CSE 401 – Compilers

MiniJava Parser and AST
Hal Perkins
Winter 2009
Abstract Syntax Trees

- The parser’s output is an abstract syntax tree (AST) representing the grammatical structure of the parsed input.
- ASTs represent only semantically meaningful aspects of input program, unlike concrete syntax trees which record the complete textual form of the input.
  - There’s no need to record keywords or punctuation like (), ;, else
  - The rest of compiler only cares about the abstract structure
MiniJava AST Node Classes

Each node in an AST is an instance of an AST class
- IfStmt, AssignStmt, AddExpr, VarDecl, etc.

Each AST class declares its own instance variables holding its AST subtrees
- IfStmt has testExpr, thenStmt, and elseStmt
- AssignStmt has lhsVar and rhsExpr
- AddExpr has arg1Expr and arg2Expr
- VarDecl has typeExpr and varName
AST Class Hierarchy

- AST classes are organized into an inheritance hierarchy based on commonalities of meaning and structure.
- Each "abstract non-terminal" that has multiple alternative concrete forms will have an abstract class that's the superclass of the various alternative forms.
  - Stmt is abstract superclass of IfStmt, AssignStmt, etc.
  - Expr is abstract superclass of AddExpr, VarExpr, etc.
  - Type is abstract superclass of IntType, ClassType, etc.
AST Extensions For Project

- New variable declarations:
  - StaticVarDecl

- New types:
  - DoubleType
  - ArrayType

- New/changed statements:
  - IfStmt can omit else branch
  - ForStmt
  - BreakStmt
  - ArrayAssignStmt

- New expressions:
  - DoubleLiteralExpr
  - OrExpr
  - ArrayLookupExpr
  - ArrayLengthExpr
  - ArrayNewExpr
Automatic Parser Generation in MiniJ ava

- We use the CUP tool to automatically create a parser from a specification file, Parser/minijava.cup
- The MiniJ ava Makefile automatically rebuilds the parser whenever its specification file changes
- A CUP file has several sections:
  - introductory declarations included with the generated parser
  - declarations of the terminals and nonterminals with their types
  - The AST node or other value returned when finished parsing that nonterminal or terminal
  - precedence declarations
  - productions + actions
Terminal Declarations

- Terminal declarations we saw before:
  /* reserved words: */
  terminal CLASS, PUBLIC, STATIC, EXTENDS;

  ...

  /* tokens with values: */
  terminal String IDENTIFIER;
  terminal Integer INT_LITERAL;
Nonterminals are similar:

nonterminal Program Program;
nonterminal MainClassDecl MainClassDecl;
nonterminal List/*<...>*/ ClassDecls;
nonterminal RegularClassDecl ClassDecl;
...
nonterminal List/*<Stmt>*/ Stmts;
nonterminal Stmt Stmt;
nonterminal List/*<Expr>*/ Exprs;
nonterminal List/*<Expr>*/ MoreExprs;
nonterminal Expr Expr;
nonterminal String Identifier;
Java Generics and MiniJava

- CUP did not support Java generics when the MiniJava starter code was written, so there are some hacks

- An example: we’d like to write

```
nonterminal List<Expr> Exprs;
```

but instead the code has

```
nonterminal List/*/Expr*/ Exprs;
```

- There are other hacks. Deal with them as gracefully as you can
  - Don’t make pointless changes to the code – save your energy for more interesting things
Precedence Declarations

- Can specify precedence and associativity of operators
  - equal precedence in a single declaration
  - lowest precedence textually first
  - specify left, right, or nonassoc with each declaration

Examples:
- precedence left AND_AND;
- precedence nonassoc EQUALS_EQUALS, EXCLAIM_EQUALS;
- precedence left LESSTHAN, LESSEQUAL, GREATEREQUAL, GREATER_THAN;
- precedence left PLUS, MINUS;
- precedence left STAR, SLASH;
- precedence left EXCLAIM;
- precedence left PERIOD;
Productions

- All of the form:

\[
LHS ::= \text{RHS1} \{ : \text{Java code 1} : \} \\
| \text{RHS2} \{ : \text{Java code 2} : \} \\
| \ldots \\
| \text{RHS}_n \{ : \text{Java code } n : \};
\]

- Can label symbols in RHS with :\texttt{var} suffix to refer to its result value in Java code
  - \texttt{varleft} is set to line in input where \texttt{var} symbol was
Productions (cont.)

Example

Expr ::= Expr:arg1 PLUS Expr:arg2
   {: RESULT = new AddExpr(arg1, arg2, arg1left);:}

| INT_LITERAL:value{: RESULT = new IntLiteralExpr(
   value.intValue(),valueleft);:}

| Expr:rcvr PERIOD Identifier:message
   OPEN_PAREN Exprs:args CLOSE_PAREN
   {: RESULT = new MethodCallExpr(
   rcvr,message,args,rcvrleft);:}

| ... ;
Error Handling

- How to handle syntax error?
- Option 1: quit compilation
  - easy
  - inconvenient for programmer
- Option 2: error recovery
  - try to catch as many errors as possible on one compile
  - difficult to avoid streams of spurious errors
- Option 3: error correction
  - fix syntax errors as part of compilation
  - hard!!
Panic Mode Error Recovery

- When finding a syntax error, skip tokens until reaching a “landmark”
  - landmarks in MiniJava: ;, ), }
    - FOLLOW sets can be a useful source of “landmarks”
  - once a landmark is found, hope to have gotten back on track

- In top-down parser, maintain set of landmark tokens as recursive descent proceeds
  - landmarks selected from terminals later in production
  - as parsing proceeds, set of landmarks will change, depending on the parsing context
Panic Mode Error Recovery

- In bottom-up parser, can add special error nonterminals, followed by landmarks
  - if syntax error, then will skip tokens till seeing landmark, then reduce and continue normally

- E.g.

  Stmt ::= ... | error ; | { error }
  Expr ::= ... | ( error )
EBNF Syntax of initial MiniJava

Program ::= MainClassDecl { ClassDecl }

MainClassDecl ::= class ID {
    public static void main
        ( String [ ] ID ) { { Stmt } }
}

ClassDecl ::= class ID [ extends ID ] {
    { ClassVarDecl } { MethodDecl } }

ClassVarDecl ::= Type ID ;

MethodDecl ::= public Type ID
    ( [ Formal { , Formal } ] )
    { { Stmt } return Expr ; }

Formal ::= Type ID

Type ::= int | boolean | ID
Stmt ::= Type ID ;
| { {Stmt} }
| if ( Expr ) Stmt else Stmt
| while ( Expr ) Stmt
| System.out.println ( Expr ) ;
| ID = Expr ;
Expr ::= Expr Op Expr
| ! Expr
| Expr . ID( [ Expr { , Expr } ] )
| ID | this
| Integer | true | false
| ( Expr )
Op ::= + | - | * | /
| < | <= | >= | > | == | != | &&