Project Compiler - Semantics

Out: 23 Nov 2011
Semantic Analysis Due: 9 Dec 2011, 11:59PM
Compiler Due 16 Dec 2011, 11:59PM

0 Motivation

You have made use of our tokenizer and created your own parser, building an Abstract Syntax Tree and catching Lexical and Syntax errors. This isn't all you must do, though. Now that you know that the code satisfies the Blaise grammar, you must use the AST you generated to ensure that the code "makes sense". Additionally, you must gather information necessary for code generation.

Consider the following example program.

```
procedure int main() {
  int a;
  b := 4;
}
```

This program would generate a valid parse tree, which you can confirm with your parser. But clearly something is wrong here, since `b` has never been declared.

One goal of semantic analysis is to find such errors.

1 Requirements

- Modify the provided Java code such that it realizes the described specification.
- Hand in your fully functional SemanticVisitor by the date at the top of this handout.
- Hand in test programs that FULLY test your semantic visitor.
- Your handin must include a `README` file that
  - describes the bugs in your program (or asserts their absence)
  - lists any changes made to the support code (i.e. classes added or edited)
  - explains your test program(s)
  - explains anything else the grader needs to know

2 Details

2.1 Support

2.1.1 Class overview

In addition to the classes used for your Parser, the following will be used in this portion of the assignment.

- **Scope** A dictionary of declared variables that is used for semantic analysis. It allows you to make sure that variables are declared correctly and are used consistently with their declarations. How you make use of this class, if you choose to at all, is up to you.
- **Procedure** This represents a procedure and its list of arguments. Since all arguments and return types in Blaise are ints, the arguments are represented only by their identifier Strings. Additionally, this class will be used to determine where to store each local variable for a procedure. You will use this important class in both semantic analysis and in code generation, so be sure to look it over. Below, you'll find more information about its methods and how to use it.
- **Visitor** This is the interface for your `SemanticVisitor` and later for your `CodeGenVisitor`. Details are below.
  - **SemanticVisitor** This is the class you will modify in this portion of the assignment. We have provided code for some of the simpler methods to get you started. Look it over so you understand what you must do.
- **Exception*** In this stage, you will create and throw `SemanticExceptions`. As before, make sure that you catch the exception in your `main` method!

2.2 The Visitor pattern

Once one has a parse tree, there are many ways to access its data. Some involve having the tree do the work. Others involve large controller classes. For this assignment we will use the Visitor pattern as described in the Compiler II lecture.

With the Visitor pattern, the nodes are sort of playing a game of "hot potato," where an active object is passed throughout a group.

Each node contains a `visit()` method, which gets passed a `Visitor`; the job of this method is to invoke the visitor on the node's `visit()` method by calling the appropriate `handle..()` method on the visitor. This pattern allows for a polymorphic approach to dealing with the differing node types.

This technique is known as the visitor pattern.

The provided `PrintingVisitor` and `GraphvizVisitor` classes are examples of the Visitor pattern, so look at them to get a sense of how to use this pattern.

If you are comfortable with the Visitor pattern then understanding the design of compiler will be much easier!

For some additional reading on the subject, take a look at *Design Patterns*, Gamma et al. There's a copy at the back of the sunlab.

2.2.1 Visitor pattern advantages and drawbacks

We use the visitor pattern when we want to make it easy to add new operations (all this requires is writing a new visitor class), but we will rarely want to change the classes that make up the tree (this would require changing any affected nodes, plus adding methods to every visitor).

So the big advantage is that it's easy to add operations; the big disadvantage is that it's hard to add node types.

2.3 Semantic analysis

A `SemanticVisitor`, which you must implement, has the responsibility of checking the syntax tree for semantic correctness. If an error is detected, you should print a message to the user and exit with an exception (though you do not have to associate a line number with a semantic error; think about why this is difficult). Be sure you are catching exceptions in `main` and exiting gracefully!

Here is a summary of all the tasks your `SemanticVisitor` must perform:

1. Ensure that all rules for procedures are upheld (e.g. each program has a suitable `main` procedure)
2. Detect scoping errors (e.g. referencing a variable that doesn't exist)
3. Detect type errors (e.g. trying to add an integer to a boolean)
4. Gather information about memory allocation to pass to the Code Generation phase

More information on all of these is provided in the coming sections.

2.3.1 Procedure Semantics

Within the `SemanticVisitor`, you will need to make sure that the rules of procedures are followed.

Typing: All procedures in Blaise have return type `int`. Any arguments must be of type `int` as well.

Main: A valid Blaise program must contain a procedure with the following header:

```
procedure int main()
```

`main` must take no arguments. This will be the procedure that is run by your generated MIPS code.

Procedure Calls: Your procedures should be able to call other procedures, as well as themselves recursively. However, a procedure may call only procedures that have already been declared earlier in the code (meaning no mutual recursion). Remember that there is no forward declaration[1] permitted by the grammar. You'll find that this simplifies your semantic analysis.

Arguments: As already mentioned, arguments must be of type `int` (but see the extra credit handout). Your procedures should be able to have an unlimited number of arguments.

Returning: The instruction `return` ⟨expr⟩ `;` is used to return from a procedure. Other instructions may follow a `return` statement, even if they will never be executed. You should not check to make sure that a procedure actually returns a value. If the end of a procedure is reached without a `return` instruction, your procedure should implicitly call `return 0;`.

Overloading: Blaise does not allow procedure overloading. That is, Blaise code containing two procedures with the same name is invalid, regardless of the arguments of the procedures.

Examples: Several examples of valid and invalid uses of procedures are available in the stencil. Each file has been commented with an explanation of why it does or doesn't work.

Note that you need to ensure that the procedure being invoked has already been declared and that the correct number of arguments are provided. We have provided you with a `Map` of `Procedure` objects for this purpose. You will need to pass this `Map` to the code generation stage, as the map will contain information needed in code generation. See below for a description of the `Procedure` class.

You should perform extensive testing here to ensure that all of the requirements for procedure semantics are met.

2.3.2 Static Scoping

In Blaise, variables can be declared anywhere in the program. Moreover, the location of the variable's declaration affects what code can access that variable.

There are two types of variables: global variables and local variables.

Global variables are declared at the beginning of the program, before any procedure declarations, using the keyword `var`. These variables should be accessible within all of your Blaise procedures. That is, they should exist in the outermost scope of your program.

Local variables exist only in the block (and thus procedure) in which they were declared (and any blocks therein). A block is any section of code contained in curly braces. In addition, remember that `if` and `else` statements, `while` loops, and procedure declarations implicitly create blocks for everything contained within them, whether or not they are followed by curly braces. You should consider procedure arguments to be local variables defined within the scope of their procedure.

Variable shadowing, or the declaration of a variable in a scope when a variable of the same name already exists in an outer scope, is not allowed in Blaise. Note that an array and an `int` in the same scope may not have the same name. This means that it is your responsibility here to ensure that a variable is not declared multiple times in the same scope. Examples of programs that are invalid because of variable shadowing can be seen in the stencil.

[1] A forward declaration is a signature without a procedure body. These allow programmers to refer to procedures that have not been fully defined earlier in the code in languages like C.

Variable shadowing isn't the only scoping problem you might encounter. You must make sure that all variables exist in any scope in which they're used. Again, see the stencil for specific examples.

Note that your compiler does not need to keep track of what variables have been assigned to a value in each scope; as long as a variable has been declared, it may be used. A variable whose value has not been set will just take on the value present at the memory location at which it is being stored. This can be somewhat confusing, so let us know if you have any questions.

To implement scoping, your semantic analysis phase must maintain a stack that keeps track of the current scope's variables; whenever a variable, including a procedure argument, is declared, that declaration must be added to the top of the stack. When a given scope ends, all the variables declared in that scope must be removed from the stack.

For examples of scoping, see the commented examples in the stencil.

2.3.3 Type Analysis

Blaise expressions come in two varieties, logical (boolean-valued) and arithmetic (integer-valued). For instance, `3 + 5 * (32 + a)` is an arithmetic expression — its "semantic value" is an integer. On the other hand, `(a == b) && (3 < (4 + c))` is a logical expression; its semantic value is a boolean. Note that the logical expression has an arithmetic expression as a subpart, but the final result is still logical.

Remember that one cannot create a bool as a variable - they only exist inside of `if` or `while` statements.

Consider the following expression: `a + (b == c)`. This expression is perfectly valid from the parser's perspective, but is clearly meaningless[2]. We are trying to add an integer (`a`) to a boolean (`(b == c)`). In this case, the Blaise grammar accepts a *superset* of the programs that are actually valid Blaise programs. Just as with the aforementioned erroneous attempt to access undeclared variables, it is the responsibility of the semantic analysis phase to detect these errors.

Why can't we have the parser catch this class of errors? Because there is no LL(k) grammar that parses such a language, that's why[3]!

To this end, we consider the *semantic type* of each expression in the parse tree. We can start at the bottom of the tree (by walking our way down from the root) and work our way up, checking the types of each expression recursively. The semantic type of an integer constant is, obviously, an integer. Likewise, the semantic type of a scalar variable or an array access is an integer[4]. All the arithmetic operators (`+`, `-`, `*`, `/`, and `%`) operate only on integers, and produce an integer result (in addition, relation operators, such as `==`, should only work on integers, not arrays). Thus, the semantic type of the result of any arithmetic operation is an integer. Finally, all relational and logical expressions produce boolean results, so their semantic type is a boolean. Consider the following examples:

[2]except in C

[3]for a fun exercise, write an LALR or GLR parser. Or a parser generator.

[4] In a more complex language with multiple types of variables, and/or operator overloading (such as Java or C++), we would have to track the type of each variable and check it against the operations being performed on it.

`a`	integer
`5`	integer
`a + 5`	integer
`a + 5 - (b * 32)`	integer
`a > b`	boolean
`a >= (b + 3)`	boolean
`(a >= (b + 3)) && (c < 19)`	boolean
`!(c == 0) || (foo == 3)`	boolean

In some languages, the result of an application of an expression might depend on the types of its operands. For instance, in Java, the expression `a + 5` would have an integer semantic type if the variable `a` was declared as an integer, but would have a floating-point semantic type if `a` was a `double`. In Blaise, however, each operator can only produce one type of value: integers or booleans. Therefore, each node class that inherits from `NodeExpr` implements a method `getSemanticType()` that returns its semantic type.

Note that Blaise also has arrays of integers. Array indices are zero-based in Blaise. You are not, however, responsible for checking to see that an array index is "within range" (can you see in what cases this would not be possible?). Additionally, unlike in Java, arrays cannot be assigned to other arrays. For example, the following program should generate a semantic error:

```
procedure int main() {
  int a[4];
  int b[4];
  a := b;
}
```

You must ensure that your arrays are used as arrays and your singles are used as singles. If `a` is a single, then `a[2]` has no meaning. If `b` is an array, then `b := 3` is invalid. The `input` instruction cannot take in an array, only a single.

The semantic visitor must, in addition to checking variable references, check the types of all the operands of a given expression. For instance, when it processes a `NodeExprPlus`, it should check the semantic types of both the left and right subexpressions — if either is not `INTEGER`, then the semantic visitor should reject the expression and produce an error.

`output` takes only integer-valued expressions. It is thus an error to say: `output(a == b)`.

For a more formal treatment of type safety, take CS173, Introduction to Programming Languages.

2.3.4 Memory Allocation

Part of the job of your Semantic Visitor is to gather information about what variables are declared, their size, and where they are used, and to pass that onto the Code Generation phase so that you can correctly generate the MIPS declarations.

Memory allocation in a Blaise program, however, is not trivial.

Global variables and local variables will be handled differently during codegen, so we want to keep them separate here.

First, let's consider global variables. Global variables will be declared in the `.data` section. You will need to keep track of their names and how large they are when you encounter them. We have provided a `Map` called `_blockSizes` for this. This will eventually be passed to the code generation phase. It's up to you exactly how you use it.

Now, we must consider local variables. Unlike global variables, local ones will be stored on the stack in a frame, as taught in lecture. Frames will be explained in more detail in the codegen handout. In short, space is allocated on the stack for our local variables, arguments, and saved registers whenever we enter a procedure, so it's necessary to know how much space to allocate and where in the frame each variable is.

Our frames act like those on hw09. That is, the frame pointer points to the word on the stack after the frame. From the frame pointer to the end of the frame are, in order, the stored registers, the local variables (in the **reverse** order in which they were encountered), and arguments, also in reverse order. One needs the offset of a variable or argument from the frame pointer to access the variable or argument. All of these offsets will be positive integers.

For example, the Blaise procedure

```
procedure int useFrames(int a, int b) {
    int c;
    int d;
}
```

would be associated with the following frame, with the frame pointer at the bottom and the stack growing downward:

Frame
a (4 bytes)
b (4 bytes)
c (4 bytes)
d (4 bytes)
pushed registers (40 bytes)

Therefore, for each local variable, we will want to know the name of the variable, how much space to allocate for it, where on the stack it will be, and which procedure it's in. To keep track of which procedure a variable is in, you will want to make use of the `_currentProcedure` field we have provided you with. We will use the `Procedure` class to determine how much space to allocate and where on the stack it will be.

However, this task is not as easy as it sounds. For example, we can have multiple variable declarations of the same name within a procedure.

Consider the following program:

```
procedure int main() {
  if(true) {
    int a;
    a := 0;
  } endif
  if(true) {
    int a[10];
    a[0] := 5;
  } endif
}
```

We clearly cannot place both variables called `a` in the same block of memory, since one is an array (and takes up more space) and one is a scalar. There are several ways to deal with this problem. The one that we have implemented for you in the `Procedure` class is to allocate a block of 10 words for both variables, and use all ten when `a` is an array, and ignore all but the first when `a` is a scalar. In this case, the semantic analysis phase needs to keep track of the largest block used by a variable of a given name, and then pass this information to the code generation visitor.

As you go through each procedure, you should use the appropriate `Procedure` object in the manner described below. At the end of semantic analysis, the information stored in your `Procedure` objects will be passed to code generation.

This can be confusing, so please ask us if you have any questions!

2.4 The Procedure class

You will find the `Procedure` class to be very helpful throughout semantic analysis. This class stores the name and argument names (and so number of arguments) for a procedure. This will let you know how many arguments should be passed, which will help you when ensuring the semantics of a procedure call are correct.

This class is also useful for memory allocation for local variables. Every time you encounter a local variable or argument declaration, you should call the `insert()` method for the appropriate `Procedure` object. Once you have finished performing semantic analysis on that procedure, you must call `finish()`. This combination of method calls will determine how much space is needed for each variable on that procedure's frame, as well as how far that variable is from the frame pointer. This information is important for code generation.

Although you won't use this method now, the `Procedure` class also has a `lookup()` method. This will be used to obtain a variable's offset from the frame pointer once `finish()` has been called.

3 Handing in

To hand in, type `cs031_handin comp_semantic`. It will prompt you for confirmation and then hand in your entire current working directory.

As with Parser, we will release the TA solution once the handin deadline has passed.

You **may** hand in this portion of the assignment late and may use your late days.