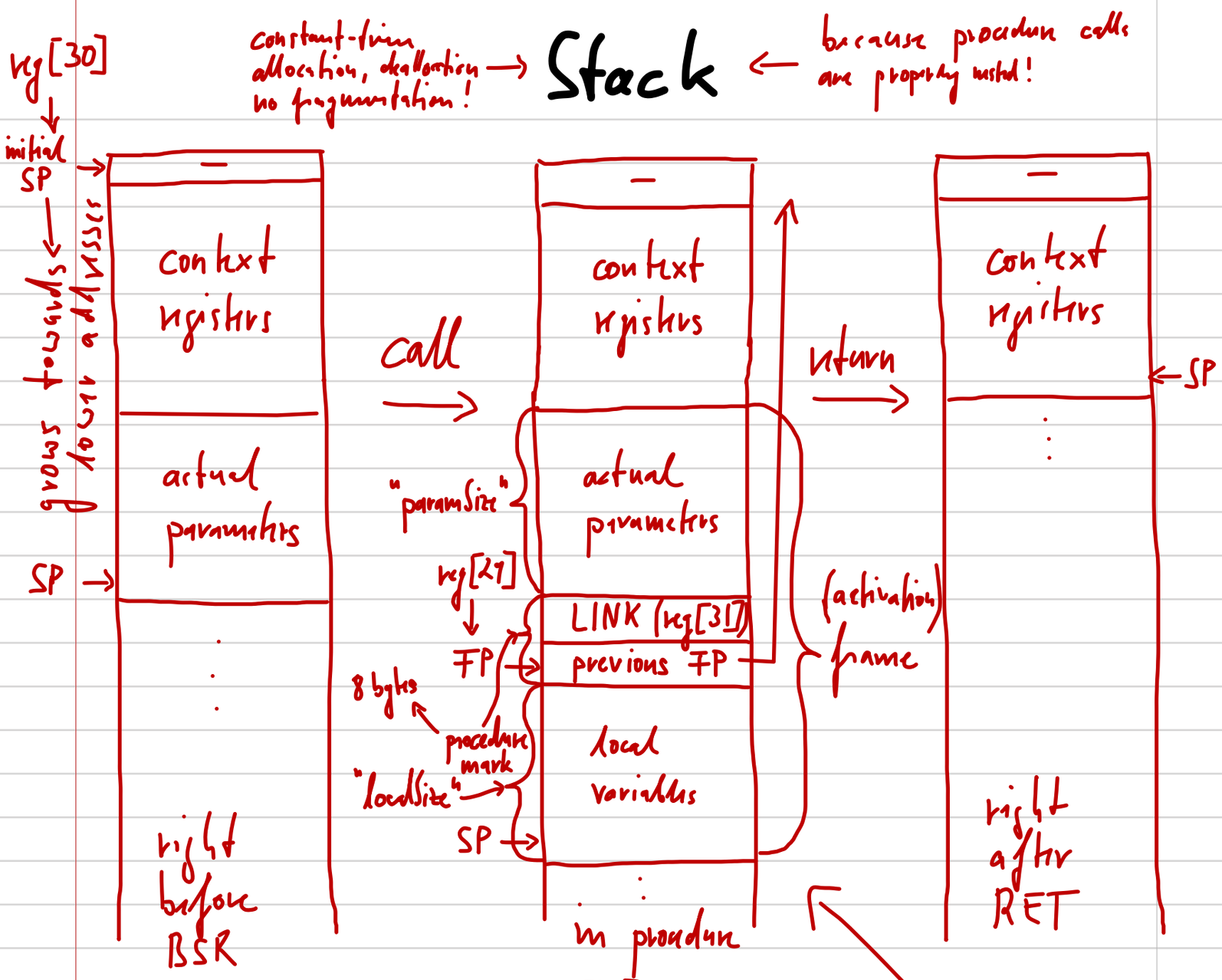
← create fixup chain for return calls

number of local variables

check uniqueness of local variables and parameters

← for checking return type in return calls

number of parameters




```
prologue(int localSize) {
 put(PSH, LINK, SP, 4); // save return address
 put(PSH, FP, SP, 4); // save caller's frame
 // assumes: reg[0]==0 for MOV semantics
 put(ADD, FP, 0, SP); // allocate callee's frame
 put(SUBI, SP, SP, localSize); // allocate callee's local variables
}

epilogue(int paramSize) {
 // assumes: reg[0]==0 for MOV semantics
 put(ADD, SP, 0, FP); // deallocate callee's frame and local variables
 put(POP, FP, SP, 4); // restore caller's frame
 put(POP, LINK, SP, paramSize + 4); // restore return address
                                    // deallocate parameters
 put(RET, 0, 0, LINK); // return
}
```

reg[31]

efficient with 4, parameter addressing

0

FP not necessary (→SP) unless there are dynamic types

Parameter Addressing

```
formalParameters(struct object_t* object) {
 int numberOfParameters;
 struct object_t* nextParameter;

 numberOfParameters = 0;

 if (symbol == LPAREN) getSymbol(); else error("missing '('");

 nextParameter = object->params;

 if ((symbol == INT) || (symbol == CHAR) || (symbol == BOOL) ||
    (symbol == IDENTIFIER) || (symbol == STRUCT)) {
  nextParameter = formalParameter(object, nextParameter);
  numberOfParameters = numberOfParameters + 1;

  while (symbol == COMMA) {
   getSymbol();
   nextParameter = formalParameter(object, nextParameter);
   numberOfParameters = numberOfParameters + 1;
  }
 }

 object->value = numberOfParameters;
 nextParameter = object->params;

 while (nextParameter != NULL) {
  numberOfParameters = numberOfParameters - 1;
  nextParameter->offset = numberOfParameters * 4 + 8;
  nextParameter = nextParameter->next;
 }

 if (symbol == RPAREN) getSymbol(); else error("missing ')'");
}
```

```
struct object_t {
 ...
 struct object_t* params;
}
```

check type if parameter has already been created

determine parameter offsets relative to FP

n: number of parameters

$$address(p[i]) = \sum_{i<j\leq n} size(p[j]) + 8$$

procedure mark

Formal Parameter

```
struct object_t* formalParameter(struct object_t* object,
                                 struct object_t* formalParameter) {
 struct type_t* type;

 type = basicArrayRecordType();

 if (symbol == IDENTIFIER) {
  if (formalParameter != NULL) {
    if (type != formalParameter->type)
      warning("type mismatch in procedure declaration and call");

    if (findObject(object->params, identifier) != NULL)
      error("parameter name already used: ", identifier);

    formalParameter->name = identifier;
  } else
    formalParameter = createFormalParameter(object, type, identifier);

  getSymbol();
  formalParameter = formalParameter->next;
 } else error("identifier expected");

 return formalParameter;
}
```

already determined elsewhere

now we know the name!

Return

```
procedureReturn() {
 struct item_t* item;

 if (symbol == RETURN)
  getSymbol();
 else
  error("return statement expected");

 if ((symbol == ADD) || (symbol == SUB) ||
    (symbol == IDENTIFIER) || (symbol == INTEGER) ||
    (symbol == LPAREN) || (symbol == NEG) || (symbol == STRING)) {
  item = malloc(sizeof(struct item_t));
  expression(item);

  if (item->type != procedureContext->type)
   warning("return type mismatch");

  if (item->type == BOOL_TYPE)
   unloadBool(item);

  load(item);

  // assumes: reg[0]==0 for MOV semantics
  put(ADD, RR, 0, item->reg);
  releaseRegister(item->reg);
 }

 returnFJumpAddress = fJumpChain(returnFJumpAddress);
}


int fJumpChain(int branchAddress) {
 put(BR, 0, 0, branchAddress);

 return PC - 1;
}
```

First<expressions>

set in procedure implementation

← that's right, even works for Booleans!

reg[27]: "return register"

jump to epilogue, not yet known

Procedure Call

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

 object = findProcedureObject(symbolTable, identifier);

 if (object == NULL) {
  warning("undeclared procedure: ", identifier);

  object = createObject(symbolTable, identifier);

  object->class = PROC;
  // TODO: infer return type
  object->type = UNKNOWN_TYPE;
  object->offset = 0;
 }

 item->mode = REG_MODE;
 item->type = object->type; // type of return value

 pushUsedRegisters();
 actualParameters(object);

 if ((object->offset != 0) && !isBSR(object->offset))
  sJump(object->offset - PC);
 else
  object->offset = sJump(object->offset);

 popUsedRegisters();
 item->reg = requestRegister();

 // assumes: reg[0]==0 for MOV semantics
 put(ADD, item->reg, 0, RR);
}
int sJump(int branchAddress) {
 put(BSR, 0, 0, branchAddress);

 return PC - 1;
}
```

let's anyway gather as much information about the procedure as possible

address of procedure code not yet known

save registers used in the calling context of the call

procedure code has already been generated

otherwise, create fixup chain

restore registers (in correct order as they were pushed)

load return value

Anonymous Parameter

```
actualParameters(struct object_t* object) {
 struct object_t* nextFormalParameter;
 struct item_t* item;

 if (symbol == LPAREN) getSymbol(); else error("missing '('");

 nextFormalParameter = object->params;

 if ((symbol == ADD) || (symbol == SUB) ||
    (symbol == IDENTIFIER) || (symbol == INTEGER) ||
    (symbol == LPAREN) || (symbol == NEG) || (symbol == STRING))
{
  nextFormalParameter =
   actualParameter(object, nextFormalParameter);

  while (symbol == COMMA) {
   getSymbol();

   nextFormalParameter =
    actualParameter(object, nextFormalParameter);
  }
 }

 while (nextFormalParameter != NULL) {
  warning("actual parameter expected");

  item = malloc(sizeof(struct item_t));

  item->mode = CONST_MODE;
  item->type = INT_TYPE;
  item->value = 0;

  pushParameter(item);

  nextFormalParameter = nextFormalParameter->next;
 }

 if (symbol == RPAREN) getSymbol(); else error("missing ')'");
}
```

First <expression>

1. check actual vs. formal type
2. check unnamed formal parameter, if unknown anonymous!

push dummy values if there are too few actual parameters

Actual Parameter

```
struct object_t* actualParameter(struct object_t* object,
                                 struct object_t* formalParameter) {
 struct item_t* item;

 if ((symbol == ADD) || (symbol == SUB) ||
    (symbol == IDENTIFIER) || (symbol == INTEGER) ||
    (symbol == LPAREN) || (symbol == NEG) || (symbol == STRING)) {
  item = malloc(sizeof(struct item_t));
  expression(item);

  if (formalParameter != NULL) {
   if (item->type != formalParameter->type)
    warning("type mismatch in procedure call");
  } else
   formalParameter =
    createAnonymousParameter(object, item->type);

  pushParameter(item);

  formalParameter = formalParameter->next;
 } else error("actual parameter expected");

 return formalParameter;
}

pushParameter(struct item_t* item) {
 if (item->type == BOOL_TYPE)
  unloadBool(item);

 load(item);

 put(PSH, item->reg, SP, 4);

 releaseRegister(item->reg);
}
```

first expression

formal parameter without name

yep, Booleans too!

Bootstrapping

1. implement system procedures (see target machine lecture)
   (need to understand how stack works for parameter exchange)
2. create entries in symbol table for all system procedures
3. linker needs to resolve the BSRs to system procedures
   (just like the BSRs to any imported procedures)
   ↳ separate compilation is next!