Abstract Syntax Trees

The parser’s output is an abstract syntax tree (AST) representing the grammatical structure of the parsed input.

But first a digression.

Intermediate Representations

- Front end - produces an intermediate representation (IR)
- Middle end - transforms the IR into an equivalent IR that runs more efficiently (usually consists of several passes)
- Back end - transforms the IR into native code
- The IR encodes the compiler’s knowledge of the program at any point in time

Typical Implementation of a Compiler

Source Program

Tokens

Abstract syntax tree

AST and symbol tables

Intermediate Code Generator

Code Optimizer

IR

Back end

Target Program

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

Concrete Syntax vs. Abstract Syntax

- Concrete syntax: what the programmer wrote
  => Parse Tree
- Abstract syntax: what the compiler needs
  => Abstract Syntax Tree

Parse trees and abstract syntax trees

- Graphically represent grammatical structure of input program
  - Parse tree: tree representation of grammar derivation
  - AST: condensed form of parse tree
  - Operators and keywords do not appear as leaves
  - Chains of single productions are collapsed
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 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

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

Notes on MiniJava Project

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:
preference left AND_AND;
preference nonassoc EQUALS_EQUALS;
preference left LESSTHAN, LESSEQUAL, GREATEREQUAL, GREATERTHAN;
preference left PLUS, MINUS;
preference left STAR, SLASH;
preference left EXCLAIM;
preference left PERIOD;
Productions

All of the form:

\[
LHS ::= \text{RHS}_1 \{ \text{Java code 1} \} \ |
\text{RHS}_2 \{ \text{Java code 2} \} \ |
\ldots \ |
\text{RHS}_n \{ \text{Java code n} \};
\]

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

• `varleft` is set to line in input where `var` symbol was

E.g.: `Expr ::= Expr:arg1 PLUS Expr:arg2`  
\[
\{
\text{RESULT} = \text{new AddExpr}(\text{arg1, arg2, arg1left});
\}
\]

`| INT_LITERAL:value { \text{RESULT} = \text{new IntLiteralExpr}(\text{value.intValue(), valueleft}); } | Expr:expr PERIOD Identifier:message OPEN_PAREN Expr:exprs CLOSE_PAREN`  
\[
\{
\text{RESULT} = \text{new MethodCallExpr}(\text{rcvr, message, args, rcvrleft});
\}
\]

Extra Slides Start Here

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`