ASM 2

Motivation

- Develop the full hierarchy of expressions for arithmetic and relational operators

Arithmetic

- What happens if we want to do arithmetic?
- Ex:

  x = y + 4
- Seems easy:

```
mov rax, [y]
add rax, 4
mov [x], rax
```

Arithmetic

- What if the expression is a bit more complex?
  x = (y+4) * (z-2)
- We can do this using a second register to hold a temporary value:

```
mov rax,[y]
add rax,4
mov rbx,[z]
sub rbx,2
imul rax,rbx
mov [x],rax
```

Problem

- What happens when we run out of registers?
  - There are only 16
- Solution: The *stack*

Stack

- When program starts up, OS allocates some RAM as a stack
- rsp register points to topmost item on the stack
- We can store temporaries to the stack with push:
  push rax
- And we can remove them with pop:
  pop rax
- **Important**: Change return-stmt, loop, and cond to get their values from the stack, not from rax!
  - It's as easy as just adding a "pop rax" in the right places

Stack

- We can discard items from the stack:
  add rsp,8
  - This works because stack grows down in memory
  - Each item is 64 bits (=8 bytes)
- We can read items from the stack without popping:
  - mov rax, [rsp] ;get topmost item
  - mov rax, [rsp+8] ;get next item down
  - mov rax, [rsp+16] ;get third-from-top

Code

- We need to define some semantics
- Change the expr production and add additional nonterminals for various mathematical and relational operations:
  expr → orexp
  orexp → ~~orexp OR andexp~~ | andexp
  andexp → ~~andexp AND notexp~~ | notexp
  notexp → ~~NOT notexp~~ | rel
  rel → ~~sum RELOP sum~~ | sum
  sum → sum ADDOP term | sum MINUS term | term
  term → term MULOP neg | neg
  neg → ~~MINUS neg~~ | factor
  factor → ~~ID~~ | NUM | LP expr RP | ~~func-call | ID LB expr-list RB | STRING-CONSTANT~~
- For now, we'll ignore the parts of the grammar that are struck out

Attributes

- Recall: Synthesized attributes and inherited attributes
- sum, term, factor produce synthesized attributes
  - Value: The value of the arithmetic operation
    - This will always be on top of the stack
  - Type: What's the type of the result
    - For now, this is always a number, but later we'll add additional types
- We'll define an enumeration:
  `enum VarType{ NUMBER };`

Code

- Suppose we write some functions to process the various tree nodes
- Some of them are just stubs for now...

```
void exprNodeCode(TreeNode n, out VarType type){
    return orexpNodeCode(n, out type);
}
void orexpNodeCode(TreeNode n, out VarType type){
    return andexpNodeCode(n, out type);
}
void notexpNodeCode(TreeNode n, out VarType type){
    return relNodeCode(n, out type);
}
void relNodeCode(TreeNode n, out VarType type){
    return sumNodeCode(n, out type);
}
void negNodeCode(TreeNode n, out VarType type){
    return factorNodeCode(n, out type);
}
```

Factor

- We're ready to write the code for factor
- Only two possibilities in our cut-down grammar

```
static void factorNodeCode(TreeNode n, out VarType type){
    //factor -> NUM | LP expr RP
    var child = n.Children[0];
    switch( child.Symbol ){
        case "NUM":
            double d = Convert.ToDouble(child.Lexeme);
            string ds = d.ToString("f");
            if(ds.IndexOf(".") == -1 )
                ds += ".0";
            emit("mov rax, __float64__({0})", ds);
            emit("push rax");
            type = VarType.NUMBER;
            break;
        case "LP":
            exprNodeCode( n.Children[1], out type );
            break;
        default:
            throw new Exception("?");
    }
}
```

Mathematics

- Next, we'll look at the code for sum
- We now need to decide: Will we do integer math or floating point math?
  - Floating point is often very useful for users
  - So we'll use floating point operations here
  - This also allows us to show how FP numbers are processed

Floating Point

- FP operations use different registers from integer operations
- Sixteen registers: xmm0, xmm1, … , xmm15

Load	movsd xmm0, [x]	
Store	movsd [x], xmm0	
Add	addsd xmm0, xmm1	; xmm0 ← xmm0 + xmm1
Subtract	subsd xmm0,xmm1	; xmm0 ← xmm0 - xmm1
Multiply	mulsd xmm0, xmm1	; xmm0 ← xmm0 * xmm1
Divide	divsd xmm0, xmm1	; xmm0 ← xmm0 ÷ xmm1
Int→FP	cvtsi2sd xmm0, rax	; Converts integer value to double
FP→Int	cvtsd2si rax, xmm0	; Truncates double to integer
XMM→GPR	movq rax, xmm0	; rax gets bit pattern for double
GPR→XMM	movq xmm0, rax	; bit pattern moved unchanged to xmm0
XMM→Mem	movsd [x], xmm0	
Mem→XMM	movsd xmm0, [x]	

Stack

- We can't use PUSH or POP with xmm's directly
- We need to do one of these for push:
  - movq rax, xmm0
    push rax
  - sub rsp,8
    movsd [rsp], xmm0
- We do one of these for pop:
  - pop rax
    movq xmm0, rax
  - movsd xmm0, [rsp]
    add rsp,8

Note

- Pay careful attention to the difference between cvtsi2sd and movq!
  - cvtsi2sd *converts an integer to a double precision number*
  - movq moves the bit pattern unchanged

Sum

```
void sumNodeCode(TreeNode n, out VarType type){
    //sum -> sum ADDOP term | sum MINUS term | term
    switch( n.Children[0].Symbol ){
        case "term":
            termNodeCode(n.Children[0], out type);
            return;
        case "sum":
            ...more code...
        default:
            error
    }
}
```

Sum

- If we're processing sum $\rightarrow$ sum ADDOP term or sum $\rightarrow$ sum MINUS term we must first evaluate the two child nodes to get their values

```
VarType t0,t1;
sumNodeCode( n.Children[0], out t0 );
termNodeCode( n.Children[2], out t1 );
```

Sum

- Next, we verify the types

```
if( t0 != VarType.NUMBER || t1 != VarType.NUMBER )
    error!
```

Sum

- We can now move the two operands from the stack to xmm registers so we can perform FP math

```
emit("pop rax");    //second operand
emit("movq xmm1, rax");
emit("pop rax");    //first operand
emit("movq xmm0, rax");
```

Sum

- We then decide whether to do addition or subtraction

```
switch( n.Children[1].Lexeme ){
    case "+":
        emit("addsd xmm0,xmm1");
        break;
    case "-":
        emit("subsd xmm0,xmm1");
        break;
    default:
        ICE
}
```

Sum

- We defined the math operations to leave their results on the stack, so we must now move the value from xmm0 to the stack

```
emit("movq rax, xmm0");
emit("push rax");
```

- We can then return our synthesized attribute

```
type = VarType.NUMBER;
return;
```

Question

- What if the parse tree was generated with an LL parser?
- In that case, the grammar rules are probably more like this:
  sum $\rightarrow$ term sum'
  sum' $\rightarrow$ ADDOP term sum' | MINUS term sum' | λ
- The logic for sum would be tweaked a bit

Sum

```
void sumNodeCode(TreeNode n, out VarType type){
    //sum -> term sum'
    VarType type1;
    termNodeCode(n.Children[0], out type1);
    sumprimeNodeCode( n.Children[1], type1, out type);
}
```

Sum’

```
void sumprimeNodeCode( TreeNode n, VarType type1, out VarType type){
    //sum' -> ADDOP term sum' | MINUS term sum' | lambda
    if( n.Children.Count == 0 ){
        type = type1;
        return;
    }
    VarType type2;
    termNodeCode( n.Children[1], out type2);
    if( type1 != type2 )
        error
    emit("pop rax");    //second operand
    emit("movq xmm1, rax");
    emit("pop rax");    //first operand
    emit("movq xmm0, rax");
    switch( n.Children[0].Lexeme ){
        case "+":
            emit("addsd xmm0,xmm1");
            break;
        case "-":
            emit("subsd xmm0,xmm1");
            break;
        default:
            ICE
    }
    emit("movq rax, xmm0");
    emit("push rax");
    type = VarType.NUMBER;
}
```

Term

- The logic for term is similar, so it's left as an exercise for you

Comparisons

- What about logical operators?
  rel $\rightarrow$ sum RELOP sum | sum
  - RELOP is one of >, <, >=, <=, !=, ==
- Notice sum is on both sides of the operator so things like "x>y>z" are not valid
- We need to define semantics of relational operators
  - We'll do like C: True is nonzero value; false is zero

rel

- We can begin with an outline that's very similar to the code that we've seen before for sum
- Evaluate the two operands and move them to registers

```
void relNodeCode(TreeNode n, out VarType type){
    //rel -> sum RELOP sum | sum
    if( n.Children.Count == 1 )
        return sumNodeCode(n.Children[0]);
    VarType t0,t1;
    sumNodeCode( n.Children[0], out t0 );
    sumNodeCode( n.Children[2], out t1 );
    ...check types of t0 and t1...
    emit("pop rax");
    emit("movq xmm1, rax"); //right hand operand
    emit("pop rax");
    emit("movq xmm0, rax"); //left hand operand
    ...more code...
}
```

rel

- Floating point compare is implemented via the cmpXXsd mnemonics
- Takes two registers to compare
- First operand gets either 0x0 or 0xffffffffffffffff for true or false
- Instructions:
  - cmpeqsd (=)
  - cmpltsd (<)
  - cmplesd (<=)
  - cmpneqsd ($\neq$)
  - cmpnltsd ($\geq$ i.e., “not less than”)
  - cmpnlesd (> i.e., “not less than or equal to”)

rel

- We can use a switch statement:

```
string mnemonic;
switch(n.Children[1].Lexeme){
    case "==": mnemonic = "cmpeqsd"; break;
    case "<": mnemonic = "cmpltsd"; break;
    case "<=": mnemonic = "cmplesd"; break;
    case "!=": mnemonic = "cmpneqsd"; break;
    case ">=": mnemonic = "cmpnltsd"; break;
    case ">": mnemonic = "cmpnlesd"; break;
    default:        throw new Exception("?");
}
emit("{0} xmm0,xmm1",mnemonic);
```

Problem

- 0xffffffffffffffff doesn't correspond to a valid floating point number
- Doubles are stored as:
  - 1 sign bit
  - 11 exponent bits
  - 52 mantissa bits
- If all exponent bits are 1's: The number represents either a NaN or infinity, depending on pattern in mantissa
- The value 1.0 is represented by sign=0, exponent = 01111111111, mantissa = 0
- So we'll do a bitwise AND to convert the NaN to 1.0

rel

```
emit("movq rax, xmm0");
emit("mov rbx, __float64__(1.0)");
emit("and rax,rbx");
emit("push rax");
type = VarType.NUMBER;
```

Boolean

- The only part left is the boolean operations (and, or, not)
  orexp $\rightarrow$ orexp OR andexp | andexp
  andexp $\rightarrow$ andexp AND notexp | notexp
  notexp $\rightarrow$ NOT notexp | rel

Evaluation

- Most modern languages implement *short circuit evaluation*
- Idea: As soon as result of boolean expression is known, stop evaluating
- If short circuit evaluation was not implemented, we couldn't write things like:
  if(x != 0 and y/x > 10){ ... }
  - We'd get divide by zero even though we're trying to prevent that

orexp

- We'll examine orexp; andexp and notexp are similar
- First, we deal with the easy case...

```
void orexpNodeCode(TreeNode n, out VarType type){
    //orexp -> orexp OR andexp | andexp
    if( n.Children.Count == 1 )
        andexpNodeCode(n.Children[0], out type);
    ...more code...
}
```

orexp

- We then evaluate the left side of the OR

```
VarType t0;
orexpNodeCode(n.Children[0], out t0);
...verify t0 is correct type...
emit("pop rax");
emit("cmp rax,0");
```

- We're ready to do the comparison

orexp

- If rax holds a nonzero value:
  - We don't want to evaluate child 2; we want value of entire orexp to be nonzero
- If rax holds zero value, we must evaluate child 2 in case it ends up being true
  - The result of the entire orexp is whatever child 2 produces
- This is going to involve some jump operations

orexp

- Create a label and pop result from evaluating first child

```
string lbl = label();
emit("pop rax");
```

orexp

- If first child gave nonzero, skip over the second child's code
- Otherwise, fall through and execute code for second child, leaving result in rax

```
emit("cmp rax,0");
emit("jne "+lbl);
VarType t1;
andexpNodeCode(n.Children[2], out t1);
...verify t1 is correct type...
emit("pop rax");
emit(lbl+":");
```

orexp

- Final step: Make sure the stack gets the result of the entire expression and return our attributes

```
emit("push rax");
type = VarType.NUMBER ;
```

Notice

- This is not very efficient: We could have a pop immediately followed by a push of that exact same thing
- Here's the assembly code written in one place:

```
    ...code for first child...
    pop rax
    cmp rax,0
    jne lbl12345
    ...code for second  child...
    pop rax
lbl12345:
    push rax
```

- Push-Then-Pop going to be a no-op for us
- So we can tweak the code to eliminate that pop-then-push...

Code

```
void orexpNodeCode(TreeNode n, VarType type){
    //orexp -> orexp OR andexp | andexp
    if( n.Children.Count == 1 )
        andexpNodeCode(n.Children[0], out type);
    VarType t0;
    orexpNodeCode(n.Children[0], out t0);
    ...verify t0 is OK...
    string lbl = label();
    emit("mov rax, [rsp]");
    emit("cmp rax, 0");
    emit("jne {0}",lbl);
    emit("add rsp,8");
    VarType t1;
    andexpNodeCode(n.Children[2], out t1);
    ...verify t1 is OK...
    emit("{0}:", lbl);
    type = VarType.NUMBER;
}
```

Explanation

- Suppose the evaluation of child 0 leaves value v on the stack
- We copy v to rax and compare to zero
- Suppose v is zero
  - We do not take the branch
  - We pop the stack (by adding 8 to rsp) and fall through to the andexp node's code
  - That will leave its result on top of the stack, so this becomes the result of the entire orexp
- What if v is nonzero?
  - We take the branch. The result of child 0 is still on top of the stack
  - We are at the end of the orexp code, so we're done.

Optimizing

- This last example shows the concept of *optimizing* code
- We'll discuss optimization in more detail later, but essentially amounts to trying to choose fastest code sequence
- In general, register operations are fastest
- Accessing RAM is much slower (ex: variable access; push/pop)
- We'd prefer to keep as many operations as possible in registers

Alteration

- The code as we've described it is not very good: It uses memory (the stack) heavily
- We could modify our code to *spill* values to the stack only when necessary
- Ex: Maybe we devote registers r8-r15 to temporaries
- We keep track of which registers are in use and which are free
- When we need a temporary, we use one of the registers if one is available
- Otherwise, use the stack
- We'd need our Attributes structure to also tell where we put the value

Analysis

- This can make generated code much more efficient
- But: It’s also more complex!

Assignment

- Complete the code for the rest of the arithmetic hierarchy (except for factor: Leave it as just NUM and LP expr RP)
- If you want to be impressive, use registers instead of the stack for temporaries
- Use the test harness: Main.cs, ExeTools.cs, GrammarData.cs, and inputs.txt
- As before, you can't assume the existence of grammar.txt, but you can embed the full grammar in your executable as a C# string

Sources

- Intel Corp. Intel Reference Manual.

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org