Semantic Analysis

1 Overview

In the previous project, you have completed the tasks of lexical and syntax analysis, which end up with a syntax tree representation for a syntactically-valid bpl program. Informally, semantic analysis ensures that the program has a well-defined meaning, in other words, via semantic analysis, your compiler finally knows whether an input bpl program is legal or not.

Unlike your experience in the previous project, to implement the semantic analyzer, you will not be provided with any handy tools. Instead, you are expected to write all the code by yourself. Semantic analysis is not the most difficult task for implementing the bpl compiler, however, it is the most laborious one. To realize the semantic analysis effectively and efficiently, you need to design various data structures, such as symbol table and data type representation, and carefully consider what is their most appropriate implementation.

We have defined bpl tokens by *regular expressions* and bpl syntax by a *context-free grammar*. In this project, we will not use more expressive formal languages (i.e., context-sensitive grammar) for semantic analysis. Instead, we use *attribute grammar*, which assigns each symbol in the CFG a set of actions. Actually, we already used attribute grammar for syntax tree generation during parsing in the previous project. Assigning all symbols with solely *synthesized attributes* allows you to make a one-pass parser, which builds the syntax tree and performs semantic analysis simultaneously. Running in one-pass is definitely more efficient, though it is preferable to separate these two stages (build the tree and then traverse it to analyze semantics) for better modularity and flexibility.

You will start semantic analysis based on the previous project, and will continue the subsequent parts of your bpl compiler on top of this work, hence it is important to keep your code maintainable and extensible.

2 Lab Environment

We Please continue to use the virtual environment used in the previous lab to complete this project. We will evaluate your work on exactly the same virtual environment.

3 Symbol Table

3.1 Overview

A *symbol table* maps a name to its associated information. During semantic analysis, the compiler will continuously updates the table with information about what is in scope. Here, the notion "name" is the *identifier* which includes (but not limited to): variable names, function names, user-defined type names, labels; "information" has a broader meaning, such as the type of identifier, the data type, array's dimension, numerical values, etc..

Two typical operations on the symbol table are **insert** and **lookup** a particular symbol. When the analyzer encounters a declaration or definition statement, it should insert the declared identifier into the symbol table, together with the corresponding information. Lookup is more commonly used, for example:

- Before inserting a new name, the analyzer will check whether it is already defined.
- The analyzer should check whether the actual type of an expression matches the declared type.
- In some strongly-typed programming languages, there are operations that strictly permit only specific types of operands.
- When generating code, the symbol table can provide information such as type length, structure member offset, etc..

Unlike the previous project, there is no restriction on the symbol table implementation, hence you are largely free to design the symbol table. Here, "free" has two meanings:

1. There are no definite structure for a symbol table, it can be built with any abstract data type (3.2), and there can be more than one table. You can organize all symbols into a single table, or separate the symbols for variables, functions, types, etc.. Multiple tables can be parallel, or form a hierarchical structure in case of scope checking (3.3).

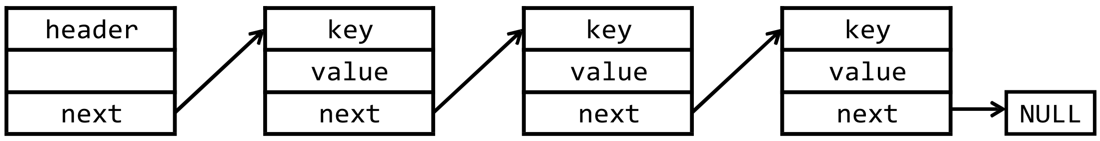
[1] http://adtinfo.org/

2. A symbol table is generally a collection of *key-value* pairs, in which the key is the symbol itself and the value can be any information related to that symbol. While you can put anything in a symbol table, we suggest that you only record the type information for a variable symbol, and the return type and parameter list for a function symbol.

3.2 Abstract Data Types

There are various of abstract data types that support insert and lookup operations. These data structures differ in space/time complexities as well as implementation difficulty. Here we list some commonly used ADTs.

3.2.1 Linked List



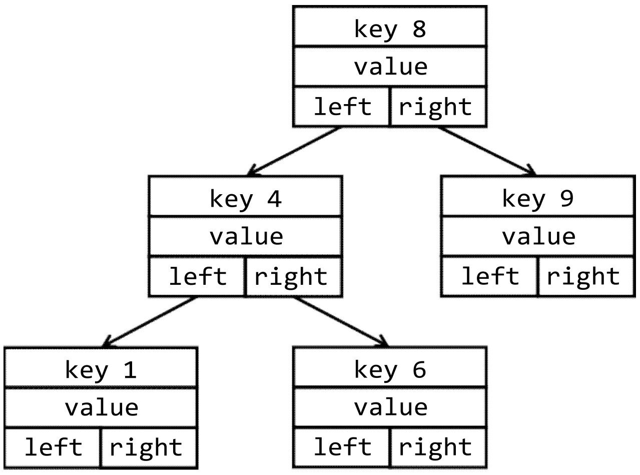
In a linked list symbol table, all symbols are organized in a sequential order. To insert a new symbol, we can simply put it right after the header node, and the time complexity is $O(1)$. However, to lookup a particular symbol, in the worst cases, we need to go through the whole list ($O(n)$ time complexity). Deleting a symbol is the same as lookup, since we should know whether the symbol exists before deleting it.

Clearly, linked list is not efficient for lookup and delete operations. When there are many symbols in the table, these operations can be time-consuming. Nevertheless, the linked list is easy to implement. If you know beforehand that there are only a few symbols during semantic analysis, using a linked list to build the symbol table may be a wise choice.

3.2.2 Binary Search Tree

Linked list only supports sequential search, while binary tree supports binary search, of which the time complexity to $O(\log n)$. A binary search tree node generally has two pointer fields, one to its left child, another to the right one. The structural property of a binary search tree is that, all keys in the left subtree are strictly less than the root's key, while all keys in the right subtree are greater than the root's key. With such a property, a binary search tree is able to lookup/delete in logarithmic time. However, the insert time is also logarithmic, since it should firstly find a suitable position to insert.

However, a binary search tree may degenerate to a linked list, depending on the order of insert and delete. To overcome this limitation, many balance strategies are proposed. Typical variants include *AVL tree* or *red-black tree*. These variants propose more strict structural properties to ensure the tree never degenerates, and the operations are in $O(\log n)$ time on average. However, implementing a binary search tree with an applicable balance



strategy is rather complicated. Binary search tree achieves good trade-off between space and time complexity, hence it is commonly used in some performance-sensitive systems. For example, JDK's TreeMap collection is implemented using red-black tree.

3.2.3 Hash Table

The time complexity of lookup in a hash table is $O(1)$. A good hash function may provide constant time for all operations (in average). Comparing to other ADTs, hash table is rather simple to implement: allocate a large consecutive memory space, design a hash function for the given key type (ASCII strings for a symbol table), then insert the key-value at the corresponding position. Note that hash table consumes more space than binary tree, though it is not critical in our project.

A hash function "compresses" the key into a fixed-range index. A good function can produce evenly distributed values with respect to the input string. How to design a good hash function is beyond the scope of our course. You can find many practical examples of hash function in https://github.com/liushoukai/node-hashes. A good candidate is the hash function proposed by P. J. Weinberger[2].

The hash function inevitably causes collisions, in which multiple keys map to the same index value. To resolve hash collisions, there are generally two approaches: **separate chaining** and **open addressing**. The former defines each table element to be the head of a linked list, and appends the collided keys to the same list. The latter is also called *rehashing*, which re-computes the hash value by alternative hash functions to find next available position. You can also employ other advanced techniques, such as *multiplicative hash function* or *universal hash function*, to make your hash table more evenly distributed.

[2]https://en.wikipedia.org/wiki/PJW_hash_function

3.3 Scope Checking

A *scope* is a section of program text enclosed by basic program delimiters, e.g.{}, in C, or begin-end in Pascal. With scope, variables are only visible within certain sections. Optionally, our bpl language can provide scoping. Consider the following code:

Listing 1: Sample bpl program (scoping)

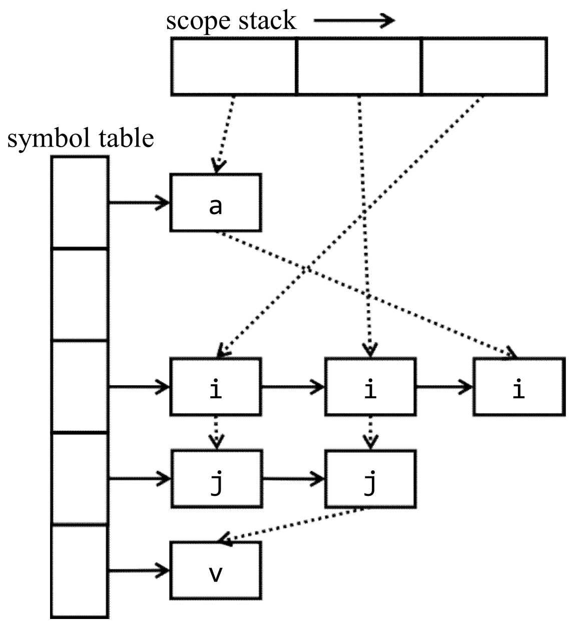
```
1    int test_2_o01(){
2        int a, b, c;
3        a = a + b;
4        if(b > 0){
5            int a = c * 7;
6            b = b - a;
7        }
8        return a + c*b;
9    }
```

The function defines variable a in line 2. Another a is defined in the if-statement's scope. For a compiler only supports global scope, it should report a semantic error at line 5 since it redefines the same identifier. As for a compiler with scope, this bpl program is semantically valid, but the three usages of a should refer to different variables. In line 3, the outer a is increased by b; in line 6, b is decreased by the inner a which shadows the outer definition; in the return statement, the outer a is referenced. However, the usages of b and c inside the if-statement should point to the outer scope definition.

Scope checking refers to the process of determining if an identifier (symbol) is accessible at that location in the program. There are two common approaches to implement a scoping symbol table.

The first approach is *imperative-style* implementation, in which only one single table is maintained globally. We can implement a hash table with separated chaining, and insert duplicated key to the head of its corresponding linked list. The innermost definition always appears at the head of list. However, when the current scope is enclosed, we should pop out all symbols in the scope. To prevent traversing the whole hash table whenever a scope is closed, you may consider to utilize *orthogonal list* data structure.
In the orthogonal list shown above, three cascading scope is defined. The outermost scope is at the bottom of the stack, in which two symbols, namely a and **i** are declared. In the second level, symbols **i**, **j** and v are declared, and this symbol **i** shadows the outer declaration. Similar for the top scope on the stack. When the top scope is popped, we can trace through the list and find that only symbols i and j are defined inside, hence efficiently remove these two keys from the head of their corresponding lists.

Another solution for imperative-style symbol table is assigning each scope with a depth number. Each name in the symbol table is inserted along with the depth number of its containing scope. A name may appear in the symbol table more than once as long as each repetition has a different depth. To lookup a certain name, the symbol table should always return the name with the innermost depth number. Such implementation is similar to an open-addressing hash table.



Another implementation style is so called *functional-style*, which separates individual symbol table for each scope. Under such implementation, we organize all these symbol tables into a scope stack, the innermost scope is stored at the top of the stack, with the next containing scope that is underneath it, etc.. When a new scope is opened, a new symbol table is created and the variables declared in that scope are inserted into the new table. We then push the symbol table on the scope stack. When a scope is closed, the top symbol table is popped. To find a symbol, we start at the top of the stack and work our way down until we find it. If we do not find it, the variable is not accessible and an error should be generated.

In addition to the obvious overhead of creating/removing symbol tables, the functional-style has a disadvantage, that is, all global variables will appear at the bottom of the stack, so scope checking of a program that accesses a lot of global variables can run slowly. Nevertheless, this approach has the advantage that each symbol table, once populated, remains immutable for the rest of the compilation process. Often times, immutable data structures lead to more robust code.

4 Type Checking

4.1 Type System

In programming language, a *type* is a set of values and a set of operations operating on those values. In bpl, there are two categories of data types:

- **Primitive Types** are the base types of the language. These types are provided directly by the underlying hardware. They are int, char and float.

- **Derived Types** are constructed by aggregating primitive types or derived types. They are arrays, structures in bpl, and pointers, union in C, etc..

Type checking is the process of verifying that each operation executed in a program respects the type system of the language. This generally means that all operands in any expression are of appropriate types and number. Much of what we do in the semantic analysis phase is type checking. Type checking can be done in compilation, during execution, or divided across both.

A type checker typically should identify the language constructs that have types associated with them, and check them against the predefined semantic rules. In bpl, there are three language constructs:

- **Variable:** All variables declared in a bpl program must have a type, either a primitive type or a derived type.
- **Function:** Each function has a return type. Each parameter in the function definition has a type, which is also associated with a corresponding argument in a function call.
- **Expression:** Each expression has a type based on the type of the variables in the expression, return types of the called functions, or the operands in the expressions.

We list the semantic rules in Section 5.1. You should carefully think about how they can be realized under the bpl type system and perform type checking at the proper syntax node.

4.2 Type Equivalence

How to tell whether two types are equivalent? The question is trivial for primitive types: an int is equivalent only to another int, a float only to another float, etc..

However, things get complicated for derived types. Actually, whether two derived types are equivalent depends on how we define "equivalence". Typically, there are two kinds of type equivalence: *named equivalence* and *structural equivalence*. For languages supporting named equivalence, two types are considered equivalent if and only if they have the same name. For example, struct st {int x; } and struct st {char y;} are equivalent types, regardless of their actual definitions. Structural equivalence is more general. Two types are structurally equivalent if a recursive traversal of the two type definition trees leads to completely the same results.

Clearly, checking types for a language supporting named equivalence is easier and quicker than doing that for a language supporting structural equivalence. But the trade-off is, named equivalence does not always reflect what is really being represented in a derived type.

4.3 Derived Types Representation

At implementation level, it is easy to represent primitive types: using constants is sufficient. As for the derived types, arrays and structures, representing them are rather complicated.

For example, we can define arrays of structures, as well as a structure with an array (what's more, a multi-dimensional array!) field.

A common technique is to store the basic information, defining the type as multilevel linked list. We provide the follow definition of a Type class.

Listing 2: Type class definition

```c
typedef struct Type {
    char name[32];
    enum { PRIMITIVE, ARRAY, STRUCTURE } category;
    union {
        enum { INT, FLOAT, CHAR } primitive;
        struct Array *array;
        struct FieldList *structure;
    };
} Type;

typedef struct Array {
    struct Type *base;
    int size;
} Array;

typedef struct FieldList {
    char name[32];
    struct Type *type;
    struct FieldList *next;
} FieldList;
```

We separate three categories, and organize their representation as a union. Note that the name is 32 bytes, since we already make the assumption that no identifier name's length can exceed 32. We can safely represent primitive types as constants. For arrays, we should record its base type and the size. Note that this representation also supports multi-dimensional arrays, since the base type can be another array. As for structure type, we record each field sequentially.

The definition above conceptually represents a derived type as a tree structure. Checking named equivalence can be done by passing the name field of two types to strncmp. For structural equivalence, you should traverse the type trees to see whether they follow the same structural properties.

Checking type equivalence is much more complicated for real-world compilers. A rigorous and practical type checker should formally infer types from multilevel language constructs, and support more language features, such as pointers, object-oriented classes, explicit or implicit type coercion, etc.. Though you are not required to implement them, you should be aware of the fact that type checking is generally much more difficult than what you are required to do in this project.

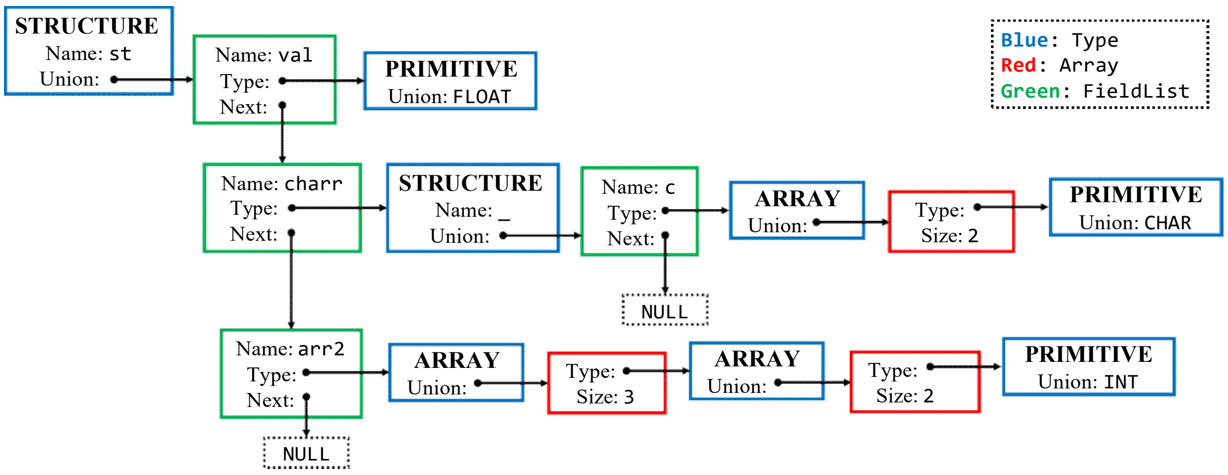


Figure 1: Tree representation of a structure type

5 Project Requirements

5.1 Semantic Rules

5.1.1 Assumptions

Before we presenting the semantic rules, we shall make some assumptions. The assumptions are restricted in the test cases, which means you don't need to consider their violations. Nevertheless, they can be your bonus rules (5.1.3), since a real-world compiler should be able to detect as many errors as possible.

Assumption 1 char variables only occur in assignment operations or function parameters/arguments

Assumption 2 only int variables can do boolean operations

Assumption 3 only int and float variables can do arithmetic operations

Assumption 4 no nested function definitions

Assumption 5 field names in struct definitions are unique (in any scope), i.e., the names of struct fields, and variables never overlapped

Assumption 6 there are only global scope, i.e., all variable names are unique

Assumption 7 using *named equivalence* to determine whether two struct types are equivalent

5.1.2 Required rules

Your analyzer should be able to detect the following semantic errors:

Type 1 variable is used without definition

Type 2 function is invoked without definition

Type 3 variable is redefined in the same scope

Type 4 function is redefined (in the global scope, since we don't have nested function)

Type 5 unmatching types on both sides of assignment operator (=)

Type 6 rvalue on the left side of assignment operator

Type 7 unmatching operands, such as adding an integer to a structure variable

Type 8 the function's return value type mismatches the declared type

Type 9 the function's arguments mismatch the declared parameters (either types or numbers, or both)

Type 10 applying indexing operator ([...]) on non-array type variables

Type 11 applying function invocation operator (foo(...)) on non-function names

Type 12 array indexing with non-integer type expression

Type 13 accessing member of non-structure variable (i.e., misuse the dot operator)

Type 14 accessing an undefined structure member

Type 15 redefine the same structure type

If your analyzer finds a semantic error in the bpl program, it should report the error type (as listed above) and line number with respect to the following format:

```
Error type [TypeID] at Line [LineNo]: [message]
```

We do not constrain the content of the error message, however, your analyzer must give the correct TypeID and LineNo. We will verify the output based on these two fields.

To show how your analyzer works, consider the following bpl program containing semantic errors:

Listing 3: bpl test case for semantic analysis

```
 1 struct Apple
 2 {
 3     int weight;
 4     float round;
 5 };
 6 int test_2_r07()
 7 {
 8     struct Apple aa;
 9     float weight_test = 1.0;
10     aa.weight = aa + 2;
11     return 0;
12 }
```

In the program, line 10 contains two errors, your analyzer may report as follow:

```
Error type 7 at Line 10: binary operation on non-number variables
Error type 5 at Line 10: unmatching type on both sides of assignment
```

Basically, variable aa is a struct Apple type, so adding it with an integer value is semantically illegal, which refers to error type 7. The expression aa + 2 then evaluates to an undefined type (or you can assign it with some customized default value, since it is, namely, undefined), which cannot be assigned to an integer type lvalue aa.weight, hence type 5 is also reported.

You must have questions like: "Which is the best error to report?" and "Is this considered a cascade from this earlier error or not?" These are perfectly valid questions, but don't be surprised if we're a bit hesitant to answer. We want you to think through and make these sorts of decisions by yourself. Put yourself in the position of someone using your compiler, and try your best to provide the most helpful error type/message.

Once an error is reported, you may suppress any subsequent errors cascading from that problem, such as ignoring type 5 error from the example above. However, if your analyzer ignores such cases in purpose, you should justify your design and explain your implementation in the report, and we will evaluate your results accordingly.

5.2 Implementation Requirement

The input format is the same as the previous project, i.e., the executable bplc accepts a single command line argument representing the bpl program path. For a semantically legal bpl program, your semantic analyzer shall not produce any output message, otherwise, a meaningful error message should be printed out.

You should compile your analyzer as the bplc target in Makefile, and move the executable to the bin directory. For example, the Makefile is placed under the project root directory, then we make the bplc target by:

```
make bplc
```

which generates the semantic analyzer executable. Then we run it by:

```
bin/bplc test/test_2_r01.bpl
```

The analyzer should output the error message in a text file test/test_2_r01.out, if any semantic error is presented. If no error, touch an empty file.

5.3 Grading Policy

The maximum score of this project, excluding the bonus part, is 100 points. Your score depends on the number of test cases your analyzer can pass. If your analyzer passed all test cases, you will get 100 points.Failing to pass some test cases may result in the decrease of your score.

6 Submission

What to submit You are required to submit your C/flex/bison source code and other related files in a .zip archive with file name format: **StudentID-projectNumber.zip** (e.g., 2023140924-project2.zip). In this project, the zip file tree is:

```
StudentID-project2/
    bin/
        bplc        // generated
    report/
        StudentID-project2.pdf
    test/
        test_2_r01.bpl
        test_2_r01.out
        test_2_r02.bpl
        test_2_r02.out
        ...
        Makefile
    ...             // C/Flex/Bison source code
```

- bin directory contains a single executable file named bplc, which is generated by an bplc target in the Makefile. Make sure it works properly in our environment (Section 2).
-

- test directory is your self-written test cases. Make sure to include your output files in the directory. The code file name should end with .bpl extension and output file name should end with .out. Remember to include your student ID in the file names.
- The Makefile is provided. You can add any targets, but most importantly, **you should ensure the** bplc **target compiles and generates an executable** bplc **in the** bin/ **directory**. Otherwise, we cannot evaluate your work and you will get a zero in project 2.
- In this project, you will write code in C/Flex/Bison, you can place them directly under the submitted directory, or under separate folders like src and include. Again, you must ensure the code can be compiled successfully, no matter where they are.

How to submit You should upload your zip file on 教学云平台 before the **11:55 PM, Dec. 10, 2023**.