PL0 Compiler Data Structures

Ian J. Hayes

April 20, 2020

While parsing a PL0 program, the compiler builds an *abstract syntax tree* (AST) structure to represent the program as well as a symbol table containing information about all the identifiers declared in the program. These structures are used for checking the static semantics (type correctness) of the program and code generation.

1 Symbol table

The symbol table (Java class SymbolTable) contains an entry (class SymEntry) for every identifier declared in a PL0 program being compiled. Every entry records: the location in the source code (line/column effectively) at which the identifier was declared; the lexical level of the scope in which it is declared; and its type. There are entries for the following.

ConstEntry for a symbolic constant, which has an expression tree used to define the constant as well as the value of the constant once the expression tree has been evaluated;

TypeEntry for a type identifier;

VarEntry for a variable, which has an offset (address) once the variable has been allocated; and

ProcedureEntry procedure definition, which has a symbol table scope for the local declarations within the procedure and the abstract syntax tree for the block representing the body of the procedure. For code generation (later in the course) the location of the start of the code for the procedure (once the code generation for the procedure has begun).

Because a procedure may have local declarations, (i.e., declarations only visible within the procedure) the symbol table is organised into scopes (class Scope), one for each procedure and one for the main program, which form a tree-structured hierarchy that matches the nesting structure of procedure declarations. All the symbol table classes may be found in the package syms.

2 Abstract syntax tree

The abstract syntax tree is represented in the compiler by three Java classes DeclNode, StatementNode and ExpNode, and their subtypes (described below). A DeclListNode groups the abstract syntax tree nodes for the procedures into a list. Each entry in the list represents a single procedure and includes the procedure's symbol table entry (see Section 1), and a BlockNode representing its body. In DeclNode.java:

DeclNode() with subclasses
- DeclListNode(List⟨DeclNode⟩ decls)
- ProcedureNode(SymEntry.ProcedureEntry procEntry, BlockNode body)

The abstract class StatementNode represents a statement in the abstract syntax tree; it has a subclass for each kind of statement. In addition, ErrorNode caters for erroneous statements. All statement nodes contain a Location, which is the location in the source code corresponding to the node. A BlockNode represents the body of a procedure (or the main program). In StatementNode.java:

StatementNode(Location loc) with subclasses
- ErrorNode()
- BlockNode(DeclListNode procedures, StatementNode body, Scope blockLocals)
- AssignmentNode(ExpNode lvalue, ExpNode e)
- WriteNode(ExpNode e)
- CallNode(String id)
- ListNode(List⟨StatementNode⟩ sl)
- IfNode(ExpNode cond, StatementNode s1, StatementNode s2)
- WhileNode(ExpNode cond, StatementNode s)

There is no tree node corresponding to a read statement; instead a read statement is represented by an AssignmentNode where the left hand side is the variable being read into and the right hand side is a special expression node (ReadNode) representing the action of reading a value from standard input.
The abstract class ExpNode represents expressions in the abstract syntax tree; it has a subclass for each kind of expression. In addition, ErrorNode caters for erroneous expressions. All expression nodes have a location (in the source code) and a type (of the expression). In ExpNode.java:

ExpNode(Location loc, Type t) with subclasses
- ErrorNode()
- ConstNode(int value)
- IdentifierNode(String id)
- ReadNode()
- BinaryNode(Operator op, ExpNode left, ExpNode right)
- UnaryNode(Operator op, ExpNode arg)
- VariableNode(SymEntry.VarEntry entry)
- DereferenceNode(ExpNode leftValue)
- NarrowSubrangeNode(ExpNode e)
- WidenSubrangeNode(ExpNode e)

IdentifierNode is used during parsing to represent a reference to either a symbolic constant or a variable. As part of the static semantics (type checking) phase it will be transformed to either a ConstNode or a VariableNode. A number of other node types are only introduced in the static semantics phase:

- a DereferenceNode represents a dereference of a variable address (left value) to access its (right) value;
- an expression of a subrange type can be widened to the base type of the subrange (WidenSubrangeNode);
- an expression of a type, T, can be narrowed to a subrange of T (NarrowSubrangeNode).

An expression like x+1 is represented by a BinaryNode After parsing the structure is

BinaryNode(ADD_OP, IdentifierNode("x"), ConstNode(1))

After static semantic analysis, if we assume x was declared as a variable, with symbol table entry Entry_x, the tree will become

BinaryNode(ADD_OP, DereferenceNode(VariableNode(Entry_x)), ConstNode(1))

All the abstract syntax tree classes may be found in the package tree.