CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 151/251
Fall 2009
Radimsky

COMPILER CONSTRUCTION

Project

The purpose of this project is to develop a compiler for the language NewL described in the appendix. You are expected to write a single-pass compiler which consists of phases, similar to those discussed in class. You are expected to use an object-oriented language such as C++ or Java. Design the appropriate objects and use exceptions to communicate error conditions and related information across the compiler. Note: references to LEX and YACC are generic to equivalent tools adapted to the implementation language you are using.

Project Schedule

Note: I am giving you dates to guide you in your progress. I will expect a “status report” and a copy of your input and output files to demonstrate debugging of the corresponding phase on each of these dates.

Part I:	Source Handler	finished by	October 8
Part II:	Scanner	finished by	October 13
Part III:	Parser	finished by	October 27
Part IV:	Error Recovery	finished by	November 12
Part V:	Symbol Table	finished by	November 19
Part VI:	Semantic Analysis	finished by	December 1
Part VII:	Intermediate Code Generation	finished by	December 15

Source Handler

Write a source handler similar to the one in the handout I gave you. Test your program with text which is not a program in any language (Test 1).

Scanner as an Automaton Simulation

After (or in parallel to) implementing the source handler, implement the lexical analyzer for NewL. You can either use LEX or do it by hand (if you are in CSc 251 you must use LEX). Unless you use LEX you will implement it by creating first an automaton that accepts the tokens. You are to develop by hand the appropriate transition table. This table will then be built in or read in, and processed by a driver which will read the input character by character.

In order to build your lexical analyzer I suggest you do the following:

(1) Make a list of the reserved words and special symbols of your language and define an appropriate internal representation for these symbols.

(2) Program the scanner as an object including a method that scans the next symbol of a program and assigns the corresponding representation which is then passed by appropriate means to the parser. You may use linear searching to recognize reserved words and identifiers. Starting to construct the symbol table would be a good idea.

(3) Write a program (Test 2) to test the scanner systematically. Make the scanner analyze this program and print the symbols.

You need to turn in by September 29 either the regular expressions you will feed to LEX or the finite automaton you created.

Syntax Analysis

If you are taking 151 you may write a recursive descent parser, if you are in CSc 251 you must use YACC, or its equivalent.

If you write a recursive descent parser.

(1) Make a list of the First and Follow symbols of each BNF rule of PL. Find the BNF rules that do not satisfy the two Grammar Restrictions imposed by the use of recursive descent and rewrite those rules. ***This is due October 15.***

(2) Extend the compiler with parsing procedures without error recovery.

(3) Write a NewL program (Test 3) to test the parsing of correct NewL sentences. Use this program to test the parser.

Error Recovery

Once you have properly debugged your parser you will introduce error recovery.

(1) Extend the parsing procedures with recovery actions for syntax errors.

(2) Write a NewL program (Test 4) to test the detection of syntax errors and the error recovery. Use Tests 2 and 3 to test the final parser.

Symbol Table

Write the symbol table as an object and test it appropriately (you may want to start this earlier).

Semantic Analysis

This is decomposed into two parts:

Scope Analysis

(1) Extend the compiler with scope analysis. Hint: Remember that the symbols **integer, Boolean, false, and true** are reserved words, not identifiers.
(2) Write two NewL programs (Tests 4 and 5) to test scope analysis of programs without and with scope errors. Use these programs to test the scope analysis.

Type Analysis

(3) Extend the compiler with type analysis. Hint: The data types of NewL are denoted by reserved words instead of identifiers. Since types are not identifierd objects they cannot be described by object records. They must be described by variant records of another type.
(4) Write two test programs (Tests 6 and 7) to test type analysis of programs without and with type errors. Use these programs to test the type analysis.

Intermediate Code Generation

(1) Extend the compiler to make it generate intermediate code described in Section 7.5, you may add new procedures if needed as long as you carefully document:

1. What they are supposed to do.
2. Why they are needed.

The compiler stores the code in a table and outputs it at the end of the compilation. Find a method for handling the forward references in jump instructions. (The basic idea is to go back and modify jump instructions in the code table whenever the compiler reaches a point where a jump address is known.)

(2) Write a NewL program (Test 8) to test the intermediate code generation.

Appendix A

NewL Language Description

NewL is a restricted version of the Java language and has a similar semantics for the parts of the language that are included.

Terminal symbols are bolded (and when they are single characters they are in quotes).

NewL Grammar

Here is a NewL grammar, you may have to modify it to suit your needs:

1. Program ::= MainClass { ClassDecl }
2. MainClass ::= **class** Ident **"{" public static void main "(" String "[" "]"** Ident **")" "{"** Statement **"}" "}"**
3. ClassDecl ::= **class** Ident **"{"** { VarDecl } { MethodDecl } **"}"** |
   class Ident **extends** Ident **"{"** { VarDecl } { MethodDecl } **"}"**
4. VarDecl ::= Type Ident **";"**
5. MethodDecl ::= **public** Type Ident **"("** FormalList **")" "{"** { VarDecl } { Statement } **return** Exp **; "}"**
6. FormalList ::= Type Ident { FormalRest } | ε
7. Formal Rest ::= **","** Type Ident
8. Type ::= **int "[" "]"** | **boolean** | **int** | Ident
9. Statement ::= **"{"** { Statement } **"}"** | **if "("** Exp **")"** Statement **else** Statement | **while "("** Exp **")"** Statement | **System.out.println "("** Exp **")" ";"** | Ident = Exp **";"** | Ident **"["** Exp **"]" =** Exp **";"**
10. Exp ::= Exp Op Exp | Exp **"["** Exp **"]"** | Exp **"." length** | Exp **"."** Ident **"("** ExpList **")"** |
    Integer_Literal | **true** | **false** | Ident | **this** | **new int "["** Exp **"]"** | **new** Ident **"(" ")"** | **"!"** Exp |
    "(" Exp **")"**
11. ExpList ::= Exp { ExpRest } | ε
12. ExpRest ::= **","** Exp
13. Ident ::= **Letter** { **Letter** | **Digit** | **"_"** }
14. Integer_Literal ::= **Digit** { **Digit** }
15. Op ::= **&&** | **<** | **+** | **-** | *****

Appendix A

Sample NewL Program

```
class Factorial {
   public static void main(String[] a) {
      System.out.println(new Fac( ).ComputeFac(10));
   }
}
class Fac {
   public int ComputeFac(int num) {
      int num_aux;
      if ( num < 1 )
         num_aux = 1;
      else
         num_aux = num *  ( this.ComputeFac ( num - 1) );
      return num_aux;
   }
}
```