CS323 Principle of Compiler - Phase 2

Semantic Analysis

陈俊峰 12011423

张闻城 12010324

唐骞 11911922

How to test my compiler

- `cd <project dir>`
- `make -j 8`
- `bin/splc <SPL source file path> 2>[output file path]`

Then output of semantic analysis will be in `[output file path]` (you need to decide where to output by yourself **qwq**).

1. Features

1 1.1 Basic Features

All required basic sematic error type detections are implemented.

2 1.2 Bonus Features

- ✓ All semantic checking is executed during bottom-up parsing, which is relatively efficient.
- ✓ There are local and global scopes. Variables in different scopes can share the same identifier. And variables defined in the outer scope will be shadowed by the inner variables with the same identifier
- ✓ Force type conversion
- ✓ Guess return value type of an undefined function
- More semantic error types:
    - ✓ `break` or `continue` statement not within a non-loop structure
    - ✓ Dereference of a non-pointer variablce
    - ✓ 32-bit signed integer overflow
    - ✓ Hex char literal overflow
    - ✓ Non-boolean expression at conditional statement

2. Design and Implementation

1 2.1 Symbol Table

We implment local scope by **multiple symbol table**. The program has a single global table. Whenever the compiler meets a **LC**, it will create a new local scope and push it into the scope stack. The symbol defined in this scope will be inserted into the corresponding symbol table. When compiler meets a **RC**, it will pop out the symbol table at the top of table stack.

When we need to check whether an identifier has been defined before, we go the stack, from top to bottom, to see if there is a definition in the corresponding symbol table.

2 2.2 Semantic Error check

We implement several class, which is related to corresponding not-terminal, such as `Stmt`, `Exp`, `Specifier` and `Def`. They are all inherited from a base class called `Container`, which has a virtual function `installChildren()` and must be overrided by every derived class. During the bottom-up parsing, whenever a production is reduced, we will call corresponding `installChildern()` function in the derived classed according to **polymorphism**. And we will do semantic check of different semantic error types in corrsponding `installChildren()`.

2.1 2.2.1 Synthesized Attribute

For synthesized attribute, bottom-up parsing will not affect its analysis, since it only needs attribute information in its children nodes, which has been calculated in bottom-up parsing when we meet a certain node. Main logic of semantic analysis are all simply in `installChildren()` function.

2.2 2.2.2 Inherited Attribute

For inherited attribute, bottom-up parsing can't simply complete its analysis when we arrive at a certain snytax tree node, since we need attribute information in other nodes, such as siblings or parents. In this project, we use a trick to complete analysis of inherited attribute. We may not know what is parent or sibling is when we arrive at a certaion syntax tree node, but we have access to the non-terminals or terminals in the parsing stack (`yystack`). We can simple look forward by reading the symbols on stack and get some attribute infromation ahead of time.

For example, current specification is $ Specifier ID LP RP, now we can reduce it to $ Specifier FuncDec. Then we now there will be a new function declaration, and we want to register this function into symbol table but we don't know its return type yet. To get its return type, we simple scan the parsing stack and read the first Specifier an get function return type information from this non-terminal. That's how we analyse inherited attribute in this project.