Project 3 - Semantic Check - Part 1 （非最终版）

2025 年 10 月 27 日

1 项目要求

在 Project 2 中，你已经成功构建了 Splc 语言的解析器，能够将源代码转化为语法树，并检查出语法错误。但是，语法正确并不意味着源代码在逻辑上（即**语义上**）是正确的。

在 Project 3 中，你将进入语义分析的第一步：实现类型系统 (Type System) 和符号表 (Symbol Table)。在 Project 4 中，你将基于前面的类型系统和符号表进行完整的语义检查。

1. **实现类型系统**：在 Java 中实现一套类型系统，并将 Parse Tree 中的变量声明（`specifier` 和 `varDec`）转化为描述该变量的类型的 Java 对象。
2. **实现符号表**：实现基于作用域 (Scope) 的符号表。
3. **执行全面的语义检查**：遍历 Parse Tree，结合符号表找出并**报告所有**发现的语义错误。在本次 Project 中，你只需要关注第三章中所描述的几种语义错误即可。

1.1 对 Project 2 的修订

在开始 Project 3 前，请在 `Splc.g4` 中为 `globalDef` 添加一条新的规则：
函数声明：`specifier Identifier LPAREN funcArgs RPAREN SEMI`

1.2 名词解释

- `type specifier`：（语法结构上的）类型说明符。对应 Project 2 文档章节 2.2 类型说明符。
- `declarator`：（语法结构上的）声明符。在 Splc 语言中包含两种：变量声明 (`varDec`)，会出现在**全局变量定义**、**局部变量定义**、**完整结构体**中的成员声明；以及函数声明 (上述修订新增)。

1.3 解释：`declaration` & `definition`

正式定义：

- [1.3.1] A *declaration* specifies the interpretation and attributes of a set of identifiers.
- [1.3.2] A *definition* of an identifier is a declaration for that identifier that:
  - for an *object* (variable), causes storage to be reserved for that object (variable);
  - for a function, includes the function body.

声明 (Declaration) 就像在电话簿的索引（符号表）里说：“我们要登记一个人，名字叫 x，他是一个 int 类型。”编译器看到声明后，会说：“好的，我认识 x 了。如果我看到有人使用 x，我知道它应该被当作一个 int 来对待。”

声明**不分配内存**（对于变量）或不提供**函数体**（对于函数）。

尽管 C 语言中可以对同一个对象（变量或函数）进行多次声明，**但是在我们的 Splc 中**，不存在变量声明，并且我们规定：同名的函数声明只能出现一次，并且其定义（若有）一定与其声明相符[1]。

定义 (Definition) 就像在电话簿的具体条目里说：“x 的具体住址是内存 0x1000（请在这里给他分配空间）。”或者“myFunc 的具体工作内容是 { ... }（请把这段代码编译成机器码）。”

定义会分配存储空间（对于变量）或提供函数体（对于函数）。

在整个程序中，同一个实体（变量或函数）[2]只能被定义一次。

一个定义同时也是一个声明。它在“创建”这个实体的同时，也“介绍”了它自己。

注意，我们只对变量和函数说 Definition，我们从不对一个完整结构体的声明说它是一个定义。

示例

```
int a() {}      // 这是函数 `a' 的定义，也是一个声明
int b(char);    // 这是函数 `b' 的声明

int b(char);    // 这是函数 `b' 的重复声明，Splc 不允许这种情况

int global_var; // 这是全局变量的定义
int main() {
    global_var = 3;
    int local_var;  // 这是局部变量的定义
}
```

1.4 项目假设

在 Project 3 & 4 中，我们做出如下假设：

1. 所有测试样例均没有**词法错误**与**语法错误**，但可能存在**语义错误**。

1.5 扩展要求

扩展部分在每个 Project 之间都是独立计分的，在后续的 Project 中移除对某扩展部分的支持不影响前序 Project 的分数。也就是说，你可以选择在前面的 Project 中完成较为简单的扩展任务；如果你发现在后续的 Project 中完成扩展部分过于困难，你可以选择不完成后续 Project 的扩展部分，这样不会影响你前面 Project 的分数。

本项目的扩展部分是 Project 2 中的延续：你需要确保你的类型系统能够正确处理 结构体 和 指针 的相关语法。并且能够检查结构体中的 *incomplete type* 的情况。

[1] 判断定义与声明是否相同是下一次 Project 的任务。

[2] 显然，同一个 Identifier 可能指向不同的实体。

2 Project 3 要求

2.1 类型系统

你的类型系统需要能够解析语法树中的 `specifier` 与 `varDec`，并表达以下类型：

- 基础类型（Primitive Type）：包含 `int` 与 `char`。
- 定长数组类型（Array Type）：包含基础类型 (*element type*) 和数组长度，如 `int a[10]` 和 `char b[2][3]`。
- 结构体类型（Structure Type）：包括完整声明的结构体（内含字段声明）与不完整声明的结构体两种。
- 指针类型（Pointer Type）：包含被指向的类型，例如 `int *a`。
- 你支持的所有类型的任意组合，例如 `struct a[10]`、`int *a[123]` 和 `int (*a)[123]`。

完整定义：

- [2.1.1] The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it. (An identifier declared to be an object is the simplest such expression; the type is specified in the declaration of the identifier.)
- [2.1.2] Types are partitioned into *object types* (types that describe objects) and *function types* (types that describe functions). At various points within a translation unit an object type may be *incomplete* (lacking sufficient information to determine the size of objects of that type) or *complete* (having sufficient information).
- [2.1.3] The `char` and `int` are collectively called the *basic types*. The basic types are complete object types.
- [2.1.4] A *function type* describes a function with specified return type. A function type is characterized by its return type and the number and types of its parameters.
- [2.1.5] Any number of *derived types* can be constructed from the object types[3], as follows:
  - [2.1.6] An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*. The element type shall be complete whenever the array type is specified. Array types are characterized by their element type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is T, the array type is sometimes called "array of T". The construction of an array type from an element type is called "array type derivation".
  - [2.1.7] A *structure type* describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.
  - [2.1.8] A *pointer type* may be derived from an object type, called the *referenced type*. A pointer type describes an object whose value provides a reference to an entity of the referenced type. A pointer type derived from the referenced type T is sometimes called "pointer to T". The construction of a pointer type from a referenced type is called "pointer type derivation". A pointer type is a complete object type.

[3]我们在这里移除了 *function types*，即它不能组成其他 Type。

2.1.1 结构体类型 要求

若你实现了 结构体类型 扩展，你需要满足以下要求：

- Project 2 文档 2.2 节 Type Specifier 中关于结构体有两种格式：
  结构体: `struct Identifier` 和 **完整结构体**：`struct Identifier { struct-declaration-list }`.
- [2.1.9] All declarations of structure type that have the same scope and use the same tag declare the same type.
- [2.1.10] Irrespective of what other declarations of the type are in the same translation unit, the type is *incomplete* until immediately after the closing brace of the list defining the content, and *complete* thereafter.
- [2.1.11] Two declarations of structure type which are in different scopes or use different tags declare distinct types.
- [2.1.12] A type specifier of the form **`struct Identifier { struct-declaration-list}`** declares a type. The list defines the *structure content*. It also declares the identifier to be the tag of that type. The type is incomplete until immediately after the `}` that terminates the list, and complete thereafter.
- [2.1.13] A declaration of the form **`struct Identifier;`** specifies a structure type and declares the identifier as a tag of that type.
- [2.1.14] If a type specifier of the form **`struct Identifier`** occurs other than as part of one of the above forms
  - , and no other declaration of the identifier as a tag is visible, then it declares an incomplete structure type, and declares the identifier as the tag of that type.
  - , and a declaration of the identifier as a tag is visible, then it specifies the same type as that other declaration, and does not redeclare that tag.
- [2.1.15] A structure shall not contain a member with incomplete type (hence, a structure shall not contain an instance of itself, but may contain a pointer to an instance of itself).
- [2.1.16] A specific structure type shall have its content defined at most once.

额外地，请区分 *declaration* 和 *type specifier* 的区别；以及请注意 declare a new structure type 和 declare an object with a structure type 的区别。

2.2 符号表

符号表是编译器存储标识符（`Identifier`，如变量名、函数名）信息的数据结构。语义分析需要利用这些信息来验证程序逻辑的正确性。

你的符号表设计应使得 `Identifier` 能满足以下要求：

- [2.2.1] An *identifier* can denote an object (variable); a function; a tag or a member of a structure. The same identifier can denote different entities at different points in the program.
- [2.2.2] For each different entity that an identifier designates, the identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. Different entities designated by the same identifier either have different scopes, or are in different name spaces. There are three kinds of scopes: *file*, *block*, *function prototype*[4].
- [2.2.3] Every identifier has scope determined by the placement of its declaration (in a declarator or type specifier).
  - [2.2.4] If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the translation unit (the compiling file).
  - [2.2.5] If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the end of the associated block.
  - [2.2.6] If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function declarator.
  - [2.2.7] If an identifier designates two different entities in the same name space[5], the scopes might overlap. If so, the scope of one entity (the *inner scope*) will end strictly before the scope of the other entity (the *outer scope*). Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is *hidden* (and not visible) within the inner scope.
- [2.2.8] Two identifiers have the *same scope* if and only if their scopes terminate at the same point.
- [2.2.9] Structure tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. Any other identifier has scope that begins just after the completion of its declarator.
- [2.2.10] If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, there are separate *name spaces* for various categories of identifiers, as follows:
  - the tags of structures.
  - the members of structures; each structure has a separate name space for its member.
  - all other identifiers.

结构体扩展：请注意 *a tag of a structure* 也是 *identifier*。

[4] A function prototype is a declaration of a function that declares the types of its parameters.

[5] 见后续。

3 Project 3 语义错误

3.1 Undeclared Use Error

`Undeclared Use Error` 发生在程序试图使用一个在当前点并**不可见** (not visible) 的标识符时。

一个标识符的作用域从其“声明点”开始：

- 对于**结构体标签** `structure tag`，它的作用域在 tag 在类型说明符 (type specifier) 中出现后立即开始。
- 对于**所有其他标识符**（变量、函数等），作用域在其“声明符 (declarator)”完成之后才开始。

具体而言，当编译器在代码的某个位置遇到一个标识符的使用时，它会根据该标识符的句法上下文（syntactic context）来确定它属于哪个命名空间 (name space)（即，它是 structure tag、structure member，还是“所有其他标识符”）。随后，编译器会检查在当前位置，是否有任何（已开始但尚未终止的）作用域中包含该标识符在对应命名空间中的声明。如果找不到这样的声明，该标识符的使用就被视为“使用未声明的标识符”。

示例 1

```
int func() {
    x = 10; // 错误：'x' 在此点不可见，其声明符尚未完成。
    int x;  // 'x'（属于“所有其他标识符”命名空间）的作用域在此点开始。
    x = 20; // 正确：'x' 现在可见。
}
struct List; // 'List'（属于“标签”命名空间）的作用域在此点开始。
struct Data {
    struct List* next; // 正确：'struct List' 标签是可见的。
};
```

示例 2

```
int main1() {
    a();    // 错误：a 尚未被声明
}
int a(struct ab *q);    // 函数 a 的声明
int main2() {
    a();    // 正确：a 已经被声明
}
int b(int a, char b) {} // 函数 b 的定义，也是一种声明
int main3() {
    b();    // 正确：b 已经被声明
}
```

在本次 Project 中，你只需要检查**所有其他标识符**这一类标识符的使用，对于**结构体标签**的使用详见后续文档，对于**结构体成员**的使用并不在本次 Project 的处理范围之内。

```
struct a { int b; };
int main() {
    struct a var;
    var.c = 123;    // 本次 Project 不要求检查出这样的错误
}
```

3.2 Redefinition Error

`Redefinition Error` 发生在两个或多个定义指向**同一个标识符**[a]，并且**具有相同的作用域时**。

[a]由于上述我们对定义的描述只包含变量和函数，所以这样的标识符一定位于 'other' 命名空间内。

在 Splc 中，适用于 'other' 命名空间的标识符仅有两种：变量名和函数名，其定义只有可能出现在：全局变量定义、局部变量定义、函数定义。

如果两个或多个定义指向同一个标识符，位于 'other' 命名空间，并且具有相同的作用域时，这属于 `Redefinition Error`。

示例 3

```
int y;
int func() {
    int x;
    char x; // Error: redefinition of `x'
}
int y;      // Error: redefinition of `y'

int z;
int z() {}  // Error: redefinition of `z', 它们都是 `other' namespace, 位于 file scope
```

如果两个定义，其标识符相同，但处于重叠的作用域 (overlapping scopes)（一个内层，一个外层），这也是合法的。根据规则，“内层作用域 (inner scope)”的实体会“隐藏 (hidden)”“外层作用域 (outer scope)”的实体。

示例 4

```
int x = 10; // file scope (outer)

int func() {
    int x = 20; // OK: block scope of func (inner)
    print(x);
    // 此处的 'x' 指的是 'x = 20' (内层), 文件作用域的 'x = 10' (外层) 在此被隐藏。
}
```

3.3 Redeclaration Error

`Redeclaration Error` 发生在

- 声明一个函数，并且该标识符已经在 'other' 命名空间内被声明。
- 声明一个包含 *structure content* 的结构体类型（即完整结构体），并且其 *tag* 在同一作用域下已被声明为一个 complete type.[a]

[a]关于什么时候是 declare xxx as a tag of a structure type，详见上述 2.1.1 章节。

示例 5

```
int func() {}   // definition is also a declaration
int func();     // Error: Redeclaration of `func'

int a();
int a();        // Error: Redeclaration of `a'

struct b;       // declaration of structure `b', but incomplete
struct b {
    int a;
    char b;
    struct b *ptr;
};              // the first full structure content declaration.

struct b {
    int a;
    char b;
    struct b *ptr;
} qwq;          // Error: Redeclaration of 'b'

int c;
int c();        // Error Redeclaration of 'c'
```

3.4 Definition of incomplete type

`Definition of incomplete type` 发生在定义一个变量，但是其类型为 *incomplete type* 时：

- The basic types are complete object types.
- The element type of an array type shall be complete whenever the array type is specified.
- A structure type of unknown content is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure tag with its defining content later in the same scope.
- A pointer type is a complete object type.

示例 5

```
struct b b1;     // OK: `struct b' 目前是 incomplete type
                 // 但是后续的 `struct b {};' 与这个是同一 Type.

struct b b2[3]; // Error: definition of incomplete type
                 // 在定义这个 Array 时它是 incomplete type

struct b *ptr;  // OK: ptr 是一个指针，它的大小是已知的，和 struct b 的大小无关。

struct d dd;     // Error: definition of incomplete type

struct b {
    char b;
    struct b *ptr;
};               // `struct b' 是 complete type
struct b b2;     // OK

struct c c0;     // Error: definition of incomplete type
                 // 此处的 'struct c' 与 main 里面的 'struct c' 声明不在同一 scope
                 // 它们不是 the same type.

int main() {
    struct c {
        int c;
        char b[12];
    } varc;
}
```

3.5 Member Incomplete Type

`Member Incomplete Type` 发生在结构体的成员类型为 incomplete type。

示例 6

```c
struct b;
struct a {
    struct b a; // Error: Member 'a' has incomplete type.
};

struct c {
    struct c c0;    // Error: Member 'c' has incomplete type.
};                  // 'struct c' is incomplete until '}'
```

3.6 Structure 相关示例

示例 7

```c
struct T {  // `T' 在此可见，它属于 `tag' namespace
    int T;  // 合法：此处的 `T' 属于 `member' namespace，它不与上述的 `struct T' 冲突
};

int Tx;     // 合法：此处的 `Tx' 属于 `other' namespace
struct Tx;  // 合法：此处的 `Tx' 属于 `tag' namespace
```

示例 8

```c
struct T {
    int x;
    int y;
};

struct T {
    int x;
    int y;
} a;        // Error: Redeclaration of 'struct T'
```