Programming Language Translation Lecture 28

Karen Bradshaw

(Covers end of Chap 11 and Chap 10, pp. 147 to end)

Driver program (1)

- The most important tasks that Coco/R has to perform are the construction of the scanner and parser.
- These must be incorporated into a complete program before they become useful.

- Any main routine for a driver program is a refinement of

```
BEGIN
  ProcessCommandLineParameters;
  IF Okay THEN
    InstallErrorHandler;
    InitializeScanner;
    InitializeSupportModules;
    Parse;
    SummarizeErrors;
    IF Successful() THEN
      ApplicationSpecificAction
    END
  END
END
```

Driver program (2)

- The customized driver frame file generally requires some additions:
  - It may be necessary to add application-specific using or import clauses, so that the necessary library support will be provided.

  - The default driver file shows one example of how the command-line option -l may be handled.
  - Other command-line parameters may be needed, and can be processed in similar ways.

- At the end of the default frame file can be found code like
  - Scanner.Init(inputName);
  - Errors.Init(inputName, dir, mergeErrors);
  - // ----- add other initialization if required:
  - Parser.Parse();
  - Errors.Summarize();
  - // ----- add other finalization if required:

- which usually needs alteration.

Driver program (3)

- For example, in the case of a compiler/interpreter:

```
Scanner.Init(inputName);
Errors.Init(inputName, dir, mergeErrors);

PVM.Init();
Table.Init();

Parser.Parse();
Errors.Summarize();

bool compiledOK = Parser.Successful();
int initSP = CodeGen.GetInitSP();
int codeLength = CodeGen.GetCodeLength();
PVM.ListCode(codeName, codeLength);
if (!compiledOK || codeLength == 0) {
            System.err.println("Cannot interpret code");
            System.exit(1);
}
else {
            System.err.println("Interpreting code ...");
            PVM.Interpret(codeLength, initSP);
}
```

Chapter 10
Practical Use of Coco/R

Error recovery (1)

- Synchronization point is specified by keyword SYNC.

- The effect is to generate code for a loop that is prepared to consume source tokens until one is found that would be acceptable at that point.

- Example:

```
Subtotal = Range { "+" Range }
                SYNC ( "accept" | "cancel" ) .
```

- The union of all the synchronization sets (denoted by AllSyncs) is used in further refinements on this idea.

Error recovery (2)

- This would generate code equivalent to:

```
void Subtotal ()
 // Subtotal = Range { "+" Range }
 //                  SYNC ( "accept" | "cancel" ) .
  {   Range ();
      WHILE ( sym.kind = plusSym) DO
            {  getSym(); Range(); }
      WHILE (sym.kind  ∉  [acceptSym, cancelSym,
                              EOFSym] )
             DO  getSym();  // skipping past incorrect syms
      IF (sym.kind ∈ [acceptSym, cancelSym] ) THEN
            getSym();
  }
```

Error recovery (3)

- A terminal can be designated to be weak in a certain context by marking it with the keyword WEAK.
- A weak terminal is one that might often be mistyped or omitted, such as the semicolon between statements.
- When the parser expects (but does not find) such a terminal, it consumes source tokens until it recognizes either a legal successor of the weak terminal, or one of the members of AllSyncs.
- Example:

```
Calc  = WEAK "clear" Subtotal { Subtotal } WEAK "total" .
```

Error recovery (4)

- This generates code equivalent to:

```
void Calc ()
  // Calc  = WEAK "clear" Subtotal { Subtotal }
  //             WEAK "total" .
{
    expectWeak(clearSym, FIRST(Subtotal));
    Subtotal ();
    WHILE (sym.kind IN [ integerSym, floatSym ] )
             DO Subtotal () ;
    expectWeak(totalSym, { EOFSym })
 }
```

Error recovery (5)

- The expectWeak routine would be equivalent to:

```
void expectWeak (Expected : TERMINAL;
                 WeakFollowers : SYMSET);
{
   IF sym.kind = Expected
     THEN getSym();
     ELSE
       {  ReportError(Expected);
          WHILE NOT (sym.kind IN [WeakFollowers + AllSyncs])
               DO   getSym();
       }
}
```

Error recovery (6)

- Frequently iterations start with a weak terminal:

  Sequence = FirstPart
  { "WEAK" ExpectedTerminal IteratedPart }
  LastPart .

- Such terminals are called weak separators and are handled specially.
- If the ExpectedTerminal cannot be recognized, source tokens are consumed until a terminal is found that is contained in one of the following three sets:
  - FOLLOW(ExpectedTerminal) (that is, FIRST(IteratedPart))
  - FIRST(LastPart)
  - AllSyncs
- Example:

  Subtotal = Range { WEAK "+" Range }
  ("accept" | "cancel") .

Error recovery (7)

- The generated code would be equivalent to

```
void Subtotal();
 {
   Range();
   WHILE WeakSeparator(plusSym,
                        [ integerSym, floatSym ],
                        [ acceptSym, cancelSym ] ) DO
              Range();

   IF Sym IN [acceptSym, cancelSym ] THEN
     getSym();
 }
```

Error checks

- Coco/R checks that:
  - each non-terminal has been defined by exactly one production;
  - there are no useless productions
  - the grammar is cycle-free
  - all tokens can be distinguished from one another (no two terminals have been declared to have the same structure)
- If any of these tests fail, no code generation takes place.

- In other respects the system is more lenient.
- Coco/R issues warnings if
  - a non-terminal is nullable (this occurs frequently in correct grammars, but may sometimes be indicative of an error);
  - the LL(1) conditions are violated, either because at least two alternatives for a production have FIRST sets with elements in common, or because the FIRST and FOLLOWER sets for a nullable string have elements in common.

Semantic errors

- The parsers generated by Coco/R handle the reporting of syntax errors automatically.
- Pure syntax analysis cannot reveal static semantic errors, but Coco/R supports a mechanism whereby the grammar designer can arrange for such errors to be reported in the same style as is used for syntactic errors.
- The parser class includes a static method that can be called from within the semantic actions, using an explanatory string as an argument to describe the error.

Semantic errors (2)

```
Range<out double r>          (. int low, high; r = 0; .)
=   Amount<out r>
  | IntAmount<out low>       (. r = low; .)
    [ ".."
      IntAmount<out high>    (. if (low > high)
                                    SemError("low > high");
                                else while (low < high)
                                  {  low++;
                                     r += low;  }   .)
    ] .
```

Next lecture …

- Please read first 2 pages of Chapter 12