Today:

Assignment 6 overview

Introduction to IR

Made some tweaks to the spec. since last lecture

Nothing major, but:

- Variables may now be initialized
- volatile eliminated
- character, integer literals assigned const type
Language and type system well-specified in homework

Read *carefully*, let spec. guide your implementation

Today: description of framework for implementing spec.

Warning: Lake is bigger than Pond; extraneous details in implementation

All types are implemented in subclasses of Type.java

Methods:
- `isAssignable` : is `lvalue` type?
- `equals` : checks type equality
- `isCompatible` : checks subtype compatibility
- `commonType` : returns lub in subtype order

Scalar types implemented in `ScalarType.java`:

Type-specific methods:
- `isConst` : is `type` `const`?
- `makeConst` : make `type` `const`
- `isSigned` : is `type` `signed`?
- `makeSigned` : switch `signed/unsigned`
- `sizeCode` : returns impl. “size” of type
Pointer types implemented in PtrType.java:

Type-specific methods:
- `baseType` : returns base of pointer type (e.g., for int ptr returns int)

Array types implemented in ArrayType.java:

Type-specific methods:
- `elementType` : returns type of array elements
- `bounds` : returns size of array type

Function types implemented in FuncType.java:

Type-specific methods:
- `params` : returns parameter DeclatorList
- `returnType` : returns function return type
Function types implemented in FuncType.java:

Type-specific methods:
- Params : returns parameter DeclatorList
- returnType : returns function return type

ParseExpr.java and subclasses contain computeType method

computeType assigns type to expressions
computeType called when expression parse trees constructed

Various conditional statements subclasses of Statement.java

Conditions automatically checked to ensure they're scalar types

Declarator.java and subclasses contain declare() method
declare() adds declaration to current scope
Scope is an interface

Methods:
- enterScope : extend current environment
- exitScope : retract current environment

Remember symbol table discussions.

Scope implemented by:
- Block.java (extends statement)
- FunctionDef.java (extends declarator)
- FileScope.java, with additional method currentScope
Matching specification to implementation:

Subtype compatibility implemented by isCompatible

Type assignability implemented by isAssignable

Type judgements for exprs implemented by computeType

Matching specification to implementation:

The binding function implemented in parser actions (assignment 5)

The bindings function implemented in parser actions:
- declare() adds bindings to environment
- Add parser action to type check variable initialization
- Use enterScope(), exitScope() to localize declarations
Statements mostly taken care of (conditions checked automatically), but:
- Add parser actions to typecheck
  
  \[
  \text{returned exprs (use parser.RETID)}
  \]
- Localize scope of blocks in parser actions

Use embedded actions to localize declarations:

```java
block ::= LBRACE p
  |
  | Block b = new Block(p);
  | b.enterScope();
  | ··
  | decls:stmts:SS RBRACE
  |
  | RESULT = (Block)FileScope.currentScope;
  | RESULT.setDecs(ds);
  | RESULT.setStmts(ss);
  | RESULT.exitScope();
  | ·;
```

If error encountered in type analysis:

Make call to Lake.error

Appropriate error message library in Messages.java
Extra credit: *structural checking*

- Check that case, default only occur inside switch
- Check that continue only occurs inside loop
- Check that break only occurs inside switch or loop
- Add control objects to statements during construction

Relevant methods in FunctionDef.java:

- currentFunction : returns function currently in scope
- currentControl : returns control statement currently in scope
- currentLoop : returns loop currently in scope
- currentSwitch : returns switch currently in scope

Structural checking implemented in parser actions:

```java
stmt ::= CONTINUE:pos SEMI
{
    Statement loop =
        FunctionDef.currentFunction.currentLoop();
    if (loop == null)
        Lake.error(pos, Messages.notInLoop, "continue");
    RESULT = new ContStmt(pos, loop);
};
```
Next topic: intermediate code representation (IR)

Lexing →
Parsing →
Type checking →
Intermediate code generation →
...

Parse tree representation:
  Chosen to be “easy” to build/check
  Specialized to Pond

Two goals for compiler construction:
  Want to share back-ends
  Want to share front-ends

? is intermediate representation (IR) or intermediate code
IR must isolate language semantics
Semantic rules implicit in parse trees, e.g.
compound arithmetic/assignment opns
Explicit in IR

IR must express
All constructs in all source languages
All machine instrs in all target arch's

IR should be low level
Easy step to machine-level

Popular structure for intermediate code:
“Three-address code”

\[ x := y \text{ op } z \quad // \text{ x,y,z lang. objects} \]

Perform op on values referred to by y and z, store in x

The IR scheme we will use for Pond is related, but distinct:

For the theoretically minded:
- Pond fairly expressive language
- Any language can be represented in three-address code
- Universe of PL features reducible to basic computational “atoms”
Pond IR very similar to parse tree

Still has 2 hierarchies
Expressions
Statements

Fewer subclasses now

IR statements lower level
Move
Expr
Jump
CJump
Label
Seq

IR expressions also lower level
Const (int)
Name
Mem
Temp
Reg
Binop
Eseq
Call
Variables, params others become locations

May eventually be on stack
May eventually be in register

Has associated info
   For param: where it came from
   For any: whether it can be in register