COMP 520 - Compilers

Lecture 14 (Thu April 7, 2022)

Code Generation

- Reading
  - PLPJ Chapter 7 Code Generation
    - Secn 7.1 – 7.4 (pp 250 - 301)

- Project
  - PA4 assignment is online

Topics

- Code generation overview
  - objectives and approach
  - entity descriptions
  - TAM stack machine and interface for object code generation
  - miniTriangle code generation examples

- Triangle code generation
  - information flow in AST traversal
  - entity descriptions
  - TAM details
  - Triangle examples

- miniJava code generation
  - simplifications and complications
  - mJAM

Code generation overview

- Code generation task
  - synthesize “object code” from decorated AST for a stack machine
    - every Identifier and Reference are linked to a declaration
      - add information about *runtime location*
    - every Expression has a type
      - determines *instructions* to be used

- Object code representation
  - binary
    - instructions for physical machine, e.g. MIPS
    - instructions for abstract machine, e.g. TAM
  - textual
    - assembler code

- Object code conventions
  - memory layout
  - procedure linkage
  - loading, execution, and debugging

What needs to be done

- Determine size and location of variables
  - how much space does a variable occupy?
  - *where* is it allocated?
  - *when* is it allocated?

- Generate object code for each construct in the AST
  - control structures
    - if, while, block statements, etc.
  - expressions
    - predefined operators
  - procedure and function call
    - caller prolog, epilog
    - callee prolog, epilog
  - variable reference
    - Reference can be read or assigned
  - space allocation
    - scope entry/exit

Approach (Triangle)

- Traverse AST using visitor
  - information flow
    - inherited: activation record (frame) size in fixed units (words)
    - synthesized: a declaration returns the size of the declared variable
  - visit declarations before references
    - create an “entity description”
      - size and location of the entity in object code units
      - access mode and value
        - » known value or unknown value at compile time?
        - » accessed in current frame or in global frame or in heap?
        - » contents are data (what type) or data address or code address?
    - update sizes of runtime structures as declarations are encountered
      - Update frame size to accommodate locals
      - Update object size as fields are encountered
  - visit commands, expressions
    - use entity descriptions to generate appropriate code for references
    - generate appropriate instructions for expression evaluation and command execution

Implementation of Entity Descriptions

- An entity description
  - `public abstract class RuntimeEntity { public int size;  ... }`

- Specialized to specific types of values and access modes
  - `public class KnownValue extends RuntimeEntity {`
    `public int value;` /* the known value */
    `...}`
  - `public class UnknownValue extends RuntimeEntity {`
    `public Address address;` /* the address of the value on the stack */
    `...}`

- Allow (some) AST classes to be decorated with an entity description
  - `public abstract class AST { public RuntimeEntity entity; ... }`
    - might be restricted to Declaration subclass

Entity descriptions (Triangle)

```
let
   const b ~ 10;
   var i: Integer;
in
   i := i * b
```

```
LOAD 4[SB]
LOADL 10
mult
STORE 4[SB]
```

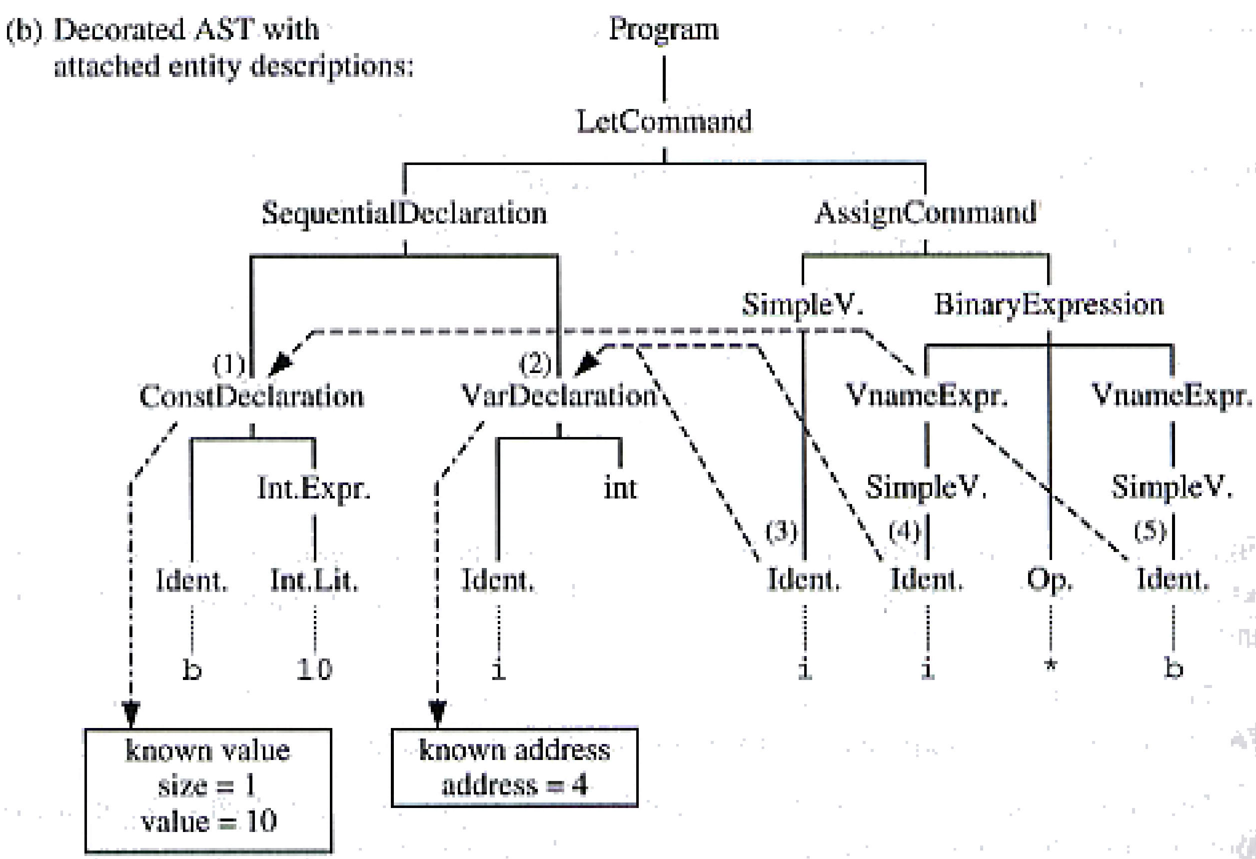


Figure 7.1 Entity descriptions for a known value and a known address.

Entity description (Triangle)

```
let
   var x: Integer
in
   let
      const y ~ 365 + x
   in
      putint(y)
```

```
LOAD  5[SB]
LOADL 365
CALL  add
STORE 6[SB]

LOAD  6[SB]
CALL  putint
```

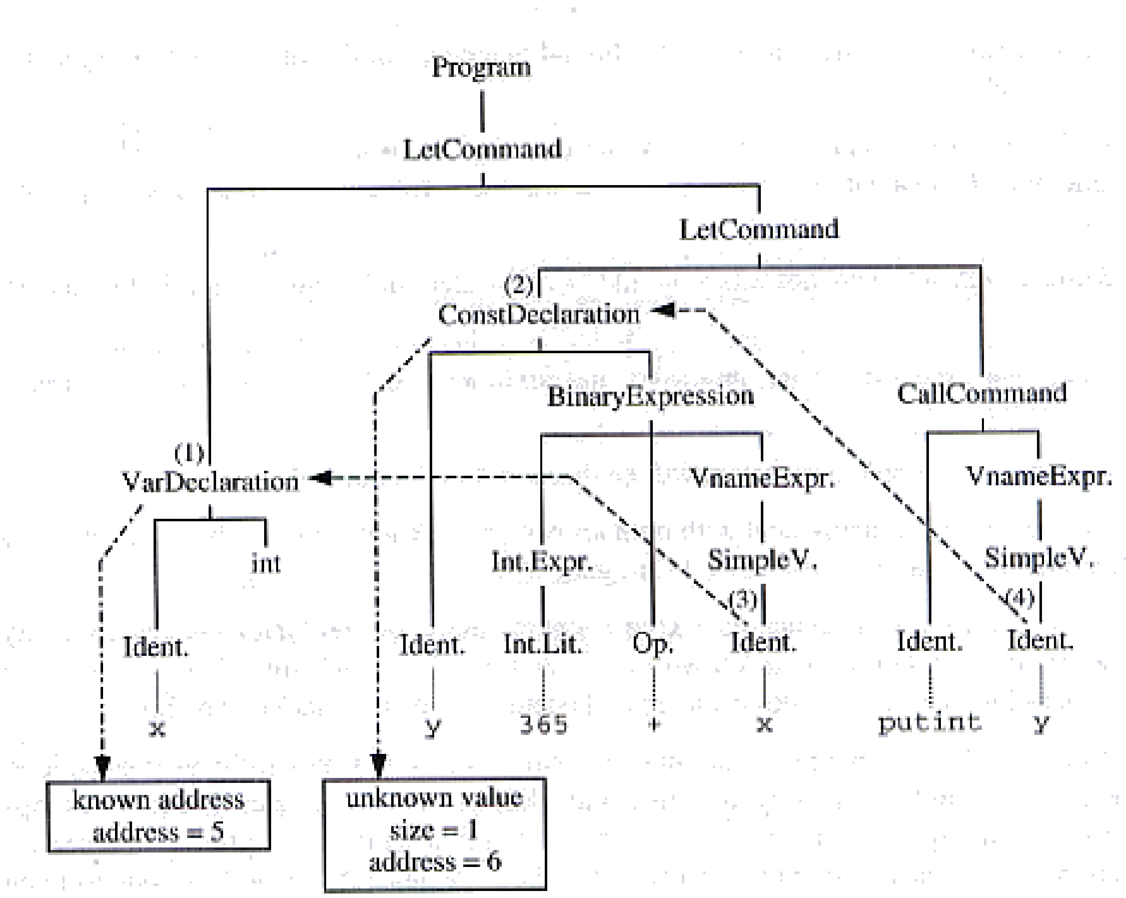
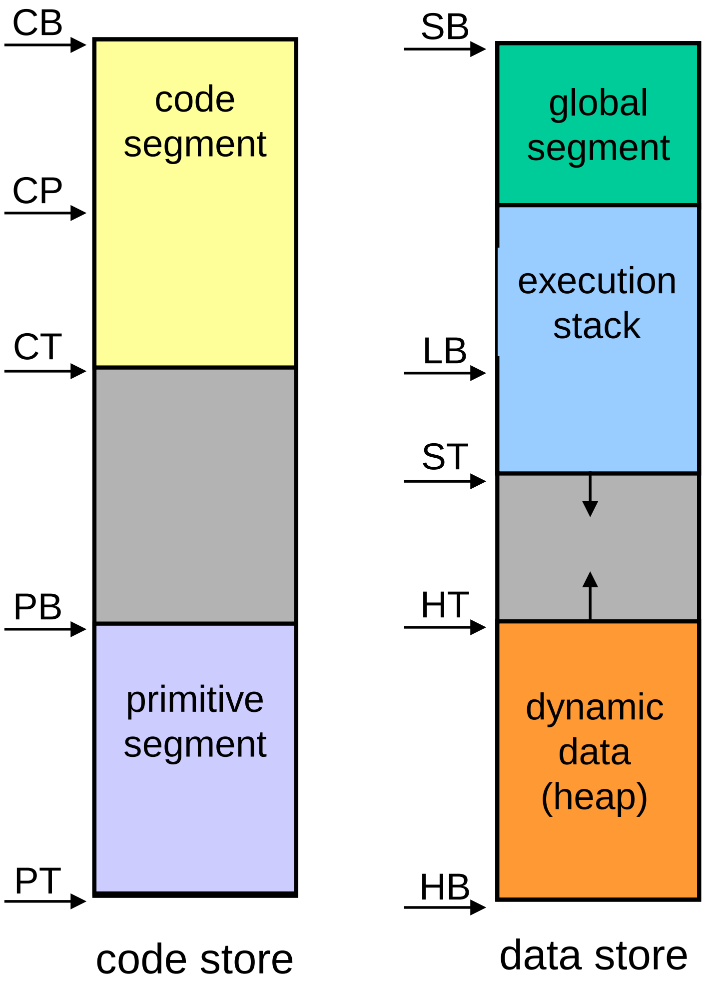


Figure 7.2 Entity descriptions for a known address and an unknown value.

Recall TAM memory organization

- Two separate memories
  - Code store
    - compiler-generated program is loaded into code segment
    - predefined runtime functions are located in the primitive segment
    - TAM can not write into code store
  - Data store
    - static constants and variables are loaded into global segment
    - procedure invocation and expression evaluation uses execution stack
      - expands downwards
    - dynamically allocated values are allocated on the heap
      - expands upwards
      - memory for deleted values can be reused
- ABI defines fixed addresses and usage conventions
  - various locations in memories are accessed relative to machine registers (CB, SB, HT, etc.)



Triangle Abstract Machine (TAM)

- TAM
  - stack machine
  - 16 registers with fixed definitions
    - CB – code base, CT – code top, CP – Code pointer
    - PB – primitives base, PT- prim top

    - SB – stack base, ST – stack top
    - HB – heap base, HT – heap top
    - LB – locals base,
      - L1 .. L6 – locals base of up to 6 lexically enclosing procedure scopes (cache for static chain)

- Instruction format (32 bits)
  - operation *op* (4 bits)
  - register *r* (4 bits)
  - size *n* (8 bits)
  - value *d* (signed 16 bits)

TAM – Triangle Abstract Machine

Table C.2 Summary of TAM instructions.

Op-code	Instruction mnemonic	Effect
0	`LOAD(`*n*`)` *d*`[`*r*`]`	Fetch an *n*-word object from the data address (d + register r), and push it on to the stack.
1	`LOADA` *d*`[`*r*`]`	Push the data address (d + register r) on to the stack.
2	`LOADI(`*n*`)`	Pop a data address from the stack, fetch an *n*-word object from that address, and push it on to the stack.
3	`LOADL` *d*	Push the 1-word literal value *d* on to the stack.
4	`STORE(`*n*`)` *d*`[`*r*`]`	Pop an *n*-word object from the stack, and store it at the data address (d + register r).
5	`STOREI(`*n*`)`	Pop an address from the stack, then pop an *n*-word object from the stack and store it at that address.
6	`CALL(`*n*`)` *d*`[`*r*`]`	Call the routine at code address (d + register r), using the address in register *n* as the static link.
7	`CALLI`	Pop a closure (static link and code address) from the stack, then call the routine at that code address.
8	`RETURN(`*n*`)` *d*	Return from the current routine: pop an *n*-word result from the stack, then pop the topmost frame, then pop *d* words of arguments, then push the result back on to the stack.
9	–	(unused)
10	`PUSH` *d*	Push *d* words (uninitialized) on to the stack.
11	`POP(`*n*`)` *d*	Pop an *n*-word result from the stack, then pop *d* more words, then push the result back on to the stack.
12	`JUMP` *d*`[`*r*`]`	Jump to code address (d + register r).
13	`JUMPI`	Pop a code address from the stack, then jump to that address.
14	`JUMPIF(`*n*`)` *d*`[`*r*`]`	Pop a 1-word value from the stack, then jump to code address (d + register r) if and only if that value equals *n*.
15	`HALT`	Stop execution of the program.

- Instructions

  - *a* denotes a data address
  - *c* denotes a character
  - *i* denotes an integer
  - *n* denotes a non-negative integer
  - *t* denotes a truth value (0 for *false* or 1 for *true*)
  - *v* denotes a value of any type
  - *w* denotes any 1-word value

TAM – Triangle Abstract Machine

Table C.3 Summary of TAM primitive routines.

Address	Mnemonic	Arguments	Result	Effect
PB + 1	`id`	w	w'	Set $w' = w$.
PB + 2	`not`	t	t'	Set $t' = \neg t$.
PB + 3	`and`	t_1, t_2	t'	Set $t' = t_1 \wedge t_2$.
PB + 4	`or`	t_1, t_2	t'	Set $t' = t_1 \vee t_2$.
PB + 5	`succ`	i	i'	Set $i' = i + 1$.
PB + 6	`pred`	i	i'	Set $i' = i - 1$.
PB + 7	`neg`	i	i'	Set $i' = -i$.
PB + 8	`add`	i_1, i_2	i'	Set $i' = i_1 + i_2$.
PB + 9	`sub`	i_1, i_2	i'	Set $i' = i_1 - i_2$.
PB + 10	`mult`	i_1, i_2	i'	Set $i' = i_1 \times i_2$.
PB + 11	`div`	i_1, i_2	i'	Set $i' = i_1 / i_2$ (truncated).
PB + 12	`mod`	i_1, i_2	i'	Set $i' = i_1$ modulo i_2.
PB + 13	`lt`	i_1, i_2	t'	Set t' = true iff $i_1 < i_2$.
PB + 14	`le`	i_1, i_2	t'	Set t' = true iff $i_1 \leq i_2$.
PB + 15	`ge`	i_1, i_2	t'	Set t' = true iff $i_1 \geq i_2$.
PB + 16	`gt`	i_1, i_2	t'	Set t' = true iff $i_1 > i_2$.
PB + 17	`eq`	v_1, v_2, n	t'	Set t' = true iff $v_1 = v_2$ (where v_1 and v_2 are n-word values).
PB + 18	`ne`	v_1, v_2, n	t'	Set t' = true iff $v_1 \neq v_2$ (where v_1 and v_2 are n-word values).
PB + 19	`eol`	–	t'	Set t' = true iff the next character to be read is an end-of-line.
PB + 20	`eof`	–	t'	Set t' = true iff there are no more characters to be read (end of file).
PB + 21	`get`	a	–	Read a character, and store it at address a.
PB + 22	`put`	c	–	Write the character c.
PB + 23	`geteol`	–	–	Read characters up to and including the next end-of-line.
PB + 24	`puteol`	–	–	Write an end-of-line.
PB + 25	`getint`	a	–	Read an integer-literal (optionally preceded by blanks and/or signed), and store its value at address a.
PB + 26	`putint`	i	–	Write an integer-literal whose value is i.
PB + 27	`new`	n	a'	Set a' = address of a newly allocated n-word object in the heap.
PB + 28	`dispose`	n, a	–	Deallocate the n-word object at address a in the heap.

- Primitives

  - a denotes a data address
  - c denotes a character
  - i denotes an integer
  - n denotes a non-negative integer
  - t denotes a truth value (0 for *false* or 1 for *true*)
  - v denotes a value of any type
  - w denotes any 1-word value

TAM object code interface

- An instruction

```
public class Instruction {
   ... definitions of op-codes and registers
   public Instruction(byte op, byte n, byte r, short d) { ... }
}
```

- Interface provided to code generator

```
private Instruction[] code = new Instruction [1024];
private int nextInstrAddr = 0;

public void emit(byte op, byte n, byte r, short d) {
   code[nextInstrAddr++] = new Instruction(op, n, r, d);
}
```

  – Requires instructions to be emitted in linear order!

Code generator using visitor (miniTriangle)

- Traverse AST, emit instructions
  - visit top-level program node

```
public Object visitProgram(Program prog, Object arg) {
   prog.C.visit(this, arg);
   emit(Instruction.HALTop, 0, 0, 0);
   return null;
}
```

  - visit integer expression (which contains an IntegerLiteral)

```
public Object visitIntegerExpression
       (IntegerExpression expr, Object arg) {
   short v = valuation(expr.IL.spelling);
   emit(Instruction.LOADLop, 0, 0, v);
   return null;
}
```

Code generator using visitor methods (miniTriangle)

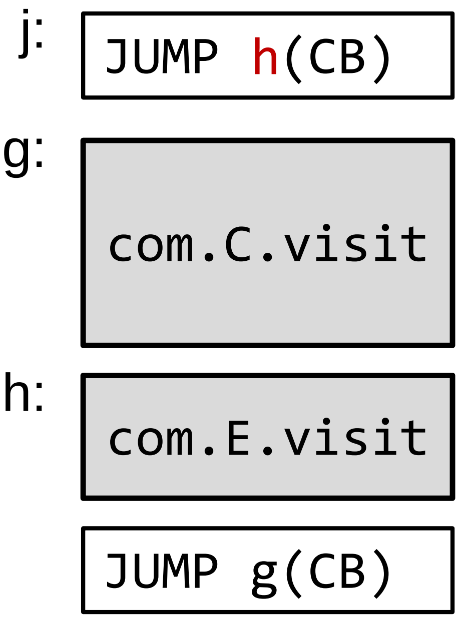
– visit unary expression

```
public Object visitUnaryExpression
       (UnaryExpression expr, Object arg) {
   expr.E.visit(this, arg);
   short p = address of primitive routine corresponding to expr.operator
   emit(Instruction.CALLop, Instruction.SBr,
        Instruction.PBr, p);
   return null;
}
```

Code generator using visitor methods (miniTriangle)

- visit while command

```
public Object visitWhileCommand(WhileCommand com, Object arg) {
   short j = nextInstrAddr;
   emit(Instruction.JUMPop, 0, INSTRUCTION.CBr, 0)  // patchme
   short g = nextInstrAddr;
   com.C.visit(this, arg);
   short h = nextInstrAddr;
   patch(j, h);
   com.E.visit(this, arg);
   emit(Instruction.JUMPop, 1, Instruction.CBr, g);
   return null;
}
```



Code generator using visitor methods (miniTriangle)

- Use visitor argument and result to track space usage
  - visit variable declaration

```
public Object visitVarDeclaration
       (VarDeclaration decl, Object arg) {
   short gs = shortValueOf(arg);
   short s = shortValueOf(decl.T.visit(this, null))
   emit(Instruction.PUSHop, 0, 0, s)
   decl.entity = new KnownAddress(proc nesting level, gs);
   return new Short(s);
}
```

  - visit multiple declarations

```
public Object visitSequentialDeclaration
       (SequentialDeclaration decl, Object arg) {
   short gs = shortValueOf(arg);
   short s1 = shortValueOf(decl.D1.visit(this, arg));
   short s2 = shortValueOf(
         decl.D2.visit(this, new Short(gs+s1)));
   return new Short(s1 + s2);
}
```

TAM code generation in the Triangle compiler

- How does it differ from our miniTriangle examples so far?
  - Triangle has
    - nested procedures and functions
      - non-local variable reference
      - static link management in procedure and function call
    - parameter passing by reference and by value
      - increases complexity of value access and update
    - arguments that are procedures or functions
      - pass as a *closure*: (code address, static link)
    - composite types
      - records, arrays
      - field and element selectors in reference and assignment
      - non-unit value size
        - » needed in assignment, equality, parameter passing
      - Triangle simplification: all values of a given type have the same size

Information passed through AST traversal

- The visit method permits an argument to be passed in and a value to be returned
  - What is passed in and returned?
    - Declarations
      - argument: frame description
      - yields: amount of storage allocated by declaration
    - Commands
      - argument: frame description
      - yields: null
    - Expressions
      - argument: frame description
      - yields: size of result
    - V-names (references)
      - argument: frame description
      - yields: runtime entity description of reference

Entity descriptions in the Triangle compiler

- Every entity has
  - size *n* (determined by type)

- Known value adds
  - constant with literal value (e.g. intLit, CharLit, …) with *n* = 1.
    - entity not allocated in any frame, fetched via LOADL

- Unknown value adds
  - constant with value computed at run time
    - entity allocated in some frame, fetched via LOAD (n) of known address

- Known address adds
  - (decl level s, displacement d)
    - entity fetched via LOAD (n) d(frame-base)   frame-base $\in$ LB, L1, L2, … , SB
    - entity stored via STORE (n) d(frame-base)

- Unknown address adds
  - an indirect address, the contents of the known address (s, d)
    - entity fetched via LOAD d(frame-base); LOADI (n);
    - entity stored via LOAD d(frame-base); STOREI (n);

Entity descriptions in the Triangle compiler

- Known routine adds
  - a code address
- Unknown routine adds
  - code address and static link (decl level s, displacement d) in a known location
    - arises when functions are passed as values
- Primitive routine adds
  - TAM-specific known code address for primitive operation
- Type representation adds
  - size
    - fixed for all values of the type
- Field adds
  - offset and size
    - in V-names

Procedure call

- Program cg1.tri

```
let
  var n: Integer;
  proc p() ~
      n := n * 2
in
  begin
  n := 9;
  p();
  end
```

- TAM code cg1.tam

```
0:   PUSH          1
1:   JUMP          7[CB]
2:   LOAD   (1)    0[SB]
3:   LOADL         2
4:   CALL          mult
5:   STORE  (1)    0[SB]
6:   RETURN(0)     0
7:   LOADL         9
8:   STORE  (1)    0[SB]
9:   CALL   (SB)   2[CB]
10:  POP    (0)    1
11:  HALT
```

Parameter passing

- Program cg4.tri

```
let
  proc p(var x: Integer,
              i: Integer) ~
      x := x + i;
  var y : Integer
in
  begin
    y := 2;
    p(var y, 5);
    putint( y );
  end
```

- TAM code cg4.tam

```
0:    JUMP          8[CB]
1:    LOAD   (1)    -2[LB]
2:    LOADI  (1)
3:    LOAD   (1)    -1[LB]
4:    CALL          add
5:    LOAD   (1)    -2[LB]
6:    STOREI(1)
7:    RETURN(0)     2
8:    PUSH          1
9:    LOADL         2
10:   STORE  (1)    0[SB]
11:   LOADA         0[SB]
12:   LOADL         5
13:   CALL   (SB)   1[CB]
14:   LOAD   (1)    0[SB]
15:   CALL          putint
16:   POP    (0)    1
17:   HALT
```

miniJava vs Triangle

- Classes

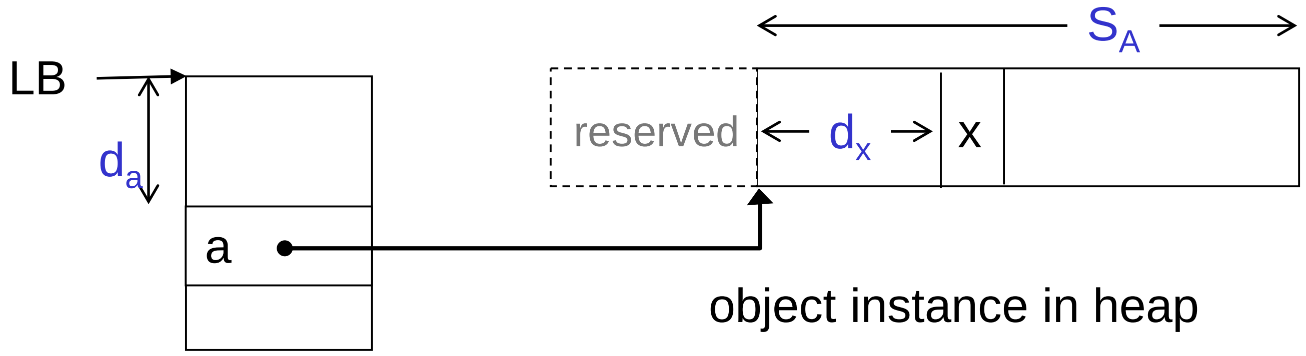
  ```
  class A {int x; void p(){x = 3;} }
  ```

  – runtime entity descriptions in AST
    - class A : S_A = size of class A (# fields)
    - field x: d_x = displacement of field x in heap-allocated instance
    - method p: d_p = displacement of code for p in code store

- Objects
  – instances are created on the heap

  ```
  A a = new A();
  ```

runtime layout



activation record on stack

Some considerations for implementing miniJava

- Simplifying properties
  - All miniJava values on the stack have the same size
    - one word
  - All miniJava values are passed by value
    - the value of an object is its address in the heap
  - All stack references are relative to LB, or possibly to SB (when?)
    - no need for Triangle nested procedure links, L1 …. L6

- Complications
  - implicit parameter `this` in every non-static method invocation
  - complex handling of References
    - encodeFetch
    - encodeStore
    - encodeMethodInvocation
  - (dynamic method invocation)

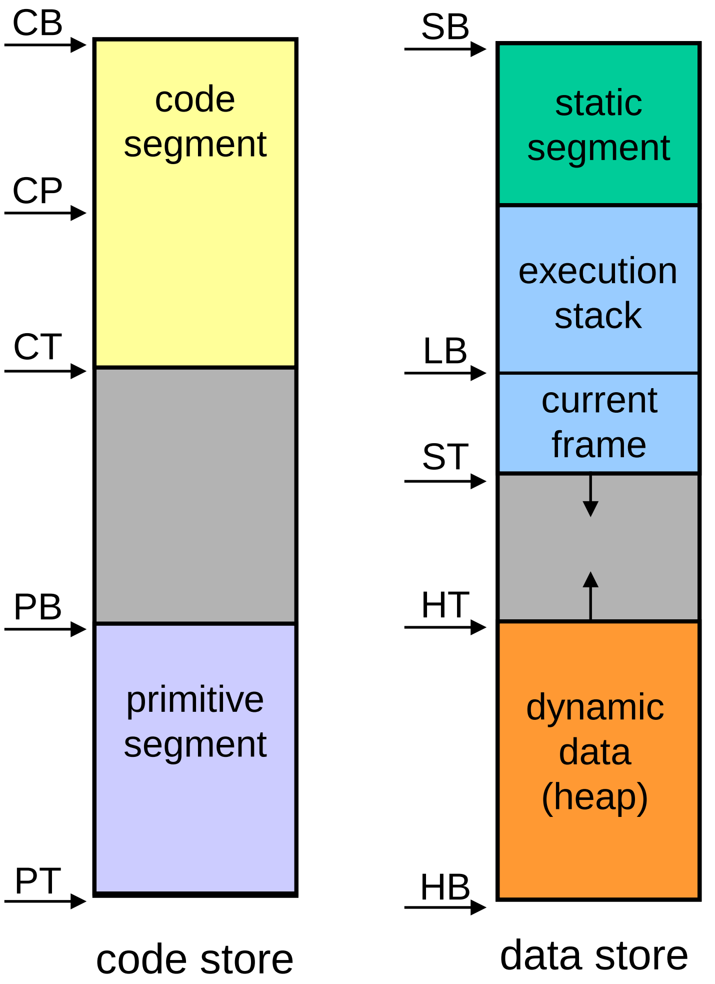
MiniJava and TAM

- miniJava compiler could target TAM
  - but some things will be tedious
    - references to instance members within a method
    - call sequence
    - int values x with |x| > 32,767
    - (dynamic method invocation)

- better target: mJAM, a Java Abstract Machine
  - implemented as a small modification to TAM
    - remove L1 … L6 registers and static link maintenance
    - extend int values to full word
    - add a register OB for object base
      - holds value of **`this`**
      - preserved/restored in method invocation
    - Method call
      - CALL for static methods
      - CALLI for instance methods
      - (CALLD for dynamic method invocation)

mJAM memory organization

- Two separate memories
  - Code store
    - compiler-generated program is loaded into code segment
    - predefined runtime functions are located in the primitive segment
    - mJAM cannot write into code store
  - Data store
    - static constants and variables are loaded into static segment
    - method invocation creates a frame
    - expression evaluation occurs at stack top
      - expands downwards
    - object instances are dynamically allocated on the heap
      - expands upwards
      - (no garbage collection)
- ABI defines fixed addresses and usage conventions
  - various locations in memories are accessed relative to machine registers (CB, SB, LB, ST, etc.)



miniJava code generation: available information

- AST phrase class (LHS NT of AST grammar)
  - Package, Statement, Reference, Expression, Declaration, Terminal

- AST attributes
  - Every Identifier and Reference link to a Declaration
    - Reference
      - IdRef, ThisRef, QualRef
    - Declaration
      - ClassDecl, MethodDecl, FieldDecl, ParameterDecl, VarDecl
  - Every Declaration has a type
    - TypeKind ∈
      - Int, Boolean, Array, void, Class
    - ArrayType (τ)
    - ClassType (*name*)
  - Every Declaration has a <u>runtime entity description</u>
    - Describes where/how to find value in memory

miniJava code generation

- AST node type (phrase) and AST attributes determine code generation for each node
  - examples of code functions for miniJava (cf. PLPJ Table 7.1)

Phrase class	Code function	Effect of generated code
Package P	run P	Call main method and HALT upon return
Statement S	execute S	Execute statement, updating variables, no change in frame size on termination except VarDeclStmt which extends frame by 1
Expression E	evaluate E	Evaluate expression E, leaving its result at stack top
Reference R	fetch R	R denotes a LocalDecl or FieldDecl, load value at Decl at stacktop
Reference R	assign R	R denotes a LocalDecl or FieldDecl, pop value from stack top and store it in R
Reference R	call R	R denotes a MethodDecl, CALLI or CALL with needed args
….	…..	…..

CodeGenerator implementation

- The CodeGenerator is yet another visitor of the AST
  1. Traverse all Declarations creating a runtime entity descriptor (RED) for each declaration
     - offset relative to LB for local variables and parameter variables
     - offset relative to SB for static fields
     - offset relative to OB for instance variables
     - offset relative to CB for methods

  2. Generate instructions in code store for each method in each class
     - method linkage – establishing a new frame, and returning
     - generate code for all statements
       - generate control flow
       - generate expression evaluation
       - generate reference evaluation
       - generate assignment or variable declaration statement
       - generate method or primitive invocation