RHODES UNIVERSITY

Examinations - January 2017

Computer Science 301 - Paper 1

(Free information made available to the students 24 hours before the formal examination)

The "24 hour" exercise in November 2016 asked students to extend a Parva compiler to incorporate a "set" type, This compiler was required to support simple set manipulations, including the ability to create empty sets or sets initialised by a list of values, add elements to an existing set, check for membership of a set, create the union of two sets, display the members of a set and so on.

The "24 hour exercise" for the supplementary extends the November system still further, but has fewer tasks, only three to be precise:

(a) Extend the Parva compiler so that it will also support forming the intersection of two (or more) sets and forming the "symmetric difference" of two sets (sometimes call forming the "xor" or "exclusive or" of these sets).

(b) Providing a way of removing all elements from a set easily.

(c) Providing support for mutually recursive functions.

Programmers schooled on languages that impose a policy of "declare before use" - which is really for the benefit of the compiler writer - learn to accept this as natural after a while, but there are situations where it becomes impossible to write code that conforms to this requirement. The classic example is provided by situations in which functions or methods are mutually recursive - function `one` wishes to call function `two`, which in turn must be able to call function `one` recursively.

Handling mutually recursive functions is not particularly difficult if the compiler first builds an AST for the whole program. An alternative strategy that is often adopted relies on a syntactic device, the so-called **forward declaration** of a function. The programmer is required to provide just enough information about the function so that it can be referenced even before it is fully defined. The device will be familiar to C, C++, Modula-2 and Pascal programmers, as all these languages provide for it. In C a forward declaration is known as a **function prototype**, and the idea is illustrated below.

Here is a simple program that illustrates all of the extensions for today's fun (`sample.pav`).

```
void one();       // function prototype
void two(int x);  // function prototype

void main() {
  one();
  two(3);

  set a = set{ 1, 2, 3, 4 };
  set b = set{ 3, 4, 5, 6 };
  writeLine( a + b );  // union        { 1, 2, 3, 4, 5, 6 }
  writeLine( a - b );  // difference   { 1, 2 }
  writeLine( a * b );  // intersection { 3, 4 }
  writeLine( a / b );  // xor          { 1, 2, 5, 6 }
  Clear(b);            // remove all elements
  writeLine(b);                       { }
} // main

void one() {
  two(1);
} // one

void two(int x) {
  if (x > 1) two(x-1);  // recursive call
} // two
```

The function prototypes declared ahead of function `main` can lead to the construction of symbol table entries that provide enough information for a single-pass compiler to generate code for evaluating and checking arguments and generating calls to these functions in all respects other than supplying the address needed in the actual call - a

minor complication that can be handled in the same way as for the forward branches needed in the compilation of an *IfStatement* or *WhileStatement*.

The syntactic change we would make to the Parva grammar to allow for forward declarations of this sort is almost trivially simple. One possibility is:

```
FuncDeclaration    = "void" Ident "(" FormalParameters ")"  ( Body | ";" )
FormalParameters   = [ OneParam { "," OneParam } ] .
OneParam           = Type Ident .
```

The implementation of this particular extension will provide you with a useful exercise for today, but we can reveal the general idea here. The *Entry* class is extended to include a Boolean field `defined` that can be used to signal whether the declaration of a function has incorporated the associated *Body* or has stopped short at the prototype stage. The *FuncDeclaration* parser, as before, prepares a symbol table entry in much the same way as in the compiler you have in your kit, with the exception that it does not immediately define the `entryPoint` field. Before inserting the entry into the symbol table a search is made to see whether an `oldEntry` for a function of the same name exists. If this is the case, a check of the `oldEntry.defined` field will highlight any invalid redeclaration, while if no such entry exists the new entry can be added to the symbol table. Analysis of the formal parameter list can take place as before. After this, if no *Body* is detected the `entry.defined` field can be set to `false`, in anticipation of the check needed when the later complete declaration is made. However, if the *Body* is present, it is necessary to update any extant entry to mark it as now fully defined and to define the `entryPoint` (which for our system will have the convenient side effect of backpatching any partially generated calls to the function).

As usual, in all three of these tasks you should pay heed to the semantic checks needed to prevent meaningless programs from being accepted.

Hints:

(a) The examination kit includes all the files needed to build a working Parva compiler similar to the one developed in your last practical exercise, and with the extensions needed to bring it up to incorporate the solutions to the extensions asked for in the November examination (it also has some, but not all of the extensions you were asked to make in the last practical, but you need not bother to try to add the missing extensions again).

(b) Depending on your approach, your solution may require modifications to any or all of the grammar and support files in the exam kit.

(c) Later in the day - at 16h00 - we shall release more information, to help those of you who may not have completed the exercises to do so. The examination tomorrow will include a set of unseen questions probing your understanding of the system, both as developed in November and possibly as extended today.

(d) Rest assured that you will not be expected to reproduce a complete Parva compiler from memory under examination conditions, but you may be asked to make some additions or improvements to the system.

(e) Remember Einstein's Advice: "Keep it as simple as you can but no simpler" and Terry's Corollary: "For every apparently complex programming problem there is an elegant solution waiting to be discovered".

RHODES UNIVERSITY

Examinations - January 2017

Computer Science 301 - Paper 1

(Free information made available to the students 16 hours before the formal examination)

Here are suggestions for how the problems posed this morning might be solved. Full details are not given here, but are easily added - simple additions to the list of opcodes in the PVM, and corresponding additions to the code generator. In tomorrow's formal exam you will receive a version of the Parva sources in which these insertions have all been made.

To handle set intersections and symmetric set differences, the production for *MultExp* can be modified as follows

```
MultExp<out int type>                 (. int type2;
                                         int op; .)
=  Factor<out type>
   { MulOp<out op>
     Factor<out type2>                (. if (IsArith(type) && IsArith(type2)) {
                                           type = Types.intType;
                                           CodeGen.BinaryOp(op);
                                         }
                                         else if (IsSet(type) && IsSet(type2)) {
                                           switch (op) {
                                             case CodeGen.mul : CodeGen.BinaryOp(CodeGen.isx); break;
                                             case CodeGen.div : CodeGen.BinaryOp(CodeGen.xor); break;
                                             default: SemError("invalid set operation"); break;
                                           }
                                           if (type != type2) type = Types.setType;
                                         }
                                         else {
                                           SemError("arithmetic operands needed");
                                           type = Types.noType;
                                         } .)
   } .
```

To clear all elements from a set, a simple *ClearStatement* can be added to the collection of statements

```
ClearStatement                        (. DesType des; .)
=  "Clear" "(" Designator<out des>    (. if (des.entry.kind != Kinds.Var)
                                           SemError("variable designator required");
                                         if (des.cannotAlter)
                                           SemError("you may not alter this variable");
                                         if (des.type != Types.setType)
                                           SemError("set designator needed");
                                         CodeGen.Clear(); .)
   ")" WEAK ";" .
```

Finally, to handle function prototypes one can modify the *FuncDeclaration* production as follows:

```
FuncDeclaration
                                      (. StackFrame frame = new StackFrame();
                                         Entry function = new Entry(); .)
=  "void" Ident<out function.name>    (. function.kind = Kinds.Fun;
                                         function.type = Types.voidType;
                                         function.nParams = 0;
                                         function.firstParam = null;
                                         Entry oldEntry = Table.Find(function.name);
                                         bool completing = !oldEntry.defined;
                                         if (!completing) Table.Insert(function);
                                         Table.OpenScope(); .)
   "(" FormalParameters<function> ")" (. frame.size = CodeGen.headerSize + function.nParams; .)
   (                                  (. if (completing) {
                                           oldEntry.entryPoint.Here();
                                           oldEntry.defined = true;
                                         }
                                         else {
                                           function.entryPoint = new Label(known);
                                           function.defined    = true;
                                         }
                                         if (function.name.ToUpper().Equals("MAIN")
                                             && !mainEntryPoint.IsDefined()
                                             && function.nParams == 0) {
                                           mainEntryPoint.Here();
                                         } .)
       Body<frame>
     | ";"                            (. function.entryPoint = new Label(!known);
                                         function.defined = false; .)
   )                                  (. Table.CloseScope(); .) .
```