Exercise Compiler Construction (List 11)

Due: 16.02.2014

Complete the following tasks. They are not programming tasks! They are about the compilation algorithm in slides **compilation.pdf**. If you don't understand the material or what should be done, then ask me by email. Don't wait until the last moment if you have problems. If you are ready to show your tasks to me, contact me some time (a few days) in advance.

1. Consider the following implementation of **fact**: (It is not an inline function.)

```
int fact( int i )
{
   int res = 1;
   while( i != 0 )
   {
      res = res * i;
      i = i - 1;
   }
   return res;
}
```

   (a) Give for each of the statements in this function (there are 4, if you don't count the return statement) the syntax tree.

   (b) Now apply the type checking algorithm in **typechecking.pdf**, slides 18-20 on the syntax trees. Give for each of the four statements the output of the type checking algorithm. The algorithm modifies the syntax tree (I want to see the new trees), and it assigns types to each subtree (which I also want to see).

   At the bottom of this task list is a list of inline functions that you can use. (I think that the list is complete. If you think that something is missing, then mail me.)

   (c) Give the output of the compilation algorithm in **compilation.pdf** on the trees given under (b). Use the list of inline functions at the bottom of the task list. Don't try to optimize the resulting code, because I want to be able to see that you applied the algorithm correctly.

2. Do the same as in the previous task (give the syntax trees, apply type checking on them, construct the translations) on the following modified **fact** function:

```
int fact( int i )
{
   int res = 1;
   while( i != 0 )
      res *= ( i -- );
   return res;
```

3. Same for the **sum** function below. There are 4 statements (not counting the return statement.) Assume that `p[ i++ ]` is an abbreviation of `*( p + ( i ++ ))`.

```
int sum( int* p, int len )
{
   int sum = 0;
   int i = 0;
   while( i != len )
   {
      sum += p[ i ++ ];
   }
   return sum;
}
```

4. (a) Consider the following **struct** definition:

```
struct list
{
   int val;
   list* rest;
};
```

Define the field functions list_val and list_rest for **list** as inline functions. Because they are inline functions, they should have types

```
list_val:  func( reg( ref( int )); reg( ref( list ) ))
list_rest: func( reg( ref( pointer( list ))));
                 reg( ref( list )) )
```

Follow the lay out of the inline functions at the bottom of the task list.

(b) Do the same as in task 1 for the following function that prints the integers in a linked list. (Give the syntax trees, the typechecked trees, and the compilations of the three statements.)

```
int sum( list* l )
{
   while( l != 0 )
   {
      print( l -> val );
      l = ( l -> rest );
   }
   return res;
}
```

Assume that `l->val` is an abbreviation of `(*l).val`, and that `l -> rest` is an abbreviation of `(*l).rest`. Assume that print is an outline function with type `print: func(void; int)`. Use the field functions that you gave in task 4a.

Available Inline Functions

Arithmetic Operators

```
operator + : func( reg(int); reg(int), reg(int) )
out: @J
in: @I1, @I2
   @J = @I1 + @I2;

operator * : func( reg(int); reg(int), reg(int) )
out: @J
in: @I1, @I2
   @J = @I1 * @I2;

operator - : func( reg(int); reg(int), reg(int) )
out: @J
in: @I1, @I2
   @J = @I1 - @I2;

operator + : func( reg(pointer(int)); reg(pointer(int)), reg(int) )
out: $Q;
in:  $P, @I;
local: @J;
   @J = @I * sizeof(int); $Q = $P + @J;

operator == : func( reg( bool ); reg(int), reg(int) )
out @B;
in @I1, @I2;
   @B = ( @I1 == @I2 );

operator != : func( reg( bool ); reg(int), reg(int) )
out @B;
```

```
in @I1, @I2;
   @B = ( @I1 != @I2 );

operator == : func( reg( bool ); reg( pointer(int)), reg( pointer(int)) )
out @B;
in $P1, $P2;
   @B = ( $P1 == $P2 );

operator != : func( reg( bool ); reg( pointer(int)), reg( pointer(int)) )
out @B;
in $P1, $P2;
   @B = ( $P1 != $P2 );
```

Assignment Operators

```
operator = : func( reg(ref(int)); reg(ref(int)), reg(int) )
out: $Q
in: $P, @I;
   [ $P ] = @I;  $Q = $P;

operator = : func( reg(ref(pointer(int)));
                   reg(ref(pointer(int))), reg(pointer(int)) )
out: $R
in: $P, $Q;
   [ $P ] = $Q; $R = $P;

operator += : func( reg(ref(int)); reg(ref(int)), reg(int) )
out: $Q;
in: $P, @I;
local: @J;
   @J = [ $P ]; @J = @J + @I; [ $P ] = @J; $Q = $P;

operator *= : func( reg(ref(int)); reg(ref(int)), reg(int) )
out: $Q;
in: $P, @I;
local: @J;
   @J = [ $P ]; @J = @J * @I; [ $P ] = @J; $Q = $P;


operator X++ : func( reg(int); reg(ref(int)) )
out: @I;
in: $P;
local @J;
   @J = [ $P ]; @I = @J; @J = @J + 1; [ $P ] = @J;

operator X-- : func( reg(int); reg(ref(int)) )
out: @I;
```

```
in: $P;
local @J;
   @J = [ $P ]; @I = @J; @J = @J - 1; [ $P ] = @J;

operator ++X : func( reg(ref(int)); reg(ref(int)) )
out: $Q;
in: $P;
local @I;
   @I = [ $P ]; @I = @I + 1; [ @P ] = @I; $Q = $P;

operator --X : func( reg(ref(int)); reg(ref(int)) )
out: $Q;
in: $P;
local @I;
   @I = [ $P ]; @I = @I - 1; [ @P ] = @I; $Q = $P;
```

Read/Write

```
read : func( reg(int); int )
out: @I;
in: $P;
   @I = [ $P ];

write : func( int; reg(int) )
out: $P;
in: @I;
   [ $P ] = @I;

read: func( reg( pointer( int )); pointer( int ))
out: $Q;
in: $P;
   $Q = [ $P ];

write: func( pointer( int ); reg( pointer( int )) )
out: $Q;
in: $P;
   [ $Q ] = $P;

// I think that no other Read/Write operators are needed.
```

Inline Copy Constructors

```
inline-copy: func( reg(int); reg(ref(int)) )
out: @I;
in: $P;
   @I = [ $P ];
```

```
inline-copy: func( reg(pointer(int)); reg(ref(pointer(int))) )
out: $Q;
in: $P;
   $Q = [ $P ];

inline-copy: func( reg(pointer(list)); reg(ref(pointer(list))) )
out: $Q;
in: $P;
   $Q = [ $P ];
```

Prefix $\star$ (Used for Pointer Lookup)

```
*X: func( reg(ref(int)); reg(pointer(int)) )
out: $Q;
in: $P;
   $Q = $P;

*X: func( reg(ref(list)); reg(pointer(list)) )
out: $Q;
in: $P;
   $Q = $P;
```