Project 3: Semantic Analysis

Introduction to Compilers
TA: Sanghyeon Lee (sanghyeon@snu.ac.kr)

Released: November 12, 2024
Due: December 4, 2024

Contents

1 Introduction **1**
1.1 Semantic analysis . . . 1
1.2 Building a semantic analyzer . . . 1

2 Environment Setup **2**
2.1 Pull the repository . . . 2
2.2 How to build and test . . . 2

3 List of Semantic Checks **2**
3.1 Undeclared variables and functions . . . 2
3.2 Redeclarations . . . 2
3.3 Type checking . . . 2
3.4 Structure & structure pointer declarations . . . 4
3.5 Function declarations . . . 4

4 Goal of the Project **4**
4.1 Output format . . . 4

5 Submission **5**

6 Tips **5**

1 Introduction

1.1 Semantic analysis

In programming languages, the term "semantics" generally refers to the behavior of a program during execution. Semantic analysis is the process of validating the semantics of a program, after the lexer and parser have finished analyzing its structure.

Here are some examples of the static semantic checks performed by the semantic analyzer at compile-time.

- Type checking
- Keeping track of the declarations
- Checking the scope of an object

1.2 Building a semantic analyzer

Parser generators can automatically build a parser with only a few lines of grammar rules. Similarly to lexers, there is no need to build a parser from scratch. GNU Bison is a widely used open-source LALR parser generator which is upward compatible with POSIX yacc (Yet Another Compiler-Compiler), a standard parser generator on Unix. Throughout this course, we will utilize Bison to generate parsers for subC.

Unfortunately, there is no automatic semantic analyzer generator available. Automatic compilation of a language's semantics is currently beyond the state of the art. One reason for this is that a program's semantics

are not fully determined at compile-time. For example, out-of-bound accesses for dynamic arrays only can be deteremined at run-time. Therefore, a semantic analyzer should be built using some heuristics.

In this project, we will implement semantic analysis using GNU Bison, as a continuation of the previous project.

2 Environment Setup

2.1 Pull the repository

Attach to the container and pull the repository to set up the environment for this project.

```
docker exec -it 2024-compilers bash
cd 2024-compilers
git pull
```

2.2 How to build and test

Navigate to the `src` directory, and build the project using `make` command.

```
cd ./src
make all
```

Once the build is completed, you can test it with the following command. If you do not specify the `[input_file]`, `stdin` will be used as the input stream.

```
./subc [input_file]
```

3 List of Semantic Checks

This section describes the semantic validations which should be implemented in this project and the corresponding error messages.

3.1 Undeclared variables and functions

- If a variable or a function is used before its declaration:

  ```
  input.c:10: error: use of undeclared identifier
  ```

  Please assume that there are no implicitly declared functions, recursive functions or structs which contain themselves as a member.

3.2 Redeclarations

- If a variable is redefined in the same scope:
- Or, if a function or struct is redefined in any scope:

  ```
  input.c:10: error: redeclaration
  ```

  Please assume that functions cannot be overloaded.

3.3 Type checking

3.3.1 Assignment operator (=)

Semantic checks for the assignment operator must be performed in order from top to bottom in the list below.

1. If the left-hand side of an assignment operation is not a variable:

   ```
   input.c:10: error: lvalue is not assignable
   ```

2. If the right-hand side is `NULL` but the left-hand side is not a pointer type:

   ```
   input.c:10: error: cannot assign 'NULL' to non-pointer type
   ```

3. If the types of the left-hand side and right-hand side are not the same:

```
input.c:10: error: incompatible types for assignment operation
```

Please assume that there are no assignments of a string constant to a pointer type.
e.g. `char *s = "Hello World";`

3.3.2 Binary and logical operators (+, -, *, /, %, &&, ||, !)

- If either operand is not an `int` type:

```
input.c:10: error: invalid operands to binary expression
```

3.3.3 Unary operators (-, ++, --)

- If the operand of 'unary `-`' is not an `int` type:
- Or, if the operand of '`++`, `--`' is not an `int` or `char` type:

```
input.c:10: error: invalid argument type to unary expression
```

3.3.4 Relational operators (>=, >, <=, <)

- If the operands are not both `int` or both `char` types:

```
input.c:10: error: types are not comparable in binary expression
```

3.3.5 Equality operators (==, !=)

- If the operands are not both `int`, both `char`, or both pointers of same types (including `struct*`):

```
input.c:10: error: types are not comparable in binary expression
```

Please assume that there are no operations between the pointer type and array type, or between the array type and array type. For example:

```
int *a;
int b[9];
a = b;
a == b;
```

3.3.6 Indirection operator (unary *)

- If the operand is not a pointer type:

```
input.c:10: error: indirection requires pointer operand
```

3.3.7 Address-of operator (unary &)

- If the operand is not a variable:

```
input.c:10: error: cannot take the address of an rvalue
```

3.3.8 Member access operators (->, .)

- If the left-hand side of '`.`' is not a struct type:

```
input.c:10: error: member reference base type is not a struct
```

- If the left-hand side of '`->`' is not a struct pointer type:

```
input.c:10: error: member reference base type is not a struct pointer
```

- If the right-hand side is not a member of a struct:

```
input.c:10: error: no such member in struct
```

3.3.9 Subscript operator ([,])

- If a subscript operator is used on non-array variable:

```
input.c:10: error: subscripted value is not an array
```

3.4 Structure & structure pointer declarations

- If an object of an undefined struct type is declared:

```
input.c:10: error: incomplete type
```

3.5 Function declarations

- If the declared return type is different from the type of the return value in its definition:

```
input.c:10: error: incompatible return types
```

  Please assume that there are no functions without a return statement. Also, struct type functions do not need to be type checked.

- If a non-function entity is called:

```
input.c:10: error: not a function
```

- If the function arguments differ from the parameters:

```
input.c:10: error: incompatible arguments in function call
```

4 Goal of the Project

You have to implement semantic analysis in addition to your existing parser. The parser should be able to detect semantic errors in the input program and print the error messages. Additionally, the parser should continue analyzing the program until it reaches the end of the program, even after encountering a semantic error.

Also, you need to implement a scoped symbol table to validate the semantics of a program. You may implement any type of scoped symbol table as long as it supports effective semantic analysis.

Complete the code in `subc.l`, `subc.y`, and `hash.c`. You may add or modify any files in the `src` directory as needed, but please make sure that the project can be built using `make all` command.

In the report, please briefly describe how you implemented the semantic analysis in two pages or less.

4.1 Output format

Each time the parser encounters a semantic error, print the corresponding error message to `stdout` in the following format.

```
$ ./subc <file_name>
<file_name>:<line_number>: error: <description>
```

- Please insert a **space** character, right before and after '`error:`'.
- `<file_name>` is the second command line argument. (i.e. `argv[1]`)
- Utilize `get_lineno()` function to get a line number.
- You don't have to consider the errors that occur across multiple lines.
- If there are multiple errors in the same line, print only one error generated from the earliest reduced production.

Helper functions for each type of semantic error are available in the skeleton code. Please consider using them to avoid typos in error messages. A demonstrative example of an input-output pair is given below.

Disregarding the output rules will result in a penalty to your project score.

I/O Example

Input code: `input.c`

```
int main() {
  int a;
  a = 5;
  a = 'a';  /* Semantic error */
  return 0;
}
```

Output message:

```
$ ./subc input.c
input.c:4: error: incompatible types for assignment operation
```

5 Submission

- Place your report somewhere in the project directory. The file name must be `report.pdf`.
- Run `./submit.sh <student_number>` in the container. For example, `./submit.sh 1234-56789`. It will compress `src` directory and `report.pdf`.
- Submit the compressed archive on eTL.
- Please make sure that the name of the archive is `project3_<student_number>.zip` and your student number is correct. **Incorrect student number may result in a penalty to your project score.**
- For delayed submissions, there will be a 20% deduction in scores per day.

6 Tips

- **Midrule Actions** It is highly recommended to utilize the midrule actions of bison. You may need to perform some actions in the middle of the productions to analyze the semantics.
- **Modifying the Grammar** Midrule actions may introduce new conflicts in the grammar. It is possible to modify the grammar to resolve the conflicts. But, please ensure that the modified grammar **MUST** remain syntactically equivalent to the original. That is, the order of reductions must not be changed after the modification, so that the order of error messages remain unchanged when the multiple errors occur in a line. If you modified the grammar, please describe how you modified the grammar in the report.
- **Tips** The overall design in this project will be carried over to the next project, so please thoroughly review the lecture slides (ch. 7 & 8) and the subC grammar before starting this project.

Appendix

GNU Bison Manual