COMP 520  -  Compilers

Lecture 14 (Tue March 26)

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
    - 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 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 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 value 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

Figure 7.1 Entity descriptions for a known value and a known address.
Entity description (Triangle)

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

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 \( \text{op} \) (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.

<table>
<thead>
<tr>
<th>Op-code</th>
<th>Instruction mnemonic</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LOAD ( n ) ( d ) ( r )</td>
<td>Fetch an ( n )-word object from the data address ((d + \text{register } r)), and push it on to the stack.</td>
</tr>
<tr>
<td>1</td>
<td>LOADA ( d ) ( r )</td>
<td>Push the data address ((d + \text{register } r)) on to the stack.</td>
</tr>
<tr>
<td>2</td>
<td>LOADI ( n )</td>
<td>Pop a data address from the stack, fetch an ( n )-word object from that address, and push it on to the stack.</td>
</tr>
<tr>
<td>3</td>
<td>LOADL ( d )</td>
<td>Push the 1-word literal value ( d ) on to the stack.</td>
</tr>
<tr>
<td>4</td>
<td>STORE ( n ) ( d ) ( r )</td>
<td>Pop an ( n )-word object from the stack, and store it at the data address ((d + \text{register } r)).</td>
</tr>
<tr>
<td>5</td>
<td>STOREI ( n )</td>
<td>Pop an address from the stack, then pop an ( n )-word object from the stack and store it at that address.</td>
</tr>
<tr>
<td>6</td>
<td>CALL ( n ) ( d ) ( r )</td>
<td>Call the routine at code address ((d + \text{register } r)), using the address in register ( n ) as the static link.</td>
</tr>
<tr>
<td>7</td>
<td>CALLI</td>
<td>Pop a closure (static link and code address) from the stack, then call the routine at that code address.</td>
</tr>
<tr>
<td>8</td>
<td>RETURN ( n ) ( d )</td>
<td>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.</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>(unused)</td>
</tr>
<tr>
<td>10</td>
<td>PUSH ( d )</td>
<td>Push ( d ) words (uninitialized) on to the stack.</td>
</tr>
<tr>
<td>11</td>
<td>POP ( n ) ( d )</td>
<td>Pop an ( n )-word result from the stack, then pop ( d ) more words, then push the result back on to the stack.</td>
</tr>
<tr>
<td>12</td>
<td>JUMP ( d ) ( r )</td>
<td>Jump to code address ((d + \text{register } r)).</td>
</tr>
<tr>
<td>13</td>
<td>JUMPI</td>
<td>Pop a code address from the stack, then jump to that address.</td>
</tr>
<tr>
<td>14</td>
<td>JUMPIF ( n ) ( d ) ( r )</td>
<td>Pop a 1-word value from the stack, then jump to code address ((d + \text{register } r)) if and only if that value equals ( n ).</td>
</tr>
<tr>
<td>15</td>
<td>HALT</td>
<td>Stop execution of the program.</td>
</tr>
</tbody>
</table>

- \( a \) 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 \textit{false} or 1 for \textit{true})
- \( v \) denotes a value of any type
- \( w \) denotes any 1-word value
### Table C.3 Summary of TAM primitive routines.

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

- **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

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

• Interface provided to code generator

```java
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
    
    ```java
    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)
    
    ```java
    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
  
  ```java
  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**

  ```java
  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
    
    ```java
    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
    
    ```java
    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. \( \text{intLit} \), \( \text{CharLit} \), …) with \( n = 1 \).
    • entity not allocated in any frame, fetched via \( \text{LOADL} \)

• Unknown value adds
  – constant with value computed at run time
    • entity allocated in some frame, fetched via \( \text{LOAD} \) (n) of known address

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

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

```plaintext
let
  var n: Integer;
  proc p() ~
    n := n \times 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**

```lisp
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
    • method p: \( d_p \) = displacement of code for p

• Objects
  
  – instances are created on the heap
    
    \( A a = \text{new } A() \);
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
    • no need for static 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 can 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)

- 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)
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, lxIdRef, ThisRef, QRef, lxQRef
    • 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 runtime entity description
  • Describes where/how to find value in memory
miniJava code generation

- AST node type (phrase) and AST attributes determine code generation for each node

- Summary of code functions for miniJava (cf. PLPJ Table 7.1)

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

....
