ASTs

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 program

- no need to record keywords or punctuation like `()`, `;`, `else`
- rest of compiler only cares about abstract structure

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 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 in 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 MiniJava

We use the CUP tool to automatically create a parser from a specification file, `Parser/minijava.cup`

The MiniJava `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 and nonterminal 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;
```

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, GREATERTHAN;
precedence left PLUS, MINUS;
precedence left STAR, SLASH;
precedence left EXCLAIM;
precedence left PERIOD;
```

Productions

All of the form:

```
LHS ::= RHS1 {: Java code 1 :}
      | RHS2 {: Java code 2 :}
      | ...
      | RHSn {: Java code n :};
```

Can label symbols in RHS with `:var` suffix to refer to its result value in Java code

- *var*`left` is set to line in input where *var* symbol was

E.g.:

```
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
- avoid streams of spurious errors

Option 3: error correction
+ fix syntax errors as part of compilation
- hard!!

Panic mode error recovery

When find a syntax error, skip tokens until reach a “landmark”

- landmarks in MiniJava: **;**, **)**, **}**
- 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

In bottom-up parser, can add special error nonterminals, followed by landmarks

- if syntax error, then will skip tokens till see landmark, then reduce and continue normally

E.g.

```
Stmt    ::= ... | error ; | { error }
Expr    ::= ... | ( error )
```