

Programming Assignment 3

CS 322 Compiler Construction
Winter Quarter 2006

Due: February 25, 11:59pm

–Objectives

- Update the symbol table structure to store semantic information.
- Code and test the semantics for declarations (types, variables, functions).

–Before you start

Read this handout and study the provided code. Look at the test cases and the standard output and make sure you understand what you need to do.

–Evaluating semantics

In part 2 of the project, we used bison utilities to evaluate the *node* attribute for each grammar symbol. However, there is additional information that we need to collect and attributes to evaluate. Each identifier and expression has a *type* attribute that describes its type. Each expression has a *code* attribute that is the code that should be generated by that expression. In this part of the project, you will evaluate the *type* attributes (i.e. collect type information) and perform some basic type checking.

Since bison allows us only one attribute and we have used it to create the AST, the type analysis will be performed by traversing the AST in a depth-first manner.

The *Visit* methods that you implemented in part 2 were used to verify that your tree has been built correctly. We won't be needing them any longer. Instead, after the tree has been created, we will call a function to evaluate the *code* attribute for the whole program. This evaluation will consist of two parts: type analysis and some checking (which is the subject of this assignment) and more extended type checking and code generation (which will be done in a subsequent assignment).

–The grammar file

Use the grammar file you created in part2, with the following modifications:

The semantic action for `pascal_program` should be:

```
pascal_program : TPROGRAM ident TSEMI block TDOT
{
    Program* AST=new Program((Ident*)$2,(Block*)$4);
    AST->EvalCode();
}
```

```
    return 0;
};
```

`openFiles()` should also be modified. See the supplied code for the new version.

–The symbol table class

The following changes have been made to the symbol table implementation:

- A `SymbolTableEntry` has an additional member, a pointer to an `IdentInfo` object, which will contain information about the identifier.
- `IdentInfo *GetInfo(int index)` – This method returns a pointer to an identifier’s current `IdentInfo`.
- `void Attach(int theIndex, IdentInfo *theInfo)` – This method attaches a newly created `IdentInfo` node to the beginning of the list of `IdentInfo` nodes for the identifier with lexeme index `theIndex`
- `void Remove(int theLevel)` – This method removes all `IdentInfo` nodes for nesting level `theLevel` from the symbol table.
- `void Dump()` - This method prints the contents of the symbol table. It will be used to verify that your program works correctly.
- You will need to provide an additional `Find()` that returns (via the argument list, since the return value is already an index) a reference to an entry. This will enable us to easily access the `IdentInfo` for the identifier.

–The IdentInfo class

A `IdentInfo` object contains attribute values for an identifier. All identifiers share the following attributes:

- `lexemeIndex` – the index of the identifier in the lexemes array
- `scopeNumber` – the current nesting level for the identifier
- `type` – the type of the identifier (for functions, this is the return type)

In addition,

- variables have a `stackOffset` attribute that holds their stack frame offset
- functions have a `paramCount` attribute that holds the number of parameters and a `label` attribute which is a counter that will be appended to the name of a function during code generation.

See the provided files for more comments. This class is fully implemented.

–The TypeDescr class

A `TypeDescr` object contains information about a basic or user-defined type. All type descriptors contain a `size` and the lexeme index for the name of the type. In addition,

- subrange types contain the low and high values of the range.
- pointer descriptors contain the lexeme index and type descriptor of their base type. Note that we only allow pointers to named types. For example, `type ptr = x;` is legal, but `type ptr = array[1..2] of integer` is not.
- array descriptors contain descriptors for the component type and subscript type of the array.

See the provided files for more comments. This class is fully implemented.

–Attribute : program.code

The code for the whole program will be generated by `Program::EvalCode` and functions called from within this method. For now, only some basic link code is generated. See `ast.cpp` for more information.

–Attribute : block.code

The code for a block is generated in `Block::EvalCode`. For now, only some basic link code is generated. See `ast.cpp` for more information.

–Attribute: function_decl.type

In `FuncDecl::DeclareFunc()` you will collect semantic information related to function declarations. See `ast.cpp` for more information.

–Attribute: var_decl.type

In `VarDecl::DeclareVars()` you will collect semantic information related to variable declarations. See `ast.cpp` for more information.

Note : Before parsing of variable declarations begins, the value of the first stack offset (i.e. the next location in the stack frame) should have been set to

$1 + \text{FUNC_STORAGE_OVERHEAD} + \text{number_of_parameters}$ where `FUNC_STORAGE_OVERHEAD` is 2.

–Attribute: param_decl.type

In `ParamDecl::DeclareParams()` you will collect semantic information related to parameter declarations. See `ast.cpp` for more information.

–Attribute: .type for specific types

The `EvalType` method for each type creates (if necessary) and returns an appropriate type descriptor.

–Fixing pointers

Consider the following declarations:

```
type ptr = ^integer;
var x : ^ptr;
```

When the type of `x` is evaluated, a type descriptor (`PointerDesc*`) is created for it. You must then make sure that the `targetType` for `ptr` points to that type descriptor.

Now consider:

```
type ptr = ^int;
    int = integer;
```

When the first type definition is traversed, the type descriptor of 'int' is not known. After the second type definition is traversed, your program should be able to update the type descriptor information for 'ptr'.

You must decide when and how to fix forward declarations such as this.

–Dumping the symbol table

You must implement the debugging `Dump` function for the symbol table. It essentially goes through the table and prints out the identifier and type information for each name. The `Dump` methods for the `IdentInfo` and `TypeDescr` classes are already done.

The output will be printed in a file called `test.?d` all Look at our sample output to see what kind of format we expect. You may print to this file using `dump <<`

–Global variables

The following variables have been defined in `globals.h`

`next_stack_offset` – holds the next location (for a variable or parameter) in the stack frame.

`next_label` – is just a label that will be attached to the name of each function at the last phase of the project. It should be incremented every time a new function is encountered.

`scopeNumber` – helps you keep track of which identifiers are active at any given time. It is initialized to -1 and should be incremented and decremented according to the depth of nesting of functions.

`intDescr`, `charDescr`, `boolDescr` – Type descriptors for the basic types. These are initialized in `SymbolTable::Initialize()`.

The type descriptors for the three basic types are already created in `SymbolTable::Initialize()`. You must figure out where and how to initialize the rest.

–Files

The distribution directory contains the following files:

- `ast.*` – The abstract syntax tree class definitions and implementations. You will edit both `ast.cpp` and `ast.h`
- `typedescr.*` – The definitions and implementations for the type descriptor class and related subclasses. You do not need to edit them.
- `globals.h`
- `grammar.y` – Note the changes described above.
- `identinfo.*` – The definitions and implementations for the `IdentInfo` class and related subclasses. You do not need to edit them.
- `syntab.*` – The symbol table class definition and implementation. You will edit `syntab.cpp` and possibly `syntab.h`
- `flex.l` This is provided only to ensure that the given code compiles. Obviously, you will use your own `flex.l`

–Submitting your code

Tar and email all files for this project to `c22@cs.northwestern.edu`