Assignment 5

Was:

1. Add still more grammatical specification for new language forms
2. Add actions to obtain a parser

Enough with the grammars already.

Plenty of practice so far

Only one aspect among many in the compilation process

Let’s move on…
Assignment 5

Now:

1. Add actions to Assignment 4 grammar to obtain parser

Smaller language (Pond?), fewer syntactic headaches

Some uncertainty for remainder of assignments:

Same topics, not sure about coverage (Lake or Pond)

“Answer Canon”: Lake

May only provide solutions for Lake (superset of Pond)

3 basic tasks

Build parse trees

Build types/declarations

Define actions for error productions
A lot of code in assign5.tar.gz
Don’t get intimidated
  Don’t try to read it all, though task-driven exploration will be necessary

4 basic categories
  Expr (13 classes)
  Stmt (15 classes)
  Type (10 classes)
  Symbol table (5 classes)

Tremendous similarity
  Understand the top level abstract

For parse trees, 3 major hierarchies:
  expressions (*Expr.java)
  statements (mostly *Stmt.java)
  types (mostly *Type.java)

Misc. others:
  Declarator, FunctionDef

For the future:
  symbol table support
  Lists, Iterators
  Lake, LakeFlags, message support

Code hierarchy documented in *.html

Use these to help exploration

For top-level documentation:
  overview-tree.html
  Index-all.html
Parse tree classes contain fields relevant to type checking and structural analysis

Don’t worry about them until Assign. 6

For now, mostly will just need to use constructors

Start with ParseExpr.java

implements SourceItem

2 constructors
  pos+type, pos

1 abstract function
  computeType (abstract)

Accessors for id, type, position

Formatting code
  toString, format, format(args), dumpExpr
  mostly abstract

Consider subclass ArrayExpr.java

simple example

Constructors w, w/out pos

Implements:
  computeType
dumpExpr
  format

New fields with accessors
  base, index
Statement.java
much like ParseExpr
1 constructor, 2 fields
position
id (not initialized)
Display functions:
dumpStmt
toString
format

while loop in WhileStmt.java
2 constructors
optional pos + condition, body
2 fields
condition
body
Implements
dumpStmt
format

Type.java top of type hierarchy
No interesting constructors
3 fields
isConst, isVolatile, id
makeConst, makeVolatile
Display code
dumpType
format, formatBase, formatPre, formatPos
toString
Pointer type is PtrType.java
2 constructors
   with and without volatile, const
1 field
   base
Implements the functions from Type.java

Declarator
3 constructors
   everything, name+pos, just name
Fields
   name
   position
   type
   visibility
   id
   initValue

Display functions
   dumpDecl
   toString
   format
Declaration function
   declare (not necessary for Assignment 5)
Scope is an interface
    cannot implement anything of its own
Scope maintenance functions
    enterScope, exitScope, parentScope
Symbol table functions
    sets for types, decs, labels
    enter, find, extend
Mostly relevant to Assignment 6, but...

FunctionDef
    Implements Scope
    Subclass of declarator
3 constructors
Fields
    body
    _ft
    _file
Override insertType
    Some type checking done

Structural checking support
    maintain current loop, switch
    maintain gotos
Scope interface
    implement all functions
    initScope
    declareParameters
Again, Assignment 6 stuff
Assignment 5 now prints something!

Give option -pp to pretty print code
Give option -pt to dump parse trees

java lakec.Lake -pp test1.c

*Note: these won’t work until parsing program generates a parse tree

If a subtree of a parse tree is not being constructed properly, Null exception

Hints/suggestions

To guide your programming, look at the grammar type declarations in Lake.cup

For terminals, these specify type of values of token (available for parser actions)

terminal String ID, CHAR_CONST ...
terminal integer INTCONST, LONG_INT_CONST;
terminal Double FLOAT_CONST;
terminal LakePos INT, CHAR, FLOAT, DOUBLE ...

For non-terminals, type specifications define parse tree structure

non terminal DeclaratorList program;
non terminal DeclaratorList decs_defs, decl, decls ...
non terminal FunctionDef def;
non terminal Declarator var_dec, array_decl, primitive_decl;
non terminal ParseExpr expr, bool_or_expr, assign_expr ...
To define action for non-terminal, refer to class type specification

Use class constructors to build parse tree object

```java
non terminal ParseExp bool_or_expr

bool_or_expr ::= bool_or_expr1 BAR_BAR :pos bool_and_expr2
  { RESULT = new BinExpr(pos,e1,e2,BinExpr.boolOr); } ;

bool_or_expr ::= bool_and_expr1
  { RESULT = e1; } ;
```

Note that for keywords, punctuation, LakePos is value of token

```java
terminal LakePos BAR_BAR

bool_or_expr ::= bool_or_expr1 BAR_BAR :pos bool_and_expr2
  { RESULT = new BinExpr(pos,e1,e2,BinExpr.boolOr); } ;
```

For constants, situation is different

Must explicitly obtain position
Obtaining position for constants is somewhat painful
Position is on tokens, not value of token
Java cup “hides” actual tokens

That is, in base level declaration production:
base_decl ::= ID;
Only token has value already
Need string from ID

How to get position of ID token?

Ideal: notion of token built into parser
Special notation to access token itself
Some OO parser generators do this
Java cup reality: need a hack
Can directly access parse stack
Token is pushed on stack

Stack is CUP$parser$stack
Top of stack is CUP$parser$top

constant ::= INT_CONST:i
{;
    LakeSym token =
        (LakeSym)
        CUP$parser$stack.elementAt(CUP$parser$top-0);
    RESULT = new ConstExpr(token.position(),i);
};
Many places for lists
   Statements in a block
   Param lists
   Arg lists

A class for each list
   StatementList
   DeclaratorList
   ParseExprList

General guide line:
Create an empty list using constructors
New lists default to empty
Don't use null

Three common functions
   size
   iterator
   add

Typical iterator usage
   while (iter.hasNext())
     { Object elem = iter.next();
       ...

   Also hasPrevious(), previous()
   See full interface for more
Each list class has own iterator
  StatementIter
  ParseExprIter
  DeclaratorIter

Each has extra functions
  nextStmt, previousStmts
  nextExpr, previousExpr
  nextDecl, previousDecl

Avoids casts (don’t use array access)

For example, to construct singleton:

```
decl_def := defd
{}: RESULT = new DeclaratorList(1); RESULT.add(d); ;
```

Always want to build something
  Even for error productions
  Want as much real structure as possible

For error in type
  use ErrorType
For error in expr
  use ErrorExpr
For error in statement
  use null statement
For example, if error in a statement:

\[
\text{stmt} ::= \text{error}\text{ SEMI,pos}
\]
\[
\quad \text{RESULT} = \text{new ExprStmt(pos,null)};\}
\]

**Final suggestions:**

Worry about correct grammatical rules first
Keep the problem manageable
Deal with error productions later
Start with simplest grammatical forms, build top-level parse trees
Test with:
\[
\text{java lakec.Lake -pp <testfile>}
\]