COP5621 Project Phase 3
Semantic Analyzer

Purpose:

This project is intended to give you experience in symbol table manipulation and semantic error detection as well as bring together various issues of semantic analysis discussed in class.

Project Summary:

Your task is to write a static semantic analyzer that creates a symbol table for the source program, augments the abstract syntax tree (produced by the parser) with appropriate semantic information, checks that the source program abides by some of the semantic rules of the PASC language, and reports any violations.

Scoping rules:

The scope of a name is the region of the program in which the name can be used. For objects declared within a procedure and formal parameter names of a procedure, the scope is from the point of declaration to the end of the procedure in which the name is declared, and objects of the same name in other procedures or declared in the main program's declarations are unrelated. Variables, symbolic constants and procedures declared in the main program are said to be *global*, accessible from the point of declaration to the end of the program, including the main program body as well as the bodies of procedure that do not redeclare the same name. The scope of the field identifiers of a record is the record type itself and as such they need not be distinct from identifiers declared outside the record type. Procedures can call procedures declared earlier in the program, and a procedure can be either directly or indirectly recursive, by using the forward declaration. The lifetime of object declarations within a procedure is limited to the execution of the procedure (i.e. the values are discarded upon exit), while objects declared as global last until the end of program execution.

Static Semantics:

The following static semantics must be performed.

- All names declared within the same block (i.e., main program, procedure, or record) must be unique.
- All names used within the program must be declared such that using the scoping rules, there exists a corresponding declaration visible from each use of a name (i.e., there should be no uses of undeclared names).
- The number of array indices in an indexed component should match the number of dimensions declared for that array.
- A name declared as a symbolic constant must not be assigned a new value.
- Identifier(s) in a subrange (used as array bounds or subrange type definition) must be previously defined as integer constants.

Symbol Table Creation:

Both static semantic checking and code generation require semantic information about names defined by the user. The *symbol table* is a data structure that stores the semantic information for each name declaration. A new entry is added to the symbol table as each declaration is encountered during semantic processing. Since code will be generated in a separate phase (in this project) using the information stored in the symbol table (and possibly adding more information to the table), symbol table entries are never removed after being created. If a given name X is defined in two different places in a program (say, as a global constant and then as a local variable within a procedure), then the final symbol table will contain 2 separate entries for X with the appropriate semantic information.

Semantic attributes are added to the entry for a particular name at various times. Every entry has *NameAttr* and *TypeAttr* attributes which are entered in response to the name's declaration and contain the unique index into the string table and (a pointer to) the appropriate type tree, respectively. Each entry also has a boolean attribute *PreDefinedAttr* which is true iff the name represents one of the predefined identifiers of the PASC language. Since PASC has no input/output defined as part of the language itself, the input/output routines are accessible to all PASC programs as a library. The identifiers are predefined by the PASC language as described by the following table. The symbol table should be initialized with an entry for each of these identifiers, where each entry for a routine contains *TypeAttr* set to point to a procedure (ProceOp) or function (FuncOp) declaration tree with appropriate formal arguments and result type and *PreDefineAttr* set equal to true. All other entries in the symbol table will have *PreDefineAttr* set to $false$.

Identifier	**Description**
integer	The predefined integer type.
char	The predefined character type.
string	The predefined string type. Implemented as an array of 256 (0..255) chars. *TypeAttr* must point to the corresponding $ArrayType$ tree.
write	May have 1 to n arguments. Print arguments of integer, char or string type and a <return>.
read	May have 1 to n arguments. Read integers and characters for the given actual arguments and then skip all the characters in the current line, until a <return> is read.
chr(i:integer) : char	Return the character of ASCII code i.
ord(ch: char) : integer	Return the ASCII code of character ch.

Table 1: Predefined Identifiers in PASC

Read(Write) is represented by a ProceOp subtree for its forward declaration. The specs subtree is null for these two procedures since the number and the types of their arguments are not fixed. *Chr* and *ord* should be represented by normal forward function declaration trees. Each entry in the symbol table should have a *LevelAttr* which is entered in response to the name's declaration and is an integer indicating the level that the symbol was defined (nesting level). Predefined entries should have the smallest nesting level. In order to distinguish between symbolic constants and variables (since they cannot be distinguished by *TypeAttr*, a boolean *IsConstAttr* attribute could be attached to all entries. Similarly, you may want a boolean *IsFormalAttr* attribute to distinguish between formal arguments and local variables.

From the description of some of the attributes, it is clear that an attribute could have a value of integer,

boolean, or ILTree. In addition to the standard attributes, your semantic analyzer may add other attributes to entries as necessary for semantic processing and code generation such that the number of attributes of a given entry may vary as you want to store different additional information for different entries. An acceptable way to design a symbol table entry is as a linked list of attribute values that can be of any type above. The order that attributes are added to the list for a given entry will be insignificant if you store an additional integer field with each attribute value. This field can indicate what attribute is stored there. For example, if the integer is 1, then the attribute value represents *TypeAttr*; if the integer is 2, then the attribute value represents *LevelAttr*, ... A symbol table entry could also be implemented as a row in a table.

There will be two flavors of lookup into the symbol table. When processing a new declaration, you will want to perform a lookup operation restricted to the current block of declarations (i.e. the name declared locally within this block thus far) to determine whether the symbol is multideclared. When processing the use of a name, you will want to perform a lookup over all currently accessible names starting with the local declarations, and if not found, continuing to the nonlocals. Thus, two lookup procedures are needed, and the symbol table must be organized to include block information such that the local declarations are found first.

One way to approach this is to maintain the symbol table as a stack that always contains the set of accessible declarations with respect to the current block being processed during symbol table creation and syntax tree augmentation. The stack will grow and shrink by the number of local declarations as processing enters and leaves a block, respectively, and the latest instance of a variable declaration in the currently accessible blocks will always be the one closest to the top of the stack. However, since the complete symbol table is needed for the code generation phase, the symbol table entries cannot be deleted during this phase. In order to retain symbol table entries that are no longer active (i.e. accessible) with respect to the current block being processed, the symbol table stack can be maintained as a separate data structure with references into the permanent symbol table structure.

For example, a stack of records can be maintained where each record contains a *boolean marker*, and *id* and a *pointer* to the entry for what id in the symbol table. When the semantic analyzer begins to process the declarations of a new procedure definition, an *OpenBlock* routine pushes a record (true, undefined, undefined) onto the stack to indicate the beginning of a new block. Processing of subsequent declarations within that procedure includes pushing records of (false, id, pointer) onto the stack. Note that the procedure name should be entered into the symbol table before calling Openblock to signal the start of a new block because the procedure name is actually defined where it is declared. When the limited form of lookup is called, it scans the stack from the top until it finds the desired id or the last true marker that was pushed onto the stack (indicating that the identifier is not yet declared in the current block). After processing all statements within a procedure definition, the *Closeblock* routine pops all records since the last true marker, and then pops the *true* marker.

The following symbol table utility routines have been written for you and are available from the file proj3.c.

```
STInit()
/*
 * Called before any other symbol table routine to initialize the symbol
 * table. You also need to modify this routine to entry the predefined
 * symbols
 */

InsertEntry(ID : integer) return STIndex
/*
 * Add an entry to current symbol table for identifier ID (unique index into
 * string table), returning index of entry in symbol table. This entry has
 * no attributes initially. ID should not have been defined since last
```

```
 * OpenBlock.
 */

LookUp(ID:integer) return STIndex
/*
 * Return most recently added instance of ID in symbol table, or NullST(i.e. 0)
 * if none.
 */

LookUpHere(ID: integer) return STIndex
/*
 * Return most recently added instance of ID in symbol table since last
 * OpenBlock. Return NullST if none.
 */

LookUpField(ID:integer; ST: STIndex) return STIndex
/*
 * Return the symbol table entry of record field TD. ST indicates where the
 * search starts in the symbol table. Return NullST if none.
 */

OpenBlock() /* Start a new block of symbols in the symbol table */

CloseBlock()
/*
 * Restore the symbol table to the set of STIndices prevailing
 * before the last OpenBlock
 */

IsAttr(ST: STIndex; AttrNum: integer) return boolean
/*
 * Return true iff ST has an attribute numbered AttrNum.
 * For a newly-defined symbol, ST, !(IsAttr(ST, n)) is true for all n
 */

GetAttr(St: STIndex; AttrNum: integer) return integer, boolean or ILTree
/*
 * return value of attribute attrnum of ST; value may be
 * integer, boolean or ILTree
 */

SetAttr(ST: STIndex; AttrNum: integer; V: integer, boolean, or ILTree)

STPrint() /* Print all symbols and attributes in the symbol table */
```

Syntax Tree Augmentation:

The semantic information to be added to the tree consists of (1) converting all leaves of type IDNode to leaves of type STNode, (2) replacing the integer value stored in each IDNode by a unique pointer into the symbol table, and converting certain subtrees into other subtrees. Each occurrence of an id is replaced with a reference to the symbol table entry containing the necessary semantic information for

that id. This may also require minor modifications to your tree manipulation routines of the second phase of this project to handle STNodes. Function calls which were handled as variable accesses (or variable accesses which were handled as function calls) must be converted from variable-access tree to function call tree (or vice verse). These tree structure modifications can be done as the types of the id's become apparent when replacing IDNode's by STNodes.

Static Semantic Checking

Uniqueness of id declarations can be checked when adding new entries to the symbol table for each constant, variable, or procedure declaration. **Undeclared id's** can be found when modifying the IDNodes in statement subtrees (and subrange type subtrees) to point to the symbol table. When an undeclared id is encountered, it should be entered into the symbol table to prevent repeat error messages. **Procedure calls with the incorrect number or type of arguments and nonvariable arguments passed as call by reference parameters** can be checked when augmenting the procedure name's IDNode leaf in the call with a pointer to the symbol table. Similarly, and **incorrect number of indices** in an indexed component can be detected when processing the array name's IDNode in the indexed component. **Assigned symbolic constants** can be detected by keeping an *AssignAttr* in each symbol table entry for a symbolic constant, initially set to $false$ and subsequently set to *true* upon replacing the left operand's IDNode in an assignment subtree. Immediately after processing the statement subtree for that block, the current block's symbol table entries are scanned, emitting a warning for each *AssignAttr* equal to *true*.

Type checking is **not** required in this assignment.

Testing the semantic Analyzer:

Your semantic analyzer should output the complete symbol table and the augmented syntax tree.

For grading ease, your syntax tree print routine should adhere to the same specifications as the second project with the following modification. Since each IDNode has been replaced by an STNode, **[ID:tokenvalue, lexeme] should be replaced by [STID: symbol table index, lexeme].** You will need to change the tree print routine to handle this.

You might also have to change the symbol table printer that is provided. The changed format should be a clear, concise format including all attributes of every entry. All the attributes should be clearly defined.

Error Handling:

Your semantic analyzer should print descriptive error messages and recover from all detected static semantic errors to find any additional semantic errors. Error messages should be as descriptive as possible, given the available information at the time of error detection.

Assignment Submission and Deadline:

The due date for this project is March 28 when you must hand-in your programs and do a demo.

Grading policy:

- Recognize correct programs (20)
- Correctly perfrom the six required semantic checks (5x6).
- Generate correct enhanced abstract syntax tree (20)

- Generate correct symbol tables (20)
- Error reporting (10).