UCT CSC305 2004 :: Compilers :: Test 2 [35 marks] :: 5 May

Error Recovery

1. Describe Burke-Fisher error repair. [4]

Any of the four marks below:
This form of error repair tries every possible single-token insertion, deletion or replacement √ at every point that occurs no earlier than K tokens √ before the point where the parser reported the error .√
Example: With K=15, if the parser gets stuck at the 100th token of the input, then it will try every possible repair between the 85th and 100th tokens.
The correction that allows the parser to parse furthest past the original reported error is taken as the best error repair √
Example: If a single token substitution of `var` *for* `type` *at the 98th token allows the parsing engine to proceed past the 104th token without getting stuck, the repair is a successful one. Generally, if a repair carries the parser R=4 tokens beyond where it originally got stuck, this is "good enough". √*

Abstract Syntax Trees

2. Describe the Visitor pattern and its use. [4]

Any of the four marks below:

A visitor is an object which contains a `visit` *method for each syntax-tree class√ Each syntax-tree class should contain an* `accept` *method. √*

An `accept` *method serves as a hook for all interpretations. √*

The `accept` *method is called by a visitor and it has just one task – to pass control back to an appropriate method in the visitor. √ (Thus control goes back and forth between a visitor and the syntax-tree classes)*

Intuitively, the visitor calls the accept method of a node and asks "what is your class?" √

The `accept` *method answers by calling the corresponding visit method of the visitor√*

Summary: With the Visitor pattern a new interpretation can be added without editing and recompiling existing classes√, provided that each of the appropriate classes has an `accept` *method. √*

Symbol Tables

3. What is a symbol table? Give one example of the type of problem it helps to solve when writing a compiler. [4]

A symbol table is a mapping of names/symbols to attributes [2]

Problems it checks for (any one worth 2 marks):

Is X declared before it is used?
Are any names declared but not used?
Which declaration of X does this reference?
Is an expression type-consistent?
Do the dimensions of a reference match the declaration?
Where can x be stored? (heap, stack, ,,,)

*Does *p reference the result of a malloc()?*
Is x defined before it is used?
Is an array reference in bounds?
Does function foo produce a constant value?

4. In terms of non-local name resolution, what is the difference between static and dynamic scope? [2]

Non-local names resolved by static scope depend on the lexical nesting of subprograms while with dynamic scope, resolution depends on the call sequence.

5. Explain how entries in a recently closed scope (assuming static scope) can be removed from an imperatively designed symbol table, implemented as a hash table. Draw a diagram to support your explanation. [5]

```
+----+        ---
| -- |         ^
| -- |         |
| HT |       +----+     +----+
| -- |  -->  |    | --> |    | --> |
| -- |       +----+     +----+
| -- |         ^
| -- |         |
| -- |       +----+
| -- |  -->  |    | --> |
| -- |       +----+
|    |         ^
|    |         |
+----+    Nodes in Current Scope  [2]
```

Connect together all nodes inserted into the hash table in a single scope using a linked list. Then, when the scope ends, traverse the linked list and remove each node from the hash table. [3]

Activation Records

6. What is an activation record? [2]

A list of all the data (local variables, return values, parameters, static links, etc.) needed to support the invocation of a subprogram/function/procedure/method.

7. With non-reentrant subprograms, why is a stack not necessary for activation records? [2]

Because there is only ever one activation record instance for each subprogram/function/procedure/method so these can occupy a fixed area of memory or the same area of memory.

8. Draw the stack of activation records corresponding to the following Pascal-like program when it is at “breakpointX”. [5] (Assume static chains and include all parameters).

```
program main ()

   subprogram funca ()
   {
      funcb ();
   }
   subprogram funcb ()
   {
      subprogram funcc ( int x )
      {
         x = x + 1;
      }
      funcc (6);
      // breakpointX
   }
   funca ();
}
```

```
funcb      static link       -----+   [1]
           dynamic link      --+  |   [1]
           return (funca)      |  |   [1/2]
                             <-+  |
funca      static link       -----+   [1]
           dynamic link      --+  |   [1]
           return (main)       |  |   [1/2]
                             <-+  |
main                              |
                             <----+
```

Intermediate Representations

9. Assuming the IR tree language in the attached page, convert the following statements/expressions to equivalent IR trees. (Assume a and b are stack frame variables at offsets k0 and k1 respectively from the frame pointer special temporary *fp*) Provide the final trees and do not use the Nx/Cx/Ex expression types/objects. [8]

   a. a+b

   b. while (a<1) { b = b + 1; }

a. [3] one mark for main tree, one for left subtree, one for right subtree

```
BINOP (+, MEM(BINOP(+, TEMP(fp), CONST(k0))), MEM(BINOP(+, TEMP(fp), CONST(k1))))
```

or

```
+ (MEM(+ (TEMP(fp), CONST(k0))), MEM(+ (TEMP(fp),CONST(k1))))
```

or a tree representation of the same

b. [5] one mark for labels, one for correct “b=b+1” statement, one for conditional jump, one for JUMP, one for nested SEQs

```
SEQ (SEQ (SEQ (SEQ (SEQ (
LABEL (top),
CJUMP (<, MEM (+(TEMP(fp),CONST(k0)), CONST(1), NAME(t), NAME(f))),
LABEL (t),
MOVE (MEM(+(TEMP(fp),CONST(k1)), +(MEM(+(TEMP(fp),CONST(k1))), CONST(1)))),
JUMP (top),
LABEL (f))
```

or a tree representation of the same